– Fachbereich Informatik –

Distributing Non-Deterministic Simulation Processes in a Computing Cluster using Event Sourcing and Containerization

Thesis to obtain the academic degree

Master of Science (M.Sc.)

submitted by

Alpay Yildiray

Matriculation No.: 4434706

Examiner	:	Prof. Dr. Udo Frese
Co-Examiner	:	Prof. Dr.-Ing. Vasily Ploshikhin

Universität Bremen

Eigenständigkeits- und Einverständniserklärung zur Überprüfung mit Plagiatssoftware sowie die Erklärung zur Veröffentlichung bei Bachelor- und Masterarbeiten

Declarations of Authorship and Consent for Checking with Plagiarism Software and the Declaration of Publication for Bachelor's and Master's Thesis

Studierenden-Angaben / Student Information:

Matrikelnr./ Student ID	4434706
Nachname / Surname	Yildiray
Vorname / First Name	Alpay

Titel der Arbeit / Title of Thesis

Distributing Non-Deterministic Simulation Processes in a Computing Cluster using Event Sourcing and Containerization

A) Eigenständigkeitserklärung / Declaration of Authorship

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Teile meiner Arbeit, die wortwörtlich oder dem Sinn nach anderen Werken entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht. Gleiches gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet, dazu zählen auch KI-basierte Anwendungen oder Werkzeuge.
Die Arbeit wurde in gleicher oder ähnlicher Form noch nicht als Prüfungsleistung eingereicht.

I hereby affirm that I have written the present work independently and have used no sources or aids other than those indicated. All parts of my work that have been taken from other works, either verbatim or in terms of meaning, have been marked as such, indicating the source. The same applies to drawings, sketches, pictorial representations and sources from the Internet, including AI-based applications or tools. The work has not yet been submitted in the same or a similar form as a final examination paper.

☒ Ich habe KI-basierte Anwendungen und/oder Werkzeuge genutzt und diese im Anhang "Nutzung KI basierte Anwendungen" dokumentiert.

I have used AI-based applications and/or tools and documented them in the appendix "Use of AI-based applications".

B) Erklärung zur Veröffentlichung von Bachelor- oder Masterarbeiten
Declaration regarding the publication of bachelor's or master's thesis

Die Abschlussarbeit wird zwei Jahre nach Studienabschluss dem Archiv der Universität Bremen zur dauerhaften Archivierung angeboten. Archiviert werden:

Two years after graduation, the thesis is offered to the archive of the University of Bremen for permanent archiving. The following are archived:

1) Masterarbeiten mit lokalem oder regionalem Bezug sowie pro Studienfach und Studienjahr 10 % aller Masterarbeiten
Master's theses with a local or regional focus, as well as per subject and academic year 10% of all Master's thesis

2) Bachelorarbeiten des jeweils ersten und letzten Bachelorabschlusses pro Studienfach und Jahr.
Bachelor's thesis for the first and last bachelor's degrees per subject and year.

☐ Ich bin damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.
I agree that my thesis may be viewed by third parties in the university archive for academic purposes.

☐ Ich bin damit einverstanden, dass meine Abschlussarbeit nach 30 Jahren (gem. §7 Abs. 2 BremArchivG) im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.
I agree that my thesis may be viewed by third parties for academic purposes in the university archive after 30 years (in accordance with §7 para. 2 BremArchivG).

☒ Ich bin **nicht** damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.
I do not consent to my thesis being made available in the university archive for third parties to view for academic purposes.

C) Einverständniserklärung zur elektronischen Überprüfung der Arbeit auf Plagiate
Declaration of consent for electronic checking of the work for plagiarism

Eingereichte Arbeiten können nach § 18 des Allgemeinen Teil der Bachelor- bzw. der Masterprüfungsordnungen der Universität Bremen mit qualifizierter Software auf Plagiatsvorwürfe untersucht werden.
Zum Zweck der Überprüfung auf Plagiate erfolgt das Hochladen auf den Server der von der Universität Bremen aktuell genutzten Plagiatssoftware.

Submitted papers can be checked for plagiarism using qualified software in accordance with § 18 of the General Section of the Bachelor's or Master's Degree Examination Regulations of the University of Bremen. For the purpose of checking for plagiarism, the upload to the server is done using the plagiarism software currently used by the University of Bremen.

☐ Ich bin damit einverstanden, dass die von mir vorgelegte und verfasste Arbeit zum oben genannten Zweck dauerhaft auf dem externen Server der aktuell von der Universität Bremen genutzten Plagiatssoftware, in einer institutionseigenen Bibliothek (Zugriff nur durch die Universität Bremen), gespeichert wird.

I agree that the work I have submitted and written will be stored permanently on the external server of the plagiarism software currently used by the University of Bremen, in a library belonging to the institution (accessed only by the University of Bremen), for the above-mentioned purpose.

☒ Ich bin **nicht** damit einverstanden, dass die von mir vorgelegte und verfasste Arbeit zum o.g. Zweck dauerhaft auf dem externen Server der aktuell von der Universität Bremen genutzten Plagiatssoftware, in einer institutionseigenen Bibliothek (Zugriff nur durch die Universität Bremen), gespeichert wird.

I do not consent to the work I submitted and wrote being permanently stored on the external server of the plagiarism software currently used by the University of Bremen, in a library belonging to the institution (accessed only by the University of Bremen), for the above-mentioned purpose.

Das Einverständnis der dauerhaften Speicherung des Textes ist freiwillig. Die Einwilligung kann jederzeit durch Erklärung gegenüber der Universität Bremen, mit Wirkung für die Zukunft, widerrufen werden. Weitere Informationen zur Überprüfung von schriftlichen Arbeiten durch die Plagiatsoftware sind im Nutzungs- und Datenschutzkonzept enthalten. Diese finden Sie auf der Internetseite der Universität Bremen.

Consent to the permanent storage of the text is voluntary. Consent can be withdrawn at any time by making a declaration to this effect to the University of Bremen, with effect for the future. Further information on the checking of written work using plagiarism software can be found in the data protection and usage concept. This can be found on the University of Bremen website.

Mit meiner Unterschrift versichere ich, dass ich die obenstehenden Erklärungen gelesen und verstanden habe und bestätige die Richtigkeit der gemachten Angaben.

With my signature, I confirm that I have read and understood the above explanations and confirm the accuracy of the information provided.

14.08.2025

Datum / Date Unterschrit/ Signature

Abstract

This thesis investigates how a modern, microservice-based software architecture can support non-deterministic metal 3D-printing simulations across a distributed computation cluster. Using event sourcing, containerization, and Domain-Driven Design (DDD), the proposed architecture separates system responsibilities into clear layers and bounded contexts. It addresses challenges caused by unpredictable simulation outcomes by combining traditional state-oriented data storage with an event-driven model for real-time orchestration and monitoring. A fully functioning prototype was built that can start and supervise simulation jobs, as well as report live status updates and process logs. Although the architectural concept proved advanced and demanding, unlikely to suit teams with little background in software architecture, it met technical as well as non-functional requirements defined in the first phase of the project. The evaluation shows that the prototype scales horizontally, remains fault-tolerant under node failure, and is maintainable thanks to cleanly adhering to established design patterns and automated CI/CD pipelines, testing, verifying code, and automating deployments iteratively. In conclusion, this master's thesis demonstrates clean architectural design and best practices for developing a distributed computing platform suitable for high-performance workloads. The resulting architecture proved to be versatile, and this thesis can also serve as a practical guide for future projects that require process distribution and monitoring, even beyond simulation workloads.

Contents

1 Introduction

In the field of computer science, software architecture is increasingly moving away from monolithic structures and towards scalable microservices-based architectures. Today almost every application must serve a global user base, integrate with numerous external services, and evolve continuously without noticeable downtime. Traditional monolithic designs struggle under that load, because even a small defect in one module can bring the entire application offline and every update requires rebuilding and redeploying the whole binary. For these reasons the industry has moved steadily toward microservices-based architectures that break a large code base into smaller, independently deployable services. Each service focuses on one bounded context, communicates through lightweight protocols, and can be scaled horizontally on commodity hardware. This shift is largely driven by the growing demand for global connectivity, 24/7 availability, and the ability to adapt quickly to the fast-paced changes in modern software development through regular updates. Major cloud providers such as Amazon, Google, and Microsoft have proven that this style of architecture supports millions of concurrent users while also supporting rapid iterative releases and a strong fault isolation. These industry leaders heavily use microservices to horizontally scale their applications and support multi-user online platforms in an efficient and performant manner. The numerous advantages of the microservices approach include a reduction in dependencies, enhanced fault tolerance, and improved scalability. Although the commercial sector is already deep into the adoption phase with the majority of offered online services already migrated to microservices-based architectures, research and engineering applications are still far behind in adoption. Research Institutes are geographically distributed, processes are running around the clock, and experimental data must be shared safely among collaborators who may not have the ability to meet physically.

Metal additive manufacturing, or metal 3D printing, illustrates that demand particularly well. Before a single layer of powder is fused by a laser or electron beam, engineers need to predict thermal gradients, residual stresses, shrinkage, and geometric distortion. These simulations improve support-structure design and help with various process parameters, reducing material waste and machine time. Legacy simulation software typically ships as a monolithic Windows installer or a simple archive with all necessary binaries inside. Every researcher must configure identical libraries, license files, and driver versions, and can run only as many simulations as their own workstation can handle. The algorithms involved are computationally intensive, highly nonlinear, and sometimes even non-deterministic, making it difficult to transform existing applications from their current monolithic form to a microservices-based architecture. Moreover, metal 3D printing simulation algorithms often exhibit non-deterministic behaviour: the same input parameters may produce slightly different thermal fields because of floating-point rounding, integrated pseudo-

randomness, or stochastic powder-bed models.
This thesis investigates whether modern software architecture can overcome those limitations through the design and evaluation of a suitable state-of-the-art software architecture for distributing and monitoring metal 3D printing simulations in a computing cluster via a multi-user web interface. The proposed architecture incorporates state-of-the-art methods and design patterns, including the event sourcing pattern combined with an external event streaming service and containerization. The central hypothesis is that a microservices-based platform, designed around the event sourcing design pattern in combination with event streaming, and containerisation, can enable the distribution of metal 3D printing simulations across a compute cluster while providing real-time feedback, high availability, and elastic scalability. Event sourcing captures every state transition as an immutable message, potentially giving a complete audit trail for regulatory compliance and scientific reproducibility. An external event-streaming platform decouples producers and consumers, letting the web dashboard, process orchestrators, computing nodes and even the database services subscribe independently. This architectural idea promises flexible live updates without downtime, supports almost limitless architecture-integrated scalability, as well as flexibility in deployment and resource management. Containerisation isolates runtime dependencies, so a tool that works on a laptop will also work on the cluster and, if needed, in the public cloud. By incorporating a microservices-based approach, the architecture separates individual complexities into manageable components that can be independently developed, tested, and scaled. The core features of this architecture include the ability to dynamically expand computational resources, high-availability support, and seamless integration of real-time updates for users in a 0-downtime manner.
Designing such a system raises several research challenges:

- **First**, a significant challenge addressed in this work is the non-deterministic nature of some simulation algorithms, which complicates orchestration, state management, and fault tolerance.
- **Second**, what data model structure satisfies the needs of front-end components, orchestration logic, and long-term archives without leaking implementation details across layers.
- **Third**, how do we ensure continuous availability when the message broker, identity provider, or database fails.
- **Fourth**, how well does horizontal scaling behave when cluster nodes differ greatly in CPU count, clock speed, or memory capacity.

To answer those questions the thesis makes four concrete contributions. It specifies a detailed requirement list by reviewing the legacy workflow currently used at the Airbus Endowed Chair for Integrative Simulation and Engineering of Materials and Processes.

It proposes a layered architecture that blends Domain-Driven Design, containerisation, event sourcing, and an external event stream while explicitly addressing non-deterministic workloads and designing the platform architecture around the mentioned constraints. It implements a complete prototype in C# and .NET 8, including a web front end, computing-cluster-node workers, CI/CD pipelines, and role-based access control via Keycloak. Finally, it evaluates that prototype on four heterogeneous machines, measuring job-launch latency, execution throughput, analyzing data flows during different events like node faults, and evaluating scalability of the computing cluster. This evaluation will highlight the strengths and weaknesses of the proposed system in addressing the complexities of distributed metal 3D printing simulations.

The prototype achieves zero-downtime live updates by using dynamic container redeployments through Docker, shows near-linear throughput scaling when additional high-performance nodes join the cluster, and tolerates sudden loss of a worker or any individual microservices without any data corruption or process interruption. At the same time the project exposes the practical limits of the chosen design patterns: with the non-deterministic nature of the metal 3D printing simulation processes event sourcing alone cannot provide deterministic replay. Containerisation fails for legacy software requiring a Windows desktop environment, and a single RabbitMQ broker creates a star-pattern dependency that must be addressed using a high-availability deployment in production. For both of the latter two issues the thesis provides a suitable solution.

By extending cloud-native practices into the domain of metal 3D printing simulation, this thesis aims to demonstrate that scientific applications can achieve the same agility, resilience, and scalability already common in commercial web services. The findings promise to provide valuable insights into the viability of a microservice-based architecture using event sourcing and containerization for solving real-world computational problems related to automatic process distribution and monitoring. This work could help other researchers modernise their workflows, whether they study additive manufacturing, or any other computing-intensive, data-driven research topic. In general, this thesis aims to contribute to research on microservices-based software architectures, as well as highlight disadvantages of some of the most popular software design patterns in our current time.

2 Background and Technological Trends

2.1 Departure from Monolithic Architectures

The evolution of software development has seen a dramatic shift in architectural design, with monolithic architectures increasingly being replaced by modular and scalable alternatives such as microservices. Monolithic architectures, which historically dominated the software development area, are characterized by a single, tightly-coupled codebase where all components of an application are developed, deployed, and maintained as a single unit [39]. While this approach was sufficient in earlier eras of software development, the rapid growth in the complexity and scale of modern applications has rendered it increasingly inadequate.

One of the most significant drawbacks of monolithic architectures is their lack of scalability. As the number of users and the size of the applications grow, monolithic systems struggle to efficiently handle the increased load. Scaling a monolithic application typically requires scaling the entire system, even if only a small subset of functionality is under heavy demand. This results in inefficient use of resources and increased operational costs [31].

Another major limitation is the tight coupling inherent in monolithic architectures. All components in a monolithic application are interdependent, making it difficult to modify or update one part of the system without risking unintended consequences elsewhere. This lack of modularity significantly hampers the ability to implement new features, fix bugs, or adopt new technologies [39]. In industries where time-to-market and continuous delivery are critical, this rigidity becomes a serious bottleneck.

The deployment and maintenance of monolithic systems also present substantial challenges. Because all components are bundled together, even small changes require redeploying the entire application. This not only increases the risk of downtime but also complicates debugging and error resolution. Furthermore, as applications grow in size, their codebases become increasingly complex and difficult to manage, resulting in what is often referred to as "spaghetti code." This further exacerbates issues related to development velocity and system reliability [12].

The software industry has recognized these limitations and is progressively moving towards more flexible, scalable, and maintainable architectures [16]. One of the key drivers of this transition is the growing demand for applications that are globally accessible, available 24/7, and capable of rapidly adapting to changing user requirements. To achieve these goals, companies have started to adopt distributed and modular approaches, such as microservices, which address many of the inherent weaknesses of monolithic architectures [16].

Microservices architectures decompose an application into smaller, independent services that can be developed, deployed, and scaled independently. This modularity allows organizations to isolate failures, making it easier to identify and resolve issues without

affecting the entire system. It also enables teams to work on different services concurrently, facilitating faster development cycles and greater organizational agility [11].
The adoption of cloud computing and containerization technologies has further accelerated this shift. Cloud platforms provide on-demand scalability and fault tolerance, while containerization tools like Docker [28] and orchestration systems like Kubernetes [8] simplify the deployment and management of microservices-based applications [47]. These advancements have made it easier than ever for organizations to transition away from monolithic architectures and adopt more modern methods.
Prominent technology companies, including Google, Microsoft, and Netflix, have led this transformation. Their success in building scalable, resilient, and user-centric systems using microservices has caused widespread adoption across the digital industry [16]. Today, even small and medium-sized companies are following suit, using microservices to meet the demands of modern software.
In summary, the departure from monolithic architectures is not just a trend but a necessity driven by the limitations of traditional systems in meeting the requirements of modern applications. The move towards microservices and other modular approaches reflects the need for more scalable, flexible, and maintainable software solutions provided in an increasingly connected and dynamic digital infrastructure.

2.2 The Rise of Microservices

The concept of microservices emerged as a response to the challenges and limitations of monolithic architectures, slowly becoming popular in the early 2010s [16]. While the underlying principles of modularity and separation of concerns are not new, the term "microservices" was popularized as companies sought to modernize their architectures to address scalability, maintainability, and deployment issues. Leading organizations such as Netflix, Amazon, and Twitter adopted microservices as part of their transition to more resilient and scalable systems. These early adopters demonstrated that breaking down monolithic applications into smaller, loosely coupled services could revolutionize software development and operation.
Microservices are characterized by their focus on small, independent services that communicate with each other through well-defined APIs [35]. Each service is designed to perform a specific business function, and it is often developed, deployed, and scaled independently of other services. This independence allows teams to work autonomously on different parts of the system, enabling faster development cycles and reducing interdependencies. Moreover, microservices architectures embrace the principles of domain-driven design, where each service closely matches a specific domain or subdomain within the overall system.
One of the most significant advantages of microservices is scalability [39]. Unlike mono-

lithic systems, which require scaling the entire application, microservices allow for selective scaling of individual components based on demand. For example, a service responsible for handling user authentication can be scaled independently from a service that processes data analytics. This targeted scalability not only optimizes resource usage but also reduces operational costs [31].

Fault isolation is another key advantage. Since microservices are loosely coupled, a failure in one service is less likely to impact the entire system [31]. This isolation enhances the overall reliability and resilience of the application [3]. Additionally, microservices support polyglot programming, allowing different services to be implemented using the most suitable programming language or framework for their specific requirements [37]. This flexibility allows developers to always utilize the best tools for each task [1].

Microservices also improve the deployment process. Each service can be deployed independently, enabling continuous delivery and reducing the risk of deployment-related failures [35]. This agility is particularly valuable in fast-paced applications where time-to-market is a critical factor. Moreover, by using containers and orchestration tools such as Docker [28] and Kubernetes [8], microservices can be deployed consistently across various environments, from local development machines to large-scale cloud infrastructure [47].

A common question is how microservices differ from virtual machines. While both technologies support modularity and resource isolation, their approaches differ fundamentally. Virtual machines provide hardware-level isolation by virtualizing an entire operating system instance for each application, which often results in significant resource overhead [20]. In contrast, microservices rely on containerization, where lightweight containers share the host operating system while isolating individual services [37]. This makes microservices far more efficient in terms of resource utilization and startup times [17] [20].

Microservices are widely regarded as the current state of the art in software architecture [16]. The industry has increasingly adapted this development, with organizations restructuring their development processes to better match with microservice principles. Furthermore, advancements in cloud computing, serverless technologies, and event-driven architectures have further accelerated the adoption of microservices [25].

In conclusion, the rise of microservices represents a shift in how modern software is designed, developed, and maintained. By addressing the limitations of monolithic architectures and improving flexibility, scalability, and reliability, microservices have become the foundation for building software systems in an increasingly dynamic and interconnected world. As software architecture continues to evolve, it increasingly adapts to the principles and practices of microservices.

2.3 Developments in Software Architecture

The field of software architecture has undergone big transformations over the years, cuased by technological advancements and the growing complexity of modern software systems [10]. Among the most influential of these developments is the adoption of microservices, which has fundamentally altered how applications are designed, developed, and deployed [16]. This shift has changed the evolution of software architecture into increasingly modular, scalable, and maintainable structures.

Historically, software architectures were dominated by monolithic designs [10], often adhering to the classic Model-View-Controller (MVC) pattern [13]. While MVC provided a structured approach to organizing code, it struggled to scale effectively with the rising demands of large-scale, distributed systems. The rise of microservices offered a much-needed alternative, emphasizing modularity and independence among components. This modularity has not only improved scalability but also resulted in the adoption of new design patterns that are better suited to overcome the challenges of distributed systems.

One such design pattern is event sourcing, which has gained traction alongside microservices [25]. In event sourcing, the state of a system is derived from a sequence of immutable events rather than being directly stored in a database. This approach provides numerous benefits, including enhanced traceability, support for real-time updates, and the ability to rebuild system states from historical events [25]. When combined with external event streaming platforms like Apache Kafka [22], event sourcing enables robust communication between services and ensures consistency across distributed components [25].

Domain-Driven Design (DDD) is another critical method in shaping modern software architecture [18]. Using DDD, software is being modeled around the core business domains and their associated logic [50]. DDD promotes clearer boundaries between services, which is especially valuable in microservices architectures [44]. Bounded contexts, a key concept in DDD [50], help ensure that services remain loosely coupled and focused on specific responsibilities, further enhancing modularity and maintainability [44].

The move away from monolithic architectures has also led to significant changes in how developers approach user interfaces and client-side development. The concept of microfrontends extends the principles of microservices to the frontend, allowing different parts of the user interface to be developed and deployed independently. This approach fits well with modular backend architectures, enabling developers to iterate on specific features without impacting the entire application [27].

Software systems have also become more modular by incorporating specialized services for tasks such as authentication and storage. For example, authentication is increasingly being handled by external OpenID services like Keycloak [48], which offers scalable and secure identity management solutions. Similarly, object storage solutions like Amazon S3 provide reliable and scalable storage for unstructured data, allowing applications to offload storage

responsibilities while increasing high availability functionality and performance [36]. This trend of outsourcing specialized functionalities has further reduced the complexity of individual systems, allowing developers to focus on core business logic.
The increasing modularity of software is further supported by advances in infrastructure and deployment technologies. Containerization and orchestration platforms like Docker [28] and Kubernetes [8] have simplified the deployment and scaling of modular components. These tools ensure consistent environments across development, testing, and production, reducing deployment risks and enabling rapid iterations. Serverless computing has also gained popularity, allowing developers to run individual functions in response to events without managing servers [33]. This further aligns with the modular principles of modern architecture.
In addition to these technological advancements, there has been a shift in software development processes to accommodate modular architectures. Continuous Integration and Continuous Deployment (CI/CD) pipelines [46] have become standard practice, enabling developers to automate testing and deployment across distributed systems [19]. Infrastructure as Code (IaC) tools like Terraform and AWS CloudFormation further allow developers to define and manage modular infrastructure components programmatically, ensuring consistency and repeatability [49].
In summary, technological advancements like microservices, event sourcing, domain-driven design, and microfrontends have significantly influenced the evolution of software architecture. These innovations have caused a change, from rigid, monolithic systems towards modular, scalable, and maintainable designs. By increasing modularity and dividing monolithic software into specialized services, modern architectures are better equipped to handle the demands of scalability, fault tolerance, and rapid development cycles, ensuring their relevance in an increasingly complex and interconnected software ecosystem.

3 Motivation and Requirements

3.1 Initial Inspiration

Last semester I took a course on metal 3D printing. I was introduced to many algorithms that prepare a part before it is built. These include simulations for heat distribution, deformation, shrinkage, and other physical effects. Such simulations reveal potential problems early in the workflow and let engineers adjust a design or its printing parameters while there is still time to improve the outcome.
During the course I also learned about the work carried out at the Bremen Center for Computational Materials Science by the Airbus Endowed Chair for Integrative Simulation and Engineering of Materials and Processes (ISEMP). The ISEMP, founded in 2009, is an interdisciplinary unit at the University of Bremen that links natural science and engineering departments and focuses on Computational Materials Science, from mechanical simulations to full process modeling. Much of its effort goes into the structural design of new materials, and many projects are pursued together with other institutes and industrial partners.
The ISEMP improves industrial practices through fundamental research, utilizing physics, computer science, materials science, and production engineering. Thanks to close cooperation with companies such as Airbus and Siemens, the chair tackles real manufacturing challenges in additive production and laser processing. Students and researchers routinely work on industry based projects and refine existing methods with computer aided tools.
At the ISEMP one of the main simulation programs is AmFem, which supports detailed analysis of metal additive manufacturing. AmFem can predict overheating, distortion, and the need for support structures, making it possible to lower production costs and improve print quality.
I used AmFem extensively during the course and soon saw that, despite its strengths, the software struggles with scaling, integration, and the coordination of large simulations. Those weaknesses sparked the idea for this thesis, to find out how modern software techniques, especially event sourcing together with containerization, could improve AmFem and programs like it. Running resource hungry simulations on a cluster that many people can share is becoming increasingly important in research and industry, so a fresh architectural approach seemed worth exploring.
My classroom experience, time spent in industry, and day to day work with AmFem all fed into the central question of this study. The main focus is how a microservices design that records events can handle non deterministic metal printing simulations, and how containerization can help deploy those simulations in a scalable, efficient, and fault tolerant way on a computing cluster. These concerns shaped the entire project and guided the design and evaluation of a new architecture for the ISEMP.

3.2 Properties of Metal 3D Printing Simulations

Metal 3D printing simulations often exhibit a non-deterministic nature, meaning that two runs of the same simulation can, at times, produce different outcomes [38]. This non-determinism arises from various sources, such as the use of pseudo-random numbers in certain algorithms, floating-point rounding inconsistencies in parallel computations, and the influence of multi-threaded processes where subtle differences in execution order can alter the results. Even minor changes in initial conditions, such as material properties, laser scanning parameters, or environmental settings, can amplify over time and lead to divergent simulation paths [38].

Another key property of these simulations is the unpredictability of their runtime. It is often difficult to estimate how long a given simulation will take before it is actually started. Some reasons include the complexity of the model, which may involve several coupled physical phenomena like heat transfer, phase transitions, and mechanical stress, all of which can vary significantly from one print geometry to another. Moreover, different hardware capabilities, variations in parallelization efficiency, and potential load imbalances across computational resources can further complicate runtime predictions. These uncertainties underscore the need for flexible, robust system architectures that can handle fluctuating workloads and adapt to changing computational demands [38].

Additionally, the diversity of objectives and constraints in metal 3D printing simulations adds another layer of complexity. For instance, some simulations might emphasize thermal effects at high fidelity, while others prioritize structural integrity or fluid dynamics, leading to varying resource requirements [38]. Such factors contribute to the inherently dynamic nature of metal 3D printing simulations, making scalability, resilience, and real-time adaptability crucial considerations when designing software architectures to handle them.

3.3 Issues with the Current Monolithic Architecture

The existing software architecture of AmFem can be described as a monolithic system. All core functionalities, such as simulation execution, licensing, user access control, and data management, are bundled into one large application that must be installed and maintained as a single unit. While this approach simplifies initial development to some extent, it poses several significant challenges when it comes to distribution, scalability, security, and user support.

A first major drawback is that the entire program needs to be copied onto the user's computer in order to run a simulation. This imposes logistical hurdles, as each user must manually install and configure the software, which can be both time-consuming and prone to error. Additionally, the software only runs on Windows operating systems, severely limiting accessibility for users working in Linux or macOS environments. From the developer's perspective, it also becomes more complex to offer a uniform experience

or to leverage container-based deployment strategies.

Another issue concerns licensing. Currently, each installation of the software must be explicitly licensed, but once the license is distributed, the developer has little insight into who is using the program, when they are using it, or precisely what operations they are performing. Restricting access to specific features or revoking a license altogether can be cumbersome since the monolith itself lacks granular control mechanisms. Moreover, once the software is handed out, there is no straightforward method to prevent unauthorized usage or to disable features after the fact, unless the developer issues a new release.

The monolithic structure also complicates rolling out updates. Because every user has a standalone copy of the software, they must reinstall or patch their local installation whenever a new version is released. This process can cause downtime, introduce potential inconsistencies between users running different versions, and require significant coordination to ensure a smooth transition. In many cases, a single bug fix or minor feature addition may necessitate a full new release, further increasing overhead.

A particularly serious limitation is the requirement that the user supplies their own computing resources to run simulations. Given that these simulations can be resource-intensive and lengthy, the user's local machine may be occupied for extended periods, preventing other work. Remote status checks and monitoring are also not well supported in the current setup, making it difficult for a user to track or manage a running simulation from another device or location.

Finally, there is the challenge of ensuring that every user's environment is properly set up. System requirements, such as correct library versions, adequate hardware configurations, and specific operating system settings, can vary, leading to failures or inconsistent performance. These configuration headaches are common with monolithic applications that require exact replicas of their environment to operate correctly.

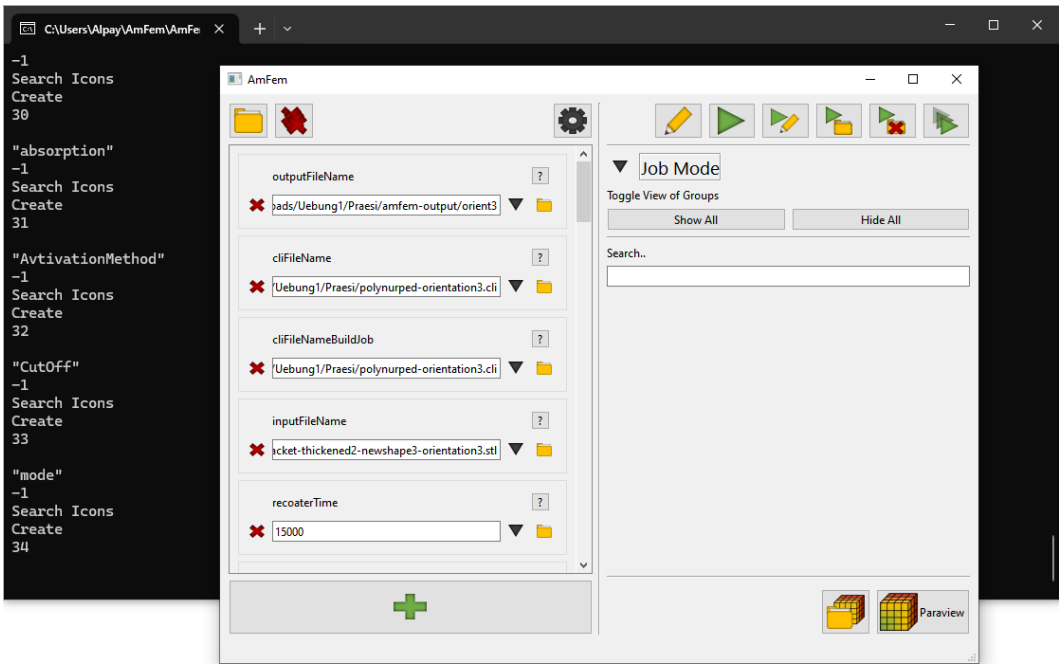

Figure 1: Graphical user interface of AmFem

Figure 1 shows the current GUI of AmFem. It appears to be very crude, functionality is limited and the interface can be described as clunky in general. File and folder selections do not work properly, the file paths of some input fields need to be filled out manually and the GUI regularly freezes. These issues are not directly inherent to the architecture itself, but a part that needs improvement nonetheless. The GUI of AmFem aligns with the general issues the applications has.
Taken together, these points show how the current monolithic architecture hinders flexibility, maintainability, and user satisfaction. It restricts the ability to scale simulations on demand, complicates access management, and forces both the developer and the user to cope with a less-than-ideal deployment model. A new architecture that addresses these issues by enabling remote execution, dynamic resource allocation, granular licensing, and easier updates, would significantly improve the overall user experience and practical efficiency when working with the software, both for the developer, as well as potential users.

3.4 Requirements of an Alternative Architectural Design

When observing the limitations posed by the current monolithic setup, it becomes clear: a new approach must address several essential requirements to ensure robustness, scalability, and usability. These requirements are derived both from the shortcomings identified in the section 2.1 from above and from best practices in modern software development.

- **Multi-User Access:** A fundamental requirement is to allow multiple users to access and utilize the system simultaneously. This means that user sessions and data must

be properly isolated, preventing accidental overwrites or unauthorized information sharing. In addition, user authentication and role-based authorization should be implemented to ensure that the different users, such as researchers, customers, or administrators, have the right level of access to both system features and simulation data.

- **Scalability on Multiple Levels:** To accommodate a growing number of simulations and users, the system should be designed with scalability in mind. This includes horizontal scaling of simulation services to handle computationally intensive tasks, as well as vertical scaling for databases or event streaming platforms. Because metal 3D printing simulations can vary in complexity and resource requirements, an elastic approach is crucial, allowing the system to automatically allocate more resources when needed and release them once tasks are finished.

- **High Availability:** High availability means that each component should remain operational despite failures or unexpected spikes in usage. Redundancy and failover mechanisms, such as running multiple instances of critical services in different zones or nodes, can help fulfill this requirement. Automated health checks, self-healing orchestration (e.g., via container orchestration platforms), and robust load balancing are additional strategies to maintain continuous uptime.

- **Provisioning of Simulation Processes:** One of the core functions is the ability to provision new simulation processes on demand. Users should be able to request a simulation run with specific configurations, while the system handles the necessary setup, including container orchestration, resource allocation, and environment variables. This provisioning should be as seamless as possible, abstracting away the complexities of hardware or operating system dependencies.

- **Input Parameter Management:** The system should offer a clear and flexible mechanism for handling input parameters to these simulations. Some simulations might require dozens of parameters, each of which can significantly influence the outcome. Providing a user-friendly interface or API for parameter specification, validation, and versioning ensures both ease of use and consistency in experiment setups.

- **Monitoring of Process States:** Since metal 3D printing simulations can take unpredictable amounts of time, real-time monitoring becomes essential. Users should be able to track the progress of their simulations, receive status updates (e.g., queued, running, completed, or failed), and retrieve partial results if available. Furthermore, a logging and event-handling mechanism should capture important runtime information for troubleshooting, performance analysis, and auditing.

- **Consideration of Non-Deterministic Outcomes and Runtimes:** A critical requirement is handling the non-deterministic nature of these simulations, where even the same model and parameters can yield varying results. The software must recognize that repeatability is not guaranteed, and be prepared to manage scenarios such as unexpectedly long simulation times or branching results. Features like retry logic, checkpointing, and dynamic resource allocation can help mitigate these uncertainties.

- **State-of-the-Art Platforms, Technologies, Methods, Tools, and Design Patterns:** To future-proof the solution and align it with current best practices, the architecture should leverage modern technologies. This could include containerization with Docker, event streaming with RabbitMQ or Apache Kafka, microservices-based design, and modular software architecture. Adopting well-established design patterns, such as event sourcing, CQRS (Command Query Responsibility Segregation), and circuit breakers, can further enhance reliability and maintainability.

Additionally to the core points outlined above, there are other relevant needs, like collecting metrics, logs, and traces in a centralized manner in order to diagnose performance, identify bottlenecks and detect errors, storing configurations in regard to different tools, their versions and input parameters while maintaining consistency across different contexts, simplifying the rollout of new features, patches, or bug fixes without extensive downtime or complex code manipulations and implementing easily configurable account access, and usage tracking to better manage how the software should be deployed.
The application should be designed to be used by three different kinds of actors.

- **Users** are the primary consumers of the simulation platform. They upload or select input files, configure simulation parameters, start new jobs, monitor job progress, and download result files or logs once a run is complete. A clean web interface or a well documented API is essential so that users can focus on research tasks instead of infrastructure details. Role based access control must prevent users from viewing or altering jobs that belong to other groups, while still allowing collaboration within their own teams.

- **Maintainers** manage the technical catalog behind the scenes. They register new tools, define available tool versions, set default input parameters, and decide which user groups may view each others jobs. Because their changes can affect every subsequent simulation, the UI must include guardrails such as form validation and confirmation prompts.

- **Administrators** look after the health of the platform itself. They create and remove user accounts, assign roles, and reset passwords. Maintainers provision and monitor

resources for the network storage and cluster nodes, manage the database, and need to make sure the platform is running smoothly in general. They also track usage metrics to spot abnormal load patterns or hardware failures and coordinate updates to the underlying operating systems and container runtimes.

Meanwhile actors follow a strict hierarchical structure. When a person has been assigned specific role permissions, they could potentially also access platform functions assigned to other user roles.

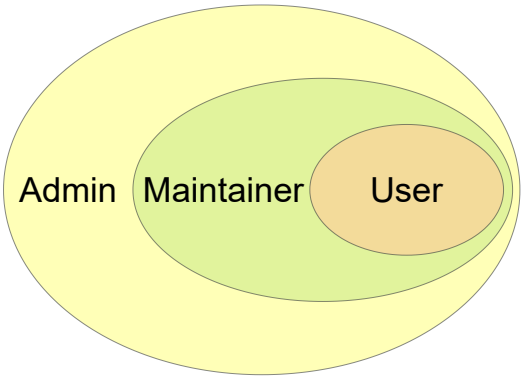

Figure 2: Hierarchical actor permissions vizualized through a Venn diagram

Figure 2 shows a Venn diagram highlighting this hierarchical structure. In this case, an maintainer may also act as a user, while an adminstrator has access to both functions assigned to users, as well as maintainers.
Clear separation of duties among Users, Maintainers, and Administrators helps prevent accidental misconfiguration and enforces least privilege. It also shapes the overall architecture, since different service endpoints, dashboards, and API scopes are tailored to each role's specific tasks.
In summary, these are the requirements that should be considered when designing and implementing an application architecture that aims to solve the issues of the current monolithic solution. By using more modern software architectural principles and technologies, the new system should be able to provide a more flexible, scalable, and reliable environment for the provisioning and monitoring of metal 3D printing simulations.

3.5 Realization of a Proof-Of-Concept Prototype

In order to evaluate the theoretical architectural concept that will be created over the course of the thesis, a prototype needs to be developed. While designing the architecture on paper is useful to clarify goals, it does not fully reveal practical challenges until the concepts are realized in practice. Therefore, building a proof-of-concept prototype allows for early testing and helps uncover potential pitfalls in areas such as deployment, resource allocation, and data flow.

A primary benefit of creating a prototype is identifying issues that might not emerge during the conceptual phase. By writing actual code and integrating different components—such as container orchestration, event streaming, and monitoring tools—developers can observe how these elements interact under real-world conditions. These hands-on insights can then be used to adjust design decisions, refine interfaces, or evaluate alternative approaches.
Another key aspect is testing the prototype against multiple criteria, including functionality, failure-resistance, scalability, and performance.

- **Functionality:** Ensuring that core features, such as provisioning new simulations or managing user access, work as intended.
- **Failure-Resistance:** Examining how the system behaves under network failures, node crashes, or misconfigurations, and whether it can automatically recover or at least provide informative error handling.
- **Scalability:** Validating how well the system adapts to growing numbers of users and simulation demands, possibly through load testing or stress testing.
- **Performance:** Measuring the prototype's responsiveness, resource usage, and throughput to confirm it can handle computationally heavy tasks efficiently.

Evaluating which components perform well, and which do not, offers valuable lessons that inform subsequent iterations of the architecture. Sometimes, certain design patterns or technologies may appear promising in theory but underperform in practice. The prototype provides concrete data to guide improvements, helping prioritize where to invest development efforts.
Additionally, developing a prototype fosters communication among stakeholders. Researchers, system architects, and industry partners can interact with a tangible implementation rather than an abstract diagram. This accelerates feedback loops, clarifies requirements, and establishes confidence in the proposed solution.
Overall, creating a proof-of-concept prototype is an essential phase of this project. It validates the architectural design, highlights unforeseen challenges, and lays the groundwork for a robust, production-ready system that can be refined even more in the future.

4 Technologies and Related Work

This section provides a comprehensive survey of the technologies, methods, and tools used this thesis. It begins by reviewing similar projects (section 4.1) and lays out the criteria used to select appropriate technologies (section 4.2). Next, it justifies the choice of programming language and framework for the prototype (section 4.3). Section 4.4 dives into the most important architectural patterns and practices: event sourcing, containerization, CI/CD, domain-driven design, MVVM, external event streaming, external authentication (OpenID/OAuth2), and automated vulnerability scanning, explaining how each contributes to scalability, reliability, and maintainability. Finally, section 4.5 enumerates the specific tools and development environment components: GitLab CI, Docker Desktop, Visual Studio 2022, RabbitMQ, Keycloak, SonarQube, and a container registry, that will be used to design and build the microservices-based prototype.

4.1 Similar Projects

In the broader landscape of process orchestration tools, the largest and most prominent project with a feature set comparable to what is envisioned in this thesis is Camunda.
Camunda is a process orchestration solution designed to coordinate diverse tasks, systems, and endpoints within a single, end-to-end process [21]. Its primary focus is on providing business process management (BPM) capabilities through tools like the Camunda Modeler, which is a kind of graphical process modeller. The system is marketed as a universal process orchestrator capable of integrating any kind of tasks, including manual tasks, microservices, AI-driven endpoints, legacy systems, and more. Camunda's principal benefits include out-of-the-box connectors, visual modeling, and built-in monitoring and analytics to offer a holistic view of running processes [21].
Despite its strengths, Camunda poses several limitations that render it less suited for the needs of the architecture outlined in this thesis. For instance, while Camunda claims to be cloud-native, it still operates under a largely centralized model for orchestration. Tasks are communicated through HTTP or similar interfaces, and high availability is achieved by running multiple identical nodes with replicated state [21]. However, these nodes do not facilitate independent scalability at a granular level, scaling a single component (e.g., the workflow engine or the monitoring service) without taking the entire Camunda cluster offline is generally infeasible. Furthermore, essential elements like the task management component and the decision engine come tightly coupled with the platform, limiting the ability to substitute or integrate a custom solution if desired [6].
From an update and maintenance perspective, Camunda's approach to high availability can still introduce downtime. When deploying a new version of a process or plugin, nodes often need to be brought down and redeployed completely, causing active process instances on that node to fail rather than automatically retrying. This approach is inconvenient for

continuous, long-running processes such as metal 3D printing simulations, which may not finish for days [38]. It also prevents seamless rolling updates where new versions can be gradually introduced without interrupting current tasks. In contrast, the architecture proposed in this thesis aims to support running multiple algorithm versions side by side, even on the same node, without stopping existing processes.

Additionally, Camunda's workflow and decision engine is focused on deterministic business workflows, which may be insufficiently flexible for highly non-deterministic, simulation-heavy workloads [21]. Such workloads benefit from specialized resource management, event-sourced state tracking, and dynamic scaling based on real-time computational demands, none of which are straightforward to implement in Camunda [6]. Because the platform has been designed towards typical enterprise process automation, it lacks native support for some features enabling dynamic computation cluster expansion at the level required for this project, features that are generally needed in these High-Performance Computing applications. For complex, domain-specific scenarios such as distributed metal 3D printing simulations, these constraints can introduce both technical friction and higher maintenance overhead [6].

Overall, Camunda does offer robust features for orchestrating and monitoring business processes. However, its monolithic approach to platform components, limited dynamic scalability, and downtime-prone update model conflict with the goals set in this thesis, a fully containerized, event-sourced system for non-deterministic simulations. The architecture proposed in this thesis aims to address these gaps by emphasizing granular scalability, uninterrupted updates, and advanced resource orchestration better suited for computationally intensive tasks.

Apart from Camunda, there are effectively no other pre-made solutions that offer similar functionalities, particularly in the realm of handling non-deterministic, computationally intensive workloads with advanced event-sourcing capabilities and dynamic scalability. In general, companies seeking a specific BPM tool use custom-made solutions. Being a relatively widespread method, the modelling of these solutions is a well-researched topic [43] [51].

However, when a company has specific requirements like these, highly custom, generally monolithic solutions are being implemented. At the same time, the method of event-streaming is also becoming increasingly more popular when working with modular event-based systems. One of the software design patterns adhering to this method is the event-sourcing pattern. Event-sourcing is becoming more popular because it records all changes in a system over time. Most organizations use event streaming services in a publish-subscribe model, which mainly sends updates one way. This works well for simpler cases, like sending notifications about data changes like changing user data when an employee enters or leaves the organization or when events like machine failures or orders are being produced.

However, when using event sourcing for process provisioning, things get more complicated. Events need to do more than just broadcast updates. They have to carry enough information to start, but also to monitor already running processes. This adds extra steps to the design. Coupled with non-determinism in the runtime and outcome of a process, this becomes even harder. Metal 3D printing simulations can produce different results, even with the same input data. This means the system must deal with branching events, unexpected runtime changes like random failures, and possibly very long running times. Most BPM solutions are not built for such unpredictable workloads [51]. They assume a more straightforward process flow and don't handle large-scale simulations very well. In contrast, the architecture proposed in this thesis suggests an architecture that pairs event sourcing with containerization and dynamic scaling. The proposed design aims to keep the benefits of event-sourcing, like integral audit logs and a seamless interface with an event-stream to connect multiple components, while also being able to handle non-deterministic simulation process distribution without downtime.
The main goal is to solve the unique challenges of distributing and monitoring non-deterministic simulation processes using an event-stream and an orchestration model. This approach aims to do things that typical business software cannot, especially when simulations are large, unpredictable, and compute-intensive are involved.

4.2 Technology Selection Criteria

When designing and implementing a modern software system, it is essential to choose technologies, platforms, tools, and design patterns that match current industry standards. Generally, companies have a strong incentive to select the "best" technology in terms of future maintainability, ease of use, and available feature sets. At the same time, choices can be influenced by existing dependencies and the need for backward compatibility. This subsection outlines the main criteria used in this thesis to evaluate and select the technologies involved.

- **Definition of "State-of-the-Art":** In this thesis, "state-of-the-art" refers to solutions that leverage the latest and most effective methodologies, technologies, or architectural approaches. Such solutions typically benefit from modern features, better support, and active communities, which can reduce both development time and long-term maintenance costs.

- **Future Maintainability:** Maintainability is key for any system expected to last a long time. Technologies that offer clear documentation, stable APIs, and active community support raise the probability of longevity. In practice, strong maintainability ensures easier bug fixes, faster feature development, and a longer lifespan for the software in general.

- **Ease of Use:** Ease of use covers both initial setup and general usage. Technologies which can be installed in a straightforward and intuitive way tend to also be more easily maintained later down the line. A shorter learning curve generally results from a technology or platform being easy to use.
- **Setup Time and Complexity:** Many organizations value how quickly a tool or platform can be installed and configured. A solution that automates or simplifies setup tasks can greatly streamline development workflows. This benefit is especially important when infrastructure or application requirements change frequently.
- **Feature Set and Extensibility:** Selecting a platform with a robust feature set minimizes the need to develop custom modules from scratch. At the same time, extensibility is crucial, as it allows developers to add specialized capabilities or integrate with external systems. This balance helps ensure the solution remains flexible and can adapt to changing project needs.
- **Industry Adoption:** Technologies widely adopted by major companies often indicate proven reliability and scalability. When a tool is already used in production environments at scale, it has likely undergone various optimizations and stability improvements. Moreover, a large user base typically means better community support and more frequent updates.
- **Backward Compatibility:** In some cases, companies stick to older technologies due to legacy systems or long-standing contractual obligations. While such choices might not align perfectly with "state-of-the-art" principles, they can be necessary to maintain continuity. In this thesis, any selection will also consider potential interoperability with older systems when applicable.
- **Security and Compliance:** Security is probably the most important aspect, especially when handling sensitive data or critical operations. Technologies with strong security features and regular updates are generally more reliable. This minimizes the risk of vulnerabilities that could compromise the system or violate regulations.
- **Community Support:** An active developer community contributes valuable tutorials, troubleshooting tips, and third-party extensions. Choosing a solution backed by a strong community can significantly accelerate the development process. Regular updates and clear roadmaps for future improvements assure long-term viability.
- **Scalability Requirements:** Given that this thesis deals with non-deterministic simulations that may require extensive computational resources, scalability is a top priority. Tools and platforms must support dynamic resource allocation, cluster orchestration, and load balancing without excessive overhead. If a solution cannot scale efficiently, it risks becoming a bottleneck as the system grows.

To sum up, the technologies chosen in this thesis must fulfill multiple criteria:they should be state-of-the-art, easy to set up and maintain, feature-rich, secure, and proven at scale. At the same time, they must leave room for future adaptability and potential backward compatibility needs. By evaluating each option against these requirements, the resulting system should combine modern design practices with practical considerations from real-world usage.

4.3 Selection of Language & Framework

For this thesis, C# and .NET 8 were chosen as the primary language and framework. This decision was done considering the criteria explained in section 4.2, particularly the emphasis on scalability, maintainability, and ease of integration with modern software patterns like microservices. These are some of the main reasons for selecting C# and .NET 8:

- **Cross-Platform Support:** While .NET was historically associated with Windows, modern .NET versions, starting from .NET 5+, which includes .NET 8, provide seamless cross-platform compatibility. This allows developers to build and run applications on both Linux and Windows servers, which is essential in containerized and cloud-native environments. Being able to deploy on multiple operating systems promotes flexibility in choosing hosting solutions.

- **Container Support:** C# and .NET 8 offer straightforward integration with container technologies such as Docker. Official Docker images and automated build processes are made readily available directly from Microsoft, making it easy to package and deploy microservices. This container-friendliness fits very well with the architectural goals of dynamic scaling and high availability set in this thesis.

- **Microservice-Friendliness:** .NET 8 includes built-in features that simplify the creation and orchestration of microservices, such as minimal APIs, gRPC support, and various configurable middlewares. These tools and libraries, also developed and maintained directly by Microsoft, make the integration of these asynchronous interfaces, cleaner separation of bounded contexts and scaling across multiple services a lot easier.

- **Dependency Injection and Modularity:** Dependency Injection (DI) is one of the main features in .NET 8, reducing code complexity by providing an easy way to manage lifetimes of components like resources, and services. In a microservices context, this modular approach simplifies testing, promotes reusability, and aids in maintaining a clean architecture aligned with domain-driven design principles.

- **Performance and Stability:** .NET 8 is designed for realtively high performance and includes technologies like Just-In-Time compilation (JIT) with seamless garbage

collection, and overall a very efficient runtime. Releases with Long-Term Support (LTS) like .NET 8 receive regular patches and security updates. This ensures stability and, with upgrades between .NET releases being generally quite easy and seamless, a clear upgrade path over time.

- **Active Development and Community Support:** Microsoft's recent investments in .NET have resulted in comprehensive documentation, extensive functionality, and a large community of developers. Tutorials, best practices, official design guidelines and many example projects are available online. Additionally, distribution of third-party packages through the package management system NuGet, widen the range of libraries available to developers even more.

- **Entity Framework for Data Access:** *Entity Framework* (EF) provides a high-level abstraction over database interactions, letting developers define models that map directly to database entities. EF supports LINQ (Language Integrated Query), making data queries a lot more intuitive and type-safe. This reduces errors and removed the need for developers to write manual database queries. Used in conjunction with domain-driven design (DDD) principles further helps maintain clean code boundaries between domain logic and data persistence.

- **Rich Feature Set and Extensibility:** As mentioned, .NET 8 offers a wide range of libraries providing solutions for all kinds of different problems. This means that it can accommodate additional features as the application grows or as new requirements emerge. When using robust design patterns and architectural principles, the integration of new features and modules without the need for significant refactoring or even having to use other languages and frameworks becomes not only possible, but also easy.

Overall, when evaluating C# and .NET 8 using the above criteria, it appears they do fit the criteria of being *state-of-the-art* better than most other frameworks, widely supported, and well-suited for building containerized microservices that integrate seamlessly into an event-sourced architecture. Their performance, ease of deployment, and the ease with which applications written using these tools can be containerized and deployed in multi-platform environments, make them an excellent fit for the distributed, non-deterministic simulation workflows explored in this thesis.

4.4 Methods & Design Patterns

When selecting methods and design patterns to implement, just like in the previous section 4.3, the selection criteria chosen and explained in section 4.2 will be used. As these methods and design patterns are vital for the construction of the applications software

architecture, the search for the right patterns and methods was done in conjunction with the construction of the theoretical software architecture outlined in section 5. However, in order to keep the logical structure of the thesis streamlined, I chose not to keep the structural content of the thesis strictly chronological, as I would have to interpolate the sections 4 and 5 with each other. In order to improve readability and logical consistency, this section will outline the final methods and design patterns selected and explain the reasoning behind choosing them.

4.4.1 Event Sourcing

This might be the most important design pattern of this thesis. The architecture revolves around this concept, without which the separation of the two client and computation cluster domains would not be possible the way it was done. In general, Event Sourcing is increasingly recognized as a *state-of-the-art* architectural pattern for applications that require granular tracking of data changes over time. Instead of storing only the current snapshot of the system, this approach logs every state modification as an event [34]. In principle, it creates abstracts the actual changes in each data modification, which is a departure from traditional state-based systems with relational databases at their core.

- **Future Maintainability & Ease of Use:** By retaining a complete history of events, developers can replay, debug, or migrate data more easily. This simplifies introducing new features or making schema changes, as the entire sequence of state transitions is available to reconstruct current or past states [34]. Over time, developers can also analyze historical events to gain insights into system behavior, user interactions, and potential areas for optimization.

- **Setup Time & Complexity:** Although Event Sourcing can initially be more complex to implement than a traditional CRUD approach, modern tooling and best practices mitigate this overhead. Given that many open-source frameworks and libraries support Event Sourcing out of the box, the setup is relatively straightforward once the fundamental concepts are understood.

- **Feature Set & Extensibility:** Event Sourcing inherently pairs well with domain-driven design (DDD) and microservice architectures, allowing for consistent and traceable data flows across services. It is easily expandable, which lets developers add new events, event consumers or projections without drastically altering the core system. This enables a modular approach to future feature development.

- **Industry Adoption & Community Support:** Event Sourcing is widely adopted in financial technologies, e-commerce, and large-scale web applications in general. There are extensive community resources available. Globally successful companies

like LinkedIn and Netflix have successfully used event-driven systems in the past, which is a strong indicator of its viability [14]. Because of its popularity, there are many example architectures and resources regarding Event Sourcing available online.

- **Scalability Requirements:** By decoupling the write model (where events are captured) from the read model (where events are consumed), developers can separate components from each other and scale them completely independently. In high-throughput scenarios, events can be processed through dedicated streaming platforms, further supporting large-scale, distributed environments [14]. This is the key reason for including Event Sourcing in the software architecture of this thesis and builds the foundation for the core idea.

4.4.2 Containerization

Through containerization, applications and their dependencies can be packaged into self-contained units. Tools such as Docker have become essential for modern software development, primarily due to their portability and consistency across environments [20]. Containerization allows developers to control the exact parameters, working environment and dependencies of applications. At the same time, using containerization in production environments enables failure resilience, scaling, monitoring and sandboxing of individual components and microservices from each other [20]. In modern software development, containerization is an essential tool [5] [16], due to its advantages and the requirements set in section 3.4, it must absolutely be included in the work done in this thesis.

- **Future Maintainability & Ease of Use:** Containers enable predictable deployments [7]. Developers can be confident that a software that works on their own machine directly works in their staging or production environments as well, minimizing issues where things break due to the software running on a different system with a slightly different environment [5]. This simplifies testing, deployment, and iterative development in general.

- **Setup Time & Complexity:** Container images define the runtime configuration once, so developers can deploy new application instances with minimal setup [5]. This drastically reduces configuration and documentation workload, which speeds up the development, deployment and maintenance process overall [7].

- **Feature Set & Extensibility:** Container-based deployments can be seamlessly integrated with orchestration tools like Kubernetes, which offer advanced features such as automatic (e.g., load-based) scaling, rolling updates, and self-healing [20]. Developers can quickly extend images with new libraries or microservices, adopting a plug-and-play approach to feature development.

- **Industry Adoption & Community Support:** In the last years, containerization effectively became the absolute standard for cloud-native applications [7]. Major cloud providers like Amazon Web Services (AWS), Microsoft Azure and Google Cloud, offer fantastic support for container deployments, which results in a broad industry acceptance and a very strong community ecosystem.

- **Scalability Requirements:** Since containers are lightweight, spinning up or scaling down replicas in response to load changes is very easy and straightforward [5]. This elasticity is perfect for environments demanding high availability and resilience, including the non-deterministic simulation workloads discussed in section 3.2.

4.4.3 Continuous Integration & Continuous Delivery (CI/CD)

CI/CD pipelines automate building, testing, and deploying software. This ensures rapid deployment of code changes and consistent build and deployment logic. This method fits perfectly for use with agile development methods. In this thesis, CI/CD pipelines will be used for the automated build and deployment of container images.

- **Future Maintainability & Ease of Use:** By integrating container image build processes, potential code analysis, and automated deployments into pipelines, important but repetitive tasks are being streamlined. This structure makes rolling out new features or patches more predictable, efficient and less prone to human error, reducing the load on developers [26].

- **Setup Time & Complexity:** Many modern CI/CD platforms (e.g., GitLab CI, Jenkins) are extensively documented, provide example configurations, and have many community plugins. Although initial configuration can be a little bit involved, the long-term benefits, like having to do fewer manual tasks and a more consistent release processes in general, far outweigh the initial setup effort [46].

- **Feature Set & Extensibility:** A well-defined pipeline can run all kinds of tasks:unit tests, integration tests, security scans, performance benchmarks, image deployments and much more in a very specifically defined and automated way [46]. Developers can also extend these pipelines to do many others tasks like generate documentation, deploy images to multiple environments, or even prevent these processes automatically if issues arise.

- **Industry Adoption & Community Support:** CI/CD is standard in large tech companies (e.g., Microsoft, Google, Netflix) and has become a best practice across various industries [46]. The available documentation, community support and various best-practice guides make it easy to adopt and maintain over time.

- **Scalability Requirements:** In general, pipelines can be parallelized, individual stages can be distributed to a large number of workers in order to handle an increasingly high workload [26]. This ensures that large projects with many contributors can still maintain structured release processes.

4.4.4 Domain-Driven Design (DDD)

Domain-Driven Design focuses on modeling software around the core business domains. It allows the logical separation of components and models based on their logical domain [18]. Using this design method, elements of the application are separated into bounded contexts [50]. Models are separated and redefined according to their logical domain. Domain-Driven Design principles promise to be the foundation for solving the main issue that was encountered when combining the Event Sourcing design pattern with the monitoring of non-deterministic simulation proceeses [18].

- **Future Maintainability & Ease of Use:** DDD enforces clear boundaries (bounded contexts), reducing transient dependencies between data models [40]. By encapsulating each subdomain's logic, the system is easier to modify and maintain, not only during the initial design, but for future maintainability as well.

- **Setup Time & Complexity:** Though DDD can introduce an initial learning curve and might require additional planning to set up these domain boundaries [40], I strongly believe it pays off by avoiding architectural designs that become unmanageable in the long run [50].

- **Feature Set & Extensibility:** Because each domain context stands on its own, developers can add new functionalities or refine existing ones without affecting unrelated domains [50]. This modular approach improves parallel development with multiple contributors, a big advantage in conjunction with microservice-oriented architectures [40].

- **Industry Adoption & Community Support:** Many books and articles in relation to Domain-Driven Design are available. There are many example projects and tutorials available online, even in C# and .NET.

- **Scalability Requirements:** By using the separation of concerns [50], Domain-Driven Design works very well when developing software for distributed systems [18]. Each bounded context can potentially be a standalone project or service. This allows horizontal scaling as needed for computationally heavy processes [50]. In combination with Event Sourcing, this will prove to be the key for making the architectural idea of this thesis work.

4.4.5 Model-View-ViewModel (MVVM)

MVVM is a widely adopted pattern for building frontends, particularly in applications with dynamic or frequently changing data. It separates the user interface (View) from the underlying data and logic (ViewModel), which binds model data (Model) to the UI [2]. By using the MVVM design pattern, input forms and data represented in the frontend can be logically disconnected from the underlying event-based architecture. This is practically a UI-specific way to separate data from different bounded contexts in the frontend.

- **Future Maintainability & Ease of Use:** This separation results in cleaner code and makes individual components more reusable. Developers can focus on UI aspects independently of core logic, reducing complexity when updating or expanding features. When changing a detail in the data representation in the frontend, the separation of ViewModels from model data prevents having to cascade modifications to the lower application layers [2].

- **Setup Time & Complexity:** Although structuring a project around MVVM may require some planning, many frontend frameworks (e.g., WPF in .NET, Vue.js, or others adopting similar patterns) make this setup a lot easier. As .NET 8 has built-in support for data binding and dynamic state management, as well as project templates available in Visual Studio 2022, MVVM becomes relatively easy to set up and use.

- **Feature Set & Extensibility:** Because each part of the pattern has a specific responsibility, adding new UI components or modifying existing ones is less disruptive to the rest of the application. As mentioned, by separating UI-specific data from model data, modifications do not cascade to the lower application layers [2]. Many plugins and community libraries further simplify the implementation of MVVM.

- **Industry Adoption & Community Support:** MVVM is a standard pattern in Microsoft's ecosystem, being central elements in frameworks like WPF, Xamarin or MAUI, which results in a strong ecosystem support. As mentioned, a multitude of tutorials, sample projects, and design guidelines exist, speeding up development.

- **Scalability Requirements:** While "scalability" in the frontend context often refers to code organization and performance under heavy data loads, MVVM supports both. Views remain lightweight and can be rendered efficiently, while ViewModels handle complex logic in a structured manner [2]. The architecture can be scaled in the context of reusing individual bounded contexts with new modules, which becomes possible because changes data representation are decoupled from the underlying data.

4.4.6 Event Streaming Through an External Service

Relying on an external event streaming platform enables real-time, decoupled communication between the individual microservices planned in this thesis. The two most used event streaming platforms are Kafka [29] and RabbitMQ [15]. When using an application architecture inspired by Event Sourcing, the separation of concerns through an event streaming platform makes the separation of the computation cluster from the main application logic possible [29]. This is another integral part of the core idea and is essential for achieving the goals set in section 3.4.

- **Future Maintainability & Ease of Use:** By offloading the complexity of event transmission to a specialized service, developers can avoid reinventing pub/sub mechanics or dealing with low-level networking details [15]. This significantly reduces system complexity.

- **Setup Time & Complexity:** Although initial setup may involve configuring things like services, topics, partitions, exchanges and consumer groups [29], I believe that the advantages of combining Event Sourcing with an external event streaming platform make this effort worthwhile and, for the goals set in this thesis, even necessary. Once in place, streaming platforms are straightforward to scale and maintain, with built-in monitoring and a great fault tolerance [52]. When events are well-defined, the event stream acts as a kind of universal interface available for any kind of producer or consumer [29].

- **Feature Set & Extensibility:** Many event streaming solutions provide advanced features like replay capabilities and durable storage (protecting against crashed and misbehaving clients) [29]. These features are very valuable for auditing and later error recovery [52].

- **Industry Adoption & Community Support:** Companies across various sectors, especially in finance and e-commerce, use event streaming for important, real-time data pipelines. The extensive user base of services like Apache Kafka [29] and RabbitMQ [42] results in good community support and many best-practice guides made available online.

- **Scalability Requirements:** Scaling usually involves adding more brokers or increasing partition counts, letting the system handle higher throughput without having to change anything architecturally. In a microservices context, each service can horizontally scale its consumers to process events in parallel, improving fault tolerance and throughput [29]. This is important for the distribution of events in the planned computation cluster. By using an event stream as the communication interface for distributing simulation tasks, built-in functions of the event streaming platform,

which will likely be RabbitMQ, can be used for routing tasks according to various parameters [42].

4.4.7 Authentication Through an External OpenID or OAuth 2.0 Service

Delegating user authentication and identity management to an external OpenID or OAuth 2.0 provider (e.g., Auth0 or Keycloak) removes much of the complexity of handling user credentials internally [41]. By doing this, the user data does not need to be integrated into the database. OpenID/OAuth2 libraries for .NET make the integration of multi-user authentication quite easy. According to my research, it appears the be the simplest, but also the most versatily solution for integrating multi-user functionality into the desired prototype.

- **Future Maintainability & Ease of Use:** Delegating the authentication task to an external authentication service means that aspects like security, identity federation, and compliance are already sufficiently handled by this external authentication service [41]. This frees developers to concentrate on core application logic rather than building and maintaining custom authentication flows.
- **Setup Time & Complexity:** Configuring OpenID can be straightforward if the chosen provider provides a well-documented setup process and SDKs, which I believe especially Keycloak excels at [48]. Both Auth0 and Keycloak have quick-start templates and sample code for .NET 5+ (which are applicable to .NET 8), reducing the initial development effort.
- **Feature Set & Extensibility:** The mentioned identity providers offer features like multi-factor authentication, SSO, and role-based access control [48]. These can be integrated without substantial code changes to the main application. This is where the service providers in .NET 8 become the most powerful. Handling not only authentication, but also claim- or role-based authorization through middlewares makes the integration of an OAuth-provider like Keycloak very elegant for the code structure as well. Because of my experience working in the industry, I am confident this practice will be the best solution for achieving multi-user goals set in this thesis.
- **Industry Adoption & Community Support:** OpenID Connect and OAuth 2.0 are industry standards with extensive support across languages and platforms [41]. Rich documentation, community forums, and many other resources help when troubleshooting or implementing even the most advanced configurations.
- **Scalability Requirements:** Managed OpenID providers can handle spikes in authentication requests without requiring changes to the application infrastructure [41]. However, a powerful service like Keycloak can also be scaled horizontally through

their multi-site deployment functionality [45]. This ensures that the service can be deployed highly-available, so as the user base grows or load increases in general, the authentication system remains stable and efficient.

4.4.8 Source Code Analysis for Detection of Vulnerabilities

Automated scans for security vulnerabilities, coding errors, and compliance issues are essential in modern software development. Especially for critical systems, small mistakes in code can become fatal for the maintainer. To verify code integrity, a code analysis tool will be integrated into the CI/CD pipelines. This helps to keep the code clean, as well as identify any issues or critical bugs.

- **Future Maintainability & Ease of Use:** Incorporating code analysis tools into the development pipeline (e.g., through CI/CD) ensures consistent code quality through iterations and prevents critical security issues in code [30]. This proactive approach helps to keep technical debt low over time and significantly increases reliability and trust in the code.
- **Setup Time & Complexity:** Many scanning tools integrate seamlessly with CI/CD platforms, requiring minimal configuration to start producing useful reports. Developers can even customize the severity of checks or add project-specific rules.
- **Feature Set & Extensibility:** Advanced code analysis solutions can detect common known vulnerabilities, perform dependency checks, and flag insecure code [30]. They often even provide plugins for popular editors and IDEs. Even the newest security vulnerabilites, like the latest iteration of the OWASP Top 10 [4], can be checked and detected with these tools.
- **Industry Adoption & Community Support:** Security scanning has become vital with any applications handling any kind of sensitive data, and many open-source and commercial products are available. Frequent updates ensure that emerging threats or vulnerabilities are quickly identified [30].
- **Scalability Requirements:** If a codebase grows beyond a certain point, running scans in parallel or distributing the load across multiple CI/CD runners becomes important for maintaining reasonable scanning times before deployment of the newest iterations of an application. Most modern analysis tools support such scalable configurations, enabling them to handle large projects effectively.

Overall, all these choices, Event Sourcing, Containerization, CI/CD, Domain-Driven Design, MVVM, Event Streaming, OpenID Authentication, and Source Code Analysis, evidently match the goal of exclusively working with *state-of-the-art* technologies set in section

3.4. Each technology, method and pattern solves specific problems with the software architecture. They each seem to offer great features regarding scalability, maintainability, ease of use, and robust feature sets. Also, the chosen methods do not seem to conflict each other. There are a few specific technicalities that will arise, especially when combining Event Sourcing and Domain-Driven Design, but overall, when analyzing each method and pattern based on the selection criteria in section 4.2, I am confident the selections are robust and fulfill the *state-of-the-art* requirement. By implementing these methods and patterns together in a coherent manner, the system becomes very modular, which hopefully makes the challenge of provisioning and monitoring non-deterministic, distributed simulations easier At the same time, the integration of pipelines and code-analysis tools promises to enalbe high standards of security, performance, and reliability.

4.5 Tools and Development Environment

4.5.1 GitLab CI

GitLab CI is an important part of GitLab and can be used to realize automated builds, testing, and deployment of code. It adheres to the principles of Continuous Integration and Continuous Delivery (CI/CD), ensuring that every commit or merge request is validated by a pipeline of checks. The pipelines can be divided into stages, each stage being responsible for a different task. As mentioned, this can include building the code (possibly named the *build* stage), testing using various tools (named the *test* stage), uploading Docker images directly to an image repository (named the *publish* stage) or even deploying them to an already set-up testing environment (named the *deploy* stage). The key benefits of GitLab CI include:

- **Integration with Version Control:** GitLab CI seamlessly connects with Git-based repositories, triggering pipelines based on defined rules for branches, tags, or merge requests.
- **Scalability:** Multiple runners can be configured to handle various workloads, making it easier to scale testing and deployment for complex projects.
- **Extensibility:** Users can write custom scripts and set up multiple stages (e.g., build, test, deploy) in a single configuration file (`.gitlab-ci.yml`), allowing developers to define their own processes as they need.
- **Monitoring:** GitLab provides a web-based interface to monitor pipeline status, logs, and artifacts in real time.

By automating repetitive tasks and standardizing the testing and deployment workflow, GitLab CI makes iterative development easier and helps to improve code quality over time.

It seems to be a great tool for tools which experience frequent updates and changes, which can be valuable especially with microservices-based applications where platforms are divided into several individual parts. The University of Bremen offers a GitLab deployment for computer science students, which will be used in this project. The already integrated, shared GitLab runners allow the integration of GitLab CI/CD pipelines without any need for set up or configuration. The only thing that is needed is the creation of a GitLab repository with a `.gitlab-ci.yml` file in the root directory.

4.5.2 Docker Desktop for Windows

Docker Desktop is a local development tool that provides an easy-to-use environment for containerizing applications. Running Docker on Windows historically involved additional layers (e.g., VirtualBox or Hyper-V), but Docker Desktop simplifies this setup substantially. Key reasons for its use include:

- **User-Friendly Interface:** Docker Desktop offers a graphical dashboard to manage containers, images, volumes, and networks. This makes managing containers and debugging easier.
- **Cross-Platform Consistency:** Although installed on Windows, the container runtime behaves similarly to Linux-based deployments, minimizing problems or inconsistencies when moving from a development to a production environment.
- **Integration with Windows Subsystem for Linux (WSL):** Developers can leverage WSL 2 for faster performance and more Unix-like behavior, improving developer experience.
- **Seamless Upgrades:** Updates and patches are handled through Docker Desktop itself, making it straightforward to stay current with new Docker releases.

This tool enables container-based development and makes testing the prototype in different environments easier and more transparent, improving code testing and better debugging in general.

4.5.3 Visual Studio 2022

Visual Studio 2022 is the primary Integrated Development Environment (IDE) chosen for this project. Developed by Microsoft, it provides a robust feature set tailored to .NET development and offers deep integration with Docker, Git, and NuGet. Important advantages are:

- **Docker Integration:** Visual Studio's built-in Docker support allows developers to debug applications running inside containers, set breakpoints, and inspect variables without leaving the IDE.

- **NuGet Package Management:** Seamless integration makes it easy to add, remove, and update .NET libraries while automatically handling dependencies.
- **Version Control:** Visual Studio's user interface for Git (and GitLab) simplifies common tasks like branching, merging, and resolving conflicts.
- **Refactoring & Code Analysis:** Advanced refactoring options and real-time code analysis help maintain cleaner, more efficient code.

While JetBrains Rider is also a powerful alternative, Visual Studio 2022 is directly supported by Microsoft, just like C# and .NET, and does not require an extra license for the core development environment. Visual Studio 2022 Community Edition can do everything that is required in this project. Having the same developer for the programming language, framework and IDE improves troubleshooting and ensures I am always working with the latest up-to-date and state-of-the-art features.

4.5.4 RabbitMQ

Though Apache Kafka is often a top choice for high-throughput event streaming, RabbitMQ is well-suited for scenarios emphasizing complex routing and flexible message-exchange patterns. Specific reasons for using RabbitMQ in this thesis include:

- **Rich Routing Features:** RabbitMQ supports *direct*, *topic*, and *fanout* exchanges, offering more nuanced control over message distribution.
- **Lightweight Footprint:** It can be easier to set up and run compared to Kafka, especially for smaller clusters or local development environments.
- **Mature Ecosystem:** RabbitMQ has been in use for over a decade, with a wide array of client libraries, plugins, and community resources, including NuGet libaries for C#.
- **Ease of Configuration:** It has a user-friendly management UI for creating and monitoring queues, exchanges, and bindings.

In the context of this thesis, where events need dynamic routing and will need to be monitored closely during evaluation, RabbitMQ's exchange model is advantageous. However, if high-throughput streaming of large data volumes was the primary focus, Kafka might be revisited, as it seems to be better optimized for that kind of application. The integrated web-based UI of RabbitMQ enables the monitoring of events and data structures without the use of external tools and therefore makes it a better fit for this project than Kafka.

4.5.5 Keycloak

Keycloak is a centralized identity and access management solution that supports modern authentication protocols like OpenID Connect and OAuth 2.0. This external authentication service addresses:

- **Single Sign-On (SSO):** Users can sign in once to access multiple services, which simplifies user management in a microservice-based environment.
- **Role-Based Access Control (RBAC):** Administrators can define roles and permissions that span multiple applications, improving security consistency.
- **Federated Identity:** Keycloak integrates with external identity providers and user databases (e.g., LDAP), offering a solution for the unification of multiple platforms.
- **Audit & Compliance:** Centralizing authentication records helps maintain a clear audit trail. Changes in user entries do not need to be synchronized across multiple platforms.

By offloading identity management responsibilities to a dedicated tool, the system remains more modular and secure. Developers focus on core logic instead of the intricacies of handling user credentials.

4.5.6 SonarQube

SonarQube provides static code analysis, measuring code quality and detecting vulnerabilities, bugs, and code smells. Its integration with CI/CD ensures that quality gates are automatically checked, enforcing consistent code quality. Key advantages are:

- **Multi-Language Support:** Although primarily used for C# in this project, SonarQube can analyze all kinds of programming languages.
- **Extensive Rulesets:** Predefined rules catch common issues such as SQL injection risks, simple null pointer exceptions, or even memory leaks.
- **Customizable Thresholds:** Quality gates can be configured to fail a build if a certain number of critical issues is detected, increasing the system's reliability.
- **Visualization Dashboards:** Developers get a clear overview of code metrics, technical debt, and overall quality trends over time.

In the context of building a large-scale platform for provisioning and monitoring of 3D print simulation processes, continuous quality assurance helps minimize risks of runtime failures and security breaches.

4.5.7 Harbor or Gitlab Container Registry as Image Repository

Harbor is an open-source container image registry that goes beyond the functionality of a simple Docker registry. It supports private repositories, vulnerability scanning, role-based access control, and replication across multiple registries. Similarly, Gitlab has a feature called Container Registry that developers can enable, which provides the most important features of Harbor in a more simple way, directly integrated into the repository. Notable features include:

- **Security Scanning:** Harbor can scan uploaded images for known vulnerabilities, adding an extra layer of defense before deployment. Using Gitlab CI/CD Pipelines, images can be scanned using external tools before being uploaded to the Container Registry as well.
- **Access Control:** Administrators can set permissions for different projects or repositories, this is especially useful in work environments with multiple developers and projects. This feature is available in both Gitlab Container Registry and Harbor.
- **Replication & High Availability:** Harbor can mirror images across different geographic regions or data centers, improving uptime and reducing latency. In terms of high-availability, it fits the research goals. As a proof-of-concept, this is not necessary, but important to consider. Gitlab Container Registry is more limited in this regard.
- **Integration with CI/CD:** Build pipelines can push images to either Harbor or Gitlab Container Registry, and automated triggers can deploy them to testing or production environments.

Harbor and Gitlab Container Registry are secure and well-organized registries for Docker images, which fits well with the *future maintainability* and *security* criteria set in section 4.2. For simplicity, Gitlab Container Registry will be used for the prototype, since advanced features of Harbor are not needed to evaluate the architectural concept of the core research idea. In theory, both platforms result in the same approach. With Harbor more difficult to set up (requiring dedicated storage, servers, etc.), while not providing any immediate additional value, Gitlab Container Registry is the more natural choice.
In summary, these tools were selected to improve development routines, security, and fit well with the desired microservice-based architecture. GitLab CI automates testing and deployment, Docker Desktop simplifies local containerization and testing, Visual Studio 2022 features great .NET development features for coding and debugging, RabbitMQ enables flexible message routing through event-streams, Keycloak centralizes identity and access management, SonarQube enforces code quality consistency, and Harbor stores container images. All these components fit the goals set in section 4.2 of this thesis, like scalability, maintainability, high-availability and more. They offer all the essential features

and requirements for achieving the goal of orchestrating distributed, non-deterministic metal 3D print simulations to a computation cluster.

5 Architectural Design

The architectural design of this application plays is the most important part in ensuring that all functionality is well-structured, maintainable, and scalable. This chapter describes the technical construction process of this architecture. It begins by demonstrating how event-sourcing principles were adapted to non-deterministic simulation workloads and how an external event stream cleanly decouples the application layer from the compute cluster. It then shows a domain-driven redesign of the entire data model, including the shift from event-centric to a hybrid approach using relational persistence and the introduction of layered, context-specific aggregates. Next, it maps every functional requirement to a service definition, explains lifetime scopes and bounded contexts. In particular, this section aims to define the key events that drive communication across services, establish clear context boundaries for domain logic, and outline a comprehensible data structure that match the requirements of an event-sourced system. We will also examine how the codebase can be divided into distinct projects, each representing a dedicated area of responsibility within the application. By identifying and detailing the various microservices, we can then assemble a final architectural blueprint that captures how these components interact with one another to orchestrate non-deterministic simulation processes efficiently. Through this approach, the application will offer both flexibility and transparency regarding real-time process provisioning, monitoring, and adaptation to all kinds of situations in a scalable way. All architectural design steps were done with high-availability compatibility in mind. Finally, the chapter assembles these pieces into a complete and fault-tolerant architecture that scales horizontally and can be redeployed or updated without any downtime. Readers should focus on how each subsection ties an identified problem to a design decision and how the combined solution satisfies the requirements listed in chapter 3.4.

5.1 Event-Sourcing

As a reminder to section 4.4.1, Event Sourcing is a design pattern in which every change to an application's state is represented as a distinct event. Rather than storing only the latest state of the data, the system keeps a time-ordered sequence of state modifications, allowing any part of the system to reconstruct past or current states by replaying these events [34]. This approach offers benefits like easier debugging and auditing, since it preserves the full history of how the system arrived at its present condition.
In the architecture of this thesis, Event Sourcing is used in combination with an external event streaming mechanism powered by RabbitMQ. This separation of event generation and transport creates a clear boundary between the core application logic and the computational cluster that will run potentially non-deterministic simulations. By streaming events through RabbitMQ, the application can orchestrate and track simulation processes without being tightly coupled to the underlying cluster infrastructure [14]. This design en-

sures scalability, maintainability, and a clean separation of concerns, which are especially important when managing complex, distributed workloads.

5.1.1 Definition of Events

Events serve as the foundational building blocks of any event-sourced architecture, representing all relevant changes in the state of the managed simulation processes. By defining a specific set of events, the system maintains a clear structure for how information flows and is represented, ensuring that each change in a simulation process is auditable and trackable. The events outlined below form the core of the application's logic and represent the data exchanged between services, including the orchestration engine, user interface, and the computational cluster.

- **Start Job Event:** This event signals that a new simulation job should be initiated. It includes key metadata, such as the type of simulation, input parameters, and any references to required data or configurations. Upon receiving this event, the system provisions the necessary resources and prepares for job execution.
- **Job Received Event:** Once a simulation job has been successfully provisioned and actually begins its run, a *Job Received Event* is emitted. This event confirms that the process is now active in the computational cluster and may include a timestamp, an assigned job identifier, and potentially the node or container responsible for the computation.
- **Status Update Event:** During the execution phase, a simulation job may go through various stages. E.g., container initialization, file copy or computation. Whenever a change in the job's execution state occurs, a *Status Update Event* is generated. This can help monitor progress (e.g., a percentage complete) or track intermediate results, allowing for insights into the simulation during execution itself. Upon successful completion of a simulation, the system also emits this event. Therefore, this event also includes a final result, potentially any relevant metrics (e.g., total runtime, resource usage), and metadata like a path for storing or retrieving the results. It can also being used to notify other services that may depend on these results, such as for reporting or visualization, that they can now proceed.
- **Job Failed Event:** If a simulation process encounters an unrecoverable error, e.g., when the user specifies an invalid input path, hardware issues, or any kind of unexpected runtime issues, the responsible computation node generates a *Job Failed Event*. Details, such as error messages, could be included to help identify the cause and potentially trigger automated rections, like retrying the job or notifying the error to a user via email.

Collectively, these events form a concise kind of ”vocabulary" for representing all significant transitions in a simulation's lifecycle. They also enable a range of features, like real-time status dashboards, automated error handling, and detailed audit trails, that are central to the architecture's flexibility and reliability. These events are the foundation of the Event Sourced architecture being built in this thesis.

5.1.2 Context Separation through an Event Streaming Layer

A key principle of this architecture is the clear division of responsibilities among the application layer, computation nodes, and the persistence layer. At the heart of the separation between application layer and cluster nodes is an event streaming platform in the form of RabbitMQ that routes events between these components.

Application Layer The application layer is responsible for initiating and orchestrating simulation requests as well as responding to updates from the computation nodes. When the system needs to start a job, the application layer publishes a *Start Job Event* to the event streaming platform. It then listens for responses, such as *Job Started*, *Status Changed*, *Job Finished*, or *Job Failed* (each described in section 5.1.1 above), which are published by the computation nodes as they perform or conclude the work. Because all communication travels through RabbitMQ (or a similar messaging service), the application layer remains largely unaware of which specific node handles the job or how exactly the simulation is executed, thus enforcing a clean separation of concerns.

Computation Nodes On the other side, the computation nodes subscribe to the relevant event queues to receive job and update requests. Upon receiving a *Start Job Event*, a node allocates resources and begins the simulation. As the job progresses, the node publishes updates back into the event streaming system. This architecture allows for flexible scaling: additional nodes can join or leave the cluster without requiring the application layer to make any direct adjustments, as their entry and exit are solely communicated through published and subscribed events.

Persistence Layer In an event-sourced environment, the persistence layer must store events in a way that preserves their chronological order and supports replayability. Rather than merely persisting the final state of a job (e.g., whether it ended successfully or failed), the system captures each significant transition as a separate event record. For example, when a *Job Started Event* arrives, the persistence layer appends it to the event store alongside timestamps and metadata, allowing future consumers to reconstruct or analyze the job's entire lifecycle.
To facilitate efficient lookups and queries, the application might employ additional read models or projections. These read models aggregate events into more compact, query-

friendly views, for instance, a data-acccess (repository) function that returns a list of *Job Models* that represents the latest data state, which can be used for monitoring. Such projections are updated whenever a new event arrives, ensuring that the system can serve real-time information while still maintaining a strictly event-based record.
Overall, the benefits of layered events are the following:

- **Loose Coupling:** By communicating solely through events, the application and persistence layer operate independently from each other.
- **Scalability:** The computation layer can be expanded or contracted simply by adding or removing nodes that subscribe to the same RabbitMQ event exchange.
- **Resilience:** Event messages can be retried, logged, or re-delivered if a node is momentarily unavailable, improving fault tolerance.
- **Auditing & Replay:** Storing events in an ordered fashion preserves historical data, enabling audits, debugging, or advanced analyses without impacting live operations.

To summarize, the event streaming layer serves as the glue connecting the application and computation contexts, each operating within its own well-defined boundary. This architectural choice not only simplifies development and maintenance but also supports the high level of flexibility required by non-deterministic execution time and potentially resource-intensive simulation processes.

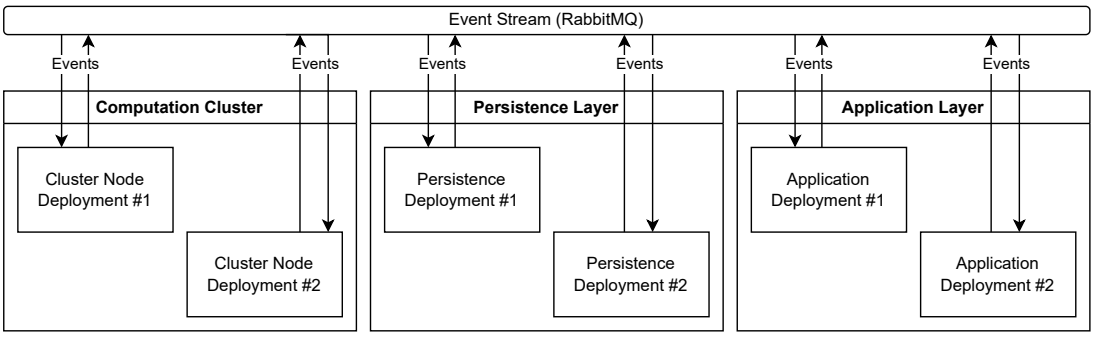

Figure 3: Cluster nodes, application and persistence layer communicating through an event stream, depicted in a level 1 graphic

In figure 3, the three individual layers described above are being depicted. In this example, two instances of each service layer have been deployed. Each deployment can be considered to be a microservice and listens to the event stream provided by RabbitMQ. All data exchanged through the event stream adheres to the event structure defined in section 5.1.1. When observing this data structure one specific problem becomes apparent: When an event is produced from one of the bounded contexts (e.g. from the application layer), it needs to be delivered to both of the other contexts. when analyzing the data structure more

closely, this is caused by the data in the persistence layer being transitively dependent on both data states of the application layer and computation cluster. A fanout exchange type would solve this, which would cause events published to that specific RabbitMQ exchange to be distributed to all consumers, but when doing this, we loose the access to routing functions. From a data consistency perspective, this is also very non-optimal. Looking even closer, it becomes apparent: Data changes relevant to the persistence layer are also always relevant to the application layer, but not to every node in the computation cluster. The application layer does need the data managed by the persistence layer for various tasks, such as process state monitoring. However, this is not relevant to the computation cluster nodes. A solution to this problem would be the separation of application and persistence layer not in a physical, but in a logical sort of way. The separation of application and persistence layer in a logical instead of physical way also removes the need for incorporating an event stream between them. Doing this will also allow other potential future components running in parallel to the application layer to access the underlying database separately from the computation nodes and the application layer itself. Additionally, if the persistence layer was a separate service listening to its own RabbitMQ exchange, the set of events that need to be defined would be significantly more complex, cluttering the event stream, as well as being more of a burden than a blessing for developers. By only separating the persistence layer logically (not physically) from the rest of the application, the persistence context and therefore database access can be referenced by the application layer directly like a library. This is very elegant from a coding-structure perspective and makes future expansion of the platform with new components that need to access the database directly without having to go through an event stream significantly easier. Since the persistence layer needs to feature many relatively complex data manipulation methods, if the architecture were to use an event streaming layer in front of the persistence layer, even slight changes in data manipulation logic would cascade to the event data structure and therefore create a transient dependency to both application layer and computation nodes, which strongly goes against modularization of the program architecture. In contrast, by funneling all communication between the application layer and the cluster nodes through an event exchange only, the application decouples service boundaries between services in the application layer and a potentially unlimited (but especially, from the perspective of the application layer unknown) number of cluster nodes, allowing for unlimited scalability, rolling updates and reducing interdependencies in general.

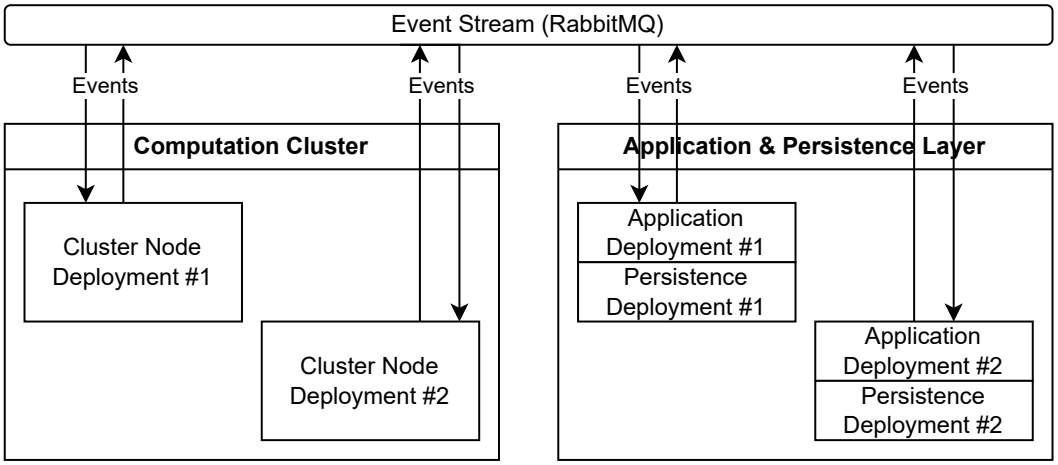

Figure 4: An alternative approach to the context separation between application layer, computation cluster and persistence layer

This change is being depicted in figure 4. Instead of the application and persistence layers being physically separated through the event stream, in this version, they are physically the same service. However, models, methods and functions are still logically separate, with the persistence layer being a completely separate project that could also be incorporated by other, new components wanting to utilize the capabilities of the computation cluster, for example through automated tasks with specific external triggers instead of a user interface. It should be noted, that this does not change the data structure fundamentally: Data between the application and persistence layer can still be exchanged through the same events. From a level 1 point of view however, the designation of the persistence context changed. This way, the separation of services through the RabbitMQ event stream stays as simple as possible and, as previously mentioned, will result in a very clean and concise code structure.

5.1.3 Incompatibility with Process Properties

One of the main advantages of Event Sourcing is that it captures all state transitions in a way that enables both traceability and replayability. By replaying the original set of events, one can theoretically reconstruct the system's state at any given point in time. However, when dealing with non-deterministic simulation processes (see section 3.2), a major incompatibility becomes apparent.

As mentioned, one fundamental feature of Event Sourcing is the possibility to replay an identical sequence of events to arrive at the same final state [34]. However, non-deterministic simulations may generate different outcomes even when started with the same *Start Job Event* and identical input parameters. In this scenario, replaying events does not guarantee the same end state, posible a significant problem for using Event Sourcing in the first place.

Another key objective of Event Sourcing is to maintain a clear audit trail of all state changes. Given the unpredictable nature of these simulation processes, the link between cause (e.g., a *Start Job Event*) and effect (e.g., final simulation results, represented with a *Job Finished Event*) becomes ambiguous if the process can diverge or fail randomly. However, capturing each event is still useful. It allows for historical tracking of processes and traceability of who does what but the inability to replicate the exact outcome does reduce the value of that event sequence as it's no longer a deterministic process and does not represent exactly how the system will behave when executed again. Since many 3D printing processes are realized in a non-deterministic way intentionally however, this reduction in traceability can be tolerated. Logging who does what and when, including the actual (although abstracted) outcome of the process, is still valuable in many different ways. The platform provider can collect better metrics and the results of previously executed processes, although not repeatable, is still a very valuable information to have for both users and the platform provider.
So, during the planning and early design of the Level 1 data structures, it became evident that a pure Event Sourcing approach would not adequately handle the non-deterministic nature of metal 3D printing simulations. Although Event Sourcing provides an elegant mechanism for capturing state changes, we cannot rely on replays to produce consistent, predictable outcomes when looking at execution history. Because of these constraints, I concluded that Event Sourcing alone cannot completely fulfil the the requirements set for this project. Adhering strictly to the classic Event Sourcing pattern means facing a mismatch between the ideal of replayability and the unpredictable behavior of metal 3D print simulations. Therefore, extra steps or modifications are necessary:

- **Augmented Audit Trails:** Additional metadata or versioning strategies could help record not just the start and end events but also the intermediate steps and results of these processes.

- **Hybrid Persistence Models:** Combining Event Sourcing with another pattern, possibly Command Query Responsibility Segregation (CQRS) or traditional state storage, might provide more appropriate data handling for non-reproducible results.

- **Domain-Driven Refinements:** In line with Domain Driven Design (DDD) principles, a domain model can capture the inherent variability of simulations. Entities or Aggregates could be designed with the possibility of multiple outcomes in mind, while still retaining a kind of audit trail to maintain the traceability advantage of the Event Sourcing pattern.

Given the project's aims, we need to try to retain most of the benefits of Event Sourcing, particularly the detailed record of state transitions in order to retain the potential for partial replay, while also acknowledging that a fully deterministic replay will not be possible.

Specific adjustments in data structures and process logic ensure that events still carry enough information for an adequate audit trail, even if the final simulation results may vary. This compromise happens to align with DDD principles, where the domain model can explicitly account for non-determinism, and events are treated more as activity logs rather than strict models deterministically represeting a state.
In short, while Event Sourcing remains valuable for capturing transitions and providing historical insights, the unique properties of non-deterministic simulations necessitate additional mechanisms or some sort of hybrid approach. These adaptations must ensure that the system's traceability and audit functions are preserved, without misleading end users into assuming that replaying the same events will always yield identical outcomes. After contemplating various different solutions I arrived at what I think is a good compromise given the project goals.

5.1.4 A Solution to the Incompatibility Problem

The incompatibility between non-deterministic processes and the exclusive use of Event Sourcing in the design of the data structure requires a more flexible approach to data persistence. While the concept of storing all state changes as events is appealing, it collides with the fact that identical simulations can produce different outcomes. To solve this issue, a change in the persistent data structure must be made.
I believe that it is best to avoid constructing the entire database schema around individual events. In a classical Event Sourcing model, each event transforms the state of an entity, and the event store itself becomes the system of record. For deterministic use cases, replaying the stored events starting from the beginning up to any given point will always result in the same final state. However, in non-deterministic scenarios, replaying those same events does not guarantee the same outcome, thus undermining one of Event Sourcing's main benefits.
Instead, the database can follow a more traditional schema, storing only the current state of each simulation job or process. Events still exist, but they serve as a mechanism for loose coupling and asynchronous communication between services, not as the sole source of truth for the database.
This design relies on persisting states rather than relying on event replay to recreate them. Following this approach should theoretically improve query speed as well. For each simulation job, the system saves its latest known data, such as run status, partial results, or metadata about its computational environment. The application logic then updates this state when it processes relevant events (e.g., *Job Started*, *Status Changed*, see section 5.1.1), but it does not store every state transition as a discrete database record. As a result, the database always reflects the most up-to-date information, circumventing the replay issue.
By moving away from an event-based persistence model, we run the risk of losing detailed historical records of how and when the data changed state. To mitigate this, each event

can still be logged in an external audit trail. Libraries for Entity Framework (EF) exist that automatically capture changes to specified entities, inserting audit records containing details like timestamp, changed fields, and user or system account responsible for the update. This also integrates well with the multi-user aspect of the desired application architecture.
Some of the advantages of using this hybrid-approach include:

- **Audit Log Storage:** These logs can be stored in a dedicated table or external service, enabling administrators or users to review or trace the evolution of any job over time.
- **Partial Traceability:** While this method does not allow perfect replay (due to non-determinism), it does preserve a chronological record of all changes.
- **Low-effort Implementation:** Libraries support the integration process, which reduces the implementation effort.

A major trade-off is that we lose deterministic replay of individual processes. Knowing the expected outcome and re-simulating until the system reaches the same expected state is theoretically possible, but highly impractical for the kind of non-deterministic processes we are working with in this thesis, as the same input may never yield the same result. Ultimately, I believe that in a system where process results are non-deterministic, pure replayability is not achievable under these conditions. Nonetheless, the chosen approach preserves many other advantages of event-driven systems as outlined in section 5.1.2, such as:

- **Loose Coupling & Scalability:** Services communicate via events, allowing horizontal scaling and resilience to transient failures.
- **Resilience:** Lost or failed events can be resent, and on a crash, the application can pick up from the last known state using audit logs.
- **Auditing:** An external audit trail ensures that all data changes are trackable. In combination with a multi-user access policy, each user can be held accountable for his own actions.

This hybrid design also indicates that the application domain will have multiple layers of representation. On one hand, the system must maintain a current-state model that mirrors the current status of each job. On the other, events and audit logs record the lifecycle of changes in a kind of history, without guaranteeing future replication of identical outcomes. Properly managing these layers and their dependencies to each other can be difficult, but is required for maintaining proper data integrity.
Given the complexities introduced by non-deterministic simulations and the proposed mixed persistence strategy, I believe the best way to establish a clear structure is utilizing

a strong domain model. Domain-Driven Design (DDD) principles (first introduced in section 4.4) guide how entities, value objects, and aggregates should be structured to maintain internal consistency. By modeling each component of the application domain and separating them according to their bounded contexts, event handling and state persistence can be logically separated and conflicts between models reduced.
In conclusion, the proposed hybrid-architecture retains many of Event Sourcing's strengths. It proposes a system perticularly for decoupling the different application layers implicitly through their code design. However, because of the non-deterministic nature of the simulation processes, we do have to accept that perfect replayability is impractical. By combining a traditional state-oriented schema with additional audit logs, this architecture proposition aims to solve the incompatibility problem posed by the non-deterministic nature of the simulation processes. This hybrid-approach promises to be a flexible software architecture, which could potantially find usage in many other applications and scenarios as well. If this concept will work in practice will need to be evaluated through the prototype.

5.2 Domain-Driven Design

Domain-Driven Design (DDD) is a method for designing software that aims to improve its structure in order to ensure data integrity, ease of development and to better define interfaces and the data that gets shared between them. Instead of just building features or tables in a database without context, DDD suggests that we first clearly define and understand the applications *domain*. Once the domain is well-defined, we shape the code and data structures around the established contexts.
In the case of this thesis, the domain revolves around *non-deterministic metal 3D printing simulations* and the processes that provision, monitor and manage them. By using DDD, we focus on how these simulations behave, what their data states look like, and how they change when interacting with other parts of the system. We then design our database entities, services, and modules to reflect those real-world interactions as accurately as possible.
The application desired in this thesis consists of many components. DDD helps break them down into clear, understandable parts (entities, value objects, and aggregates). By centering the design on the behavior and data that really matter, like how simulations start, fail, or produce results, we avoid cluttering the system with convoluted data models and creating transient dependencies between classes. As simulations or data states change, a well-defined domain model can adapt better than a system built around purely technical concerns.
Since fully event-sourcing non-deterministic processes is problematic, DDD offers a way to rebuild the data structures so that each entity reflects the true state of a simulation. We will store essential data, like current status, parameters, or partial results, directly in a more

traditional, state-oriented format. At the same time, in order to retain most of the advantages of event-sourcing (as described in section 5.1.4 above), we will maintain additional records in the form of audit logs for historical change tracking. DDD ensures these two concepts (state storage and historical tracking) are designed to match the domain's real workflows, making our architecture both practical and robust.
In the following sections, specific DDD patterns (like entities, value objects, and bounded contexts) will be used to reshape the data structure of the application. This helps to keep track of critical simulation states, while still supporting the scalability, monitoring, and especially the loose coupling required for a high-performance, distributed system.

5.2.1 Redesigning the Persistent Data Structure

When designing an application that relies on event sourcing, one might be tempted to build it in a monolithic way, where events are used to manage and store all the data in a single system.

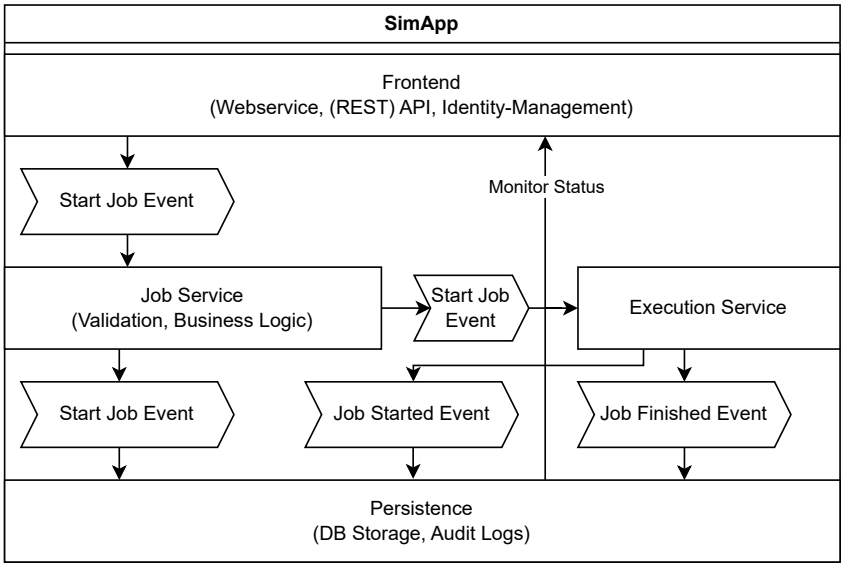

Figure 5: Abstract representation of a potential monolithic application architecture using event sourcing

Figure 5 shows an abstract example of such a monolithic architecture. Here, the application layer and persistence layer are logically separated in principle: the persistence layer receives events and handles data storage, while the application layer contains two main service functions:

1. A *Job Service*, responsible for business logic.

2. An *Execution Service*, responsible for running the underlying processes or simulations.

However, due to the downsides and outdated nature of monolithic software architectures (outlined in section 2.1), this structure is not desired. Although this approach separates the function of storing data from the function of handling business logic, it still mixes responsibilities to a considerable extent. In particular, it does not strongly enforce the principle of *responsibility segregation*, a concept that is central to Domain-Driven Design (DDD). A clearer structure emerges once we pinpoint the core tasks and isolate them into independent layers. Using the Domain-Driven Design principle of responsibility segragation [50], the software architecture will now be separated into its most abstract layers.

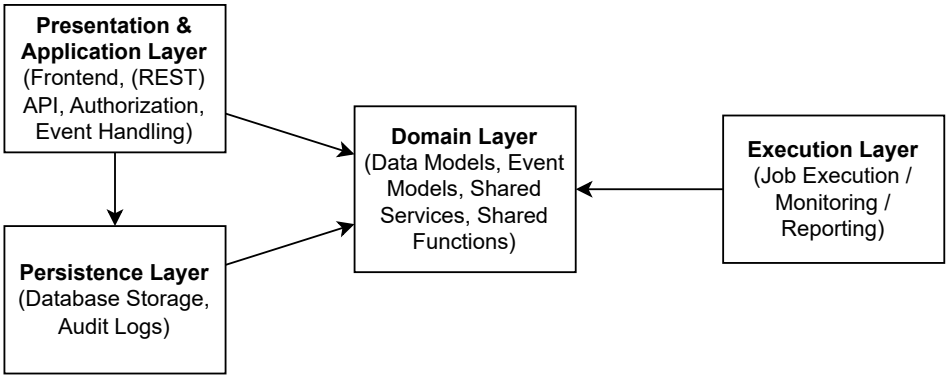

Figure 6: After responsibility segregation, four distinct layers are created

Figure 6 illustrates how four abstract but distinct layers are formed after a more careful separation of concerns. The most important addition is the *domain layer*, which contains a kind of shared language (data models, services, event models) used by all other layers. Rather than letting each layer handle its own version of the data, the domain layer provides common models that can be used as a kind of shared language between interfaces.
In this structure, the application layer still interacts with a persistence layer to handle database writes and reads. However, instead of designing the persistence layer solely around events, the system consumes events internally for orchestration or communication with other services (such as the execution service) while making more traditional *CRUD* calls to store the current state of entities. The domain layer becomes the heart of the application's design, dictating how data should be represented, validated, and exchanged across all parts of the system.

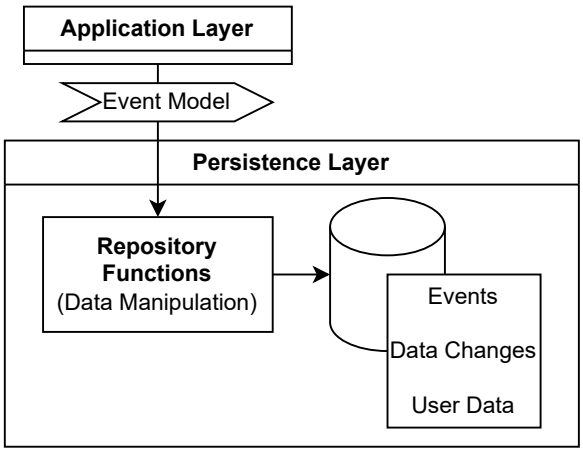

Figure 7: Rough representation of persistent data before doman-driven redesign

Figure 7 illustrates how a purely event-sourced database might look if one were to store all data changes as event records. Under this model, the repository functions accept event models as parameters and transform them into records reflecting the changes over time (e.g., which user triggered which event and how it altered the data). While this strategy can be appealing for deterministic operations (where replaying the event stream reliably reconstructs the final state), it becomes problematic for non-deterministic processes, as noted in Section 5.1.3. In non-deterministic cases, the same events might not yield the same final results upon replay, which is a problem in respect to one of the primary benefits of event sourcing.

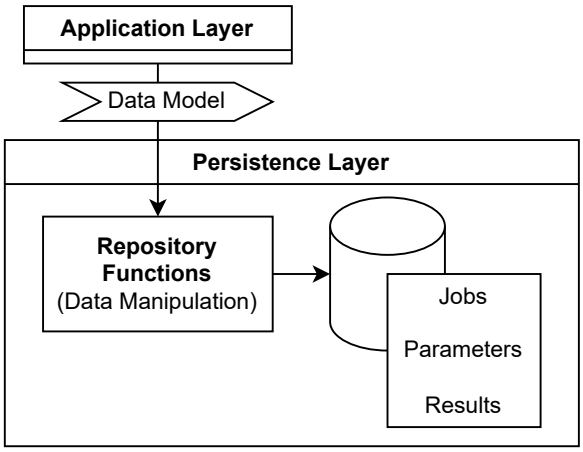

Figure 8: Rough representation of persistent data after redesign

Figure 8 shows a reworked persistence layer that uses a more traditional relational data storage. In this setup, repositories receive models representing the desired end state of the

data (such as a *Job* or *Tool* that needs updating) rather than events describing incremental changes. Each repository function then applies the modifications directly to the database, changing the currently stored data as requested. This approach avoids the overhead of parsing a potentially large chain of events just to understand the present state.
Although events remain essential for orchestrating simulations (such as "*Start Job Event*" or "*Job Update Event*"), these events mainly travel across the event stream for real-time communication between computing cluster and the application layer of the main application. Within the application itself, the persistence layer and application layer interact through a straightforward internal interface: repositories, in which domain models, instead of event models, are used to exchange data.

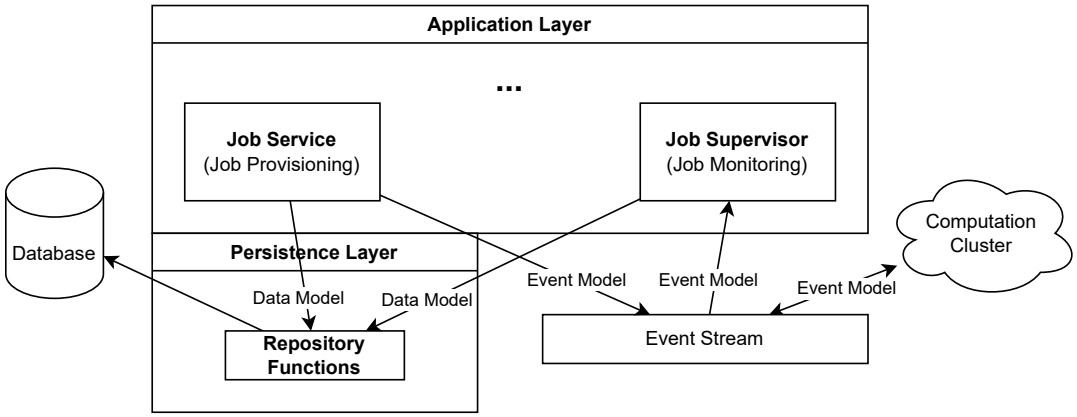

Figure 9: Representation of the type of data exchanged between application layer, persistence layer and the computation cluster

Figure 9 depicts the two distinct interfaces managed by the application: one for interacting with the persistence layer via repositories, and another for communicating with the computation cluster via events.

1. *Job Service:* Stores job information in the database, then sends a *Job Started Event* or similar to the event stream so that cluster nodes can consume it.

2. *Job Supervisor:* Consumes events like the *Job Update Event* from the event stream and updates the database entries accordingly to maintain an accurate job status in the database.

Notably, there are occasions where the application must generate or consume *both* domain-related models and event messages. For example, when a new job must be stored in the database first and then provisioned to the cluster, the application must handle two distinct but sequential operations. Should any issues arise (e.g., an error saving the job or a failure sending the event), a certain level of business logic is required to maintain consistency and avoid incomplete or inaccurate data states.

In conclusion, this hybrid solution balances the advantages of storing state-oriented data in an environment with non-deterministic processes with the key benefits of an event-driven application structure. While pure event sourcing is not fully suited for these non-deterministic simulation processes, mixing event-driven communication for orchestration with relational state-based data persistence results in a flexible software architecture that maintains the advantages of both. Domain-Driven Design ensures that each layer (application, domain, persistence, and execution) uses a shared language with consistent interfaces and data models.

5.2.2 Model Separation using Domain Logic

One of the key aspects of Domain-Driven Design (DDD) is the concept of *bounded contexts*, which are used to group related functionalities and data models. By splitting the system into multiple bounded contexts, each with its own set of entities, we can reduce complexity and keep each context focused on a specific part of the business logic. This approach also helps prevent "leaky abstractions", where changes in one area unintentionally impact another.
We begin our model separation at the persistence layer, which holds the long-term data storage for the application. Here, we define entities to represent the core components of the system:

- **Platform:** Stores information about the execution environment for a given tool or algorithm (e.g., Windows, Linux). Different Docker images can be based on distinct operating systems, so having a *Platform* entity lets the system know which cluster node is compatible with a specific tool's requirements.
- **Job:** Represents a simulation or computation task. It includes details like the algorithm name, version used, creation time, current execution status, the user to whom the job belongs, directories for logs and output, and the ID of the Docker process. This entity is central to tracking the progress and results of each submitted job.
- **Tool:** Stands for an algorithm or function that can be executed in the system. Each *Tool* references a specific *Platform*, indicating which environment it requires, and contains a Docker image name for hosting the algorithm. Tools can have multiple versions, each possibly requiring distinct parameters or Docker tags.
- **Group:** Identifies a user workgroup and contains a list of members. Users in the same *Group* can view one another's jobs, enabling collaborative or shared ownership of computation tasks.
- **ToolVersion:** Specifies a particular release of a *Tool*, including its Docker image tag, a label (e.g., "v1.0", "beta", "nightly", etc.), and possibly different sets of input parameters for each version.

- **Parameter:** Describes an input field or argument that a *ToolVersion* requires. It includes a name, description, data type, and a default value. This entity helps define the interface for running a job with certain algorithm settings.
- **ParameterValue:** When a new *Job* is created, the actual input values used for each *Parameter* get stored as *ParameterValue*. By referencing both *Parameter* and *Job*, it forms a link between the input definition and the actual arguments used in a specific simulation run.

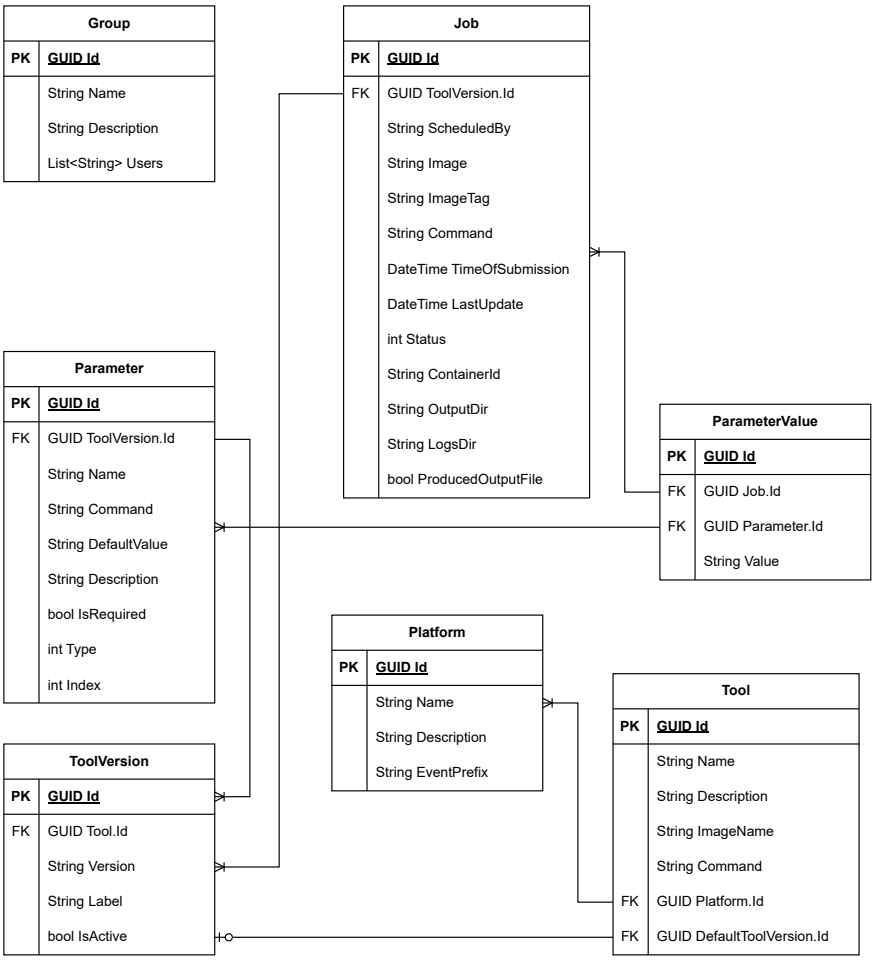

Figure 10: Entity-Relationship Model of the persistent data storage

Figure 10 shows the entity-relationship model for the persistent data structure. It's worth noting that user data, such as usernames, emails, or passwords, is not stored here, as user authentication is managed externally by Keycloak (via JWT tokens).
While the persistence entities capture the core database structures, DDD emphasizes that the *domain layer* should form its own bounded context, independent of specific database or infrastructure concerns. Accordingly, the system defines a separate set of domain models. These domain models still represent the same conceptual pieces (*Job*, *Platform*,

Tool, etc.) but do so with reduced dependencies on Entity Framework or database-oriented features. For instance, virtual properties used by EF to manage foreign keys are removed or replaced with references more suitable for domain logic.
Additionally, the domain models inherit from three foundational interfaces that come from DDD principles:

AggregateRoot An object that serves as the entry point to a group of associated entities and value objects. Changes to the aggregate are controlled through this root, ensuring consistency within that block of data.

DomainEntity A more general concept representing any object with an identity tied to the domain (e.g., a *Job* can be seen as a *DomainEntity* if it is not already an *AggregateRoot*). It still belongs to a bounded context but might delegate important operations to an aggregate root.

ValueObject Represents an immutable concept without its own identity. Typical examples include small, cohesive data groups like parameter values or configuration parameters that never change once created.

Structuring models this way provides a consistent way to handle them across different layers, reinforcing the rules and constraints of the domain rather than just reflecting database schema. In this case, following DDD principles, the Job, Group, Platform and Tool entities have been defined as AggregateRoots. Parameter and ToolVersion will be DomainEntities and ParameterValue is a ValueObject.
To manage inter-service communication, the domain layer also contains *event models* for the events described in section 5.1.1 (*Start Job Event*, *Job Update Event*, etc.). While these events carry enough data to inform other services of state changes, they do not expose unnecessary details that belong strictly to the persistence or presentation layers.
The final piece in this layered structure is the *view models*, which determine how data is displayed and manipulated in the user interface. In `.NET 8`, developers can reference these models directly in the HTML views by using an `@using <viewmodel>` tag. This feature enables easy data binding and reflection in Razor pages or MVC views, removing the need to manually sync changes between backend code and frontend forms.
The benefits of this approach are as follows:

- **Clear Separation of Concerns:** Each layer (persistence, domain, and presentation) focuses on its own responsibilities, reducing the risk of cross-layer dependencies.
- **Flexibility in Data Handling:** If the domain logic changes, the presentation layer can adapt without requiring a complete overhaul of the database.
- **MVVM Compatibility:** The data structure has been designed with the MVVM design pattern (described in section 4.4.5) in mind and closely fits its design principles. The

entity models serve as the “Model”, *domain models* operate as part of the “View”, and *view models* represent the “ViewModel” layer.

The structure of having distinct sets of models (persistence entities, domain entities, and view models) may seem elaborate and work-intensive, but it introduces several advantages:

- **Better Mapping to Business Logic:** The domain layer isolates the core business logic of the different layers from their specific technical details, making it easier to change data structures in response to changes in individual layer logic (prevent “leaky-abstractions”).
- **Resilience to Non-Deterministic Outcomes:** With domain models focusing on robust application behavior at the cost of event replay (explained in section 5.1.4), we can better handle the unpredictable nature of certain simulations while still maintaining a clean, auditable system state in the persistence layer.
- **Scalable Architecture:** By employing domain events in a separate bounded context, the system can be scaled horizontally (e.g., more computation nodes) without forcing large changes to the database schema or the user interface.

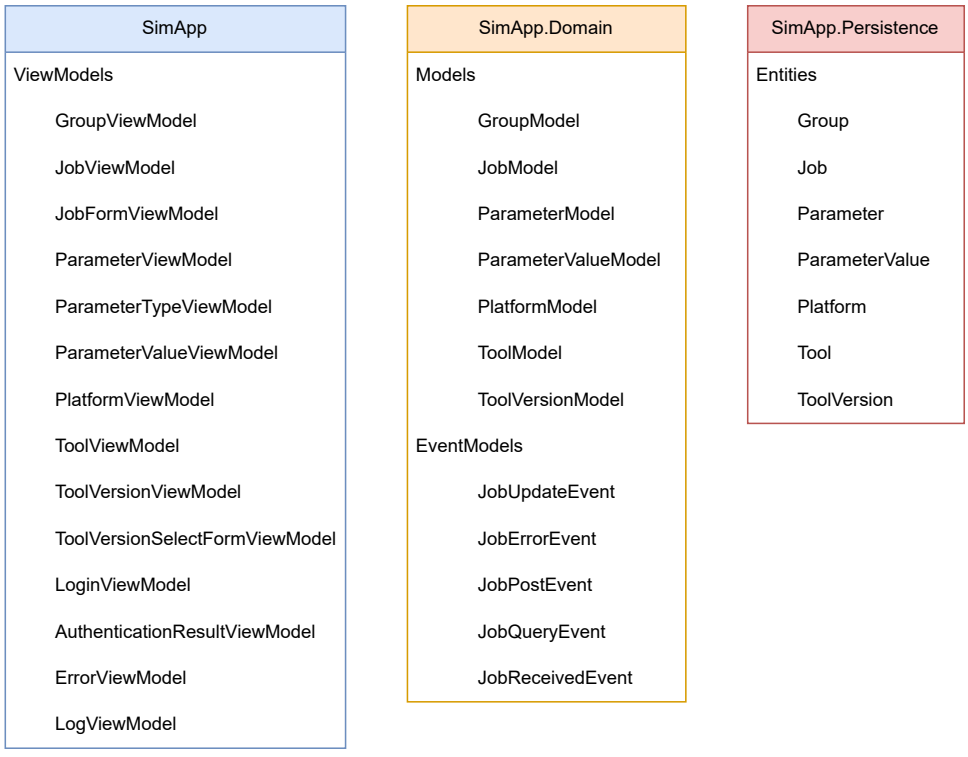

Figure 11: Data models sorted into projects based on their context designation

Figure 11 illustrates how these models fit into their respective layers, showing the flow from end-user interactions in the presentation and application layer (SimApp project) down to

the persistent storage (SimApp.Persistence project) with the domain models as a shared language in-between (SimApp.Domain project). The structure of these projects matches the layer model of figure 6 in previous section 5.2.1.
This multi-model approach, designed using DDD principles, ensures that each part of the system remains flexible and maintainable, while fulfilling the requirements defined in section 3.4. It also brings the benefits of loosely coupled services and clear data flow.

5.3 Service Definitions

Services in a `.NET` application are classes dedicated to handling specific functionalities, such as data access, business logic, or background processing. By splitting responsibilities into service contexts, the application can remain well-organized and easier to maintain. Additionally, `.NET` offers multiple service lifetimes to control how service instances are created and disposed of within the application's lifecycle: *singleton*, *scoped*, *transient*, and *hosted services*.

- **Singleton Services:** A *singleton* service is created once and shared throughout the application's lifetime. This is useful for components that must keep information consistent across all users or requests (e.g., caching mechanisms), where only one instance across the application should exists at a time.
- **Scoped Services:** A *scoped* service is tied to the lifetime of a single request in ASP.NET, or a comparable scope in another environment. Each incoming request receives its own instance, improving data isolation and reducing unintended side effects between concurrent requests.
- **Transient Services:** A *transient* service is instantiated every time it is injected or called. This is well-suited for lightweight, stateless operations that do not require shared data and can benefit from fresh and short-lived instances.
- **Hosted Services** *Hosted services* are background tasks that run alongside the main application. They often perform recurring or long-running work such as polling external APIs, cleanup routines, or processing message queues and can be started or stopped by the application's host.

In the sections that follow, we will define which tasks belong to which service context and detail how their chosen lifetime benefits the overall architecture.

5.3.1 Task Identification and Context Separation

Before assigning services to their respective lifetimes (singleton, scoped, transient, or hosted), it is necessary to map out the tasks each project or layer must handle. Breaking

down these responsibilities helps ensure that related operations stay in the same context and improves the way components communicate. In the practical realization of the application, services will be sorted into the projects defined in the previous section 5.2.2, so we will use this project structure.

SimApp Project

- **Publish SimApp-Specific Events:** Publish application-level updates (e.g., `Start Job` or `Job Query` events) to the RabbitMQ event stream.
- **Consume Job Events:** Listen to event queues for *Job Received*, *Job Error*, or *Job Update* messages and handle them (e.g., update the database, resubmit jobs).
- **Job Supervision:** Detect stale jobs, re-initiate them if necessary, and provision newly created jobs to the appropriate nodes.
- **Authentication Handling:** Manage user logins and interactions with the Keycloak-based authentication flow.
- **Business Logic & CRUD:** Implement operations for creating, reading, updating, and deleting entities in the database through the appropriate functions in the persistence layer.
- **File Management:** Offer endpoints for downloading output files, ensure the existence of output directories, and provide access to log file content.

ClusterNode Project

- **Publish ClusterNode-Specific Events:** Publish RabbitMQ-events of cluster-node-level activities (e.g., `Job Received`, `Job Update`, `Job Error` events).
- **Consume Job Start & Query Events:** React to queue messages instructing a node to start a new job or respond to status queries.
- **Docker Process Orchestration:** Start and monitor Docker processes as needed for running jobs.

Domain Project (Shared Tasks)

- **RabbitMQ Connections:** Provide a common utility or base class for establishing and managing connections with the messaging service.

Additional Repository Services (Persistence Project) Each aggregate root in the domain requires a dedicated repository to handle data manipulation tasks through Entity Framework. Examples include:

- **GroupRepository:** Manages the creation and retrieval of *Group* entities, including membership queries.
- **JobRepository:** Handles all operations related to *Job* entities, like saving new jobs, updating statuses, or retrieving job histories.
- **PlatformRepository:** Manages *Platform* entities, including lookups to ensure compatibility with tools.
- **ToolRepository:** Deals with all logic pertaining to *Tool* and *ToolVersion* entities.
- **ImageRepositoryService:** Fetches all available Docker image tags from an external image repository (e.g., Harbor or GitLab Image Repository).

By allocating each task with a respective project, the architecture clearly separates concerns across the *SimApp*, *ClusterNode*, and *Domain* layers, while the repository services unify data access in a centralized and maintainable manner.
Adhering to official Microsoft guides [32], services will be assigned the following service types:

- All repository services will be defined as scoped.
- SimApp services handling publishing of events, authentication handling, business logic and file management will be scoped.
- RabbitMQ event consumers of the SimApp project will be defined as hosted services, as they will need to run asynchronously parallel to all other SimApp processes.
- For the same reasons as with the event consumers, the SimApp job supervision will be defined as a hosted service as well.
- The Docker orchestration service of the ClusterNode project will be declared as scoped.
- Similar to the SimApp project, the RabbitMQ event consumers of the ClusterNode project will also be hosted services.

Using these service definitions, closely adhering to DDD principles, promises to achieve a strong separation of concerns, task separation using our previously-defined bounded contexts and clear definition of service lifetimes using advanced dependency injection features of .NET 8 [32].

5.4 Assembling the Final Architecture

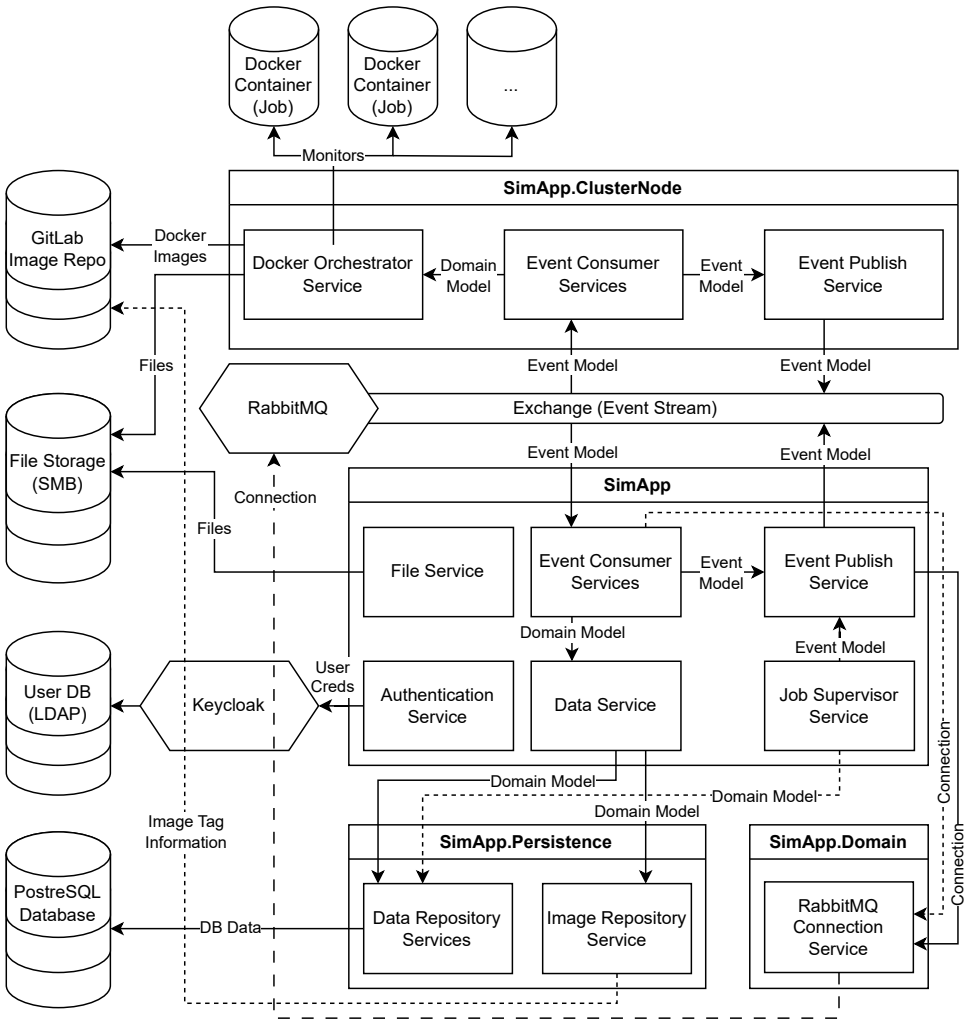

Figure 12: Architecture Diagram depicting the final platform architecture

Figure 12 illustrates how all the services, data layers, and external systems are connected to form the *final platform architecture*. The diagram shows how the different application components `SimApp.ClusterNode`, `SimApp.Domain`, `SimApp.Persistence` and `SimApp` interact with each other, as well as external services such as RabbitMQ, Keycloak, Docker image repositories, and file storage. Below is a walkthrough of the major components and how they collaborate:

SimApp and Its Core Services

- **File Service:** Responsible for managing file uploads, downloads, and storage. It integrates with the external SMB file share, ensuring that input files or simulation output directories exist and can be downloaded by the user.

- **Event Consumer Services:** Subscribes to RabbitMQ event streams, listening for updates or commands related to *Job* events, e.g., "*Job Update Event*", "*Job Error Event*" (5.1.1).
- **Event Publish Service:** Publishes new events, e.g., "*Job Created Event*", "*Job Finished Event*" (5.1.1) to RabbitMQ, keeping other components and nodes informed of changes in job status.
- **Job Supervisor Service:** Acts as the orchestrator within `SimApp`, controlling the lifecycle of each job by coordinating with the persistence layer and external resources like RabbitMQ. It supervises jobs, restarts them when they become stale and provision them to the computing cluster when they are created by the user.
- **Authentication Service:** Works with Keycloak to handle user identity and access tokens, ensuring only authorized users can modify or view job data.
- **Data Service:** A high-level service that executes domain logic, supporting the application layer with creating, reading, updating, and deleting (`CRUD`) operations on entities like `Job` or `Tool`. It centralized persistent data access between application and persistence layer.

SimApp.ClusterNode

- **Docker Orchestrator Service:** Manages the creation, monitoring, and teardown of Docker containers used to run simulation jobs. It pulls necessary images from the GitLab image repository (or a similar registry) and launches containers on-demand.
- **Event Consumer Services:** Subscribes to messages instructing the cluster node to start or stop jobs, or to respond with status updates.
- **Event Publish Service:** Sends back information in the form of events, e.g., "*Job Received Event*", "*Job Failed Event*" (5.1.1), to RabbitMQ to inform the SimApp of changes in job executions.

SimApp.Domain Houses the domain logic: interfaces, domain and event models defined in section 5.2.2 using to Domain-Driven Design (DDD). By centralizing domain rules here, both `SimApp` and `ClusterNode` can share a consistent language for exchanging information. This project also contains shared services needed in multiple other contexts, like for managing RabbitMQ connections.

SimApp.Persistence Provides repository services for each aggregate root like `Job` or `Tool` and an `ImageRepositoryService` for fetching image version information from the image repository. These repositories rely on Entity Framework (EF) to read and write data to the PostgreSQL database. Therefore, they also contain all of the entity configurations and migrations as described in .

External Services and Data Stores

- **PostgreSQL Database:** The primary data store for persistent data.
- **File Storage (SMB):** Hosts input, log, and output files.
- **Keycloak:** Manages user accounts and access tokens, providing an interface for `SimApp` to handle authentication and authorization.
- **GitLab Image Repository:** Stores Docker images (for various `Tools` and their versions) that `ClusterNode` pulls to run new jobs.
- **RabbitMQ:** In the center of the architecture, RabbitMQ's event exchange coordinates all event-based communication. `SimApp` and `ClusterNode` both publish and consume events, allowing them to stay loosely coupled while still working together.

Workflow Overview A typical workflow starts with a user authenticated via Keycloak accessing the user interface through the `SimApp`. The user initiates a new job, which gets stored in the database through the `DataService` and `SimApp.Persistence` project. The *Job Supervisor Service* then publishes a *Job Started Event* through RabbitMQ. The *Event Consumer Services* in `ClusterNode` receive this command, fetch the required Docker image from the GitLab image repository, and launch a container. As the simulation proceeds, `ClusterNode` publishes *Job Update Event* back to RabbitMQ, which gets received by one of the `Event Consumer Services` in the `SimApp` for status tracking and notifications. Once the job finishes, output files are stored in the SMB share, and any changes to the job's lifecycle (e.g., completion, errors) are also saved in the PostgreSQL database.

Benefits of This Architecture

- **Loose Coupling & Scalability:** Each major component communicates through asynchronous events, making it easier to add or remove cluster nodes based on workload demands.
- **Robust Security:** Keycloak ensures only authenticated and authorized requests reach the internal services, securing both the application layer and the cluster operations. RabbitMQ's authentication mechanics prohibit external interferance through the event stream.

- **Maintainable Codebase:** DDD principles, layered services, and separate repositories promote a clear separation of concerns, easing future changes or expansions.
- **Real-Time Visibility:** The event-driven model, combined with dedicated job updates, lets users and administrators monitor processes live as they happen.

Overall, this final architecture combines event sourcing and DDD principles through domain models separated into bounded contexts, event-based messaging, and container orchestration, which promises to be a flexible and fault-tolerant system for running complex metal 3D printing simulations.

6 Prototype Implementation

This chapter demonstrates the practical feasibility of the architectural design presented in section 5. Readers should pay special attention to four scientific contributions. First, the project structure shows how Domain-Driven Design concepts map to concrete C# projects, namespaces, repository classes, and even the file structure itself, providing an executable proof that the data-model separation can be enforced in real code. Second, the containerisation strategy realizes abstract service boundaries as reproducible Docker-based microservices, detailing the exact build process, volume mounts, and privilege flags that turn an event-sourced microservice into a self-contained process. Third, the CI/CD pipeline automates quality gates, security scans, and multi-stage image publication, showing how modern DevOps practices help with rapid iterations without sacrificing reliability or security. Fourth, a failed AmFem containerization attempt exposes a deliberate limitation of containerisation and, at the same time, showcases the flexibility of the orchestrator interface by swapping a Docker-based runner for a native Windows process without touching the surrounding services. In short, this chapter moves from diagrams to deployable binaries and should convince the reader that every design decision can be compiled, executed, and inspected on real hardware. The following sections guide the reader through environment preparation, code layout, image building, continuous-delivery automation, and multi-node deployment, ending with practical tips for debugging the components directly inside running containers, as they would be deployed in a productive environment. By the end of this section, readers should be able to reproduce the full setup, run sample simulations, and verify that key features, such as event-driven job provisioning, container orchestration, and real-time status monitoring, work as designed.

6.1 Development Environment

This subsection explains how to prepare a local workstation for building and running the prototype. Because the solution targets C# and .NET 8, as reasoned in section 4.5.3, Visual Studio 2022 is used as the primary IDE. The following steps describe how to install and configure the necessary tooling so that all projects (`SimApp`, `SimApp.ClusterNode`, `SimApp.Domain`, and `SimApp.Persistence`) compile and run without additional adjustments.

6.1.1 IDE Pre-configuration

Installing Visual Studio 2022 Begin by downloading the latest Visual Studio 2022 installer from Microsoft's official website. Launch the installer and choose the *ASP.NET and Web Development* workload, which bundles most of the web-centric tools required by the prototype.

Individual Components Even after selecting the workload, several components may not be ticked by default. Ensure that the following items are explicitly selected in the *Individual Components* tab before completing the installation. This list includes items not strictly necessary to run and debug the projects, but also components, which improve the debugging and development experience. In order to be completely transparent in which components were used during the development of the prototype, they are included:

- **.NET 8.0 Runtime** Provides the runtime environment that executes compiled applications.
- **.NET SDK** Supplies the compiler, CLI tools, and libraries needed to build .NET 8 projects.
- **Container Development Tools** Adds Docker support, enabling container debugging and image creation directly from Visual Studio.
- **NuGet Package Manager** Facilitates retrieval and management of third-party libraries, including RabbitMQ clients and Entity Framework extensions.
- **Class Designer** Offers a graphical view of class relationships—useful when modelling complex domain entities.
- **C# and Visual Basic Roslyn Compilers** Required for building source code that targets the latest language features in .NET 8.
- **.NET Debugging with WSL** Enables smooth debugging of Linux-based containers via Windows Subsystem for Linux.
- **.NET Profiler Tools** Helpful for diagnosing memory usage or performance bottlenecks in simulation jobs.
- **Just-In-Time Debugger** Allows immediate inspection of unhandled exceptions thrown by any running service.
- **ASP.NET and Web Development Prerequisites** Installs IIS Express, web templates, and related tooling used by the `SimApp` web front-end.
- **C# and Visual Basic** Core language support for editing, refactoring, and IntelliSense.
- **IntelliCode** Provides AI-assisted code completions that speed up everyday development tasks.
- **JavaScript and TypeScript Language Support** Ensures that client-side scripts inside Razor pages compile and lint correctly.

- **Razor Language Services** Powers Razor-page editing with syntax highlighting, diagnostics, and live-reload capabilities.
- **Entity Framework 6 Tools** Supplies design-time helpers for database migrations, reverse engineering, and model visualization.

6.1.2 Docker

Running and debugging the prototype relies on Docker containers, which package each microservice together with its runtime and dependencies. All services and tools run inside Docker containers, so Docker Desktop is required on development machines. Although Docker is available for macOS and most modern Linux distributions, all development and deployment activities for this thesis were performed on Windows 10 workstations with the WSL 2 backend, both my personal computer as well as workstations at the ISEMP (described in section 3). The ISEMP workstations have been set up with the following installation steps:

1. Enable hardware virtualization in the BIOS.
2. Turn on the "Hyper-V" and "Windows Subsystem for Linux" features, then reboot.
3. Install the latest Docker Desktop and select "WSL 2 backend" at first launch.
4. Add current user account to the `docker-users` group (log out/in once).
5. To test, open a terminal and run `docker run hello-world`. A success message confirms Docker is ready.

On Linux the standard Docker packages can be installed. To verify, the same `hello-world` check applies. With Docker in place, Visual Studio 2022 can now build, run, and debug each microservice inside containers that match the target production environment.

6.2 Depicting the Data Structure in Code

The logical architecture defined in section 5 must now be translated to a concrete codebase. Because each layer, bounded context, and data model has already been planned in detail, the implementation phase focuses on mapping those concepts to C# classes, interfaces, and projects. Using .NET 8 tools (e.g., project references, NuGet package management, and built-in Docker support) the transition from design to code can proceed in a highly structured way.

6.2.1 Establishing a Project Structure

To mirror the four-layer architecture (see figure 6), the solution is divided into four independent projects:

1. **SimApp (Presentation and Application Layer)** Hosts the ASP.NET web front-end, API controllers, and application-level services such as the *Job Service* and *Job Supervisor*.

2. **SimApp.ClusterNode (Execution Layer)** Contains the background worker that pulls Docker images, launches job containers, monitors their runtime, and publishes status events.

3. **SimApp.Domain (Domain Layer)** Holds all domain and event models, entities, value objects, aggregate roots, and the C# classes that represent the *Start Job*, *Job Started*, *Status Changed*, *Job Finished*, and *Job Failed* events. By concentrating the shared language here, both `SimApp` and `ClusterNode` share a single, consistent definition of business rules.

4. **SimApp.Persistence (Persistence Layer)** Implements Entity Framework Core entities, DbContext, and repository classes (*JobRepository*, *ToolRepository*, *PlatformRepository*, *GroupRepository*, and the *ImageRepositoryService*). This project is the only layer that knows about PostgreSQL schema details or EF-specific annotations.

Project References Dependencies follow the direction of the layered architecture:

- `SimApp` and `SimApp.ClusterNode` both reference `SimApp.Domain`.
- `SimApp` additionally references `SimApp.Persistence` to perform CRUD operations through repositories.
- `SimApp.Persistence` references `SimApp.Domain` so its entities can map to domain models without code duplication.

These references are added via Visual Studio's *Add → Project Reference...* dialog. MSBuild then compiles each project in the correct order and resolves NuGet dependencies automatically, preventing version conflicts or circular references.
With this structure in place, each layer can evolve independently: updates to domain logic occur in one project, database migrations reside in another, and the web UI or background workers can be modified without risking unintended side effects across the solution.

6.2.2 Application Configuration

A well-defined configuration system is critical for a distributed platform. Both the `SimApp` web service and each `ClusterNode` instance rely on typed configuration models: simple C# `class` or `record` types whose properties mirror the keys found in configuration files or environment variables. When the application starts, .NET's built-in configuration pipeline binds JSON, environment variables, and user-secret values to these models, giving strongly-typed access to settings throughout the codebase.

Configuration Sources

- **`appsettings.json`:** Holds default values that are safe to check into source control: connection strings for the database, default RabbitMQ exchange names, or the Keycloak realm.
- **`appsettings.Development.json`:** Loaded only when the `ASPNETCORE_ENVIRONMENT` variable equals `Development`. It contains overrides useful for local debugging, e.g. mapping the external Samba shares to Windows paths or enabling verbose logging. Any key present here replaces the value in `appsettings.json`.
- **User Secrets (`secrets.json`):** Visual Studio's *Manage User Secrets* feature stores per-developer secrets outside the repository (e.g. Keycloak admin passwords, RabbitMQ credentials). The secrets file lives in the user profile and is automatically encrypted on-disk.
- **Environment Variables:** When the prototype is containerised, environment variables supplied in the `docker-compose.yml` override the content in the JSON files. This technique is ideal for injecting production-only values, such as a database URL or the address of a highly available RabbitMQ cluster, without rebuilding images.

Because `SimApp` and `ClusterNode` perform different roles, each project defines its own configuration records. These records only contain the keys the respective project needs and are registered in the DI container via the `builder.Configuration.GetSection()` pattern, then injected wherever required (e.g. into hosted background services or controller constructors).

Benefits

- **Strong typing & validation:** Configuration errors surface at startup because .NET can validate required properties and data annotations on the model classes.
- **One code base, many environments:** The same binaries run unchanged on a laptop or in the cluster; only JSON files or environment variables differ.

- **Security:** Secrets stay out of Git history and Docker images, reducing accidental leaks.
- **Docker-friendly overrides:** Operations teams can adjust scaling limits, credentials, or hostnames simply by editing the Compose file used in deployment.

By standardising on configuration models and layered sources, the prototype gains flexibility in local development, CI testing, and eventual production roll-outs, without scattering configuration strings or hard-coded paths throughout the codebase.

6.3 Containerizing Simulation Algorithms

To execute a job reliably on any node in the cluster, every simulation tool is packaged as a self-contained Docker image. Each image bundles the tool's binaries, its runtime dependencies, and a tiny entry-point wrapper written in C# (.NET 8). As a proof-of-concept, three testing tools optimized for execution inside linux-based Docker containers have been written, which showcase the different tool functions available:

- **TestTool** simply receives a text input from the user and prints it to its job log.
- **OutputfileTool** also receives a text input from the user, but also creates an output file containing the user-defined text, which will need to be downloadable through the SimApp UI.
- **RandomFailer** is a tool used for performance tests. It simulates a long-running job with a non-deterministic result, receiving two numbers as its input: Maximum runtime in seconds (the final run time will be generated based on this number as the maximum) and the final failure chance of the tool as a percentage (in this case, the tool will simulate a failure).

For simulation jobs requiring the AmFem binaries, additional steps had to be made and a slightly different solution has been found. This deviation from the original container-based approach had to be done because AmFem was written in such a way, that made it container-incompatible. To make AmFem work like the proof-of-concept tools listed above, a number of adjustments need to be made to its code, which is out of scope for this project and already in the works by researchers at the ISEMP. The problems encountered regarding AmFem expose a weakness in the architecture concept and simultaneously highlight its versatility. Due to the modularity of the underlying architecture, in the final prototype, AmFem jobs can be provisioned and monitored using the SimApp in the same way container-based tools would be. The specific adjustments needed to achieve this will be described in more detailed in its respective subsection.

6.3.1 Container-Based Simulation Tools

Aligning with the underlying architecture concept, the three tools used to evaluate the prototype have been developed based on Docker images. When the `ClusterNode` service receives a *Start Job* event, it pulls the requested image (if not cached), mounts the standard input, log, and output volumes, and launches the container with the appropriate command-line arguments.
A separate `SimTools.sln` solution holds:

- one project per tool (e.g. `SimTool.TestTool`, `SimTool.RandomFailer`);
- a `Dockerfile` for each project that compiles and copies the compiled DLLs into a minimal `mcr.microsoft.com/dotnet/aspnet:8.0` base image;
- shared utility code for different tool functions required for every tool.

Input Handling At runtime, the container reads all input parameters from its command line (forwarded by `SimApp.ClusterNode`). Any input parameters containing path are mounted as Docker CIFS volumes by the cluster node immediately before container launch. The algorithm inside the container then runs the computation, writes logs into an automatically configured logfile and exits with an appropriate status code.

Log and Output Conventions Inside every tool image, two well-known paths are reserved:

- `/logs`: bound to the host's configurable *log* share. Each simulation writes a rolling text log here, allowing real-time tailing by the SimApp dashboard.
- `/output`: bound to the host's *output* share. If a tool produces a result file, it will be written here such that the user can download it through the dashboard.

Because the mount points are automatically configured and stored in the database for every job, the SimApp always knows where to fetch live logs and finished artefacts. No tool-specific paths need to be hard-coded in the UI. Using this dynamic parameter system, it's possible to start jobs with theoretically any parameters or configuration. Parameter type, default value, or description can be configured through the SimApp administrator area. Additionally, parameters can be optional/required or hidden. The latter prohibits the user from changing its value before provisioning a new job.
Alongside a virtual platform for execution of AmFem Tools, several test tools were implemented which will assist in torture testing. All images are given a preconfigured name and pushed to the same registry. This way the prototype can be validated without running proprietary code.
Developers add new tool projects to `SimTools.sln`, add a Dockerfile that follows the template, and push the image to the registry. The cluster can then run this image with no additional configuration.

6.3.2 AmFem-Based Simulation Tools

As mentioned in the introduction to this section, AmFem contains calls to functions requiring a desktop environment, so it cannot run inside a Docker container. The original `ClusterNode` project, by contrast, was written with container workloads in mind. Its main responsibility is handled by the `DockerOrchestrator` class, which starts a Docker container for each job, passes the required mount points and environment variables, and then regularly polls the container status until the job finishes. Similar to the rest of the application, to avoid tight coupling, `DockerOrchestrator` implements a small `IOrchestrator` interface that exposes just two methods, `StartAsync`, and `GetUpdateAsync`. Any class that fulfils this contract can be dropped in without changing the rest of the code base.
To support starting AmFem-based jobs, a new project named `AmFemNode` was created and configured to reference the `ClusterNode` project, which enables it to inherit all of its classes. This way, all shared helpers, background services, and event-handling code remain identical, only the `DockerOrchestrator` class is replaced. The new `AmFemOrchestrator` that takes its place still implements `IOrchestrator`, but instead of calling the Docker Engine, for every new job, it spawns a native `AmFem_cmake.exe` process, captures its standard output, and publishes periodic status events back to RabbitMQ. A small watchdog task checks every few seconds whether the process is still alive or has exited unexpectedly and raises a *Job Failed* event if needed.
Due to AmFem's restrictions regarding containerization `AmFemNode` itself cannot be containerised. Consequently, several benefits of working with containers (see 4.4.2) disappear. For example, the node must rely on the host file system rather than an isolated volume, it cannot pin a known image hash for reproducibility, and updates require a traditional installer instead of simplified commands like `docker pull`. Even so, the exercise shows that the overall architecture is flexible. Because orchestration duties are hidden behind `IOrchestrator`, swapping Docker for a native process controller required only a single class to change, while message formats, event logic, and monitoring dashboards stayed exactly the same. This capability illustrates that the platform can adapt to legacy tools or proprietary binaries while still enjoying most of the advantages of an event-driven, microservices design.

6.4 Continuous Integration / Continuous Deployment

All source code for the prototype is hosted in the GitLab instance of the University of Bremen's computer-science department at `https://gitlab.informatik.uni-bremen.de/alpay1/masterarbeit`. As described in section 4.4.3, for automated builds, tests, and image publication, a complete CI/CD pipeline is defined in the `.gitlab-ci.yml` file in the projects root directory. GitLab detects this file and runs the pipeline every time a commit is being pushed to the origin server.

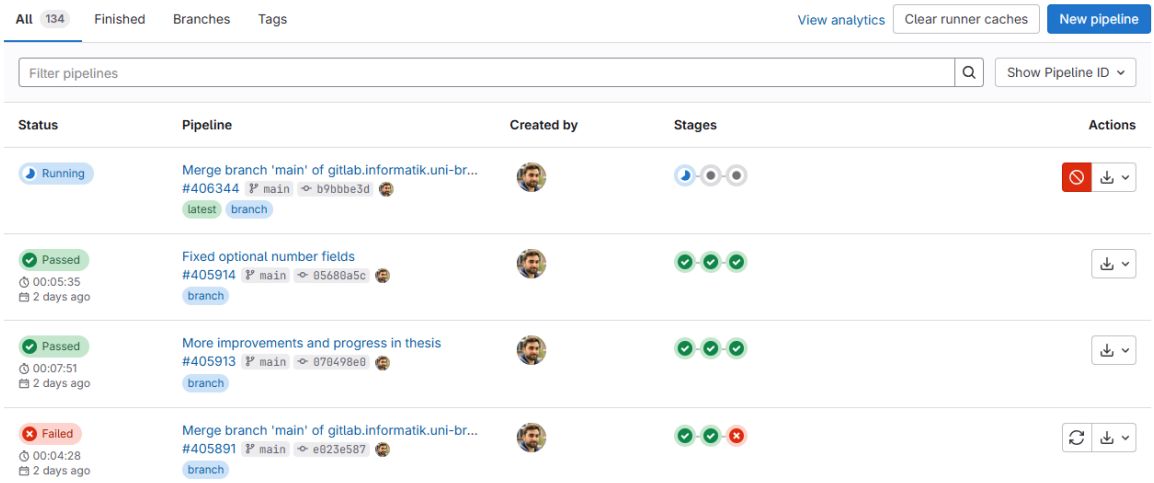

Figure 13: Different pipeline iterations as seen from the GitLab UI

Figure 13 shows the different pipeline iterations automatically created by GitLab with each repository commit. The topmost pipeline still running, each pipeline iteration contains different stages and jobs. Once all jobs in a pipeline stage are completed, the pipeline stage is displayed with a green checkmark. Pipeline stages can be cancelled, skipped or artifacts can be downloaded for completed pipeline jobs. Once a pipeline fails, it's being displayed in red with an X-mark, indicating its failure. Additionally, the user can choose to receive E-Mails when pipelines fail and subsequent jobs depending on that pipeline job or stage are interrupted. This stops the pipeline from accidentally publishing images or performing actions based on non-functioning code.

SonarQube integration: Code quality and basic security scanning are handled by a self-hosted SonarQube server at `https://sonarqube.home.alpayy.de:3455/`. First introduced in section 4.4.8, SonarQube is an open-source platform for static analysis that detects code smells, bugs, and vulnerabilities. By integrating SonarQube into the GitLab pipeline, every commit is analysed automatically, providing developers with an immediate quality gate.

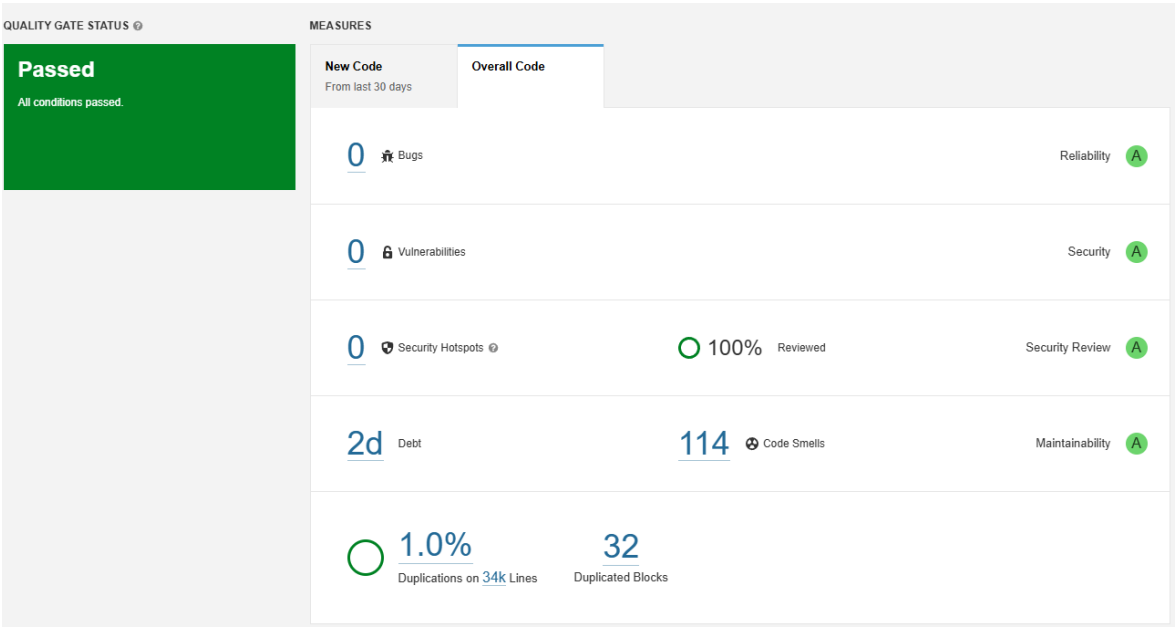

Figure 14: SonarQube analysis results

As seen in figure 14, the code in this thesis has an A rating (which is the highest rating) in all four measured security metrics: Reliability (0 Bugs), Security (0 Vulnerabilities), Security Review (0 Security Hotspots, 100% Reviewed), and Maintainability (2d Debt, 114 Code Smells).

Pipeline stages: The pipeline is divided into three stages: *Build*, *Test*, and *Publish*.

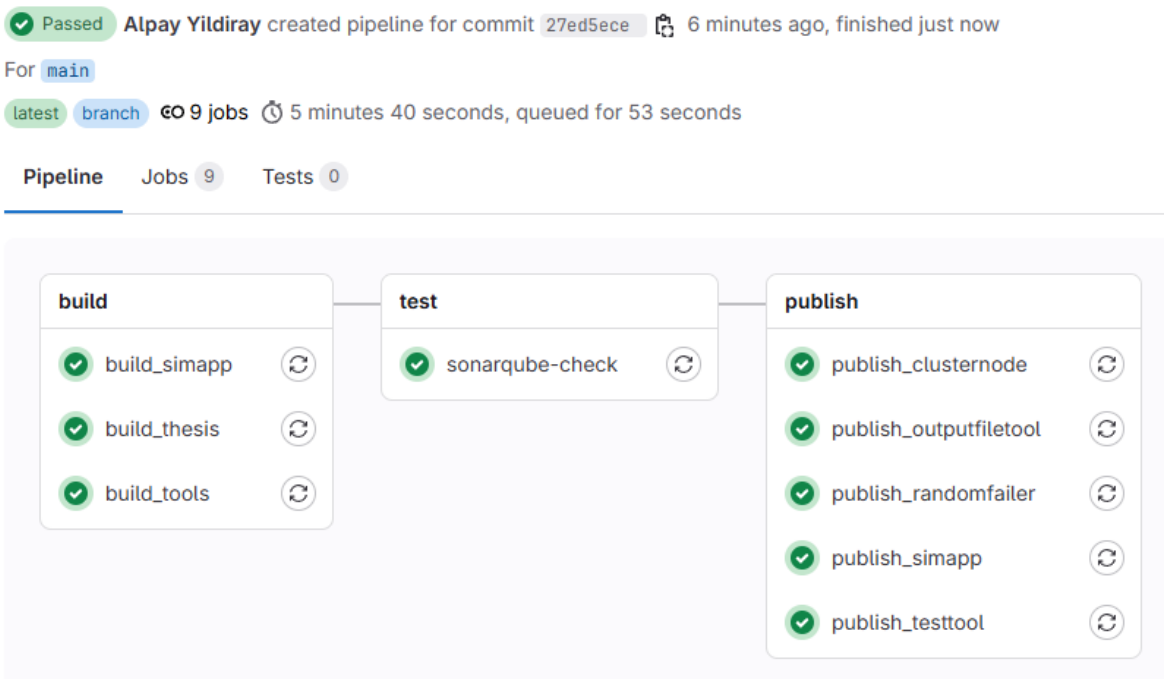

Figure 15: Jobs and stages of a pipeline iteration as seen from the GitLab UI

Figure 15 shows all jobs and stages of a specific pipeline iteration. In this case, all jobs and

stages completed successfully. The pipeline stages separate jobs into categories:

- *Build:* Three jobs run in parallel. The first compiles the complete *SimApp* solution (application, cluster node, domain, and persistence projects). The second builds the *SimTools* solution that contains the containerised simulation tools. The third compiles the LaTeX sources of this thesis to flag broken references or package problems early. If any job fails, the pipeline stops, protecting the main branch from non-compiling code.
- *Test:* A single job uploads the latest coverage and quality report to SonarQube. If SonarQube detects issues above a defined severity threshold, the job surfaces a warning in GitLab and can even block the *Publish* stage until the problems are resolved. Running this stage on every push keeps the SonarQube dashboard synchronised without extra developer effort.
- *Publish:* For each microservice or tool there is an independent image-build job. The *SimApp*, *ClusterNode*, and every tool project are converted into version-tagged Docker images and pushed to the GitLab container registry that belongs to the `alpay1/masterarbeit` git project. Production deployments could later move each tool into its own repository and pipeline, but a monorepo is sufficient for the proof-of-concept.

GitLab runner: While the university offers shared runners, their Docker support is limited. Therefore, to run the pipeline steps, I installed a GitLab runner instance on my private server, registered it to the project, and configured it with the `docker` executor.

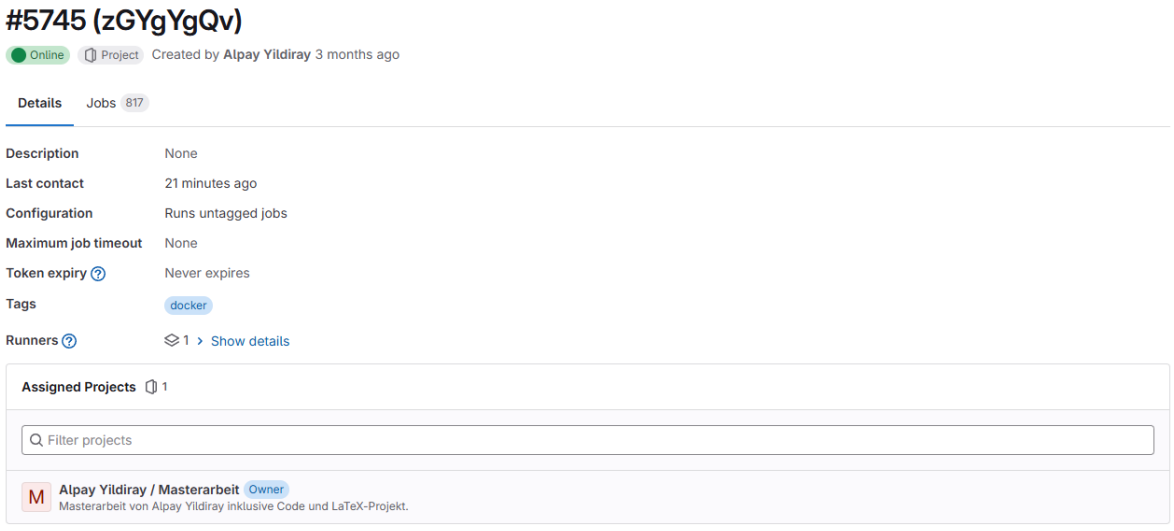

Figure 16: The GitLab runner is being displayed in the GitLab UI after successful registration

Figure 16 shows the GitLab runner after its registration and assignment to the repository of this masters thesis in the GitLab UI. This runner can build images, run unit tests in

containers, and perform SonarQube scans without restriction. From here on out, this GitLab runner will compile, test and deploy each component as a microservice for every commit made.

Image registry access: GitLab's built-in registry is enabled for the project. A project access token with pull privileges is stored in the `secrets.json` file and passed as an environment variable in the deployment stack. Both *SimApp* and *ClusterNode* use this token to query image tags or pull tool images at runtime.

Through this pipeline, every commit triggers a full chain of compilation, quality analysis, and container publication. The result is a consistently reproducible build that can be deployed to any Docker-capable environment with minimal manual intervention, fulfilling the CI/CD goals set in section 4.4.3.

6.5 Prototype Deployment

The ISEMP made four workstations available for testing the prototype. Each machine has different CPU, GPU, and RAM configurations, allowing performance to be evaluated under realistic mixed-hardware conditions. Three Samba shares, *input*, *log*, and *output*, are available to every host, providing shared storage for job data, like input and output files.

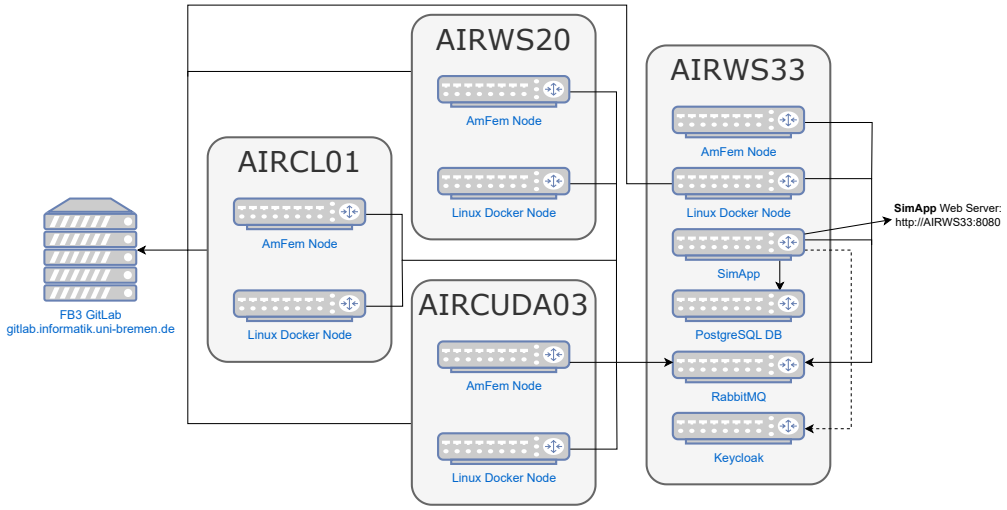

Figure 17: Distributed cluster setup used for testing

As illustrated using figure 17, one workstation will be dedicated for running the core services and the *SimApp* web front-end. Keycloak, RabbitMQ, and the PostgreSQL database all start on this node within a common Docker network, alongside the SimApp container itself. The remaining three machines exclusively form the computing cluster. Each hosts one instance of the *ClusterNode* and *AmFemNode* service respectively, the former of which pulls Linux-based Docker images for compatible tools from the registry, launches job

containers, and streams status events back to SimApp, while the latter can run AmFem jobs requiring native code execution under a Windows desktop environment. The only external dependency is the University of Bremen GitLab, which hosts the Docker container image registry, where the base images for Docker-based simulations are stored.
Although the platform could be scaled out for high availability (by running redundant Keycloak, RabbitMQ or PostegreSQL instances, as well as adding load balancers) such measures are unnecessary for the proof-of-concept. In a production setup, communication would span both a physical LAN (between hosts) and internal Docker bridge networks (between containers on the same host). For development, however, every component is deployed on a single computer using a virtual Docker network, simplifying development while still virtually reflecting the final network topology in a way.

6.5.1 Setup of External Services through `Docker Compose`

Running the prototype requires three core infrastructure services: Keycloak for authentication, PostgreSQL for data persistence, and RabbitMQ for event streaming. To ensure every developer works with the same configuration, these services are provisioned with `Docker Compose`. Compose acts as a lightweight "infrastructure-as-code" (IaC) tool (first mentioned in section 2.3, some more advanced cloud-based solutions including IaC definitions are described in [49]), defining container images, environment variables, volumes, and shared networks in a single YAML file. Launching the stack on any workstation therefore becomes a one-line command, eliminating the "works-on-my-machine" problem.

Compose Principles

- **Declarative configuration**: each container's port mappings, credentials, and volumes are codified, making changes auditable and version-controlled.
- **Single virtual network**: all services join a private Docker bridge network, so they can discover each other by name (e.g., `keycloak:8080`).
- **Reproducibility**: developers can set up an identical environment with `docker compose up -d`, regardless of host OS.

Service Overview

Keycloak Provides OpenID Connect or JWT tokens to the SimApp. A default realm, client, and user are pre-seeded via an import volume so that no additional setup is needed.

PostgreSQL Stores all persistent data. Data is being stored in a folder made available to the container through a Docker volume for durability across container restarts.

RabbitMQ Supplies the event-stream backbone. The Compose file exposes the management UI on port `15672` for easy queue inspection.

Samba (debug only) A lightweight `dperson/samba` image exports three CIFS shares (*input*, *log*, and *output*). During local development these act as stand-ins for production SMB shares, letting services exchange files without touching the host file-system. In production, actual SMB shares at the ISEMP will be used for storing files. This allows administrators to dynamically manage access permissions and provids a mature file management solution.

Launching the Stack

1. Ensure Docker Desktop is running and shows the green "Engine running" indicator in the desktop UI.
2. Open a terminal in `code/Services/` and execute:

   ```
   docker compose up -d
   ```

3. Verify that all four containers are healthy in Docker Desktop's dashboard. Since they have been deployed through a single Docker Compose file, they will be grouped visually for easier identification.

Opening the Solution

1. In Visual Studio 2022 choose *File → Open → Project/Solution* and select `SimApp.sln` from the `code/SimApp` directory.
2. Right-click the solution and choose *Rebuild*. A successful build confirms all NuGet dependencies were restored correctly.
3. From the run-configuration dropdown pick *SimApp* and click on *Run Container (Dockerfile)*. Visual Studio will build the project image, start it in a container, and open a browser to `http://localhost:8080` (or the mapped port). Verify that the application didn't throw any errors. Configuration parameters should be preconfigured, but in case of errors they should be re-checked using the `appsettings.json` file.

First Log-In A default developer account is pre-configured in Keycloak:

- **Username**: `simapp`
- **Password**: `simapp`

Logging in with these credentials should display the SimApp dashboard and an empty job list, indicating that Keycloak, RabbitMQ, and PostgreSQL are all reachable from the container.
This workflow confirms that every essential runtime, SDK, and Docker integration is correctly set up, allowing you to proceed with feature development or prototype testing on any Windows 10 machine (and, in principle, any OS that supports Docker Desktop).

6.5.2 Notes on Debugging Components in Containers

When developing in Visual Studio 2022, the prototype is launched under the *Docker* debug profile, meaning every microservice (including the web front-end and any cluster nodes) runs inside its own container, even while you are stepping through breakpoints. This section described intricate configuration choices made when setting up the debug environment. The debugging setup has not been trivial, requirements like internal network storage mounts (needed to access files), docker socket access (needed to start containers on the host system) or internal access to the required services through the virtual Docker network (e.g., Keycloak, RabbitMQ, PostegreSQL) have resulted in quite a few challenges, which have required creative solutions.
To ensure that the application code and services can see one another exactly as they will in production, all debug containers must attach to the same user-defined network created by the `docker-compose.yml` file (named `simapp`).

General Rules

- Use the `--network=simapp` argument for every service you start from Visual Studio.
- Run the container as `root` so it can mount SMB shares or create temporary folders inside job containers.
- Add the Linux capability `SYS_ADMIN` (and, for the cluster node, `DAC_READ_SEARCH`) to permit file-system operations that are normally restricted inside an ordinary container.

Debug Arguments for `SimApp` In the project's *Docker Debug Properties* window, the additional run arguments have been set:

```
--network=simapp --cap-add SYS_ADMIN --privileged -u root
```

Why each flag is needed

- `--network=simapp` - joins the container to the shared network so it can reach Keycloak, PostgreSQL, RabbitMQ, and the Samba shares by container name.

- `--cap-add SYS_ADMIN` - grants the ability to mount CIFS shares from the Samba debugging container.
- `--privileged` - required because some low-level share operations otherwise fail under the standard container security profile.
- `-u root` - forces the container to run as `root`, avoiding permission errors on mounted volumes.

Debug Arguments for `ClusterNode` The cluster-node service needs a slightly extended set of privileges, because it must be able to start additional job containers and create temporary docker volume mounts for them. The following run arguments have been added:

```
-v //var/run/docker.sock:/var/run/docker.sock --network=simapp --cap-add SYS_ADMIN --cap-add
    DAC_READ_SEARCH -u root
```

Additional considerations

- The `docker.sock` bind-mount (`-v ...`) exposes the host's Docker daemon to the container, allowing the cluster node to start and stop job containers exactly as it will do in production.
- `--cap-add DAC_READ_SEARCH` extends directory-access capabilities so the node can read files owned by other UID/GID combinations (useful when copying results out of job containers).

Outcome With these debug flags in place, pressing `F5` in Visual Studio launches the `SimApp` or `ClusterNode` code inside containers whose network, privileges, and mounted volumes mirror the real production topology. This setup lets you:

1. Step through C# logic in Visual Studio while RabbitMQ and PostgreSQL run in parallel in sibling containers.
2. Verify that file transfers to the Samba shares succeed under realistic permission constraints.
3. Confirm that `ClusterNode` can create and manage nested job containers (via the hosts Docker socket) without leaving the IDE.

In short, debugging inside Docker ensures "what you see in Visual Studio" precisely reflects the behaviour you will deploy to staging or production, reducing issues caused by variations in environments. Additionally, a shared virtual Docker network, precisely specified container privileges and configuration minimize any effort needed to be made before deployment of the application.

7 Tests and Evaluation

This section examines whether the prototype meets the requirements defined in Section 3.4. Two complementary approaches are used. First, a series of manual tests trace data flow through the user interface while impersonating each of the three actor roles, user, maintainer, and administrator. These walkthroughs confirm that administration functions, role-based permissions, job provisioning, status updates, and file downloads all behave as specified. Second, automated performance tests stress the cluster to measure how well the system scales and how it responds to high computational load. These tests launch batches of jobs with varying runtimes and parameter sets, record overall throughput and node utilization, and verify that event handling and database updates remain responsive even at peak load. Together, manual UI exploration and controlled load generation promise to provide a base for a reliable evaluation of functionality, resilience, and performance.
A third evaluation path often used in software research is a structured user study, where representative participants work through predefined tasks while researchers record ease of use, satisfaction scores, or error rates. For this project such a study would contribute little insight. First, the prototype is a specialist engineering tool designed for researchers and production engineers who already understand metal-printing workflows and are primarily interested in computational accuracy and throughput, not visual polish. Second, the user interface is deliberately minimal. It exposes only the controls needed to upload input files, set parameters, and monitor progress, because the thesis focuses on back-end architecture rather than human-computer-interaction patterns. Third, evaluating interface aesthetics or click counts does not help determine whether the system meets its key goals of high availability, horizontal scalability, and correct orchestration of non-deterministic simulations. Those objectives are best measured by observing event flows, container lifecycles, and database consistency, which can be logged and verified objectively without human preference data. Finally, organising a valid user study would require recruiting domain experts, preparing training materials, anonymising data, and securing ethics approval. All effort that would divert time from deeper technical testing while offering limited academic payoff. For these reasons the evaluation concentrates on integration checks and automated performance benchmarks, which directly probe the architectural qualities that form the scientific core of the thesis.

7.1 Integration Tests

Integration tests focus on verifying that all services work together as a coherent whole. There are different approaches to integration testing, e.g. testing procedures using in-memory databases in Entity Framework [23]. Testing is carried out through the web interface while assuming each of the three defined roles: user, maintainer, and administrator. For every role, the test walks through a typical workflow, such as uploading input

data, starting a simulation, inspecting its status, and downloading results. During each step, the Visual Studio 2022 debugger, the RabbitMQ management UI and process logs emitted by all microservices (e.g., `SimApp`, `ClusterNode`) will be used to confirm that the internal data flows and published and consumed events correctly.
Maintainer tests add tasks like registering a new tool version, updating parameter configurations, and assigning users to user groups. These actions are cross-checked in the PostgreSQL tables through database access tools (in this case JetBrains DataGrip was used) to make sure repository calls store the intended state without breaking referential integrity. Maintainer scenarios include creating user accounts, assigning user permissions, and platform maintenance functions like updating the individual microservices. System logs are inspected to verify that authentication requests reach Keycloak and role changes propagate to JWT claims. The actual process logs, debugging results and outputs will not be included in this thesis, as the process is easily repeatable using the application code and the elaborate instructions of section 6. Also, there would not be any additional value in including specific debugging information beyond what is already being vizualized and desribed through the following diagrams and their respective explanations.
Regarding the testing methodology, by combining the browser front end with service-specific consoles and log viewers, the integration tests trace the full data path from UI action to database row and back. Doing this in a containerized environment matching the future production environment gives confidence that the prototype operates correctly and exactly the same way under productional usage conditions as during this testing.

7.1.1 Testing Workflow Requirements

The final step in the integration test suite is to verify that every requirement listed in Section 3.4 is fully supported by the running prototype. Test cases are executed from the viewpoint of each actor type (user, administrator, and maintainer) to confirm that the system behaves correctly under real-world scenarios. For every role we establish a checklist that maps individual actions, such as launching a job or creating a new tool version, to the corresponding requirement, for example multi-user access or input-parameter management. We then walk through the workflow and observe database updates, event messages, log entries, and screen outputs to ensure the expected outcome occurs without errors or security violations. If a step fails it is logged with precise context so that the root cause can be traced and fixed. By systematically covering all roles and features, this sub-section demonstrates that the prototype meets all goals defined before the start of the project.

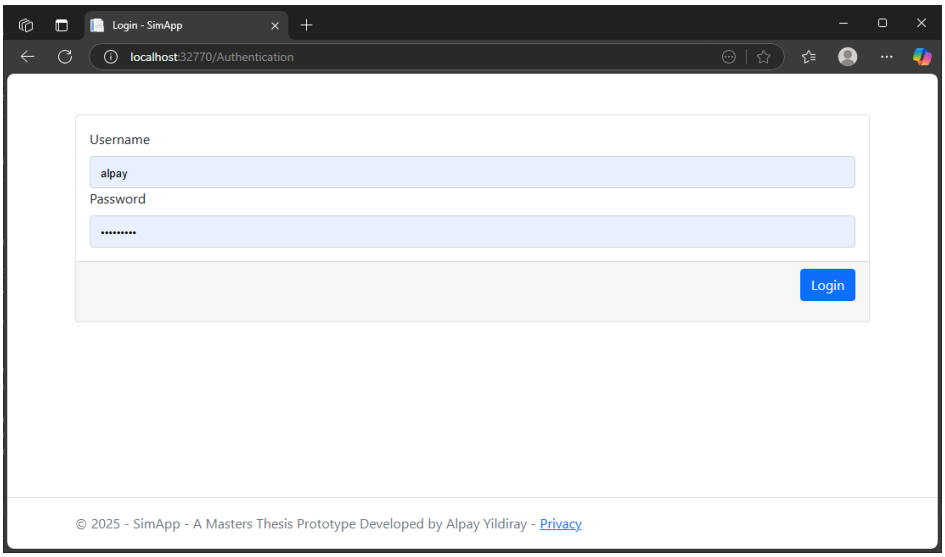

Figure 18: Browser window showing user login

User - Login Figure 18 shows the login page presented by the SimApp front-end. The user is prompted for a username and password, both of which are forwarded to Keycloak to obtain an OpenID Connect token. A successful login issues a signed JWT that is stored in the browser session and attached to every subsequent API request. The minimal layout keeps distractions low while making the authentication flow clear and quick, which is important for repetitive sign-in cycles during testing.

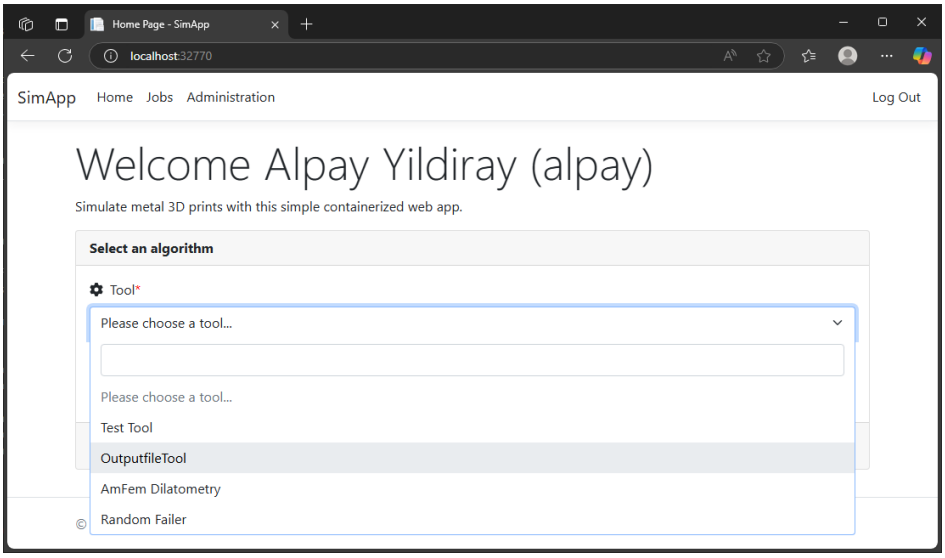

Figure 19: Browser window showing user selecting a simulation tool

User - Starting Jobs After a successful login the user is redirected to the home page, shown in figure 19. A personalised greeting confirms that the JWT token has been validated and decoded. The first step is to choose a simulation tool from the drop-down menu. Only tools that the current account is authorised to run are displayed, demonstrating role-based

filtering on the back end. Once a tool is selected, the user clicks *Next* and is taken to the parameter form.

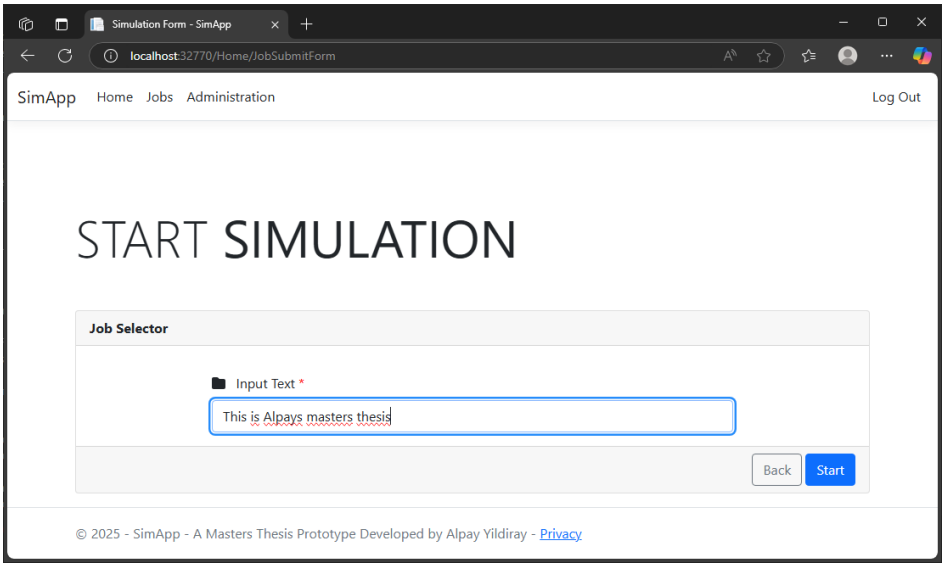

Figure 20: Browser window showing user filling out input fields

Figure 20 illustrates the input form for the chosen tool. All required parameters appear as strongly typed fields, and client-side validation highlights missing or invalid entries in real time. When the form is submitted the browser sends a `POST` request to the SimApp API. SimApp first writes a new `Job` entity to the database through the repository layer, then publishes a *Start Job* event to RabbitMQ. Almost instantly, an acknowledgement message returns, allowing the UI to redirect the user to the job-status page while the computation starts in the background.

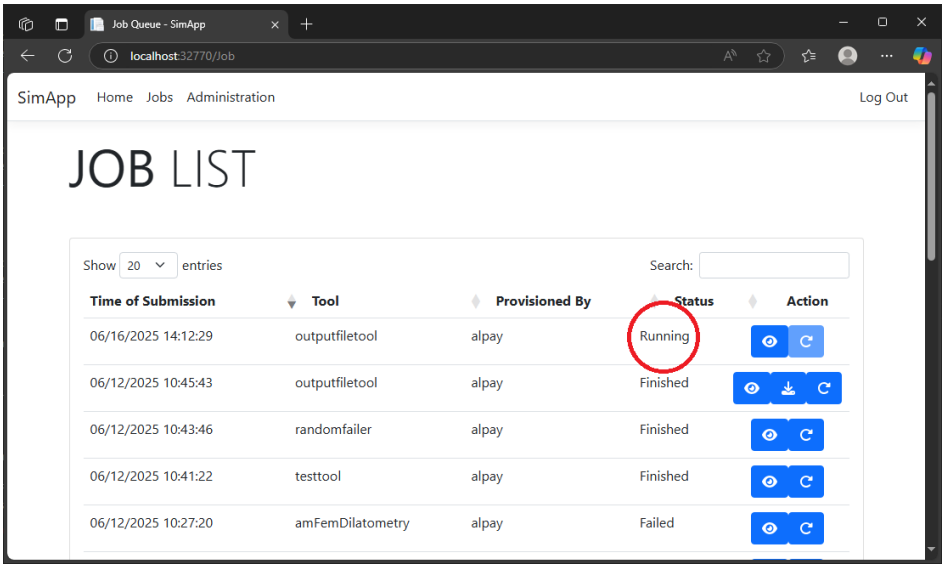

Figure 21: Browser window showing job list with the topmost job still running

User - Job Monitoring After submission the user is redirected to the job-list page. Figure 21 displays this table while the most recently created job switched to the *Running* state. The row updates in near-real time because the browser receives periodic status events pushed by the back end. Action buttons let the user inspect live logs, retry a previously-run job, or download finished output files once they are available.

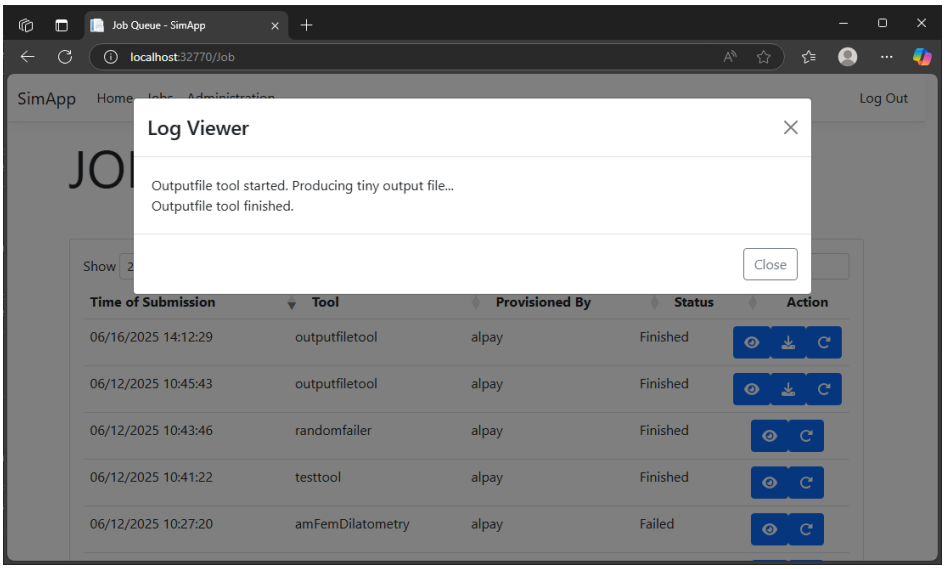

Figure 22: Browser window showing user monitoring job logs

Clicking the log-viewer icon opens the modal shown in figure 22. Here the latest lines of the container's standard output are streamed directly from the shared `/logs` volume. This confirms that the cluster node writes log data to the expected location and that SimApp can read and relay it without blocking the main UI thread.

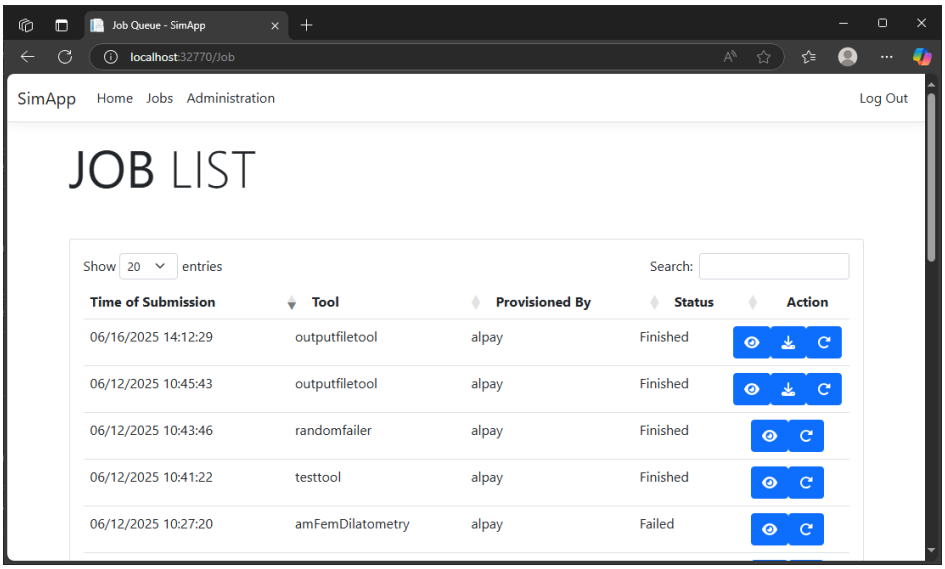

Figure 23: Browser window showing job list with all jobs finished

When the simulation completes the cluster node publishes a *Job Finished* event, SimApp

updates the row, and the status label switches to *Finished*. Figure 23 shows the updated list after all running jobs have ended successfully. The download button is now enabled, allowing the user to retrieve result files stored in the `/output` share.
These screens demonstrate a full user workflow: tool selection, parameter entry, live status monitoring, log inspection, and retrieval of finished results. Each step exercises a different architectural path: database writes, event streaming, log mounting, and file transfer, verifying that the front end, back-end services, and cluster nodes operate together as intended.

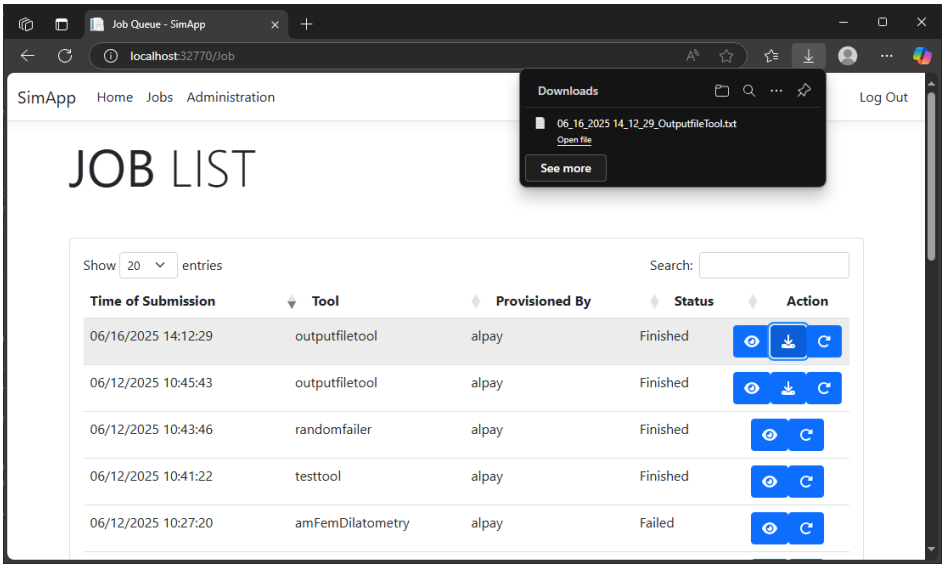

Figure 24: Browser window showing user downloading finished job artifacts

User - Downloading Artifacts Figure 24 demonstrates the final step in the standard user workflow, downloading the result files produced by a completed simulation. Once the job status changes to *Finished* and the spoecific job run produced an output file, the download icon in the *Action* column becomes active. Clicking this button triggers an API request that streams a ZIP archive, or a single file if only one artefact exists, directly from the jobs internal output directory. The browser's built-in download panel then confirms that the file has been transferred successfully and is ready for inspection on the user's local machine. This step verifies that output files are written to the expected location by the cluster node, that SimApp can resolve the correct path in the SMB share, and that role-based permissions allow the job owner (but no one else) to retrieve the artefacts. Together with the previous screens, this completes a full end-to-end test of job submission, live monitoring, and results retrieval.

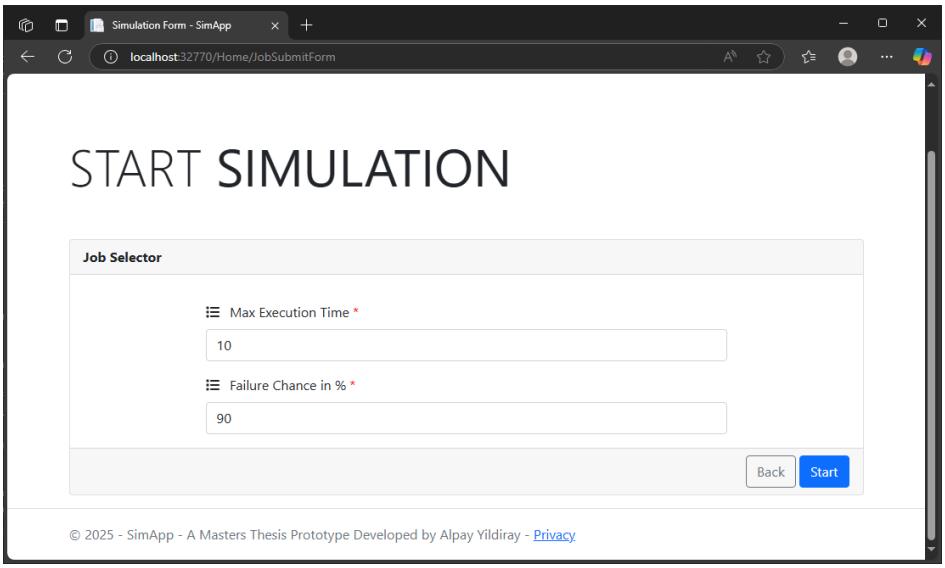

Figure 25: Browser window showing user starting a test job with high failure chance

User - Job Failure Figure 25 presents the parameter form for the `randomfailer` test tool. Here the user sets a maximum run time of ten seconds and a ninety-percent failure probability, deliberately testing the platform's error-handling. When the job is submitted the cluster node launches the container, then randomly exits with an error based on the supplied probability.

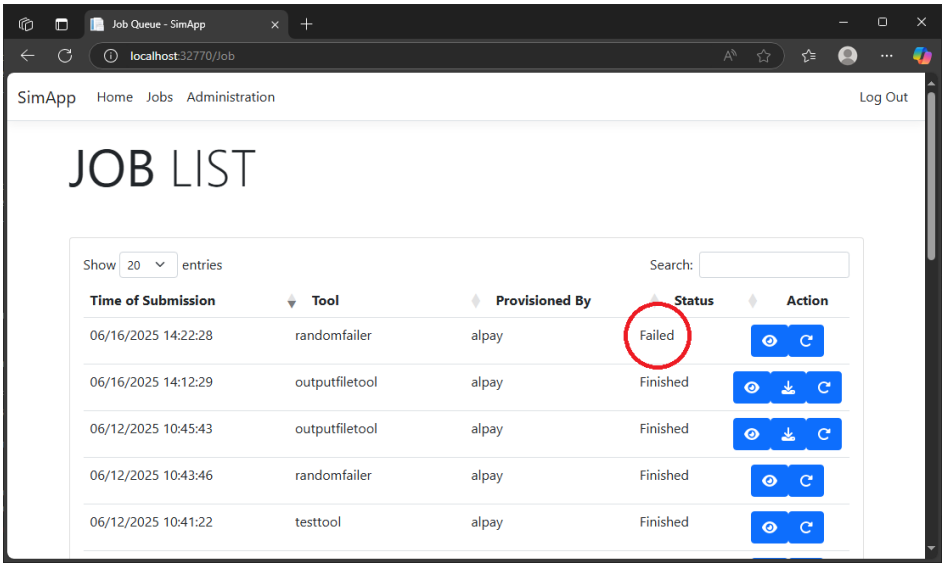

Figure 26: Browser window showing a failed job in the job list

As soon as the container terminates with a non-zero exit code the node publishes a *Job Failed* event. SimApp receives the event, marks the database record accordingly, and pushes an update to the front-end. The result appears in the job table shown in figure 26, where the status column switches to *Failed*. From this page the user can still open the live log or retry the job with a single click, demonstrating graceful degradation and quick recovery even when simulations end unexpectedly.

User - Node Failure During Job Execution This scenario demonstrates that high-availability logic is handled at the orchestration layer rather than inside individual tools. If multiple nodes are available, upon failuire, an automatic retry policy will reschedule the task on a healthy node. However, in order to test the behavior of the SimApp when no alternative node is available, for this test, only a single node has been deployed.

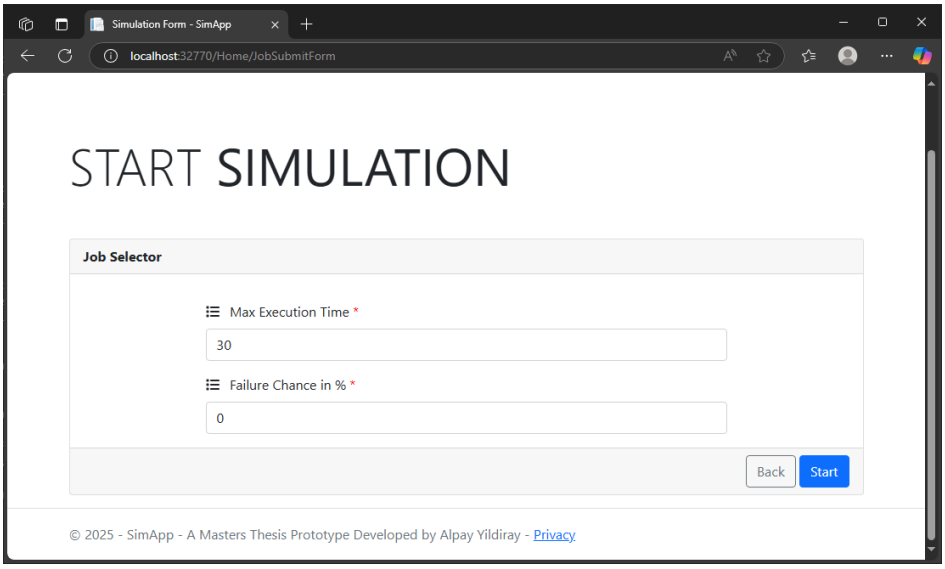

Figure 27: Browser window showing user starting a job with a 0% failure chance and maximum 30 seconds execution time

The resilience of the prototype was tested by simulating a node crash during job execution. Figure 27 shows the submission form for the `randomfailer` tool with its failure probability intentionally set to zero, guaranteeing that any later error cannot originate from the algorithm itself. A maximum execution time of thirty seconds is configured to ensure the container remains active long enough for the test.

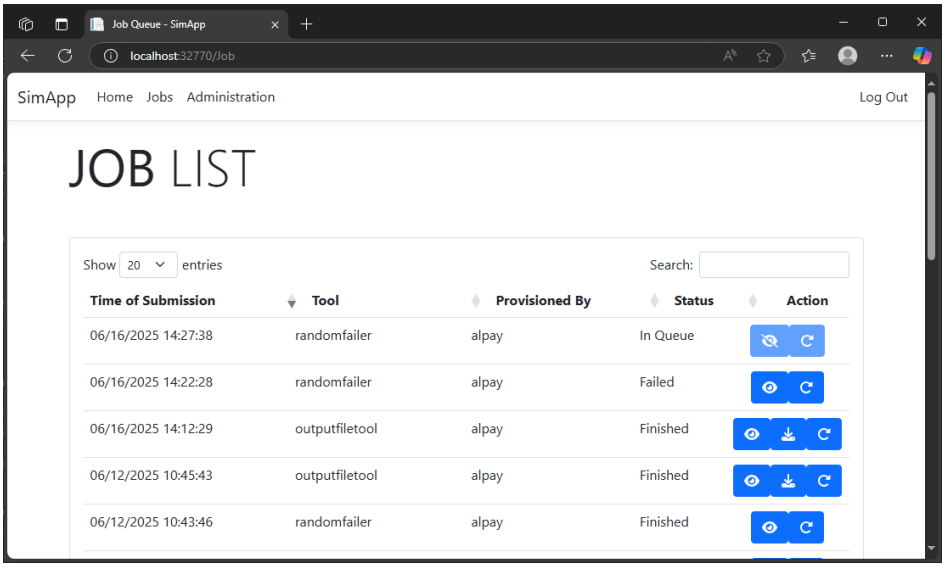

Figure 28: Browser window showing the 0% failure chance job running

Immediately after submission the job appears in the queue with status *In Queue*, then transitions to *Running* as illustrated in figure 28. At this point the container is executing normally, and the cluster node is emitting *Status Changed* events every few seconds. Roughly halfway through the scheduled runtime the node process is force-terminated with `docker kill <node-container>`, simulating a hard crash or power loss. Because job update events from that node stop arriving, the Job Supervisor detects a timeout and marks the job as *Failed*.

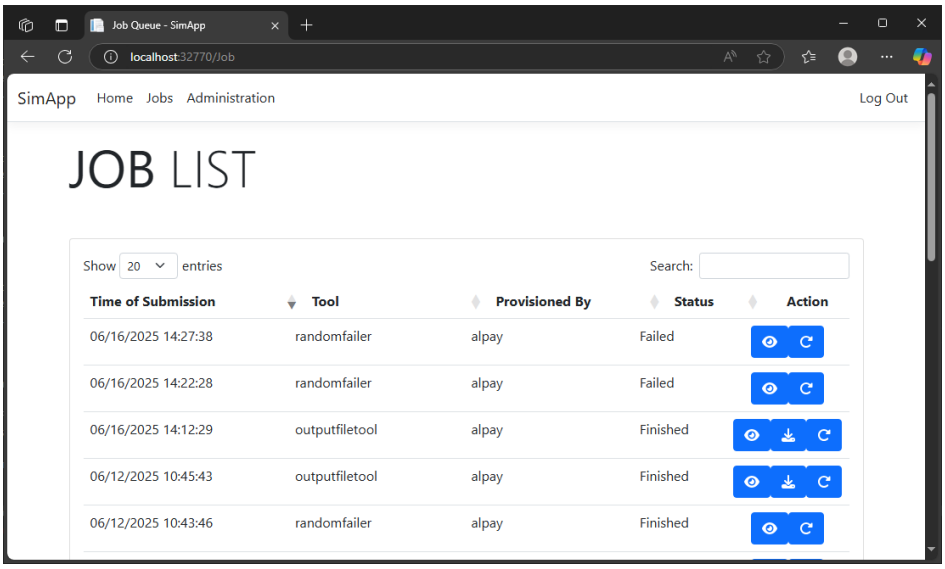

Figure 29: Browser window showing the 0% failure chance job being marked as failed

Figure 29 confirms the change: even though the tool's own failure chance was zero, the system correctly records the job as failed due to infrastructure loss.

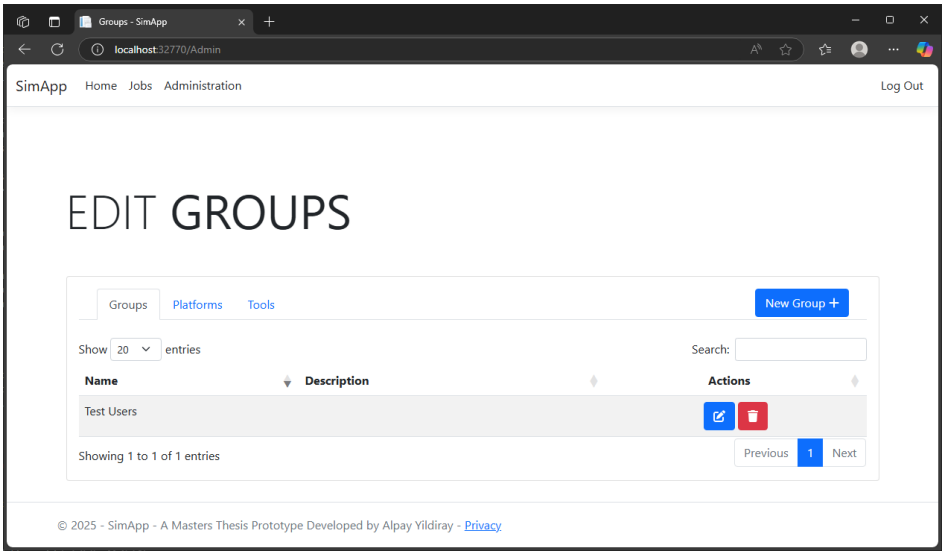

Figure 30: Browser window showing user groups on administration page

Maintainer - CRUD User Groups Figure 30 displays the administration page that is available only to maintainers. The tabbed interface separates Groups, Platforms, and Tools, making it clear which domain the maintainer is editing. Klicking *New Group* opens a form for creating a fresh workgroup and assigning an initial set of members.

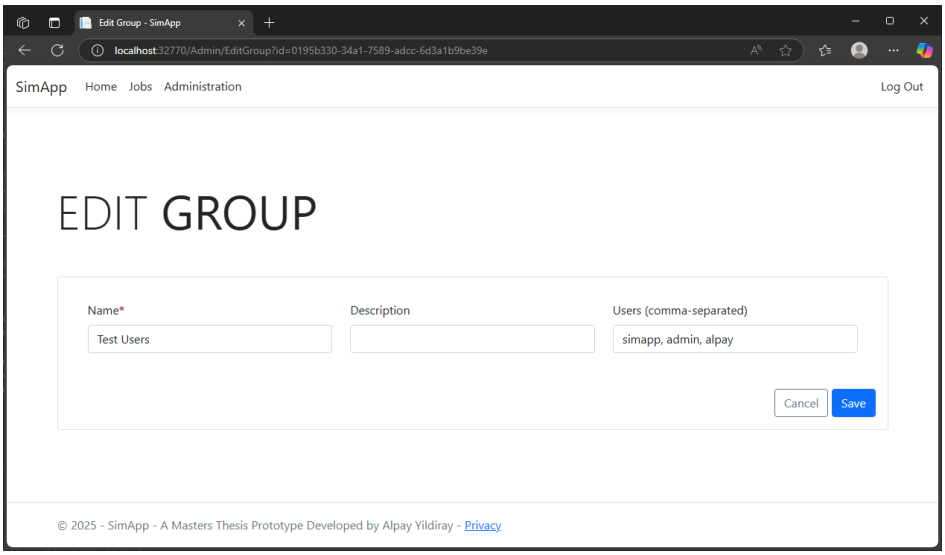

Figure 31: Browser window showing user editing a user group

Editing an existing group is shown in figure 31. Here the maintainer can rename the group, add a description, or update the comma-separated list of usernames. When the *Save* button is pressed SimApp validates that each username exists in Keycloak, then writes the update to the database through the `GroupRepository`. A success toast confirms that the change has been committed.

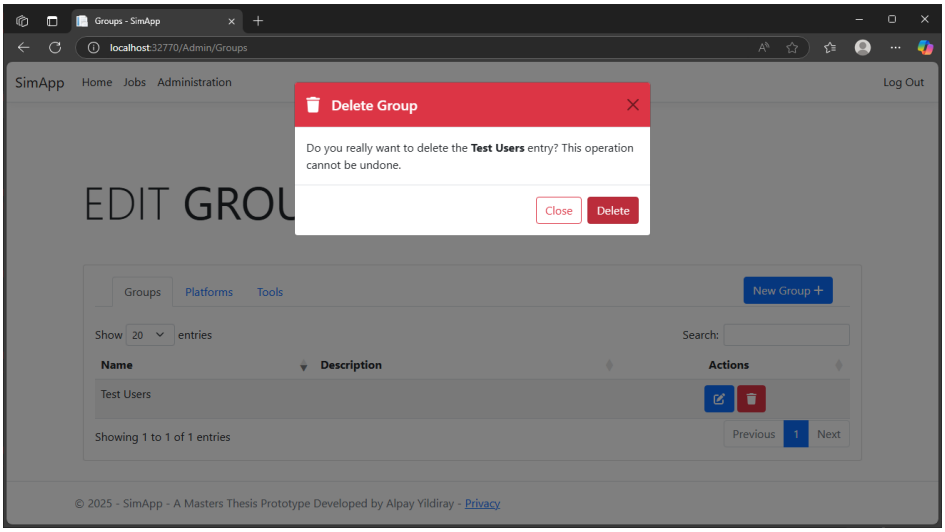

Figure 32: Browser window showing user deleting a user group

Deletion requires an explicit confirmation dialog, as illustrated in figure 32. If the maintainer clicks *Delete*, SimApp removes the group record, detaches all user associations, and

publishes a *Group Deleted* event so that running services can update in-memory caches. The modal protects against accidental data loss, reinforcing the least-privilege principle for high-impact actions.

These three screens demonstrate that maintainers can perform full CRUD operations on user groups without leaving the web interface, satisfying the requirement for easy account and permission management.

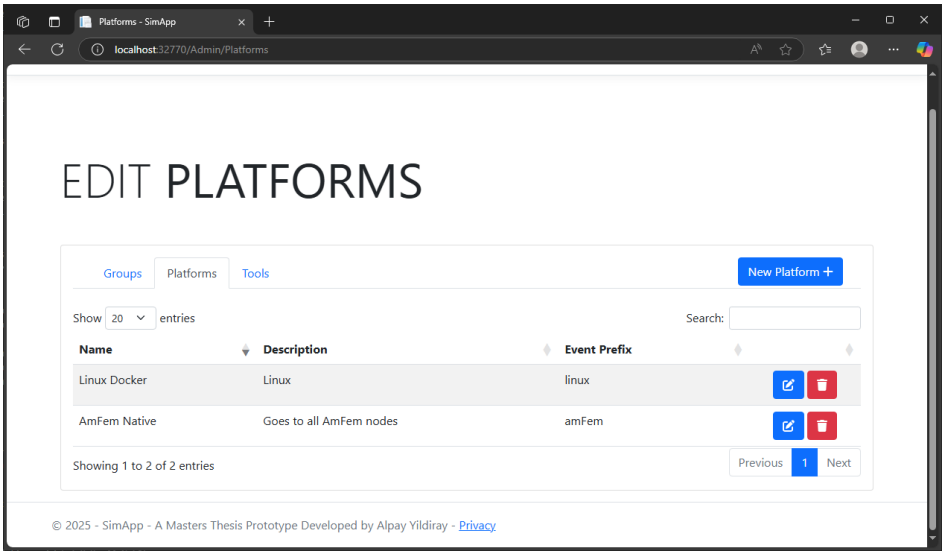

Figure 33: Browser window showing tool platforms on administration page

Maintainer - CRUD Platforms Figure 33 presents the *Platforms* administration tab, which lets a maintainer register or modify the execution environments that simulation tools may target. Each platform row stores a unique name, a description, and an event-stream prefix that RabbitMQ uses for routing events. The layout mirrors the Groups view, so maintainers need to learn only one interaction pattern for both domains.

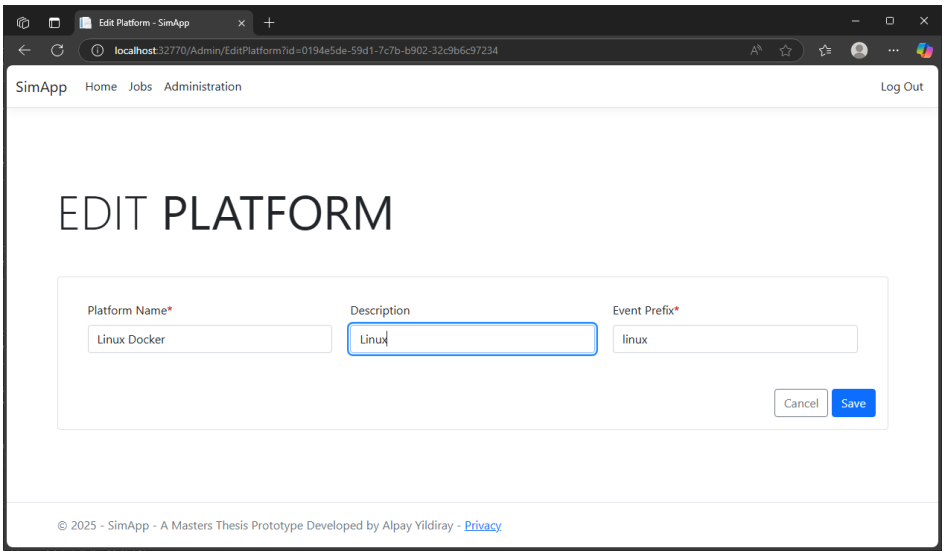

Figure 34: Browser window showing user editing a tool platform

Selecting the blue edit icon opens the form shown in figure 34. Here the maintainer can adjust the descriptive text or change the event prefix that routes jobs to the correct node type, for example from `linux` to `linux-gpu`. When *Save* is pressed the change propagates through the `PlatformRepository`, triggers an audit entry, and publishes a *Platform Updated* event so that any long-lived caches inside cluster nodes stay consistent.

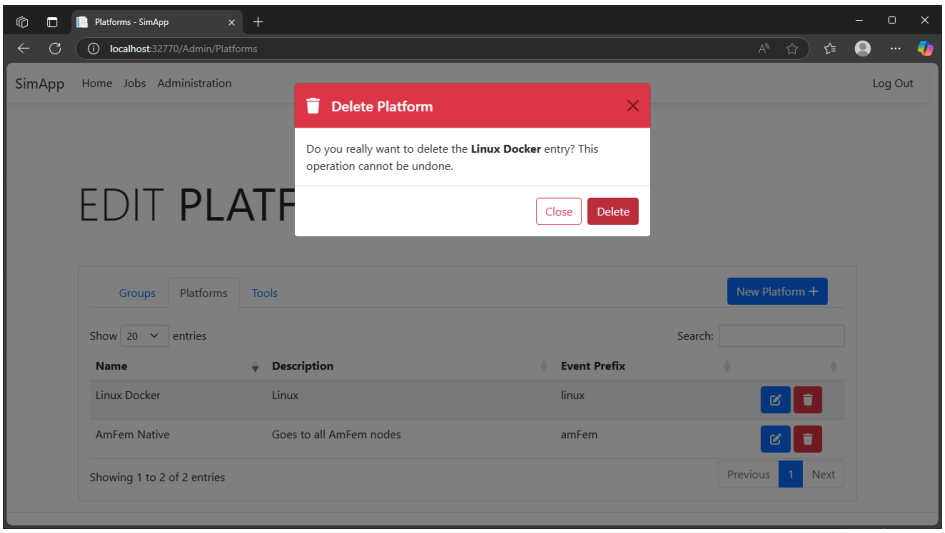

Figure 35: Browser window showing user deleting a tool platform

If a platform is no longer needed it can be removed with the red trash-can icon, which launches the confirmation dialog in figure 35. Deletion is blocked if active tools still reference the platform, enforcing referential integrity at the persistence layer. Otherwise the record is deleted, a *Platform Deleted* event is emitted, and the row disappears from the table, demonstrating safe clean-up of configuration data without orphaning related entities.

These CRUD actions confirm that maintainers can manage execution environments entirely through the web interface while the system guards against destructive changes, satisfying the requirement for easy, reliable platform administration.

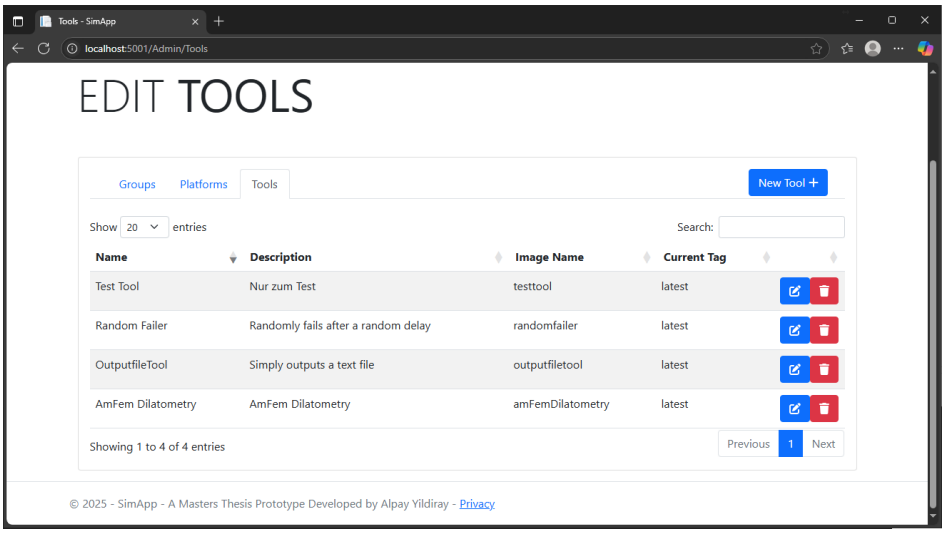

Figure 36: Browser window showing tool list on administration page

Maintainer - CRUD Tools Figure 36 lists all registered simulation tools. For each entry the table displays a descriptive name, a short explanation, the Docker-image (or AmFem executable) name, and the image tag that is currently considered “latest”. The blue pencil opens the edit dialog, while the red trash-can requests deletion.

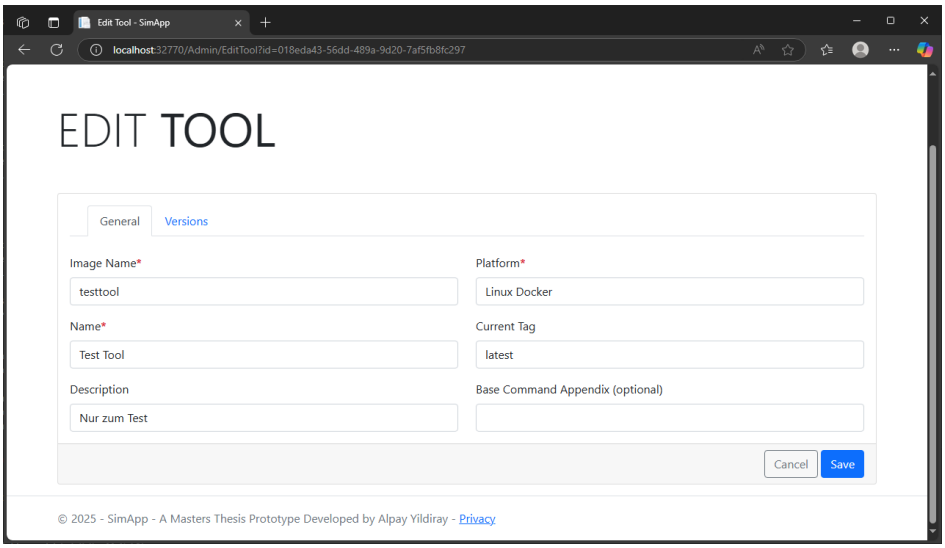

Figure 37: Browser window showing user editing a tool

Editing a tool (figure 37) lets a maintainer change the image name, switch the target platform, or update the default tag so that users get a different version as the default when selecting a tool. Because tools often evolve quickly, the form also offers a separate Versions

tab, where additional tags can be added or removed without touching this general metadata. Saving the form pushes the update to the database through the `ToolRepository`.

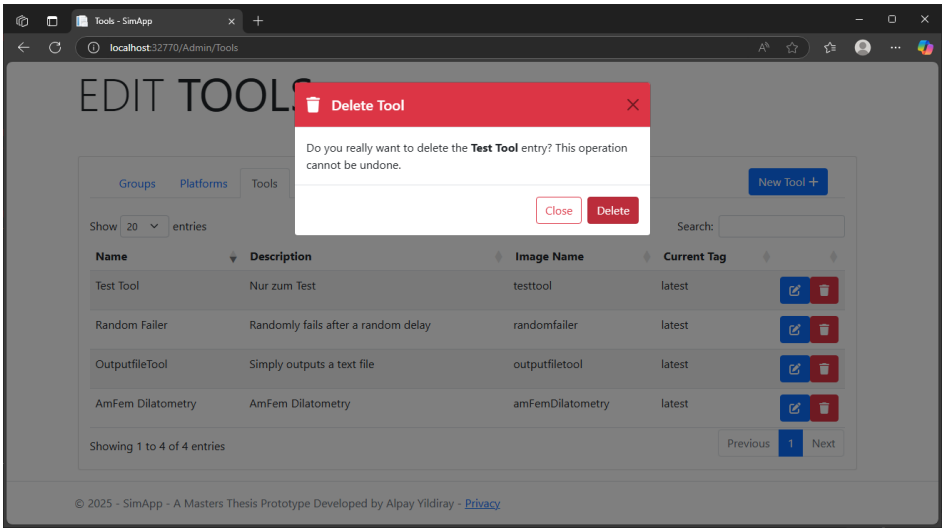

Figure 38: Browser window showing user deleting a tool

If a tool is retired, the maintainer clicks the trash-can, prompting the confirmation shown in figure 38. The back-end prevents deletion while jobs referencing the tool are still queued or running. Once the constraint is satisfied the database row is removed, a *Tool Deleted* event is published, and the UI table refreshes, demonstrating safe cleanup without leaving dangling references.

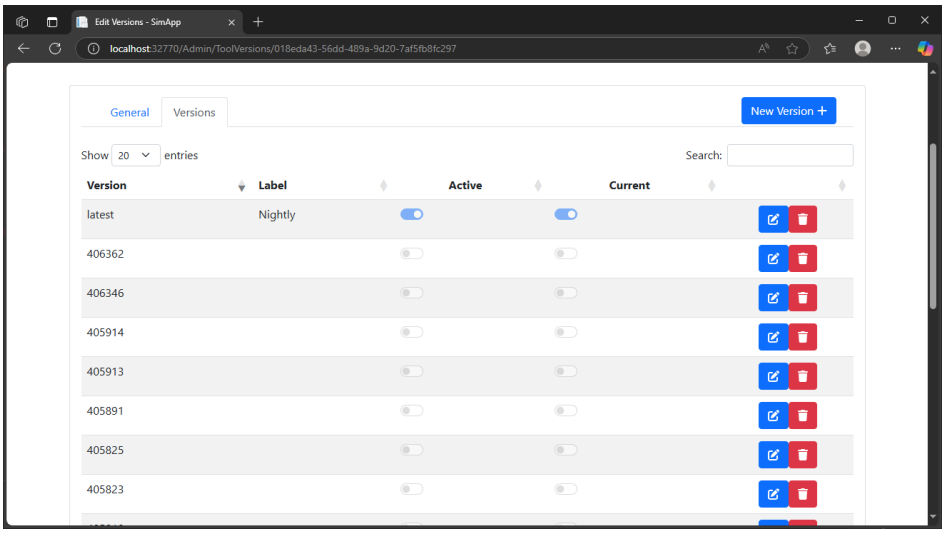

Figure 39: Browser window showing the list of available versions for a specific tool

Maintainer - CRUD Tool Versions Figure 39 lists every version tag that belongs to the selected tool. Each row shows the raw tag pulled from the container registry, an optional human-readable label, an *Active* switch that lets a maintainer disable obsolete builds, and

a *Current* switch that marks which tag should be selected by default when users select a tool on the main page.

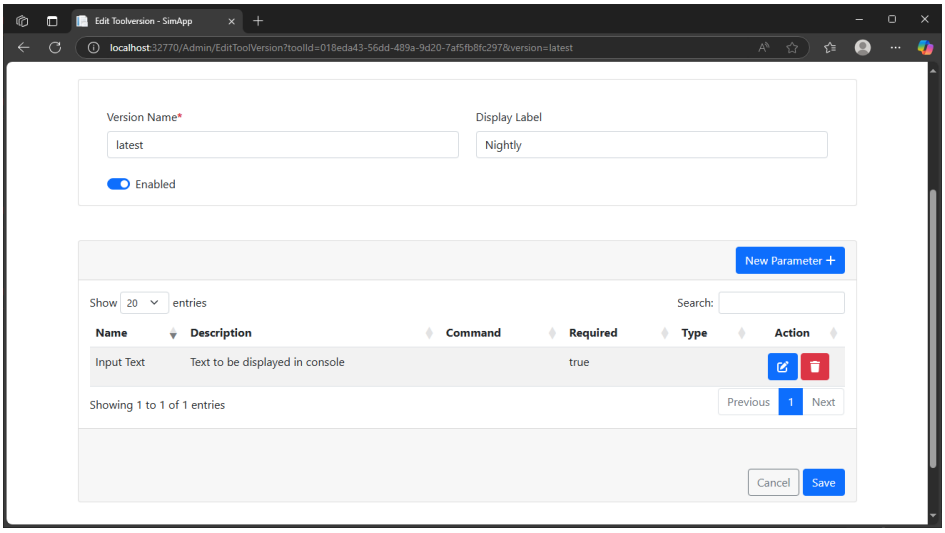

Figure 40: Browser window showing user editing a tool version

Editing a version tag is shown in figure 40. The maintainer can rename the tag, update the display label, enable or disable the version, and add or remove input parameters that are unique to this build. Because parameters may differ by algorithm revision, the form embeds a second table where each input can be edited or reordered. On save the code performs two checks: first that the tag exists in the registry, and second that required parameters have unique command-line placeholders. A success toast then confirms the update and the revised parameter schema is ready for immediate use.

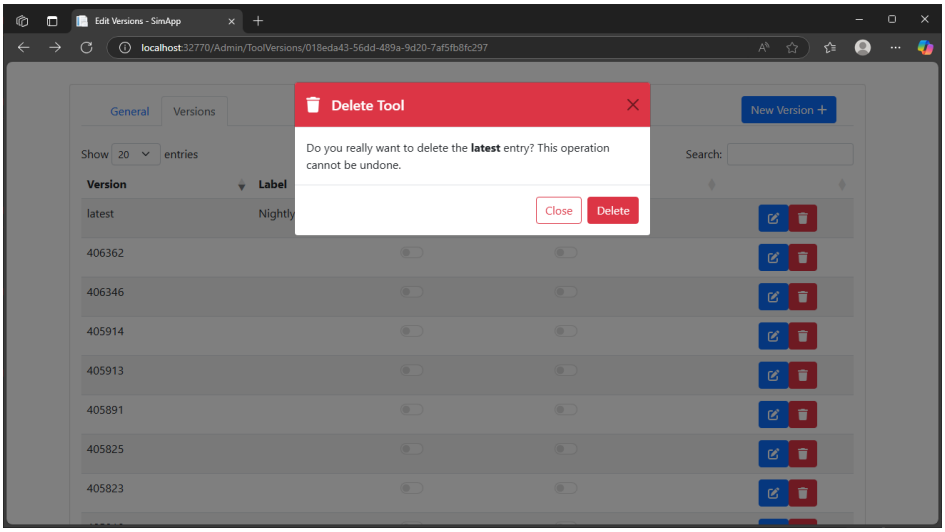

Figure 41: Browser window showing user deleting a tool version

If a tag is deprecated the maintainer clicks the red trash-can, triggering the confirmation dialog in figure 41. Deletion is blocked if jobs referencing the version are still queued or running. Once the constraint clears, the record can be removed from the database.

Using these functions, maintainers can create, disable, or retire individual versions without touching the underlying container images, ensuring that the platform can roll forward or back between algorithm builds while preventing “configuration drift” between nodes.

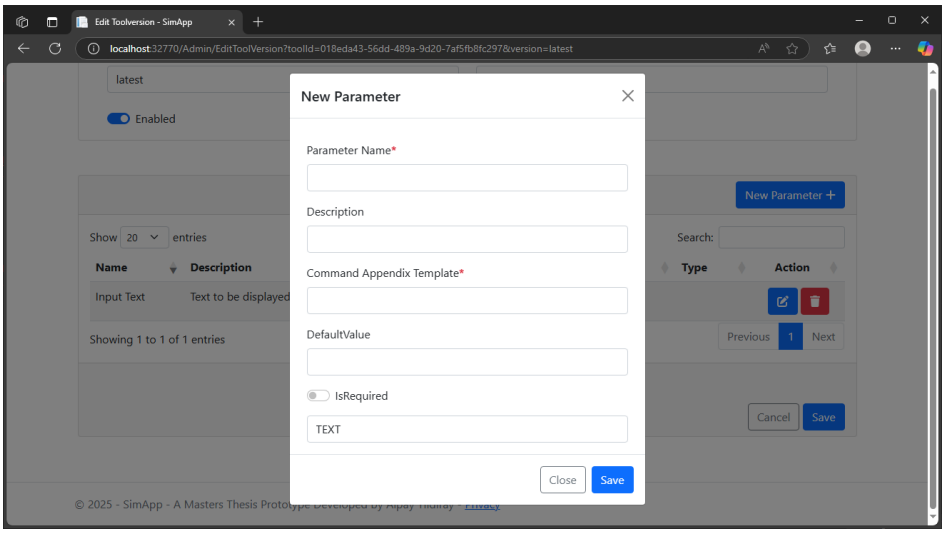

Figure 42: Browser window showing user adding a parameter to a tool version

Maintainer - CRUD Parameters Figure 42 shows the dialog that appears when a maintainer clicks *New Parameter*. Mandatory fields include a unique *Parameter Name* and a *Command Appendix Template*. The template defines how the value is injected into the command line, for example `--iterations <value>`. Optional fields capture a description, a default value, and a boolean flag to mark the parameter as required. After the form passes client-side validation it is posted to the back-end, which persists the new `Parameter` entity and returns the refreshed list to the table without a full-page reload.

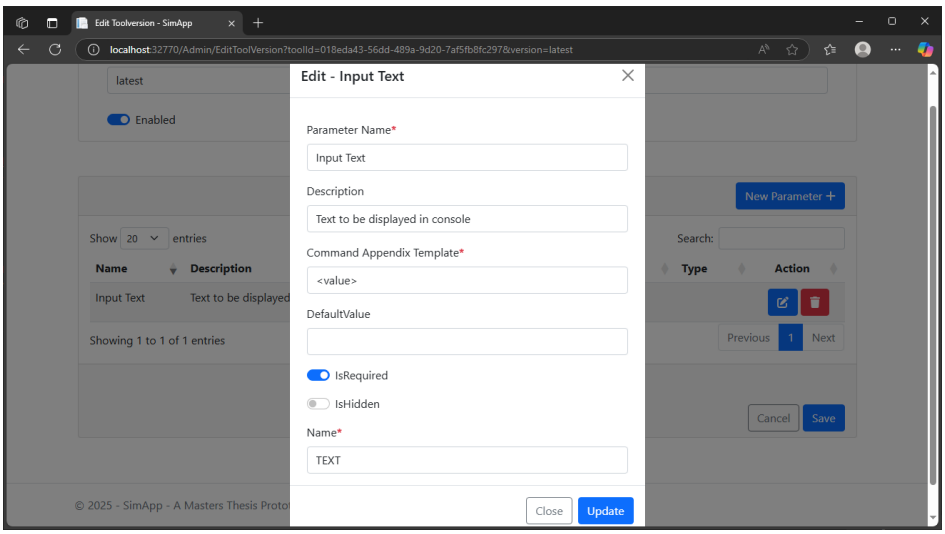

Figure 43: Browser window showing user editing the parameter of a tool version

Figure 43 demonstrates editing an existing parameter. All attributes can be changed, including whether the parameter is required or hidden. Hidden parameters stay in the schema but are not exposed to users in the input form. Pressing *Update* triggers a repository call that writes the modifications and updates the row in the table.

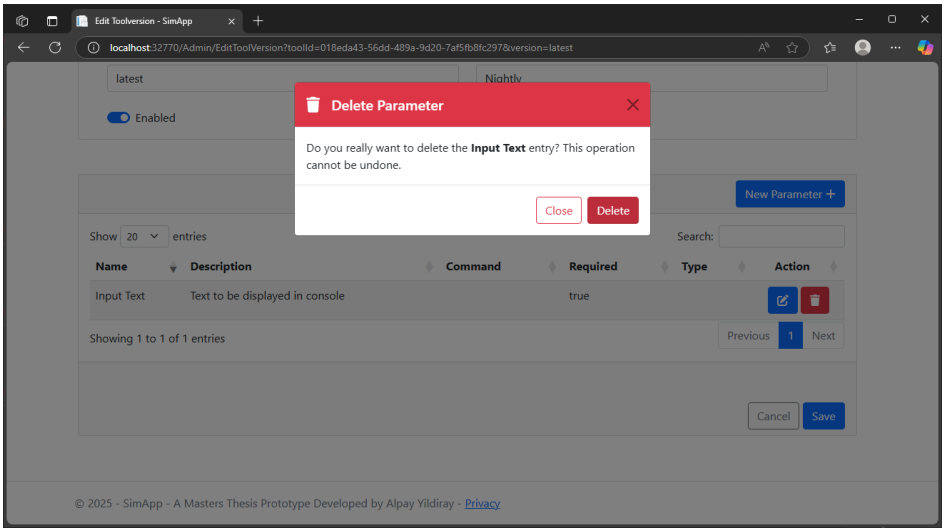

Figure 44: Browser window showing user deleting the parameter of a tool version

If a parameter becomes obsolete the maintainer selects the trash-can icon, prompting the confirmation dialog in figure 44. The record is then removed from the database through the repositories. The UI table then updates instantly, confirming that the configuration has been cleaned up safely. These add, edit, and delete flows allow maintainers to fully configure the input schema of every tool version, enabling rapid evolution of simulation interfaces while preserving data integrity and backward compatibility.
This completes the maintainer workflows, granting maintainers full control over groups, platforms, and tools via the same consistent web experience, and fulfilling the requirement for straightforward management of simulation resources and permissions.

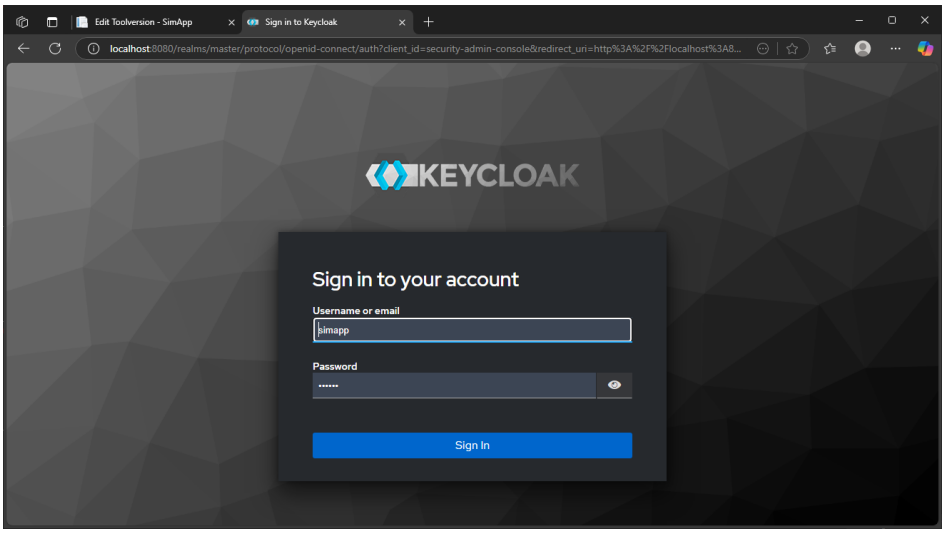

Figure 45: Browser window showing administrator logging into keycloak

Administrator - Keycloak Login Figure 45 captures the login screen for the Keycloak administration console. An administrator enters the `simapp` service account and its password to gain access (the default has been configured to be simapp/simapp). After a successful sign-in Keycloak issues an OpenID Connect token that allows the admin to create or remove user accounts, assign roles, and configure client scopes without touching the application code. This step confirms that identity management is fully externalised, satisfying the requirement for centralised authentication and role-based authorisation.

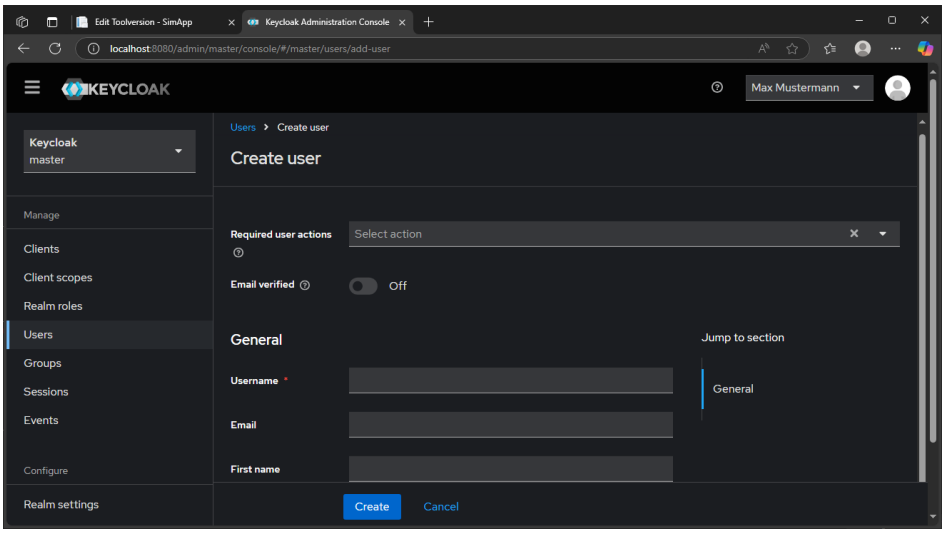

Figure 46: Browser window showing administrator creating a new user in keycloak

Administrator - Creating Users Figure 46 illustrates the Keycloak console in *Users → Create user* view. The administrator fills in a unique username, optional contact details, and assigns initial required-user actions such as password reset or e-mail verification. After pressing *Create*, Keycloak stores the account in its internal directory and returns to

the credentials tab, where the admin can set a temporary or permanent password. Because SimApp receives user roles through JWT claims, the new account becomes effective immediately, allowing users to log in right after its creation.

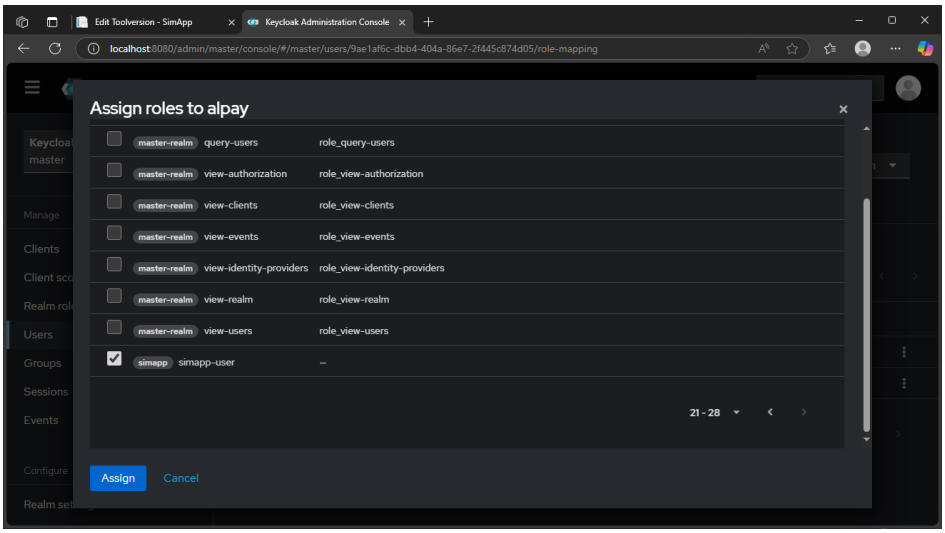

Figure 47: Browser window showing administrator assigning roles to an existing user

Administrator - Assign User Permissions Figure 47 depicts the *Role Mapping* dialog inside the Keycloak admin console. After navigating to *Users*, selecting a specific account, and opening the *Role Mapping* tab, the administrator clicks *Assign Role*. A scrollable list of realm-wide and client-specific roles appears. In this example the `simapp-user` client role is toggled for the highlighted account, granting the bearer permission to log in to SimApp and start jobs. Additional roles, such as `simapp-admin` for application administrators, can be granted the same way. Pressing *Assign* stores the mapping; on the next login the new claims are embedded in the user's JWT token and immediately recognised by SimApp without any service restart, completing the permission-management workflow.

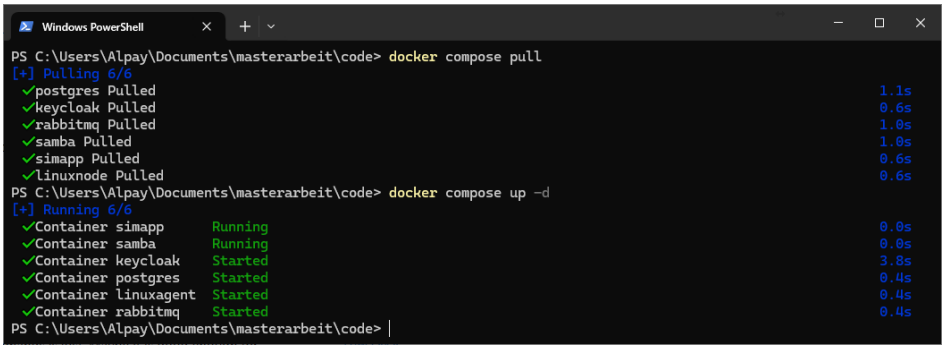

Figure 48: Terminal windows showing administrator updating the app through the Docker Compose utility

Administrator - SimApp Update Figure 48 illustrates how an administrator updates every service in the deployment stack with two simple Docker Compose commands. The first command, `docker compose pull`, fetches the newest images that were pushed by the CI/CD pipeline, covering the database, Keycloak, RabbitMQ, SimApp, and cluster nodes. Immediately after, `docker compose up -d` recreates only those containers whose image digests changed, while leaving unchanged services running. Because Compose performs an in-place restart, the overall interruption is measured in seconds, and jobs that are already running inside cluster nodes continue unaffected, as they are provisioned in their own docker containers, which are not affected by the node restart. The terminal output confirms that all six containers started successfully, demonstrating the easy maintenance process achieved using Docker Compose. This also fulfills the goal of zero-downtime updates, an advantage that was gained by choosing a microservice-based architecture.

7.1.2 Dataflow Analyzation

Model Mapping Across Bounded Contexts As designed in section 5.2.1 using figure 9, when a user submits a form, the presentation layer first converts the view-model into a domain-model that conforms to the validation rules of the *Application* context. After successful validation the domain-model is handed to the repository layer, where it is mapped to one or several entity classes and persisted in PostgreSQL. The same domain-model instance is then converted to a *JobStartedEvent* and published on RabbitMQ, ensuring that identical payloads reach the persistence layer and the computation cluster. Because every event type has a one-to-one counterpart in the domain model assembly, data remains type-safe while being transmitted across context boundaries like when an event is published and consequently consumed through RabbitMQ.

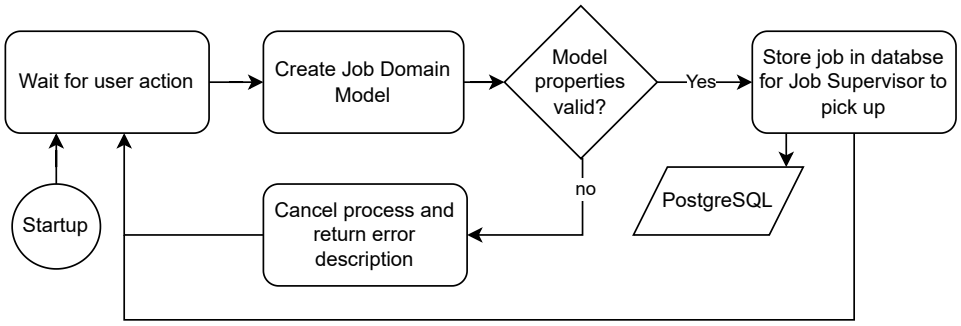

Figure 49: Flow chart of job start dataflow

Job Start (Application Layer) Figure 49 follows the path from a user action to a stored `Job` row. The job service creates a domain-model, validates required fields, and, if the object passes all checks, inserts it into the database for the Job Supervisor to pick up. If validation

fails, the service aborts the transaction and returns an error message to the UI, preventing incomplete jobs from entering the queue.

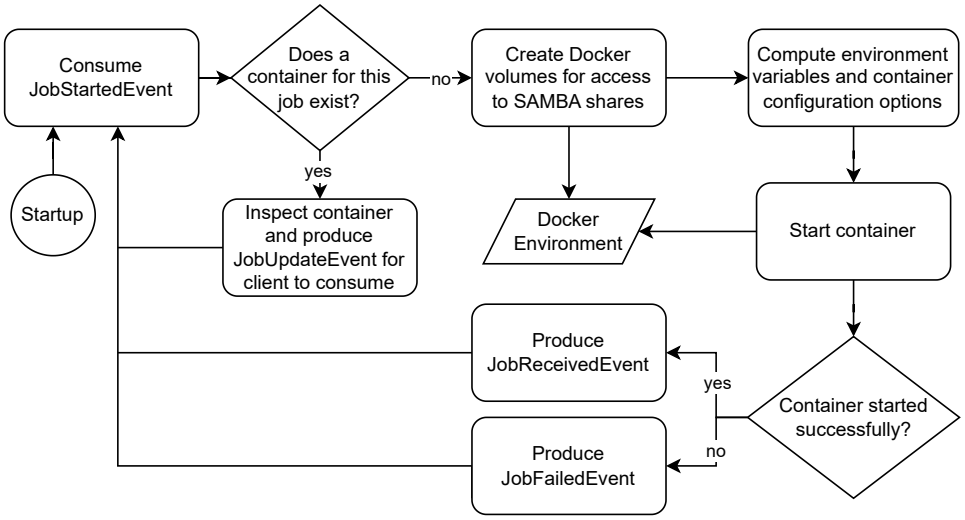

Figure 50: Flow chart of job start dataflow on a computation node

Job Start (Cluster Node) Once the Job Supervisor emits a *JobStartedEvent*, any node that subscribes to the matching platform prefix executes the flow in figure 50. The node inspects whether a container with the requested identifier already exists. If not, it mounts the three SMB shares as Docker volumes, computes environment variables, and attempts to launch the container. A *JobReceivedEvent* is produced on success; otherwise the node reports a *JobFailedEvent*, allowing the supervisor to decide on a retry or report the error to the user.

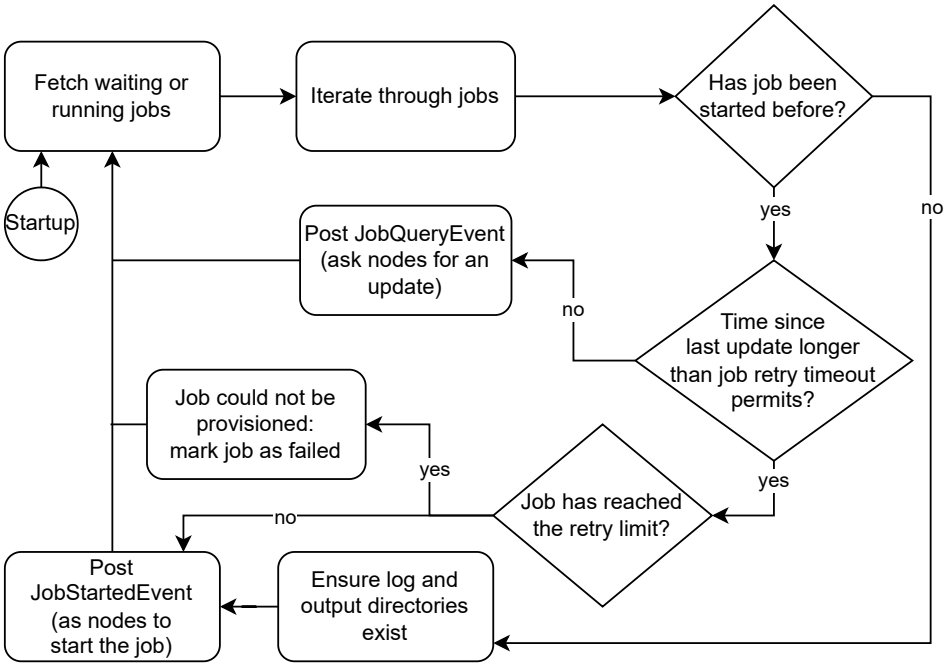

Figure 51: Flow chart of job supervision dataflow

Job Supervision Figure 51 illustrates how the Job Supervisor polls outstanding jobs. For each waiting entry the service either posts a fresh *JobStartedEvent* or a *JobQueryEvent* (status request). If no update arrives within a configurable timeout, the supervisor increments a retry counter and, once the limit is reached, marks the job as failed.

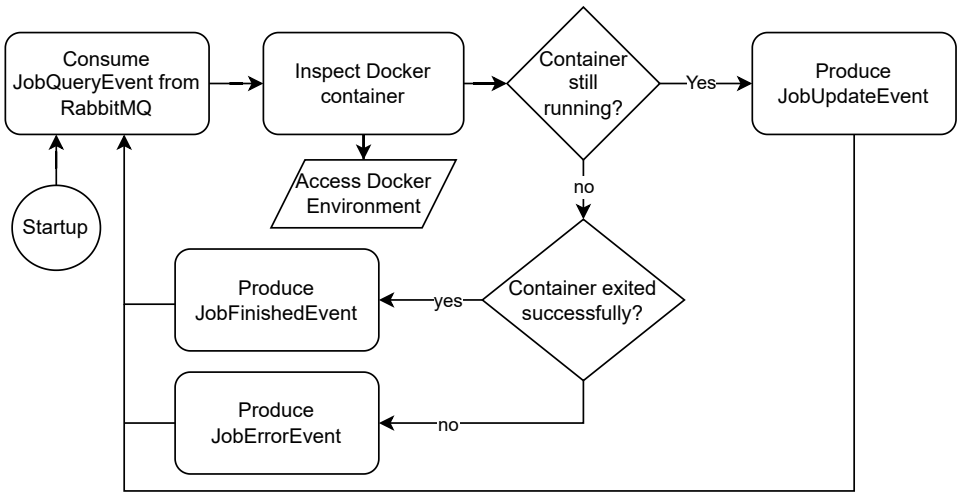

Figure 52: Flow chart of job update dataflow

Status Updates from Nodes Cluster nodes react to every incoming *JobQueryEvent* as shown in figure 52. If the container is still active, the node returns a *JobUpdateEvent*. If

the container has exited, it inspects the exit code and emits either *JobFinishedEvent* or *JobErrorEvent*. These events propagate back to the application layer, where the supervisor updates the database so the user can observe the status change through the UI.

Failure With Spare Nodes Available When a node crashes or explicitly returns a *JobErrorEvent*, the supervisor checks whether the job's retry count is below the threshold. If spare nodes are registered for the same platform, the supervisor posts a new *JobStartedEvent*. Because events are content-based and idempotent, the replacement node can reuse all original parameters without further modification, giving the system automatic fail-over while preserving traceability.

User-Initiated Restart or Copy Jobs in the job table contain a restart button: It allows the user to create a new job with a fresh identifier but pre-filled parameter set. The action instantiates a new domain-model, persists it, and emits a *JobStartedEvent* to RabbitMQ. From the supervisor's perspective a copied job is indistinguishable from any other first-time submission, ensuring consistent handling across the pipeline. The function is a UI mechanic that aims to improve user experience.

7.2 Computation Performance Tests

To understand how the prototype performs under load, a set of automated test cases was written and executed against several cluster configurations. Each test submits large batches of simulation jobs and records metrics such as job start times, delta start times between jobs, average runtime per job and total execution time for the whole test.
The first series targets a cluster with only a single node. Then, an increasing number of physical machines are added, each equipped with either AMD Ryzen processors, or with Intel Xeon Processors of varying generations and processing powers. Running the same workload on different cluster configurations highlights how much raw CPU performance, core count and available RAM influence overall performance and helps identify weaknesses in the computing cluster architecture and its scalability.
100 jobs with a target runtime of zero seconds are submitted almost instantly, saturating the RabbitMQ message queues. The test records how quickly the cluster clears the backlog, whether any status events are lost, and how gracefully the job supervisor recovers if a node fails mid-run. Through a few individual tests, 100 was deemed a sufficient number to highlight the clusters scalability behavior. Running more would only increase the total execution time with little impact on average job execution times while fewer would exit a little bit too quickly.
Together these scenarios provide a clear picture of scalability, resource utilisation, and fault tolerance, showing how the system behaves from a minimal single-node setup to a fully populated cluster operating at maximum capacity.

As illustrated in figure 17 of section 6.5, four computers will be used These computers have the following hardware:

Node Hardware				
Computer	Processor	Physical Cores	Logical Cores	RAM
AIRWS33	AMD Ryzen 3900X	12	24	32
AIRCUDA03	Intel Xeon Silver 4112	8	16	128
AIRWS20	Intel Xeon E5-1620 v4	4	8	32
AIRCL01	Intel Xeon E5-1607 v3	4	4	16

Table 1: Table depicting the hardware setup of the computers used for testing

Table 1 lists the four machines that formed the test cluster and summarises the core resources available on each node. All computers are part of the ISEMP infrastructure and are identified by internal host names, beginning with the prefix “AIR”, followed by either “WS” or “CL” for workstation-class systems, or “CUDA” for GPU-capable servers.

AIRWS33 is the most powerful workstation in the pool. Its AMD Ryzen 3900X processor provides 12 physical Zen 2 cores and 24 logical threads, which gives excellent single-node throughput for highly parallel simulation tasks. With 32 gigabytes of RAM, the system can cache large input meshes or run multiple medium-sized jobs concurrently without swapping.

AIRCUDA03 is a rack server equipped with an Intel Xeon Silver 4112. While it offers fewer cores than AIRWS33, it carries 128 gigabytes of memory and an NVIDIA GPU (not used in the current tests but available for future work). The high RAM capacity makes it suitable for memory-bound workloads or simulations that keep extensive result fields in memory until post-processing is complete.

AIRWS20 and AIRCL01 represent mid-range and entry-level nodes, respectively. Both use older Intel Xeon E5 processors with four physical cores, but AIRWS20 supports hyper-threading for eight logical threads, whereas AIRCL01 does not. These two systems are valuable for evaluating how the architecture behaves when a heterogeneous cluster mixes strong and weak nodes. They also reveal how effectively the RabbitMQ event exchange assigns tasks to different consumers to prevent slower hardware from becoming a bottleneck. AIRWS20 includes 32 gigabytes of RAM, enough for most small simulations, whereas AIRCL01’s 16 gigabytes put it at the lowest end of the memory spectrum in the test bed.

By combining nodes with very different CPU counts, thread capabilities, and memory sizes, the benchmark suite can measure three important properties of the prototype. First, it shows how much advantage the software takes from additional cores and threads when running workloads with multiple parallel jobs. Second, it demonstrates whether the event-driven job supervisor distributes work proportionally to each node’s capacity in multi-node tests. Third, it helps detect any hidden assumptions in the code that might

limit portability, such as hard-coded thread pools or RAM thresholds. The monitoring stack must accommodate machines that publish status updates at different rates due to their hardware limits.
In summary, the hardware mix in Table 1 offers a realistic cross-section of the kinds of equipment found in academic and industrial labs. It provides a solid basis for testing both vertical performance scaling on individual hosts and horizontal scaling when several nodes cooperate inside the cluster.

7.2.1 Performance Results

The four graphs in figures 53-56 plot five metrics for each of the burst tests. *Start Time* (blue) measures how many seconds elapsed between the *Start Job* event and the moment the container or native process actually entered the *running* state. *Execution Time* (orange) shows the time the container spent inside the node until it exited with a success code. *Delta Start Time* (yellow) is the difference in provisioning delay between any job n and the previous job $n-1$, so negative values mean the container began earlier than the running average. Finally, the dotted blue curve is a moving average that smooths short-term oscillations and highlights long-term trends.

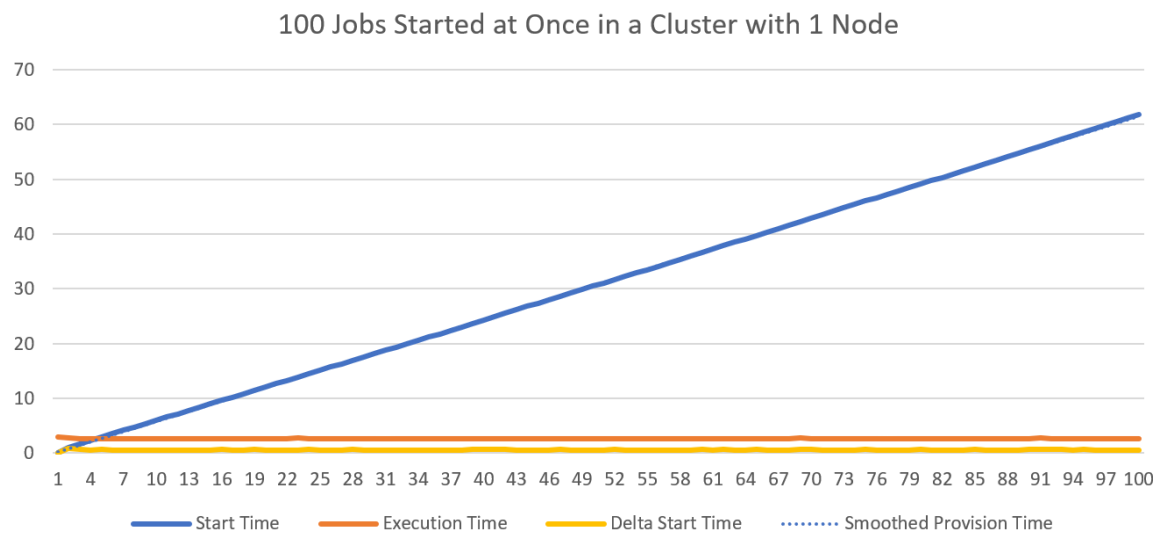

Figure 53: Test results after running the test on only node AIRWS33 (seconds/count)

One Cluster Node on AIRWS33 With only AIRWS33 online (figure 53) the blue start-time curve rises linearly. Each job gets provisioned after the previous job was processed and a container was started, which takes around **0,61 seconds**. The hundredth job launches roughly sixty seconds after the first, resulting in a total execution time of **61,74 seconds**. The tight cluster of orange execution bars, averaging **2.66 seconds**, reveals that the workload

itself is lightweight; the dominant cost is queueing. Yellow deltas hover near zero because no alternative node can overtake the baseline sequence.

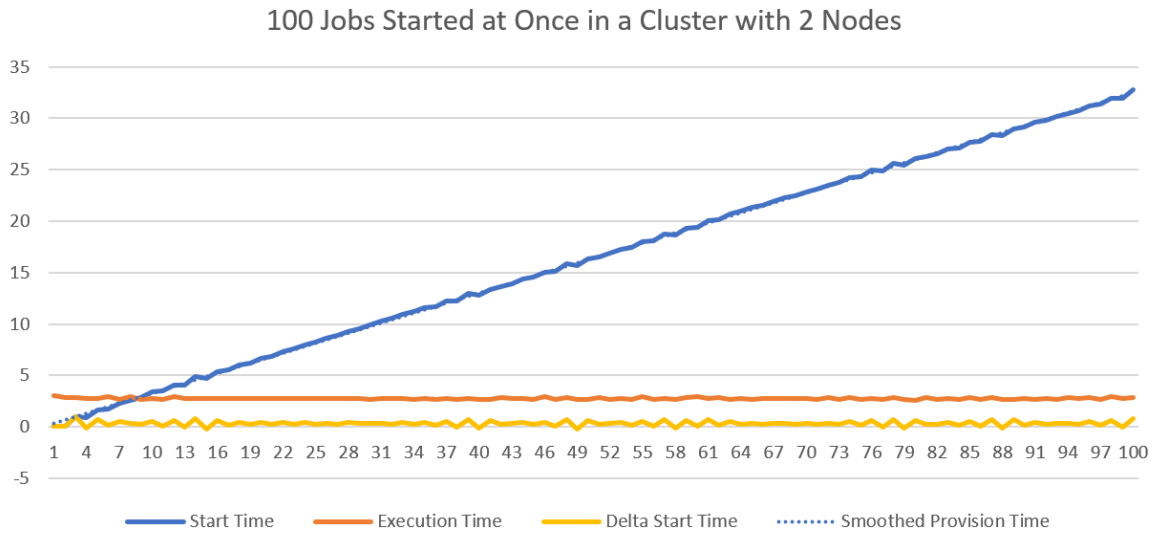

Figure 54: Test results after running the test on the two nodes AIRWS33 & AIRCUDA03 (seconds/count)

Two Cluster Nodes on AIRWS33 & AIRCUDA03 Introducing AIRCUDA03 almost halves total runtime to **32.74 seconds** (figure 54). This is almost a factor of two, which is what would have been expected if both had the exact same hardware. The slope of the blue line is very linear with very small variations between jobs. Average start time drops to **0.32 seconds**, which is almost half, so adding a second node also improved this metric almost by a factor of two. Execution time nudges upward to **2.75 seconds**. Job containers likely take slightly longer to finish execution due to the slower single-thread performance of the CPU, so the average rises. Yellow deltas begin to oscillate around zero, likely also because of the very same reason. Assuming events get distributed to the two cluster nodes in turn (so each nodes gets every second job), variations in *JobStarted* event processing times and subsequent container provisioning would lead to these oscillations.

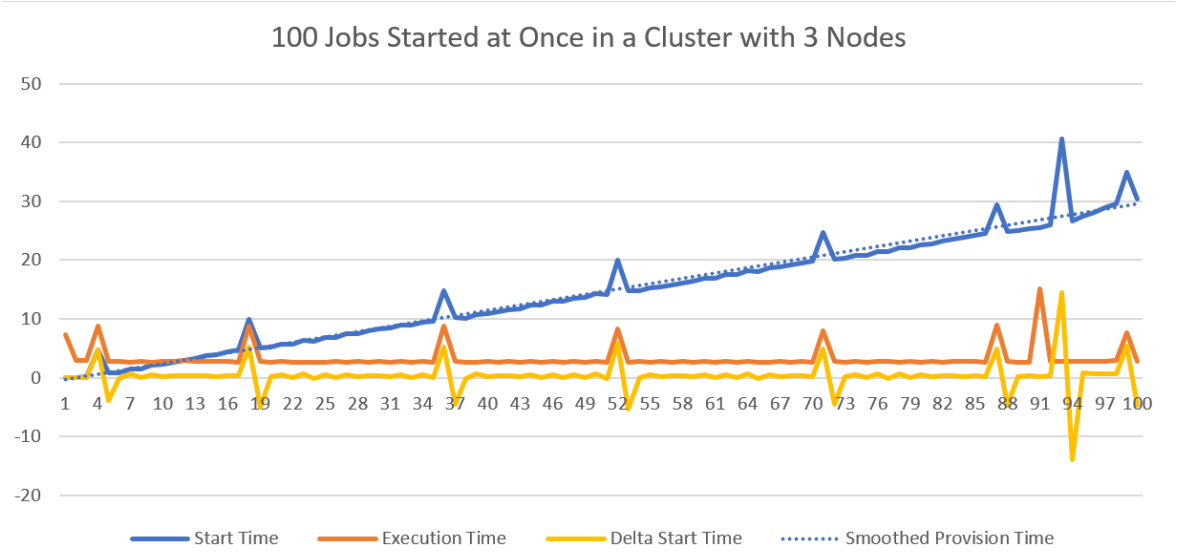

Figure 55: Test results after running the test on the three nodes AIRWS33, AIRCUDA03 & AIRWS20 (seconds/count)

Three Cluster Nodes on AIRWS33, AIRCUDA03 & AIRWS20 With AIRWS20 added (figure 55) overall runtime improves only slightly, to **30.34 seconds**. The smoothed start-time trend stays relatively linear but similar to the other metrics significant outliers appear. Average job execution time also climbs to **3.30 seconds**, showing a worse job execution time than with the previous two-node test, where significantly faster computers were used. AIRWS20 offers half as many logical cores and a drastically lower single-thread performance, so jobs routed to this node are consequentially slower. Spikes in yellow deltas also grow taller and broader. However, when counting the exact job execution times with significantly higher values, it appears that **<10%** of all provisioned jobs have been routed to this new slower node. With a completely fair, round-robin-like scheduler, we would expect a number around **33%** or $\frac{1}{3}$. This confirms that the scheduler does not send new work to a node that is already busy, leaving the slow-running node to clear its backlog slower than its partners. The only metric that actually improved by adding this third node was the average job start time, which was slightly reduced from **0.32 seconds** to **0.30 seconds**. This observation makes sense, as RabbitMQ has more nodes to route job events to. The delivery time of events improves.

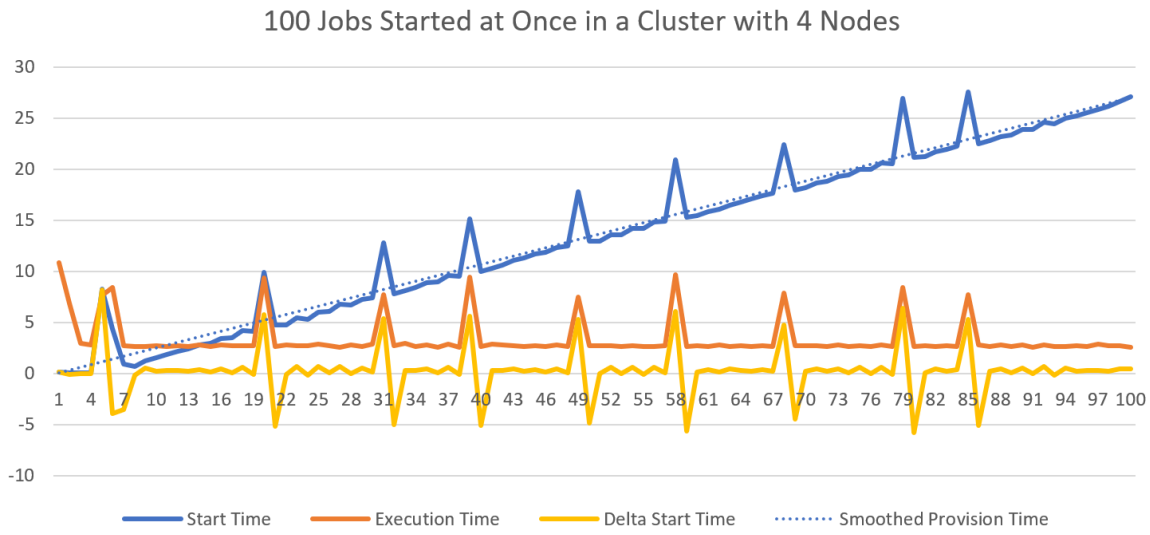

Figure 56: Test results after running the test on the four nodes AIRWS33, AIRCUDA03, AIRWS20 & AIRCL01 (seconds/count)

Four Cluster Nodes on AIRWS33, AIRCUDA03 & AIRWS20 Adding AIRCL01 (figure 56) seems to deliver minimal benefit and even slightly worse averages, from **0.30** to **0.31 seconds** to start and **3.30** to **3.46 seconds** to fully execute. Total runtime also slightly rises to **31.86 s**. AIRCL01 lacks Hyper-Threading and ships with the lowest base clock, so its jobs consistently finish last. The dotted moving-average curve stays relatively linear, but shows even more outliners, indicating that faster nodes execute their jobs unaffectedly fast, while the slower nodes try to keep up. Large yellow spikes identify the few workloads that land on the weakest hardware, and negative spikes reveal occasions where an empty fast node launches a late-arriving job almost immediately. Even still, the proportion of jobs processed by the two slower nodes is significantly lower than the expected (**<10%** instead of $\frac{\mathbf{2}}{\mathbf{4}} = \mathbf{50\%}$).

Figure 57: Time measurements in seconds for all performance tests with different cluster setups

Aggregated comparison Figure 57 aggregates the four scenarios. The blue bars (total runtime) fall by roughly fifty percent when moving from one to two nodes, showing near-ideal speed-up because both hosts have comparable CPU performance. Beyond two nodes, speed-up plateaus and even regresses because the extra nodes feature substantially lower CPU performance. The orange bars (average process start time) also first fall by a factor of 2, but then only slightly decrease with three or four nodes. By contrast, yellow bars (average delta start time) shrink as additional nodes help reduce the event queue buffer, which is why users experience faster job launches even if completion times become more erratic.
The system scales linearly *if* new nodes match the baseline in CPU speed. When heterogeneity is unavoidable, job throughput still rises, but adding slower nodes raises the number of outliers and the mean execution time.
Three core findings emerge: First, the event-driven scheduler itself remains responsive; no events were lost and queue wait never exceeded two seconds even under flood conditions. Second, the prototype would benefit from CPU-aware scheduling, for example weighting nodes by recent throughput or core count to keep stragglers from hogging long jobs. Third, administrators can tune cluster cost and performance by deciding whether more cheap nodes or fewer fast nodes give a better trade-off; the monitoring metrics collected here provide a quantitative basis for that choice.
These tests suggest that the proposed architecture supports high burst loads without functional degradation and that its performance scales predictably, though not perfectly, in the presence of uneven hardware.

7.3 Critical Evaluation of Results

The prototype fulfils every research goal set at the beginning of this thesis. All core requirements were met, including multi user access, high availability, and real time monitoring. Dynamic scaling was demonstrated in the burst tests, where total runtime dropped by roughly fifty percent after a second node of comparable power joined the cluster. When nodes provide similar compute capacity, overall throughput grows almost linearly with each additional worker.

Additionally, several constraints emerged. Adding slower hardware contributed less benefit and in some cases raised mean execution time, showing that heterogeneous clusters need CPU aware scheduling to avoid stragglers. Current tests did not fully saturate system limits. In theory, under ideal conditions and ignoring CPU and RAM limitations, AIRWS33 has an estimated limit of 50000 containers, calculated by dividing total storage capacity of the node with the storage requirements of a single exited test container. The Docker environment on AIRWS33 held fewer than 1000 containers during these tests, which is far below this limit. Similarly, none of the individual workloads exhausted system memory, as I was not able to produce a singular test that required more than 16 gigabytes of RAM. Additional efforts could introduce tools that use specific instructions sets such as AVX or AVX2 to stress CPUs to their limits, or incorporate test tools that integrates a CPU stress test tool like *FIRESTARTER* [24].

Larger clusters would give a clearer picture of how the event stream behaves under not only hundreds, but thousands or even hundreds of thousands of concurrent provision and monitoring events. Mixing compute intensive and memory intensive jobs would also reveal how well the scheduler copes with conflicting resource demands. Finally, testing with nodes that span a wider gap in CPU generations, for example pairing even faster servers with even older nodes, could highlight even bigger constraints or disadvantages of this kind of simulation task distribution.

Despite these potential improvements, the current data confirm that the architecture is robust, scales predictably under balanced hardware, and maintains functional integrity under burst load. The remaining challenges are optimisation opportunities rather than fundamental design or testing flaws.

8 Summary and Conclusion

This thesis set out to design and validate a modern software architecture that can run processes like non-deterministic metal 3D printing simulations on a distributed computation cluster. The goal was to replace a monolithic desktop tool with a flexible, event driven platform that supports multiple users, scales horizontally, and remains resilient to individual component failure. To reach that goal the thesis combined several state of the art approaches, including microservices, containerization, event sourcing, and Domain Driven Design. A complete prototype was implemented in C sharp and .NET 8 and deployed on four heterogeneous nodes provided by the Airbus Endowed Chair for Integrative Simulation and Engineering of Materials and Processes. Extensive integration and performance testing confirmed that the prototype meets all functional and non functional requirements defined in section 3.4.

8.1 Achievements of the Prototype

The system supports true multi user access. Authentication is handled by Keycloak, while role based authorisation in SimApp separates everyday users, maintainers, and administrators. User groups constrain data visibility so that research teams can collaborate without exposing confidential projects across organisational boundaries.
Horizontal scalability was demonstrated by launching one hundred concurrent jobs on clusters that ranged from a single workstation up to four mixed performance nodes. When additional nodes had similar compute power, total workload runtimes decreased almost linearly. Event flow remained stable, containers launched without error, and database writes completed well within the connection pool budget.
High availability is partially achieved through fail fast services, health checks, and automated container restarts. Even so, the RabbitMQ broker is a single logical hub in the current deployment, so full resilience would require a clustered RabbitMQ setup with mirrored queues, which was not needed for the proof-of-concept in this thesis. Instructions on how to deploy RabbitMQ in a high-availability configuration are well documented by the vendor and only require small configuration adaptions. The prototype deployment sufficiently demonstrated the versatility and robustness of the researched software architecture.
The platform allows users to provision simulation jobs through a clean and modern web interface. Users simply supply input file paths and any necessary parameters, press start, and the job supervisor handles all remaining application logic, like validating input fields, posting events to the event stream and storing job information in the attached PostgreSQL database. RabbitMQ then uses its specific routing logic to assign the work to an idle node. During execution, live status and log streaming keep users informed and allow them to observe any related job information in realtime. When a run finishes, output files appear automatically in the configurable Samba folders and can optionally be downloaded from

the web UI, if the developer configures the application in that way.

8.2 Technical Strengths

The architecture enforces strict separation of concerns. Domain models isolate core business logic from infrastructure code, reducing side effects when features evolve 5.2.2. Repository patterns hide database specifics, making it easier to swap PostgreSQL for another relational engine or even a document store 5.2.1. Domain Driven Design provides a stable language between services, so new microservices can subscribe without recompiling existing binaries 5.2.

Containerization simplifies deployment, versioning, and roll back 6.5. Every microservice ships as a single image, is fully configurable through Docker Compose files or even environment variables and includes a clear health probe 6.5.1. Developers can investigate bugs locally by launching the same code through Visual Studio in container debug mode, which was achieved by adding specific container-specific debugging options 6.5.2.

CI/CD pipelines guarantee reproducible builds. The GitLab CI compiles the entire solution, runs SonarQube scans, and pushes images to a registry on every commit 6.4. This automation enforces consistent code quality and removes human error from the release process.

8.3 Observed Limitations

The architectural concept is advanced and challenging. It assumes familiarity with Event Sourcing, microservice patterns, container orchestration, message brokers, and Domain Driven Design. Developers lacking that experience may struggle to debug event flows or understand container orchestration mechanics at the computation nodes. Detailed documentation and time is therefore necessary for new developers to fully understand the software.

Containerization is not a universal solution. The AmFem code requires a Windows desktop environment and cannot run inside a headless Linux image. Although the thesis introduced an `AmFemNode` that runs native Windows processes, that workaround sacrifices container benefits like image immutability, sandboxing and file system isolation.

Additionally, RabbitMQ introduces a star topology. If the event broker becomes unavailable, new jobs cannot start and status updates cannot propagate. High availability deployments with mirrored queues will mitigate that risk, but they also increase operational complexity and hardware requirements.

Sharing files through network folders requires a Samba server with configurable access to both users and the computation cluster nodes. Combining Windows authentication, Keycloak roles, and container mount paths creates a complex configuration setup. Centralised

storage solutions like CephFS or S3 compatible object stores could reduce that burden but require additional infrastructure and significantly more development effort.
Performance scales well only if new nodes match the baseline in CPU speed. When slower hosts join the cluster they generally increase the average execution time, because the scheduler does not yet weight job assignments by historical throughput or the computational capacity of each node.
Finally, the architecture focuses on distributing a large number of parallel simulation processes. It does not partition a single large task across multiple nodes, e.g. through sharing of CPU cores across physical systems. If a simulation requires more memory than any single workstation can provide, the current architectural design reaches its limit. Deploying nodes and configuring them in the SimApp using platforms and tool versions does provide a limited solution to this issue by letting administrators configure the application to route more demanding tasks to the most capable nodes.

8.4 Interpretation of Testing Data

Burst tests showed that, without a significant computational load, queue wait time is mainly taken up by the provisioning process. Average container start time dropped from **0.61 seconds** on one node to **0.32 seconds** on a two node cluster. After that, diminishing returns set in because additional nodes had lower clock speeds and fewer cores. Graphs in figures 53-56 reveal that start time oscillations grow when heterogeneous hardware is mixed, yet the system generally remains stable as a whole. No events were lost, job IDs remained unique, and log streaming continued uninterrupted even during peak load.

8.5 Final Assessment

The prototype achieved all functional goals, ran reliably under load, and even proved flexible enough to spontaneously integrate a legacy Windows solver. Its disadvantages are operational rather than architectural and can be mitigated with additional engineering effort. Therefore the work is mature enough for testing using actual users at the ISEMP and can serve as a blueprint for other software engineers or research groups who face similar non deterministic, compute intensive workloads. With small, incremental refinements, the platform can offer a robust foundation for future platforms and sets a practical example of applying modern software architecture to scientific computing. This thesis is an exemplary exercise in the application of modern software design through state-of-the-art methods like microservices and design patterns like Event Sourcing and Domain Driven Design.

9 Potential Improvements and Outlook

The prototype delivers a fully working proof of concept, yet several refinements could raise its performance, resilience, and usability. This final section outlines concrete improvement ideas, as well as ideas for applying the same architectural concept in other domains.

9.1 Potential Improvements

A first enhancement concerns job scheduling. Currently, tasks are distributed in round-robin fashion, unaware of hardware differences between nodes. RabbitMQ already supports message priorities and consumer prefetch limits. Combining those features with per-node metrics would let the scheduler push long jobs to fast hosts and bursty workloads to idle ones, reducing the observed execution time spikes. A CPU-aware scheduler could keep track of each node's benchmark score, track historical job execution times, or poll real-time CPU and memory metrics, then route new jobs accordingly. Faster hosts would receive heavier workloads, and slower machines would pick up lighter tasks, reducing tail latency and smoothing the execution-time curve observed in section 7.2.1.
Also, in the current deployment configuration, RabbitMQ is a single point of failure. Running a mirrored-queue cluster across two or three hosts would harden the message backbone against outages. Failover should be load-tested with rolling broker restarts to verify that no job events are lost. This change is purely configurational: there is no architecture or logic adjustment needed.
In the local network at the ISEMP that was also used for testing, Samba shares work well, but add Windows-specific administration overhead. An S3-compatible object store would bring built-in versioning and shadow-copies, lifecycle policies, and HTTP access, simplifying backup and audit. A switch to object storage would also allow the web front end to stream logs directly through signed URLs instead of mounting CIFS volumes.
The orchestrator service already decouples job execution through Docker containers or native AmFem binaries from the rest of the application, but I believe there is more room for improvement as well. Additional node types using different orchestrator services could, e.g. start Kubernetes Pods, provision individual jobs that can run on multiple nodes at once and share resources, or extend containers with GPU-acceleration support. Such improvements would open the door to new, more advanced simulation code or data-processing logic.
Stress testing can also be improved. Tools like *FIRESTARTER* can drive memory bandwidth to its limits and expose bottlenecks that the current CPU-bound micro-containers miss. Additionally, synthetic benchmarks that saturate AVX2 or AVX-512 instruction units would reveal whether the event stream copes with long stretches of maximum CPU consumption.
Cloud deployment on a managed Kubernetes service would showcase elastic auto-scaling at node counts significantly higher than the four used in testing. Autoscaling rules could use both event-queue depth and total cluster CPU utilisation to add or remove workers on

demand, minimizing hardware costs.
Wherever possible, future work should rely only on software components that run natively in containers. Introducing special cases like the `AmFemNode` complicates the stack and loses many of the benefits of this architecture concept and containerisation in general. If Windows-only binaries remain essential, isolating them in dedicated Windows containers or wrapping them behind an API gateway could restore consistency. However, for this change to be possible additional adjustments to the AmFem code need to be made.
High-availability deployments of Keycloak, PostgreSQL, RabbitMQ, and SimApp would remove the last single points of failure. Running each critical service with at least two replicas and a load balancer would keep the platform online during maintenance or unexpected crashes.
Wherever unnecessary, removing optional features can lower complexity. If administrators decide that a simpler file-management interface or fewer job states are acceptable, the corresponding UI, event types, and database tables can be pruned, shortening the learning curve for new developers and initial development time with new projects.
Finally, regarding user-experience, there are also some improvement ideas. Parameter dropdowns with pre-configured selections, shareable links to previously executed jobs, and an integrated file management interface with features like drag-and-drop file upload would make the web front end significantly easier to use. Implementing this would allow the switch to S3 storage, which would further streamline file handling because users could upload directly to the object store without first copying data onto a share.

9.2 Outlook

Despite its complexity, the architecture is mature, coherent, and extensible. It offers a strong template for future research projects that demand on-premise or hybrid-cloud simulation capacity. Because the system depends on an event stream rather than direct RPC calls, the same pattern can also be reused for data-collection pipelines, sensor networks, or batch workloads.
The technology choices (Docker, Event Sourcing, RabbitMQ, .NET 8, and DDD) proved both versatile and nicely compatible with each other. Other developers can adopt this architectural concept confidently, the layers integrate cleanly. As for difficulties with initial adoption, a simplified reference implementation, accompanied by diagrams and ready-made benchmarking scripts, would give rookie developers a head start on their own distributed applications.
Task-routing algorithms remain an open research topic. Logic based on current queue length, recent throughput, or predictive modelling of job runtimes could result in better node utilisation. Machine-learning techniques could be used to learn which node finishes a given tool fastest and adjust routing in real time.

Alternative data-exchange methods deserve exploration. As already mentioned, instead of copying files through SMB, clients could upload inputs to an object store and request signed URLs for retrieval. Additional HTTP or gRPC APIs would allow the platform to more responsively push partial results back to the UI during job execution without the need for polling.

The deployment setup could also be expanded to span multiple LANs. VPN-linked Docker networks or even Kubernetes service meshes (ISTIO provides a state-of-the-art solution [9]) would open up new possibilities for cluster expansion using mutliple data centers. Such a wide-area cluster also introduces new potential problems regarding latency, bandwidth, and dynamic replica placement, providing more opportunities for new research.

Furthermore, a Kubernetes foundation would simplify network virtualisation, rolling upgrades, and replica management. Features like Horizontal Pod Autoscaler, node taints, and topology-aware scheduling fit perfectly with some of the requirements discussed here. Kubernetes would enable more sophisticated resource control out of the box.

In summary, the work presented in this thesis lays a solid groundwork for new, state-of-the-art distributed computing platforms. With some of the mentioned improvements and further testing on even larger clusters, the architectural concept has the potential to be the foundation for a production-ready service platform that meets the demanding needs of research institutions and industrial partners alike. Fulfilling all desired features defined in the beginning of this thesis, the novel architectural concept developed, implemented, and discussed in this thesis proved to be innovative, capable, flexible, and fault-tolerant. Incorporating state-of-the-art methods, technologies, and design patterns, this thesis could be the basis for the next generation of software architectures built using the principle of distributed computing.

10 Bibliography

References

[1] Amazon Web Services. Monolithic vs Microservices - Difference Between Software Architectures. https://aws.amazon.com/compare/the-difference-between-monolithic-and-microservices-architecture/, 2023. [Online; accessed 15.03.2025].

[2] C. Anderson. The Model-View-ViewModel (MVVM) Design Pattern. In *Pro Business Applications with Silverlight 5*, pages 461–499. Springer, 2012.

[3] Arashtad. A Complete Guide to Microservice Architecture. https://medium.com/@arashtad/a-complete-guide-to-microservice-architecture-91de2410dda2, 2024. [Online; accessed 28.03.2025].

[4] M. Bach-Nutman. Understanding the top 10 owasp vulnerabilities. *arXiv preprint arXiv:2012.09960*, 2020.

[5] O. Bentaleb, A. S. Z. Belloum, A. Sebaa, and A. El-Maouhab. Containerization Technologies: Taxonomies, Applications and Challenges. *Journal of Supercomputing*, 78(2):1144–1181, 2022.

[6] A. Berti, W. van der Aalst, D. Zang, and M. Lang. An Open-Source Integration of Process Mining Features into the Camunda Workflow Engine: Data Extraction and Challenges, 2020.

[7] C. Boettiger. An Introduction to Docker for Reproducible Research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, 2015.

[8] B. Burns, J. Beda, and K. Hightower. *Kubernetes*. Dpunkt Heidelberg, Germany, 2018.

[9] L. Calcote and Z. Butcher. *Istio: Up and running: Using a service mesh to connect, secure, control, and observe*. O'Reilly Media, 2019.

[10] R. Chen, S. Li, and Z. Li. From Monolith to Microservices: A Dataflow-Driven Approach. In *Proc. 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 466–475, 2017.

[11] Codex. Building Polyglot Microservices: Using Different Languages for Different Services. https://medium.com/codex/building-polyglot-microservices-using-different-languages-for-different-services-5f6e4725cc78, 2024. [Online; accessed 26.03.2025].

[12] Cortex. Monoliths vs. Microservices: Pros, Cons, & Key Considerations. https://www.cortex.io/post/monoliths-vs-microservices-whats-the-difference, 2023. [Online; accessed 15.03.2025].

[13] J. Deacon. Model-View-Controller (MVC) Architecture. *JOHN DEACON Computer Systems Development, Consulting & Training*, 28:1–6, 2009.

[14] A. Dębski, B. Szczepanik, M. Malawski, S. Spahr, and D. Muthig. In search of a scalable and reactive architecture of a cloud application: CQRS and event sourcing case study. *IEEE Software*, 35(2):62–71, 2018.

[15] D. Dossot. *RabbitMQ Essentials*. Packt Publishing Ltd, 2014.

[16] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: Yesterday, Today, and Tomorrow. In M. Mazzara and B. Meyer, editors, *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[17] R. Dua, A. R. Raja, and D. Kakadia. Virtualization vs Containerization to Support PaaS. In *2014 IEEE International Conference on Cloud Engineering*, pages 610–614, 2014.

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

[19] B. Fedoryshyn. Strategies for Implementing or Strengthening the DevOps Approach in Organizations: Analysis and Examples. *BULLETIN of Cherkasy State Technological University*, 29(2):57–69, 2024. Available at: https://er.chdtu.edu.ua/bitstream/ChSTU/5087/1/7.pdf.

[20] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An Updated Performance Comparison of Virtual Machines and Linux Containers. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172, 2015.

[21] A. Fernandez. Camunda BPM platform loan assessment process lab. *Brisbane, Australia: Queensland University of Technology*, 124, 2013.

[22] N. Garg. *Apache kafka*. Packt Publishing Birmingham, UK, 2013.

[23] B. L. Gorman. *Unit Testing, Integration Testing, and Mocking*, pages 575–614. Apress, Berkeley, CA, 2022.

[24] D. Hackenberg, R. Oldenburg, D. Molka, and R. Schöne. Introducing firestarter: A processor stress test utility. In *2013 International Green Computing Conference Proceedings*, pages 1–9, 2013.

[25] C. Herath. Why Microservices Should Use Event Sourcing. *Bits and Pieces*, 2022. [Online; accessed 29.03.2025].

[26] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley Professional, 2010.

[27] ITNEXT. Domain-Driven Design and Micro Frontends: Organizing Autonomous Teams for Collaborative Success. `https://itnext.io/domain-driven-design-and-micro-frontends-organizing-autonomous-teams-for-collaborative-success-1aa6b5f327db`, 2023. [Online; accessed 02.04.2025].

[28] M. Kofle and B. Öggl. *Docker: Das Praxisbuch für Entwickler und DevOps-Teams*. Rheinwerk Verlag, Bonn, Germany, 3 edition, 2021.

[29] J. Kreps, N. Narkhede, and J. Rao. Kafka: A Distributed Messaging System for Log Processing. In *Proc. NetDB Workshop (Networked Systems for Developing Regions)*, 2011.

[30] M. Kulenovic and D. Donko. A survey of static code analysis methods for security vulnerabilities detection. In *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1381–1386. IEEE, 2014.

[31] LambdaTest. Monolithic vs Microservices Architecture: Advantages and Disadvantages. `https://www.lambdatest.com/blog/monolithic-vs-microservices-architecture/`, 2023. [Online; accessed 15.03.2025].

[32] K. Larkin, S. Smith, and B. Dahler. Dependency injection in ASP.NET Core. `https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-8.0`, 2024. [Online; accessed 24.03.2025].

[33] Y. Li, Y. Lin, Y. Wang, K. Ye, and C. Xu. Serverless Computing: State-of-the-Art, Challenges and Opportunities. *IEEE Transactions on Services Computing*, 16(2):1522–1539, 2023.

[34] P. Michail and C. Kalloniatis. Object Relational Mapping Vs. Event-Sourcing: Systematic Review. In *Electronic Government and the Information Systems Perspective (EGOVIS 2022), Lecture Notes in Computer Science, vol. 12429*, pages 18–31. Springer, 2022.

[35] Microsoft Learn. Microservice Architecture Style. `https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices`, 2025. [Online; accessed 28.03.2025].

[36] J. Murty. *Programming amazon web services: S3, EC2, SQS, FPS, and SimpleDB*. O'Reilly Media, Inc., 2008.

[37] S. Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015.

[38] P. Ninpetch, P. Kowitwarangkul, S. Mahathanabodee, P. Chalermkarnnon, and P. Ratanadecho. A review of computer simulations of metal 3D printing. *AIP Conference Proceedings*, 2279(1):050002, 10 2020.

[39] OpenLegacy. Monolithic vs Microservices Architecture: Pros and Cons. `https://www.openlegacy.com/blog/monolithic-application`, 2023. [Online; accessed 15.03.2025].

[40] F. Rademacher, J. Sorgalla, and S. Sachweh. Challenges of Domain-Driven Microservice Design: A Model-Driven Perspective. *IEEE Software*, 35(3):36–43, 2018.

[41] D. Recordon and D. Reed. OpenID 2.0: A platform for user-centric identity management. In *Proc. 2nd ACM Workshop on Digital Identity Management (DIM)*, pages 11–16, 2006.

[42] G. M. Roy. *RabbitMQ in Depth*. Simon and Schuster, 2017.

[43] B. Ruecker. *Practical Process Automation: Orchestration and Integration in Microservices and Cloud Native Architectures*. O'Reilly Media, 2021.

[44] SAYONE. Domain Driven Design for Microservices: Complete Guide 2025. `https://www.sayonetech.com/blog/domain-driven-design-microservices/`, 2023. [Online; accessed 29.03.2025].

[45] A. e. a. Schwartz. Multi-site deployments. `https://www.keycloak.org/high-availability/introduction`, 2025. [Online; accessed 02.02.2025].

[46] M. Shahin, M. A. Babar, and L. Zhu. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, 5:3909–3943, 2017.

[47] TechTutorTips. Polyglot Microservices. `https://www.techtutortips.com/post/polyglot-microservices`, 2025. [Online; accessed 26.03.2025].

[48] S. Thorgersen and P. I. Silva. *Keycloak-identity and access management for modern applications: harness the power of Keycloak, OpenID Connect, and OAuth 2.0 protocols to secure applications*. Packt Publishing Ltd, 2021.

[49] TrustSoft. Infrastructure as Code with AWS CloudFormation and Terraform. https://www.trustsoft.eu/blog/aws-infrastructure-as-a-code-with-aws-cloudformation-and-terraform, 2023. [Online; accessed 02.04.2025].

[50] V. Vernon. *Implementing Domain-Driven Design*. Addison-Wesley, 2013.

[51] M. Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer, 2nd edition, 2019.

[52] J. Williams. *RabbitMQ in Action: Distributed messaging for everyone*. Simon and Schuster, 2012.

A Appendix

A.1 File System Overview

```
.
├── README.md
├── thesis.pdf                    ← Digital PDF of this thesis
├── code
│   ├── Services                  ← This folder contains the necessary files required by external services like PostgreSQL
│   │   ├── data
│   │   │   ├── postgres
│   │   │   ├── rabbitmq
│   │   │   │   ├── data
│   │   │   │   └── log
│   │   │   └── samba
│   │   │       ├── input
│   │   │       ├── logs
│   │   │       └── output
│   │   └── docker-compose.yaml ← Compose file only containing services needed for debugging via VS 22, no SimApp or nodes
│   ├── SimApp                    ← Main SimApp solution and its sub-projects, also contains cluster node projects
│   │   ├── README.md
│   │   ├── SimApp
│   │   │   ├── Dockerfile        ← Each project contains its own Dockerfile used for building its corresponding Docker image
│   │   │   ├── appsettings.json ← The appsettings.json files contain configuration parameters of each corresponding service
│   │   │   ├── ...
│   │   ├── SimApp.AmFemNode
│   │   │   ├── Dockerfile        ← Non-functional Dockerfile, it remains from a failed attempt to containerize this project
│   │   │   ├── amFem             ← Windows binaries of the AmFem application, used by the service to run AmFem-based tasks
│   │   │   │   └── ...
│   │   │   ├── publish           ← Folder containing published binaries for native execution on Windows
│   │   │   │   └── ...
│   │   ├── SimApp.ClusterNode
│   │   │   ├── Dockerfile
│   │   │   ├── appsettings.Development.json ← These Development jsons are used specifically for debugging via VS 22
│   │   │   ├── appsettings.json
│   │   │   ├── docker-compose.yml ← This Compose file creates a single linux node on a machine without any other containers
│   │   │   └── ...
│   │   ├── SimApp.Domain
│   │   │   └── ...
│   │   ├── SimApp.Persistence
│   │   │   └── ...
│   │   ├── SimApp.Tests
│   │   │   ├── ...
│   │   │   ├── setups
│   │   │   │   └── singlenode.docker-compose.yml ← This Compose file adds a linux node to a running SimApp deployment
│   │   ├── docker-compose.yml ← Compose only containing what's missing in the yml file at line 15 (SimApp & one Linux node)
│   │   └── ...
│   ├── SimTools                ← The SimTools solution contains the test tools developed for testing the project
│   │   ├── ...
│   │   ├── SimTool.OutputfileTool
│   │   │   ├── Dockerfile
│   │   │   ├── ...
│   │   ├── SimTool.RandomFailer
│   │   │   ├── Dockerfile
│   │   │   ├── ...
│   │   ├── SimTool.TestTool
│   │   │   ├── Dockerfile
│   │   │   ├── ...
│   │   ├── SimTool.AmFem.Dilatometry ← Project containing code that can be used for deploying an AmFem-based tool
│   │   │   ├── ...
│   │   └── SimTool.sln           ← Solution file for the SimTool solution, open this with VS 22 to develop, or debug
│   └── docker-compose.yml        ← Main compose file containing the full service stack for local launch of the platform
└── thesis
    └── ...
```

The files accompanying this thesis are organised to separate source code, service components, test utilities, and documentation. The compiled PDF of this thesis is stored next

to the root README. The `code` directory contains the full source code, as well as configurations and Compose files required to debug, deploy and test the prototype. Within it the `Services` subfolder holds a Compose file and files for the volume mounts for external services: PostgreSQL, RabbitMQ, and a Samba server that exposes the *input*, *logs*, and *output* shares so the platform can be run entirely locally without the need for a dedicated Samba server. The Compose file (`docker-compose.yml`) is configured to start only these base services plus the Samba helper container; it is useful when debugging SimApp or a cluster node in Visual Studio, because those projects can be launched directly from the VS 22 IDE while the external services run in the background.

The largest folder is `SimApp`. It contains a solution that bundles the main .NET web application (`SimApp`), the Linux cluster node (`SimApp.ClusterNode`), the Windows-only AmFem node (`SimApp.AmFemNode`), the project containing shared domain models (`SimApp.Domain`), the persistence layer (`SimApp.Persistence`), and a unit-test project (`SimApp.Tests`). Each microservice is a separate project, therefore every project folder has a dedicated `Dockerfile` used for building the corresponding Docker image. General configuration is done in `appsettings.json` files, while a matching `appsettings.Development.json` can be used to override configuration parameters like connection strings and log levels for local debugging. Because the AmFem node cannot run in a container, the `SimApp.AmFemNode` project still includes an experimental Dockerfile that remains from an earlier attempt at containerisation; until the AmFem code gets fixed, the actual executable should launched natively.

Inside `SimApp.Tests` the `setups` directory offers a small helper compose file. The file `singlenode.docker-compose.yml` adds a standalone Linux cluster node to a running SimApp instance so that integration tests can be run with multiple nodes on the same computer. This was not used in the section 7 tests, because there were multiple computers with heterogeneous hardware available. It was determined that tests run on a cluster deployed on multiple different computers would be significantly more meaningful. At the root of the SimApp solution a second `docker-compose.yml` can be used to deploy the two microservices missing from the Compose file in line 15: SimApp, and one linux cluster node. This way individual microservices can be deployed additional to the Compose deployment in the `Services` folder.

All containerised test tools reside under `SimTools`. Each test tool, such as `RandomFailer`, `OutputfileTool`, or `TestTool`, has its own project folder and Dockerfile. Also, this folder contains the test project `AmFem.Dilatometry`, which contains code for deploying AmFem container-based and as a dedicated tool, but because of container compatibility issues with AmFem this does not work currently.

Finally, a top-level `docker-compose.yml` contains every microservice needed for deployment of the platform in one single Compose file: PostgreSQL, RabbitMQ, the mock Samba, SimApp, and one Linux cluster node. **Running `docker compose up -d` at the repository**

root starts the entire platform with the preconfigured defaults, enabling developers to launch the prototype, sign in with a test account, and execute simulations in minutes. This makes it possible to deploy the whole platform locally with a single command.
The directory structure therefore reflects the layered architecture described in the thesis: external services are isolated and can be launched separately, application code is grouped by bounded context even through file structure, and test tools are kept separate. Each folder contains the components needed to build or debug its contents independently.

A.2 Docker Compose File Containing Full Service Stack

```
services:
  simapp:
    image: gitlab.informatik.uni-bremen.de:5005/alpay1/masterarbeit/simapp:latest
    container_name: simapp
    hostname: simapp
    ports:
      - 5001:8080
    cap_add:
      - SYS_ADMIN
      - DAC_READ_SEARCH
    networks:
      - simapp
    user: root
    environment:
      - Container__OutputDirectory=//samba/SimAppStorage/output
      - Container__LogsDirectory=//samba/SimAppStorage/logs
      - Container__StatusTimeout=30
      - Container__RetryTimeout=120
      - ImageRepository__HostUrl=gitlab.informatik.uni-bremen.de
      - ImageRepository__Username=guest
      - ImageRepository__AccessToken=ypsfjuzyuLtv4oGPyHLG
      - ImageRepository__Project=alpay1/masterarbeit
      - EventStream__HostName=rabbitmq
      - EventStream__Username=guest
      - EventStream__Password=guest
      - EventStream__Exchange=simapp
      - EventStream__VirtualHost=/
      - EventStream__Port=5672
      - NetAuth__Username=guest
      - NetAuth__Password=guest
      - Keycloak__OidcUrl=http://keycloak:8080/realms/master/protocol/openid-
          connect/token
      - Keycloak__ClientId=simapp
      - Keycloak__Scope=openid
      - Keycloak__Issuer=http://keycloak:8080/realms/master
      - Keycloak__Authority=http://keycloak:8080/realms/master
      - Keycloak__AccessExpiration=30
      - Keycloak__RefreshExpiration=60
      - Keycloak__Secret=oznCDc1uPWu2f4hZgxUX9nwGCm29taaL
    restart: always

  linuxnode:
    image: gitlab.informatik.uni-bremen.de:5005/alpay1/masterarbeit/clusternode:
        latest
    container_name: linuxagent
    hostname: linuxagent
    cap_add:
```

```
      - SYS_ADMIN
      - DAC_READ_SEARCH
    networks:
      - simapp
    volumes:
      - /var/run/docker.sock.raw:/var/run/docker.sock
    user: root
    environment:
      - EventStream__HostName=rabbitmq
      - EventStream__Username=guest
      - EventStream__Password=guest
      - EventStream__Exchange=simapp
      - EventStream__VirtualHost=/
      - EventStream__Port=5672
      - Task__Platform=linux
      - Task__AgentName=Linux Agent 1
      - ImageRepository__HostUrl=gitlab.informatik.uni-bremen.de
      - ImageRepository__Username=guest
      - ImageRepository__AccessToken=ypsfjuzyuLtv4oGPyHLG
      - ImageRepository__Project=alpay1/masterarbeit
      - NetAuth__Username=guest
      - NetAuth__Password=guest
    restart: always

  postgres:
    image: postgres:17-alpine
    container_name: postgres
    hostname: postgres
    ports:
      - 5432:5432
    networks:
      - simapp
    volumes:
      - ./Services/data/postgres/:/var/lib/postgresql/data
    environment:
      - POSTGRES_PASSWORD=testing
      - POSTGRES_USER=simapp
      - POSTGRES_DB=simapp
    restart: always

  keycloak:
    image: quay.io/keycloak/keycloak:26.1
    command: start-dev --hostname http://localhost:8080/ --hostname-admin http://
        localhost:8080/ --hostname-backchannel-dynamic true
    networks:
        - simapp
    container_name: keycloak
    hostname: keycloak
    environment:
      KC_HTTP_ENABLED: true
      KC_HEALTH_ENABLED: true
      KEYCLOAK_ADMIN: admin
      KEYCLOAK_ADMIN_PASSWORD: simapp
      KC_DB: postgres
      KC_DB_URL: jdbc:postgresql://postgres/keycloak
      KC_DB_USERNAME: keycloak
      KC_DB_PASSWORD: keycloak
    ports:
      - 8080:8080
    depends_on:
      - postgres
    restart: always
```

```yaml

  rabbitmq:
    image: rabbitmq:4-management-alpine
    container_name: rabbitmq
    hostname: rabbitmq
    ports:
        - 5672:5672
        - 15672:15672
    volumes:
        - ./Services/data/rabbitmq/data/:/var/lib/rabbitmq/
        - ./Services/data/rabbitmq/log/:/var/log/rabbitmq
    networks:
        - simapp
    restart: always

  samba:
    image: dperson/samba
    container_name: samba
    hostname: samba
    environment:
      TZ: 'ETC\UTC'
    networks:
      - simapp
    read_only: true
    tmpfs:
      - /tmp
    restart: unless-stopped
    stdin_open: true
    tty: true
    volumes:
      - ./Services/data/samba:/mnt:z
    command: '-s "SimAppStorage;/mnt;yes;no;yes;all" -u "guest;guest"'

networks:
  simapp:
    name: simapp
    driver: bridge
```

The `docker-compose.yml` file in the projects top directory launches the entire prototype with one command. Every container joins a custom bridge network called `simapp`. This network lets services discover one another by hostname without exposing unnecessary ports to the host.

SimApp The `simapp` service runs the ASP.NET Core web application and job supervisor. Its image is pulled from the GitLab container registry under the `alpay1/masterarbeit` project. Port 8080 inside the container is mapped to port 5001 on the host so users can open the UI at `http://localhost:5001`. Two container capabilities, `SYS_ADMIN` and `DAC_READ_SEARCH`, are added so the process can mount CIFS shares and inspect file permissions. Environment variables specify output and log directories, RabbitMQ settings, Keycloak endpoints, and registry credentials.

Linux Cluster Node The `linuxnode` service represents a generic worker that pulls tool images and launches job containers. It mounts the host's Docker socket at `/var/run/docker.sock`, allowing nested container creation. The same capabilities ap-

plied to SimApp are granted here to enable file-system operations inside job containers. An agent name and platform tag help SimApp identify and schedule tasks for this node.

PostgreSQL Database The `postgres` service uses the official `postgres:17-alpine` image. A bind mount under `./Services/data/postgres` persists the database. Initial credentials create a `simapp` database owned by the `simapp` user with password `testing`.

Keycloak Identity Provider Keycloak runs in `start-dev` mode, which enables HTTP access on port 8080 and omits HTTPS configuration just for local testing. It depends on the Postgres container because Keycloak stores realm data in its own schema. Admin credentials are set to `admin/simapp` for convenience.

RabbitMQ Broker RabbitMQ is started with the `management-alpine` tag, giving access to the web dashboard on port 15672. Data and log directories are persisted under `./Services/data/rabbitmq`. SimApp and the cluster node both connect to RabbitMQ using the default `guest/guest` account and publish to the `simapp` exchange.

Samba File Share The `samba` service exports a single share named `SimAppStorage`. The share maps to `./Services/data/samba` on the host, which contains the `input`, `logs`, and `output` sub-directories. The container runs read-only except for an in-memory `/tmp` mount, reducing the risk of accidental file-system changes. Anonymous access is enabled with the `guest` user so that containers can mount the share without additional credentials.

Restart Policies and Health Every service except Samba uses `restart: always`. This directive ensures automatic recovery after host reboots or transient crashes. Samba employs `restart: unless-stopped` to avoid unintended restarts during manual maintenance.

Overall Behaviour When `docker compose up -d` is executed, the message broker, database, identity provider, and file share come online first. SimApp then starts, reads its configuration from environment variables, and connects to RabbitMQ and Keycloak. Finally, the Linux cluster node starts up and binds itself to the RabbitMQ exchange for incoming jobs. With all containers healthy, a user can log in through the browser, upload input files to the Samba containers mounted directory, and launch simulations through the SimApp web interface that will then be distributed through RabbitMQ to run inside nested containers orchestrated by the `linuxnode` service.

A.3 Deployment Instructions

To launch the complete platform on a single workstation run `docker compose up -d` in the `code` repository root. Docker pulls every image, creates the `simapp` bridge network, and starts all containers in the correct order. After a few seconds the SimApp dashboard is available at `http://localhost:5001` and the Keycloak admin console at `http://localhost:8080`.

For a multi-host deployment where a dedicated Samba server and several compute nodes already exist use a two-step procedure. First, change into `codes/Services` and run the local compose file found there. This starts PostgreSQL, RabbitMQ, and Keycloak without SimApp or any node containers, because those will run on separate machines. Second, run the `docker-compose.yml` located in `codes/SimApp` using `docker compose up -d` on the same machine if you want to use it as a Linux cluster node as well. Third, copy and run the `docker-compose.yml` in `codes/SimApp/SimApp.ClusterNode` to every additional machine that should join the cluster and run Linux based tools. This brings up a single `clusternode` container, which automatically registers itself with RabbitMQ and begins polling for tasks. The containers read the RabbitMQ and Keycloak addresses from preconfigured environment variables, however, when doing a multi-machine deployment, replace the RabbitMQ hostname with the name of the machine runnning the full service stack.

The AmFem solver cannot run inside a container and must be launched as a native Windows process. Navigate to `codes/SimApp/SimApp.AmFemNode/publish` and open a PowerShell window. Run `.\SimApp.AmFemNode.exe` to start the service. The executable reads its configuration from the `appsettings.json` file, connects to RabbitMQ, and registers itself as an additional worker in the computing cluster. The node window shows a console log; leave it open while AmFem jobs are running.

After all services are online verify the cluster health by logging into the SimApp dashboard. Create a test job that uses the *RandomFailer* or any other test tool. If the job appears in the queue and then moves to the running state, network connectivity, message routing, and shared storage are correctly configured.

A.4 Use of AI-based Applications

	AI-based Tool	Purpose	Aspect of the Work Affected	Example Prompt (Entry)	Comment
1	ChatGPT	Formulation and Grammar	Writing (Text)	"correct the grammar mistakes in this text and reformulate any sentences that are difficult to understand: <text>"	ChatGPT proved to be a great tool to check grammar and re-formulate badly-written sections. Other tools showing similar performance and features were paid; ChatGPT was free.