universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„A Multi-Cloud-Based Blockchain-as-a-Service“

verfasst von / submitted by

Ismail Alhamzeh BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2024 / Vienna, 2024

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066921
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Informatik
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Klas

Acknowledgements

I extend my heartfelt appreciation to my supervisor, Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Klas for his invaluable guidance and support that made this work possible. His expertise and dedication have truly been instrumental throughout this thesis.

I would also like to thank my family for their unwavering encouragement and support. Their belief in me has been a constant source of inspiration, and I am blessed to have them by my side.

Abstract

Blockchain is a shared, immutable ledger that tracks transactions and provides means to establish trust between different parties. However, building and maintaining a production-level infrastructure and hosting a blockchain network for enterprises benefiting from blockchain technology is complex and challenging. Those barriers and challenges are resolved using Blockchain-as-a-Service (BaaS). Such a cloud-based service combines the high availability, elasticity, and high computing power of cloud computing with blockchain technology's decentralization, immutability, and transparency. This technology combination gives enterprises that utilize this technology combination advantages such as reduced costs, scalability, and increased security. This thesis provides an overview of BaaS commercial and academic systems. It provides a comparative analysis based on different characteristics and parameters specified in this thesis's scope. Furthermore, this thesis presents an architectural design and a prototype implementation of a Blockchain-as-a-Service platform that offers multi-cloud, multi-region blockchain framework deployments, a property many commercial Blockchain-as-a-Service providers like AWS or Google Cloud do not provide. Finally, the BaaS platform proposed in this thesis will be evaluated based on user experiments and functional and non-functional requirements evaluation.

Kurzfassung

Blockchain ist ein gemeinsames, unveränderliches Ledger, das Transaktionen verfolgt und Möglichkeiten bietet, Vertrauen zwischen verschiedenen Parteien aufzubauen. Der Aufbau und die Wartung einer Infrastruktur auf Produktionsebene sowie das Hosten eines Blockchain-Netzwerks für Unternehmen, die von der Blockchain-Technologie profitieren, sind jedoch komplex und herausfordernd. Diese Hindernisse und Herausforderungen werden mithilfe von Blockchain-as-a-Service (BaaS) gelöst.

Ein solcher cloudbasierter Dienst kombiniert die hohe Verfügbarkeit, Elastizität und hohe Rechenleistung des Cloud Computing mit der Dezentralisierung, Unveränderlichkeit und Transparenz der Blockchain-Technologie. Diese Technologiekombination bietet Unternehmen, die diese Technologiekombination nutzen, Vorteile wie geringere Kosten, Skalierbarkeit und erhöhte Sicherheit. Diese Arbeit bietet einen Überblick über kommerzielle und akademische BaaS-Systeme. Es bietet eine vergleichende Analyse auf der Grundlage verschiedener Merkmale und Parameter, die im Rahmen dieser Arbeit spezifiziert werden.

Darüber hinaus stellt diese Arbeit einen Architekturentwurf und eine prototypische Implementierung einer Blockchain-as-a-Service-Plattform vor, die Multi-Cloud-, Multi-Region-Blockchain-Framework-Bereitstellungen ermöglicht, eine Eigenschaft, die viele kommerzielle Blockchain-as-a-Service-Anbieter wie AWS oder Google Cloud nicht anbieten. Abschließend wird die in dieser Arbeit vorgeschlagene BaaS-Plattform auf der Grundlage von Benutzerexperimenten und der Bewertung funktionaler und nichtfunktionaler Anforderungen evaluiert.

Contents

List of Tables

List of Figures

Listings

1. Motivation

1.1. Topic Motivation and Problem Description

Blockchain is a shared, immutable ledger that tracks transactions and provides means to establish trust between different parties [53]. What is being transacted can be coin in the case of cryptocurrencies, health data in the case of human health tracking blockchain, or asset ownership (car, house, and land), basically anything that has a value can be tracked and secured through the blockchain which provides trust between different authorities [53].

It is a complex and challenging task to build and maintain a production-level infrastructure that hosts a blockchain network for enterprises that aim to benefit from blockchain technology because it costs a lot to configure, deploy, manage, and do scheduled software updates for the hosted blockchain nodes [48, 18]. Therefore, BaaS - Blockchain-as-a-Service was introduced to make blockchain technology accessible for businesses and enterprises by shifting those complex tasks to the cloud provider and making businesses of different sizes focus on developing blockchain-based applications without the need to have advanced knowledge about the infrastructure [18].

BaaS is a cloud-based service that combines the high availability, elasticity, and high computing power of cloud computing with blockchain centralization, immutability, and transparency [45]. This technology combination gives enterprises that utilize this technology combination many advantages, such as reduced costs, scalability, increased security, faster time-to-market, and access to blockchain experts who can assist with developing blockchain applications.

BaaS is a new concept that has gained public traction in recent years. Microsoft Azure and IBM first offered it, and since then, other cloud providers have also provided their own BaaS solutions [18, 45]. Therefore, it is a relatively new technology, and some issues still need to be discussed and resolved, and optimizations can be done.

Many of the current BaaS products like AWS - Amazon Managed Blockchain [2], Oracle Blockchain [43], and Alibaba Cloud [11] do not provide the ability to deploy a blockchain network instance on multiple cloud providers (multi-cloud deployment: the deployment of the blockchain nodes on various public clouds) or on-premise or in a hybrid way (deploying the blockchain nodes on the cloud and on-premise).

Currently, any Blockchain-as-a-Service platform either does not support multi-cloud and hybrid deployment or supports it using commercial enterprise technologies like Redhat

Openshift or Google Anthos or using its closed-source solution [27, 28, 29, 12, 24].

A Blockchain-as-a-Service provider that allows its users to deploy the blockchain nodes only on their infrastructure creates cloud provider dependency issues and drawbacks, which can be classified into the following four categories of challenges:

1. Availability: If the public provider data centers experience an outage for any reason, then the BaaS platform becomes unavailable for businesses. However, only a specific number of nodes will be unavailable when the blockchain nodes are distributed among multiple infrastructures (cloud providers).

2. Provider Lock-In: It becomes a challenging task to migrate the blockchain nodes to a different infrastructure or cloud provider.

3. Centralization: Making a whole blockchain network depending on a single cloud provider, introduces much centralization.

4. Security: If the provider experiences a data leak or security breach, the whole BaaS instance may be compromised.

These challenges and BaaS requirements motivate us to research and find a solution focusing on decentralization, high availability, and preventing vendor lock-in of a Blockchain-as-a-Service platform. Such a solution is feasible by making the platform support multi-cloud or hybrid blockchain deployments, which is possible by designing a control plane layer that orchestrates all the different required operations that vary from provisioning infrastructure from a single cloud provider and ending by connecting multiple infrastructures and utilize them for hybrid and multi-cloud deployments.

1.2. Goals

This master's thesis aims to compile a comprehensive state-of-the-art analysis of the current Blockchain-as-a-Service platform solutions by understanding architectural solutions, identifying significant issues and challenges, and proposing a more advanced approach to BaaS, including a prototypical implementation of the proposed method.

The work to be done can be organized into the three subsequent major research questions:

1. What are the current research advances in BaaS providers, challenges, and architectures?

2. What are the various architectures and solutions for a Blockchain-as-a-Service platform that effectively address the issues and challenges mentioned in the first section, particularly those related to availability, centralization, and provider lock-in?

3. How can the answers to the previous research sub-questions be used to create a Blockchain-as-a-Service prototype that addresses most of the mentioned issues and challenges?

These three subsequent research questions can be defined as the following concrete research goals:

1. The comparison of different commercial Blockchain-as-a-Service (BaaS) providers based on specific characteristics thoroughly specified throughout this thesis: supported blockchain types and frameworks, use cases, architecture, pricing, vendor lock-in, and multi-cloud/on-premise deployment support.

2. The comparison of different academic BaaS solutions based on specific characteristics that were thoroughly specified throughout this thesis, which are: proposed architecture, to which area in BaaS they contribute (decentralization, deployment, adoption, security, and utilization of BaaS), vendor lock-in and multi-cloud/on-premise deployment support.

3. Design a Blockchain-as-a-Service solution that uses the benefits of different analyzed BaaS commercial providers and architectures, focusing on scalability, decentralization, high availability, and enabling multi-cloud and multi-region blockchain network deployments. The design must be done as a specification to enable its adoption and implementation by various frameworks and technologies.

4. Implement a concrete Blockchain-as-a-Service solution and control plane that is specialized for blockchain network deployments. The implementation must enable:
   a) Dynamic provisioning and managing of multiple kubernetes clusters from various cloud providers.
   b) Secure communication between the provisioned clusters.
   c) Single-cloud and multi-cloud-based blockchain node deployment.
   d) Integration with Google Cloud and Microsoft Azure, with the ability to integrate additional cloud providers.
   e) Usage of the custom blockchain framework, with the ability to integrate additional blockchain frameworks.

1.3. Work Process and Used Methods

The work process started by exploring the current Blockchain-as-a-Service research status and how it evolved — followed by analyzing and researching the related work by reviewing the BaaS academic and commercial providers and comparing them based on carefully selected parameters and characteristics based on the intersection of the presented parameters by each commercial provider or academic architecture since some parameters are

present in some providers but hidden in the others.

After that, we continued analyzing the different possibilities and technologies that enable us to achieve a decentralized, highly available, and no vendor-locked-in BaaS control plane that allows users to manage Kubernetes clusters across different cloud providers dynamically, provide the ability to configure and connect those clusters, and manage blockchain workloads across those distributed clusters.

After researching and analyzing the literature and suitable technologies, we started designing the BaaS multi-cloud specification, considering all the functional and non-functional requirements the end solutions must fulfill, iteratively refine, and improve based on the ongoing research and supervisor's feedback. After that, approach a meaningful design that satisfies all the requirements and resolves all the challenges the solution has to fulfill by implementing the prototype and iteratively updating it based on the applied design changes.

2. Related Work

2.1. Literature Background and Related Work

2.1.1. Blockchain

Blockchain Architecture - Block Structure

Blocks contain a set of transactions that are verified and will cause the blockchain state to change. Each block is linked and chained with the previous block hash, creating a chain of immutable blocks since the blocks are linked together with their hash [49].

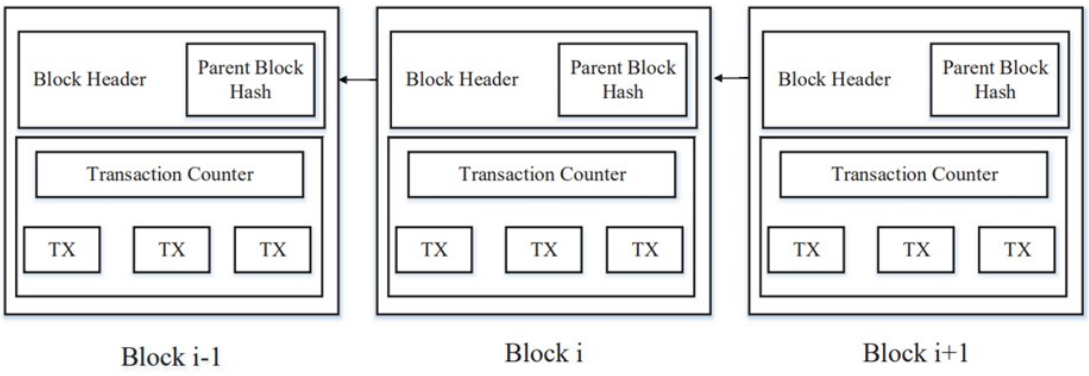

Figure 2.1.: Generic Block Structure. Source: [55].

Figure 2.1 shows a generic overview of the block structure in a blockchain. The following attributes are present in each block:

1. Block hash: Block identifier.
2. Previous block hash: Previous block identifier.
3. Merkle root: Calculated Merkle root for transactions verified and accepted in this block.
4. State root (present in the account-based-model blockchain): Calculated Merkle root for the state (blockchain accounts) after applying a batch of transactions (block) to the blockchain.

5. Validator/Miner Identifier: The identifier of the block validator or miner depends on the blockchain type and consensus mechanism.

6. Height: Height of block.

7. Transactions: Transactions batch that will be appended and accepted to the blockchain and change its state.

8. Timestamp: Block creation timestamp.

Blockchain Architecture - Transaction Structure

As explained in [49, 55], a blockchain transaction refers to an action triggered by a user and causes a change in the blockchain state (in the case of account-based-model blockchain).

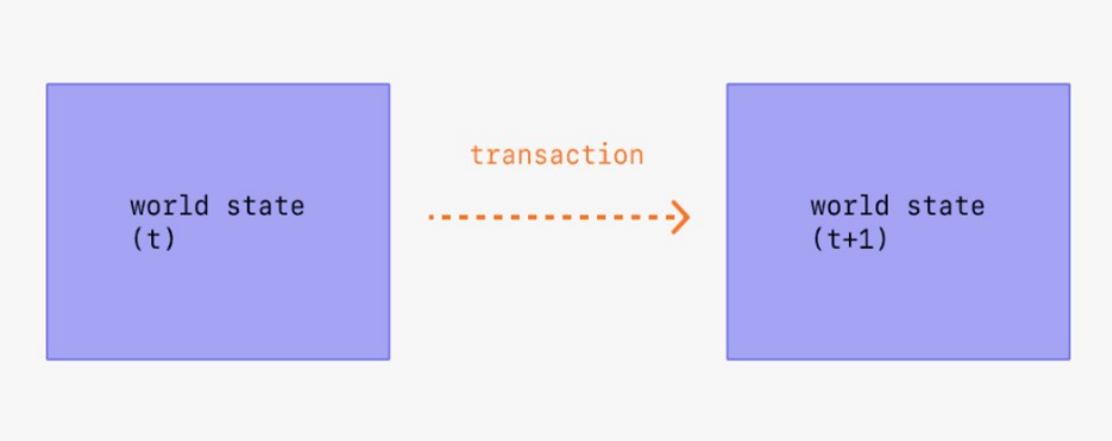

Figure 2.2.: Blockchain Transaction. Source: [49].

Each block in the blockchains contains a set of accepted transactions that will be appended to the ledger and cause the world state of the blockchain to be changed (Figure 2.2). The miner/validator nodes verify these transactions and validate when the other nodes receive the block before accepting the new one.

Depending on the blockchain type (public or private), these transactions may require a fee to get executed and accepted [49], and they change the blockchain state in case the Account Model is used.

The following attributes are usually present in different blockchain types (public, private, and hybrid) [49, 41]:

1. Sender: The sender address - the sender's public key.

2. Receiver: The receiver address - the receiver's public key.

3. Signature: The sender identifier - the sender signs the transaction with its private key to verify that he and only he created the transaction.

4. Nonce: Incrementing number to avoid duplicate transactions. Whenever a user creates a transaction, it gets incremented.

5. Timestamp: Transaction creation timestamp.

6. Transaction Hash: An identifier that can be used to locate a transaction.

7. Value: The amount of money sent from the sender to the receiver.

8. Other attributes may be included depending on the blockchain and use case requirements.

Blockchain Account Models

UTXO Model - Unspent Transaction Output Model: UTXO, the unspent transaction output model that is utilized in bitcoin [41], means that when a sender creates a transaction, then he uses one or more of his unspent transactions as input to his transaction when this transaction is confirmed and added to the blockchain. The receiver can use the outputs of this transaction.

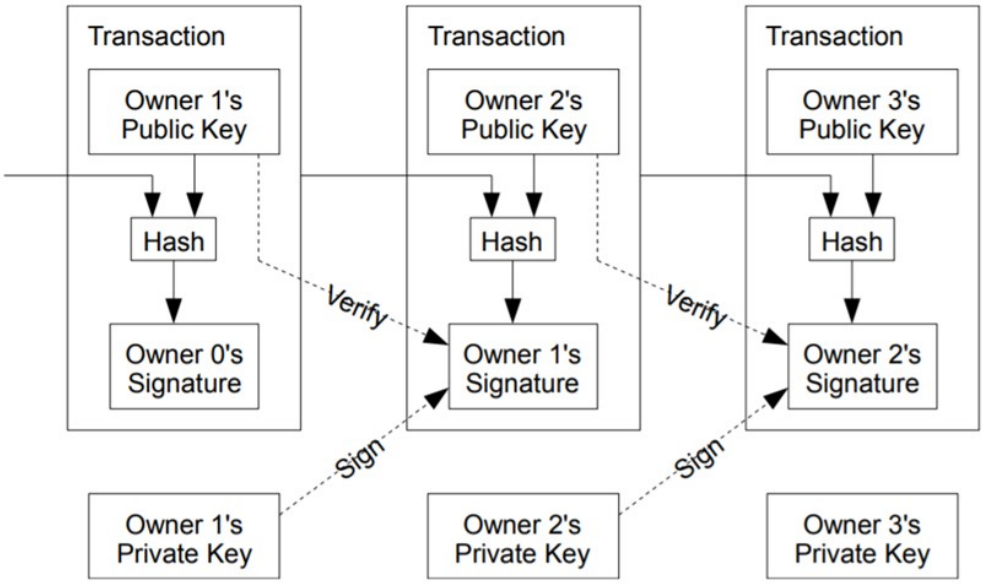

Figure 2.3.: Blockchain UTXO Account Model. Source: [55].

As Figure 2.3 shows, the sender can spend his unspent transactions by signing the hash of the last transaction with the receiver's public key and private key. In this way, the coins are transmitted from the sender to the receiver account, and only the sender can spend the new transaction output by signing it with the new receiver's public key [41, 50].

The problem is how the network can check for double-spending. A simple solution would be for a central authority to check each new incoming transaction to see if it is double-spent or not. However, in a peer-to-peer system, the solution would be to make all the nodes agree on the earliest incoming transaction to prevent double-spending. Because of the enormous number of transactions, keeping all the peers in sync after each transaction is complicated, so transactions are grouped into a block [41]. It is generally recommended to use the UTXO model when the transaction logic is simple.

Account Model: The account-based model is based on the idea of the bank account, where the state is a set of accounts with a balance, public key, and other attributes (Figure 2.4). There are two types of accounts:

1. The first is an account with a public and private key used to sign transactions.

2. The second is an account with a contract code specifying when a transaction/transactions must be created/confirmed [49].

Account attributes [49, 50]:

1. Nonce: A number that can be incremented whenever the account owner creates a transaction. This number helps the network to get information about the account (how many transactions are done by this account), and it also prevents the receiver from duplicating the transaction without the sender knowing.

2. Balance – The current account balance. Balance is always checked when an account tries to create a transaction, and the transactions will be rejected if there is insufficient balance in the sender account.

3. Code Hash (in the case of Ethereum): An account's code.

4. Public key: It is used as an account identifier of a node and can also be used to verify that the sender's private key creates a transaction.

5. Storage Root (in the case of Ethereum): The hash of the root in the Merkle tree.

6. Private key: Used to sign the transaction when the sender account creates one. In this way, no one other than the sender's account owner can create and sign a transaction.

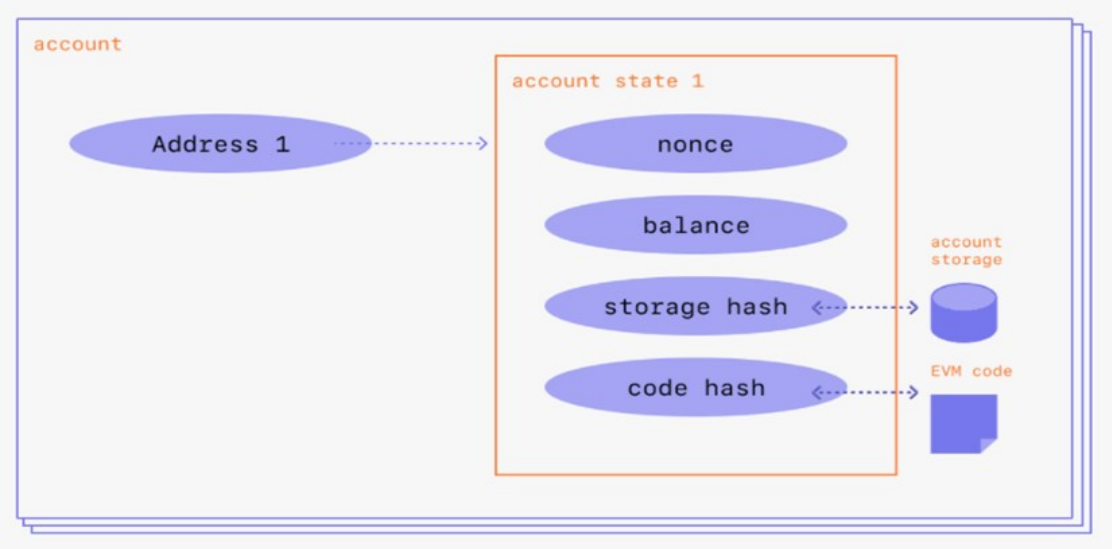

Figure 2.4.: Blockchain Account Model. Source: [49].

Merkle Tree

A Merkle tree, shown in Figure 2.5, is a one-way tree constructed starting from the children's nodes and going up until the root node is constructed. It is used in many applications within cryptography, and it provides a fingerprint for the data set [31].

With the Merkle Tree [31], two different nodes can verify for each other that data is included in the tree with the help of the Merkle path, which is constructed through pair nodes from the lead up to the root.

In a blockchain system, the Merkle tree is used with transactions; the transactions will be the leaves, and every two elements (transactions) will be hashed together until having a single hash value, which is the Merkle root [31]. After constructing the Merkle tree and Merkle root, it is then possible for a node to ask other nodes if a transaction is included in a block or not with simple payment verification and trust the other node's response by verifying the received Merkle path.

Smart Contracts

A piece of code that is stored on the blockchain can have a balance, functions, and persisted state that may change whenever the smart contract is invoked. Users can interact with the smart contract by creating a transaction and invoking a specific function. They are immutable, and the transactions that involved the smart contracts can not be revoked [55, 50].

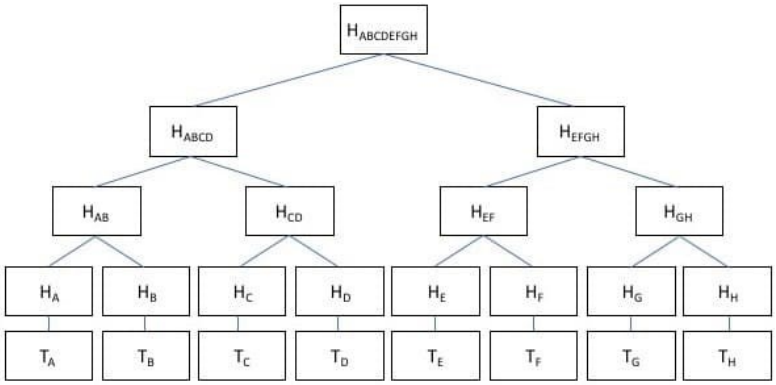

Figure 2.5.: Merkle Tree Representation and Traversal. Source: [31].

2.1.2. Blockchain Types

Different contexts and requirements use different blockchains, which provide different features, depending on if the blockchain needs to be permissioned/permissionless, users' ability to read its records, or restrict mining/validating operations for some/all nodes. There are four blockchain types:

Public Blockchain

Bitcoin [41] defines public blockchain as a permissionless blockchain that allows anyone to join. Also, anyone can read the data stored in the blockchain, create transactions, or join the mining process as a miner node. It is entirely decentralized because the blockchain will be stored in every node (except in wallet nodes) and requires a consensus algorithm for public blockchains like Proof of Work (PoW) or Proof of Stake (PoS). Anyone can verify transactions, and its source code is usually open, and anyone can participate in the source code maintaining process. Public blockchains are trustable since they rely on PoW or PoS consensus algorithms for transactions and blocks validation. Furthermore, the more nodes participate in such networks, the more secure and distributed they become [41]. Another advantage that public blockchains offer is transparency, which is achieved by allowing everyone in the network to trace its transactions and blocks [55]. Using the Proof of Work for public blockchain is expensive and consumes a lot of energy. Another disadvantage is scalability. Public blockchains process transactions slowly because the more extensive the network becomes, the slower it becomes [50].

Private Blockchain

Also called permissioned or business blockchains, are in most cases controlled by single identities. Participation in such networks (blockchains) is restricted to a limited number of identities that only join by a valid invitation [42]. Validation is also required and should be done either by a validator node (operator) or by a set of rules implemented by

the network. The difference to a public blockchain is that it controls who can join the network and who has the right to mine new blocks and read the stored data. They are also smaller than public blockchain networks [42, 55]. Private blockchains are used mostly by private entities and organizations; therefore, they were designed to be scalable and fast to process many transactions. Since the network size is small, it may be vulnerable to attacks, and another disadvantage is that private blockchains are centralized because one/multiple organizations or entities control it [42, 55]. Private blockchains are used in many applications, such as supply chain management, asset ownership, and internal voting.

Hybrid Blockchain

It combines public and private blockchains; it takes advantage of both, and its records and transactions stay private and are only accessible to the public through smart contracts [50].

In blockchain review paper [50], It is mentioned that hybrid blockchain is secure since it prevents many attacks, such as 51 percent attacks. It is also scalable and highly customizable. However, private blockchains lack transparency since some information is hidden, and therefore, it can be seen as an advantage and disadvantage. Furthermore, it is also challenging to maintain and upgrade the network.

Consortium Blockchain

Hyperledger Fabric [1] defines Consortium blockchains as enterprise-level blockchains similar to private blockchains, but they are controlled by multiple entities/organizations rather than a single private organization. Only nodes that belong to one of the organizations that control the network are allowed to participate. It is mainly used when organizations and businesses want to exchange information and create transactions.

2.1.3. Blockchain Node Types

Figure 2.6 shows a clear view of the different categories and types of blockchain nodes. Different types of nodes provide different types of functionalities. Therefore, a blockchain network will utilize various nodes depending on the blockchain type (public, private, or hybrid), requirements, and consensus mechanism.

1. Primary Node or Full Node: Blockchain full nodes maintain and store the blockchain records and blocks and act as blockchain servers. These nodes verify the network transactions and fulfill other nodes' requests, such as requesting a specific block or transaction information [5]. They are also responsible for acting as bootstrap nodes so that other nodes can discover and join the network. They also participate in the decision-making process through their ability to vote for changes in the network.

2. Miner Node: Miner nodes perform tasks to be able to verify the next block in the network and receive a specific reward for verifying it [5]. For example, in the

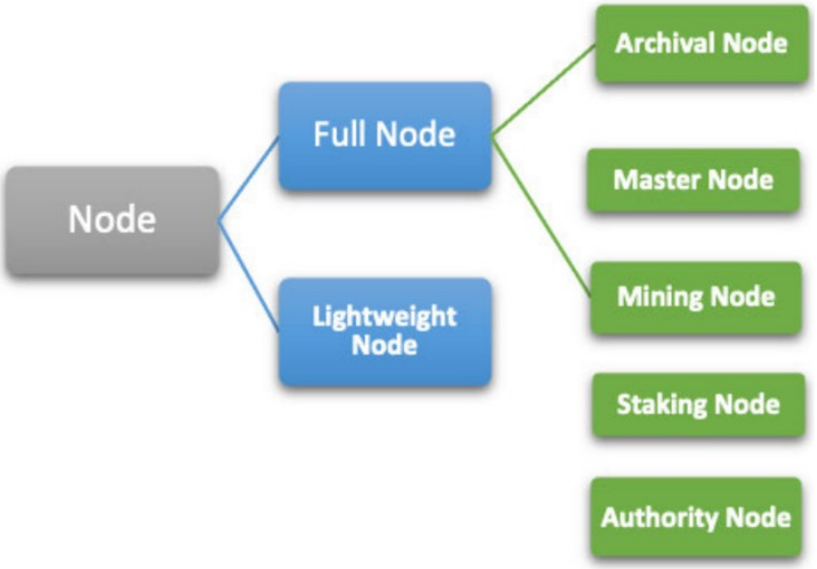

Figure 2.6.: Blockchain Node Types. Source: [5].

case of PoW consensus, miner nodes consume high amounts of energy to solve cryptographic/mathematical tasks, proving that the node is willing to contribute to the network [41]. In some cases, miner nodes are also responsible for fulfilling other lightweight node requests.

3. Staking Node: Staking nodes are known in public blockchains where a node stakes or invests some/all of its money to be able to validate the next block [5], which in return gets some money once a block is correctly validated [49].

   Like Miner nodes, staking nodes are also responsible for fulfilling some requests from lightweight or other nodes.

4. Lightweight Node: As the name says, this node type does not store the ledger or transactions. It only stores the blockchain headers and requests data to fulfill user requests. It does not require considerable space or CPU requirements and is known as an SPV node or Simplified Payment Verification node [55].

5. Authority Node: It is used in the Proof of Authority consensus mechanism. A group of nodes will be selected, called validators, responsible for verifying pending transactions and accounts and adding them to the blockchain [5]. Those nodes are authorized to act based on their reputation and identity [50].

6. Archival Node: They maintain the entire blockchain, including its transactions and state (in the case of the account-based model), and have enough space capacity to fulfill the network validation requests and other nodes requests [5, 50].

2.1.4. Blockchain Consensus Algorithms

A consensus algorithm comprises multiple rules that all the nodes follow to reach consistency and synchronization between the nodes in the network [55]. These rules determine what is considered a valid block or a valid transaction and how to resolve conflicts [49, 55]. For example, when two blocks are received simultaneously, which should be accepted? All these rules will make the network reach a standard agreement about the state of the blockchain. With consensus algorithms, trust and reliability are achieved between unknown peers in the network and ensure that all the peers agree on a block that is the only source of truth [55].

As Figure 2.7 shows, consensus algorithms can be categorized into two main sections: voting-based and proof-based algorithms.

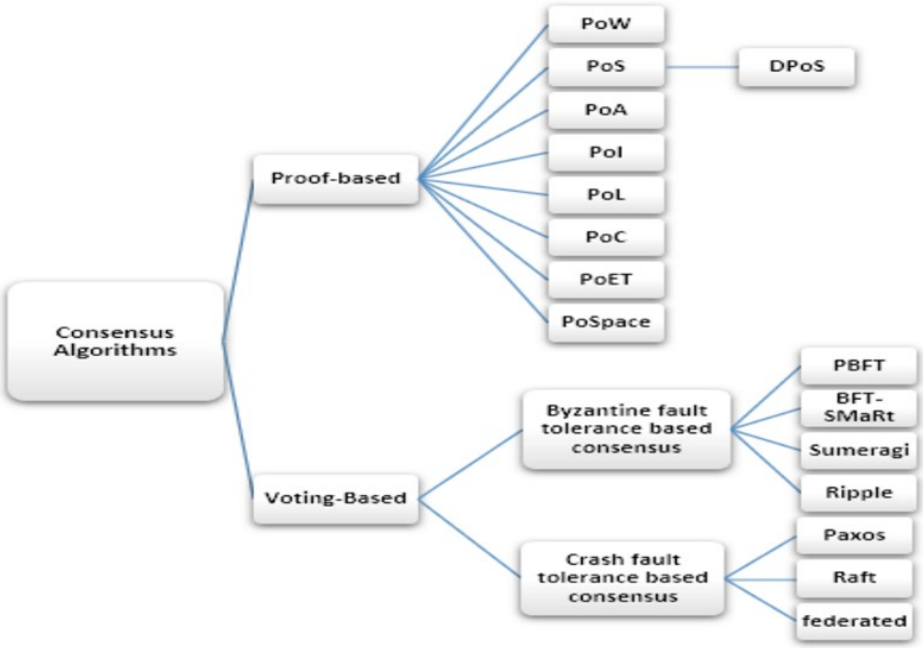

Figure 2.7.: Consensus Algorithms for Blockchain Technology. Source: [30].

Proof-Based Consensus Algorithms

1. Proof-of-Work: This is the most used consensus algorithm. It was introduced by Bitcoin [41], and its idea is that each miner node can add the next block to the blockchain by spending computing power. For example, in Bitcoin, the miner node has to find a nonce, that when it is combined with the block and calculate the hash,

then the hash must be smaller than a predefined number; once this hash is found, the miner node constructs the block and broadcasts it to the network [41]. Other nodes receive the new block with its hash value and recalculate the block hash. If it is correct, the nodes accept the block [41]. No Node should be able to have more than 50 percent of the computing power in the network because, in this way, the node can control the longest chain in the network, which means it can attack the network by double-spending transactions and which will be accepted by the network [41, 50].

2. Proof-of-Stake: Proof-of-Stake can replace PoW consensus because instead of spending computational power, miners (leaders) are selected based on the number of coins they have (stake) and their input in the network [49]. The stake is how much a node hold or deposit; a leader will be randomly selected based on its stake to do the mining process and add a new block to the blockchain [49].

3. Proof-of-Burn: It has two main functions:
   a) A function that generates a random address, so when a miner node sends coins to this address, then these coins will be no longer available (destroyed) [34].
   b) A function that verifies that the account that received the coins can't spend the coins. The main characteristics to make this consensus algorithm work are achieving a way to check if an account is only for burning (cannot spend coins), binding data to the coins that are being burned, and a way to make a difference between a regular and burning address [34].

4. Proof-of-Space: Instead of spending computational power or burning coins like in Proof of Burn, validating nodes must invest their hard drive space to be selected for mining the next block. The chance of being selected increases the more available space they have [17]. The participants are provers who store some data and verifiers who check if the provers are storing the data; verifiers will send challenges to provers who will respond with proof of storage [17].

5. Proof-of-Elapsed-Time: Two measures ensure the user has waited an amount of time. First, once generating a block, each user needs to generate proof for the waiting activity with assistance, which is submitted together with the block [9]. Second, statistical tests are employed to check whether the waiting times of a user indeed follow a specific probability distribution [9].

Voting-Based Consensus Algorithms

Figure 2.7 also shows that Voting-Based consensus can be further classified into two main sections: Crash-Fault-Tolerance (CFT) like Raft, Paxos, and Zab, and Byzantine-Fault-Tolerance (BFT) like PBFT, RBFT, and DBFT.

Crash-Fault-Tolerance consensus algorithm guarantees blockchain network resiliency and recovery from network and node failures such as node crashes, failed hardware, and failed network calls. However, CFT does not prevent Byzantine failures where malicious

actors are involved [30]. Therefore, CFT is only suitable in environments where all the node's identity is known and everyone is trusted, this is relevant in private and consortium blockchains that are used privately by businesses and enterprises [30].

On the other hand, BFT or Byzantine-Fault-Tolerance consensus mechanisms are based on the Byzantine Generals Problem, where several armies should agree on a single action, either attack or surrender. These armies can communicate with each other only through a messenger and must prevent the traitors who want to maliciously prevent the armies from achieving an agreement. BFT must have 3m+1 active nodes to tolerate m malicious nodes [30].

1. Paxos - CFT: It is a fault-tolerant algorithm based on message passing. It has three types of nodes: proposer, acceptor, and learner. The proposer is responsible for creating proposals and waiting for a response from the accepters; the acceptor is responsible for voting. The consensus execution is divided into two phases, PREPARE phase and ACCEPT phase, and it can tolerate F crashing nodes when 2F+1 nodes are present [30].

2. Raft - CFT: The raft consensus algorithm [44] is designed for enterprises and industrial applications. It is also widely adopted in academic research, and its nodes are divided into three types, leader, follower, and candidate. The candidate is responsible for initiating election voting. Once a candidate wins the elections, it is changed to a leader; the first phase is called leader election. The second phase is called log replication, where a leader receives client requests, updates its log, and sends heartbeat requests for the followers to replicate its log [44, 51].

3. PBFT: Practical Byzantine Fault Tolerance [51] is based on state machine replication and can tolerate both crash failures and byzantine failure. It is designed and optimized to work in an asynchronous system and achieve high performance. It achieves consensus through its three phases, which are pre-prepare, prepare, commit, and reply, and it can process the transactions efficiently; however, the communication overhead makes it challenging to scale since it requires identifying the participating nodes, this makes it unsuitable for public blockchains environments [30]. PBFT can guarantee the system works robustly and correctly when there are a total of 3f+1 nodes where only and no more than f faulty nodes are present in the network [30].

4. RBFT: Redundant Byzantine Fault Tolerance requires the same as PBFT, that there are no more than f faulty nodes present in the network where the total nodes are 3f+1, and each node runs several instances of PBFT (f+1 instances) where there is only one primary instance and other instances are used as backup instances, it introduces a new phase to the PBFT which in the propagate phase before pre-prepare to ensure that a request will reach all the nodes [51]. Each node now has a monitoring mechanism that switches to another PBFT instance whenever a threshold is reached for the throughput difference between the primary and a backup instance [30].

2.1.5. Blockchain Frameworks

There are different private and permissioned blockchain frameworks and each has its specific features and advantages, such as smart contracts, performance, identity management, and pluggable consensus.

With the help of the existing blockchain frameworks, many enterprises, businesses, and organizations are continuously and rapidly adopting this technology since they provide a predefined solution divided into modules that are configurable and pluggable depending on the business requirements, the most known and used frameworks are Ethereum, Hyperledger Fabric, Hyperledger Sawtooth, and Corda.

Ethereum

Private Ethereum blockchain [49] can be used by enterprises to create their private blockchain and build it on the Ethereum main net or other blockchains that are based on the Ethereum technology. Nodes deployed on the private Ethereum blockchain can achieve higher transaction throughput than the Ethereum main net. Enterprises have only to pay for maintaining their servers where the private blockchain nodes are deployed. There is no payment when the nodes connect to Ethereum's main net; however, a gas fee must be paid whenever transactions are created on the main net. It also provides privacy and node permissions by only allowing authorized nodes to join the private chain, and the stored data is not visible to anyone outside the network [49].

Hyperledger Sawtooth

Hyperledger Sawtooth is an enterprise-level blockchain platform, and it is mainly designed to achieve smart contract security and keep the ledger distributed. It separates the core system and the application domain, making it easier for the developer to develop the applications without knowing about the underlying core system [21]. It is also highly modular, offers pluggable components, and allows its user to have its preferred consensus algorithm, transaction rules, authorization, and permission rules [21]. It is focused on parallelizing the execution of the transactions by grouping them by the state part that is changed by a transaction. Furthermore, it parallelizes the execution of transactions that belong to the same changed state. It offers several other features, such as creating and broadcasting events within the network, extending the Sawtooth interoperability to deploy Ethereum smart contracts to Sawtooth. Furthermore, it offers many different consensus mechanisms depending on the enterprise requirements, such as PBFT, Raft, and Sawtooth PoET [21].

Hyperledger Fabric

Enterprises have many requirements that are not satisfied with the current public blockchain solutions like Bitcoin and Ethereum, such as identified participants, permissioned networks, high transaction throughput, and low latency for transaction confirmation. Therefore, Hyperledger Fabric has been designed [1].

Hyperledger Fabric [1] is an open-source enterprise-grade permissioned blockchain that was established under the Linux Foundation and maintained by over 35 organizations and 200 developers. Its highly modular and configurable architecture makes it suitable for a wide range of use cases like banking, finance, supply chain, and various other use cases.

It is focused on fulfilling enterprise-specific requirements like node identity, high transaction throughput, low latency, and confidentiality of transactions. It offers various features like pluggable consensus protocols, which do not require a native cryptocurrency, is highly modular, and follows the execute-order-validate transaction architecture [1]. Furthermore, it is the first to support smart contracts written in a general-purpose programming language, which improves the agility of developing applications.

2.1.6. Blockchain-as-a-Services

BaaS is a cloud-based service that combines the high availability, elasticity, and high computing power of cloud computing with the decentralization, immutability, and transparency of blockchain technology [48]. This gives enterprises that utilize this technology combination many advantages such as reduced costs, scalability, increased security, faster time-to-market, and access to blockchain experts who can assist with developing the blockchain applications [48, 18].

2.1.7. Commercial Blockchain-as-a-Service Providers

In BaaS, the state of the art is constantly improving as new cloud providers are offering BaaS, and existing cloud providers who offer BaaS are continuously improving their BaaS solutions. The leading cloud providers in BaaS solutions are:

AWS - Amazon Managed Blockchain (AMB)

It is a fully managed blockchain that makes it easy to participate in blockchain networks or create and manage private networks using Hyperledger Fabric and Ethereum [2].

It takes responsibility for the infrastructure provisioning and eliminates the overhead required for the blockchain network and node creation in minutes. It manages software updates and maintenance and allows inviting partners from other organizations based on their AWS-Account-ID.

Amazon-managed blockchain allows the provisioning of dedicated Ethereum full nodes to build in the web3 area and connect to the main net. It provides many different features such as adding/removing members based on voting API, choice of the framework (Ethereum or Hyperledger Fabric), easy to scale, and secure interactions [2].

It makes it easier for enterprises to build and deploy private blockchain networks using Hyperledger Fabric open source framework [2]. It is also supported in GovCloud, which makes it possible for government agencies to use this platform and deploy it in a specialized environment designed for users that require high security and compliance requirements.

AMB is easy to scale; new nodes can be added to meet the required capacity depending on the client's requirements. Furthermore, the user can specify each node's memory and CPU size, which makes it flexible depending on the deployed applications on the blockchain [2].

It is backed by AWS Key Management Service, which secures the Hyperledger Fabric certificate authorities, using FIPS 140 hardware security module in Key Management Service (KMS), which is integrated with AWS monitoring services where all actions and activities are traced [2].

AMB comes with its own custom Hyperledger fabric [1, 2] ordering service - AWS Quantum ledger Database (QLBD) which is optimized for storing the transactions and makes it capable of storing all uncommitted transactions immutable.

IBM Cloud - IBM Blockchain Platform

It started to offer BaaS in 2016 using Hyperledger Fabric and provides a managed full-stack Blockchain-as-a-Service with the offer of delivering it to an organization-selected environment (on-premise, IBM Cloud, other environments) [27]. It has an easy-to-use interface for managing networks, channels, and smart contracts, the customer data are stored securely because they are encrypted [29].

The IBM blockchain platform worked in September 2019 on deploying a non-vendor lock-in solution by allowing on-premise and multi-cloud blockchain deployment. IBM then announced a new version of IBM Blockchain platform software based on Red Hat's Kubernetes platform [29]. Depending on the user requirements, it is now flexible where the blockchain nodes will be deployed, whether on-premises, single-cloud, or multi-cloud architecture. Furthermore, they created software tools to build, maintain, and grow blockchain networks [28, 27].

Oracle Cloud Infrastructure - Blockchain Platform Service

It is a managed blockchain service for hosting an immutable and tamperproof ledger, built on the open-source Hyperledger Fabric, and simplifies the development of secure applications [43].

It integrates well with other Oracle cloud services and provides a robust REST API and UI, providing complete lifecycle management for smart contracts. Some of its main features are simple provisioning, high availability, easy expansion for partner organizations, and high support for smart contracts development [43].

Kaleido

Kaleido provides enterprise-grade BaaS with various configuration options such as multi-cloud, multi-region, hybrid-based BaaS deployments, SLA and 7/24 expert support, secure network isolation, and automatic failover [33]. Kaleido integrates well with different Kaleido services such as key management, monitoring, REST-API, and event-based integration tools [33].

Google Cloud - Blockchain Node Engine

It is a fully managed node-hosting service that can minimize the need for node operations. It is suitable for companies and enterprises that need dedicated Ethereum nodes, which makes it possible to create transactions and smart contracts and read and write blockchain data in a reliable way [12].

Web3 organizations will have the following benefits when using the Google blockchain node engine: streamlined provisioning (deploy the Ethereum node with a single simple operation by just applying some configurations), authorized access, fully managed operations, and high availability [12].

2.1.8. Academic Blockchain-as-a-Service Systems

uBaaS - Unified Blockchain-as-a-Service Platform

The unified BaaS platform [46] aims to simplify implementing and deploying blockchain-based applications by providing design patterns related to data management and smart contracts. Those are categorized into two primary services: Deployment-as-a-Service and Design-pattern-as-a-Service (Figure 2.8). Deployment-as-a-Service consists of blockchain and smart contract deployment services, which are platform-agonistic and avoid vendor lock-in to a specific cloud provider. Meanwhile, Design-pattern-as-a-Service comprises data management and smart contract design services to achieve blockchain properties such as data integrity, confidentiality, and transparency.

As addressed by [46], many challenges must be addressed when storing and managing data in blockchain applications, like limited storage capacity, scalability, data privacy, and confidentiality. Those challenges are resolved by utilizing the data management design patterns:

- On-chain and off-chain: Store only the sensitive, required, and small data on the blockchain while storing the others with bigger off-chain. This approach resolves the limited-storage-capacity challenge.
- Hash Integrity: All the off-chain stored data must be referenced in the on-chain data by calculating the off-chain data hash and storing it on the blockchains. This design pattern guarantees the integrity of the off-chain data by comparing the stored hash with the computed hash.
- Data Encryption: Since users continuously join the blockchain network, data privacy is a big challenge for blockchain users. It can be solved using asymmetric encryption by encrypting the data before storing it on the blockchain. This approach gives the users who store data control over who can read this data and who cannot.

UBaaS proposes the following design patterns to improve the security of smart contracts:

- Multiple Authorities: Smart contracts require the authorization of a specific number of blockchain addresses to get executed. Some addresses (authorities) may be

unavailable for some time, so the design pattern of Multiple Authorities is applied, giving some flexibility by requiring only M of N blockchain addresses to authorize the transaction.

- Dynamic Binding: When deploying a smart contract, some authorities may still be unknown that is required to authorize a transaction, which can be dynamically defined by adding an off-chain secret sufficient to permit a transaction when associated with it.

- Embedded Permission: Reject unauthorized smart contract function calls by adding permission control to every function defined in the smart contract.

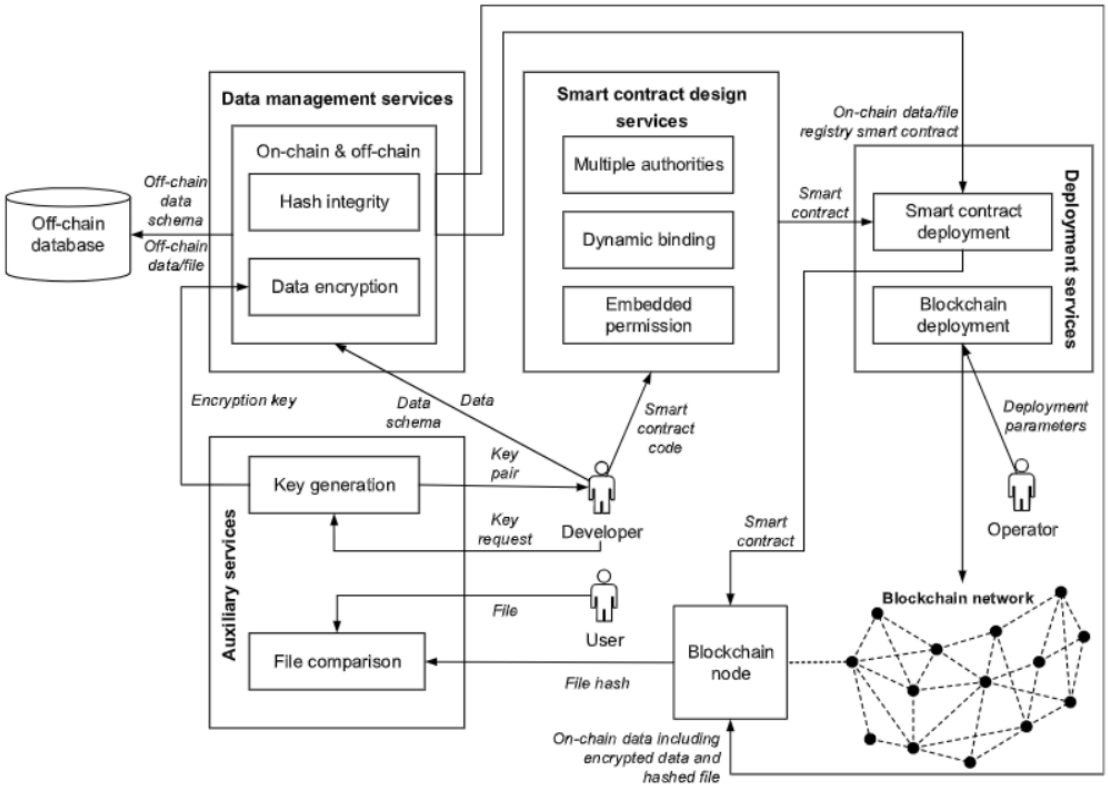

Figure 2.8.: UBaaS Architecture. Source: [46].

uBaaS contains three categories, each with its set of services with related responsibilities: deployment as a service, design pattern as a service, and auxiliary service [46].

- Deployment Service: Includes blockchain and smart contract deployment service. Blockchain deployment service takes the desired configurations from the operator and

deploys the blockchain using the infrastructure as code (IaC) approach. Meanwhile, the smart contract deployment service allows developers to select and deploy predefined smart contracts on the blockchain.

- Data Management Service
  - Using the on-chain and off-chain service, the user can determine the data schema and which data is stored on- or off-chain. Based on the user-defined data model, a smart contract is generated and deployed on the blockchain, which acts as a data registry for on-chain data. Meanwhile, it creates a data table for storing the data off-chain.
  - Integrity service generates hashes for off-chain data to guarantee their integrity.
  - Data encryption services encrypt data that should only be visible to a subset of users using asymmetric encryption.
- Smart Contract Design Service: Focus on smart contract security by applying the defined design patterns.
- Auxiliary service: Contains a file comparisons service that compares the on-chain hash value with the hash of the off-chain stored file. Furthermore, it includes a key generation service responsible for creating and distributing the keys when a user is willing to encrypt data and share it with only a subset of users.

FBaaS - Functional Blockchain-as-a-Service Platform

FBaaS [7] proposes a new service model, Functional Blockchain-as-Service, based on the Function-as-a-Service (FaaS) model. It simplifies the implementation of business logic which gives many advantages such as improving the blockchain performance, benefiting from the high availability and resilience of the FaaS network, and implementing a higher level of abstraction of the logic.

The FBaaS architecture follows the Cloud Computing Open Architecture and is implemented using the Big Data Open Architecture. As the figure 2.9 shows, the layers of BDOA that represent the FBaaS architecture are [7]:

- Infrastructure layer: Mapped to layer one and layer 2 of FBaaS, which only relies on AWS EC2 instances.
- Components layer: Mapped to layer 3 of FBaaS, which contains the essential functions frequently used in the upper layers, this layer is realized using a Container-as-a-Service solution.
- Service layer: Mapped to layers 4 and 5. In this layer, most of the BaaS functions are realized.
- Business logic layer: Mapped to layer APPs and large-scale services. This layer is based on the functions that are present in layer three.

The service layer is composed of the following functions: [7]:

- Object Storage: Used for object storage and management, done using strong consistency. Users do not worry about eventual consistency issues.
- New transaction: Add the new transaction and append it to a block to be mined. The transaction has no size limit and will get encrypted.
- Mine: A function responsible for mining a new block, done using Apache Kafka to accelerate the mining process.
- Chain Info: Fetch blockchain information.
- Resolve: A function that resolves conflicts by running consistency algorithms.
- Add Node: Responsible for adding a new node to the network by storing the state, replicating a node, and changing its configurations.

FSBaaS - Full-Spectrum Blockchain-as-a-Service Platform

Many enterprises and businesses do not prefer and are not adopting specialized private blockchains like Amazon QLDB, which belong to blockchains that work with a centralized authority [8]. However, they should be used in cases like a single organization using the blockchain. Such cases do not make a difference if there is a central authority. FSBaaS [8] integrates the two underlying blockchain runtimes, centralized and decentralized, to support consortium blockchain networks. Both runtimes are exposed using unified interfaces and REST APIs. An FSBaaS tenant can subscribe to both runtimes simultaneously and synchronize data across networks [8].

Figure 2.10 shows the architecture of an FSBaaS node, which comes with both of the runtimes bc-fabric and bc-lite sharing the same business network manager and contract executor. Hyperledger Fabric is used for bc-fabric, and MongoDB is the world state database. MongoDB stores the world state and the ledger data in bc-lite, and the client-side API server can be reached through REST API [8].

NutBaaS - Nut Blockchain-as-a-Service Platform

Most current BaaS solutions focus on reducing the effort and cost of blockchain deployment and development for businesses and enterprises. However, NutBaaS platform [52] provides a safer development environment for developers by focusing on smart contracts security.

The NutBaaS architecture is divided into four main layers. As Figure 2.11 shows, its purpose is to provide a detailed operational monitoring mechanism that covers configuration management, visual monitoring, user-defined alert, and automated deployment [52].

- Resource Layer: Provides networking, storage, and virtual machines from a cloud provider. It supports public, private, and hybrid blockchain deployment. It promotes multi-cloud deployment, making a blockchain network instance secure and

decentralized. Kubernetes is used to make the system scalable and maintainable through container orchestration. The resource layers support the deployment of different blockchain frameworks like Hyperledger Fabric and public/private Ethereum peers.

- Service Layer: All blockchain-essential services are implemented in this layer. When integrating these layers, we achieve general-purpose solutions and many other services, like the contact center, divided into three sub-layers.
  - Underlying system: Support public, private, consortium, and blockchains.
  - Developer Tools: Varies depending on the type of blockchain.
  - Service: The integration and combination of the tools provide this layer's advanced and basic services.
- Application Layer: Contains the integrated services and applications from the previous layer, which are general-purpose solutions.
- Business Layer: This layer contains existing solutions for many different industries, making it easy for enterprises to find a suitable solution for their use case.

NBaaS - Noval Blockchain-as-a-Service Platform

NBaaS provides a decentralized and secure platform that protects the blockchain's main characteristics, like decentralization and security, despite being managed and deployed on the cloud [54].

It classifies BaaS platforms into two categories: encapsulated Blockchain-as-a-Service (eBaaS) and native Blockchain-as-a-Service (nBaaS). The nBaaS is about deploying and hosting blockchain nodes on the cloud. Meanwhile, eBaaS provides interfaces like REST APIs to deploy blockchain instances on the cloud [54]. Since eBaaS affects blockchain characteristics like decentralization negatively, a new transparent eBaaS is done by nBaaS that protects the blockchain's main features.

It is done by making an eBaaS deployment a white box deployment using the deployable components approach, which allows the client to deploy the components wherever he wants, regardless of whether on-premises or on different cloud providers [54].

2.2. Comparative Analysis - BaaS Commercial Providers

The following considers the BaaS providers introduced in this paper. The comparison shows that all providers offer private blockchain types while most offer consortium blockchains. Furthermore, Hyperledger Fabric is offered in almost all BaaS providers, making them suitable for many use cases (Table 2.1).

Based on the comparison shown in Table 2.2, Kaleido and Oracle Blockchain platforms offer the cheapest price for hosting and managing a BaaS instance. Most of the BaaS providers are programming language agnostic since Hyperledger Fabric is present in most of them, which allows writing smart contracts in many programming languages.

BaaS Provider	Blockchain Type	Blockchain Framework	Use Cases
Amazon Managed Blockchain	Private, Consortium, Public	Hyperledger Fabric, Ethereum	Supply chain, Manufacturing, Retail, Trading
IBM Blockchain	Private, Consortium	Hyperledger Fabric, Custom Blockchain Frameworks	Retail, Banking, Transportation
Oracle Blockchain Platform Service	Private, Consortium	Hyperledger Fabric	Retail, Manufacturing, Finance, supply chain, Healthcare
Kaleido	Private, Consortium, Public	Ethereum, Polygon Edge, Hyperledger Fabric, Corda	Supply chain, Finance, Public sector, Healthcare
Google Blockchain Node Engine	Public, Private	Ethereum	Web3 Applications

Table 2.1.: BaaS Providers Blockchain Type, Blockchain Framework and Use Case comparison

The pricing comparison is based on the following parameters:

- Deployment Region: US
- Package Type: Starter package
- Blockchain Framework: Hyperledger Fabric
- Currency: US Dollar

As Table 2.3 shows, only some providers allow the user to deploy and manage the BaaS instance on Multiple cloud providers or on-premise, making the BaaS more decentralized and resilient. However, when inspecting more BaaS providers, only some of them are expected to offer this important characteristic.

2.3. Comparative Analysis - BaaS Academic Systems

The addressed BaaS systems work on different generic areas that contribute to the adoption and utilization of BaaS systems:

- The most desired goal is simplifying a blockchain network deployment and management process.
- Maintaining and improving the essential characteristics like decentralization and security of the blockchain in a cloud environment.

BaaS Provider	**Architecture**	**Pricing**	**Pay-as-you-go**	**Programming language agnostic**
Amazon Managed Blockchain	Component-based architecture	0.33 per hour/node	True	True
IBM Blockchain	Modular/layered architecture	180 per month/Virtual processor core	False, 12-month commitment	True
Oracle Blockchain Platform Service	Layered architecture	0.215 per hour/oCPU	True	True
Kaleido	-	0.22 per hour/node	True	True
Google Blockchain Node Engine	-	-	False	False

Table 2.2.: BaaS Providers Architecture, Pricing, Pay-as-you-go and Programming language-agnostic comparison

BaaS Provider	**Vendor Lock-In**	**Multi-Cloud / Onpremise deployment**	**Other**
Amazon Managed Blockchain	True	False	Custom ordering service- AWS QLDB
IBM Blockchain	False	True	based on open source technologies Kubernetes, Hyperledger
Oracle Blockchain Platform Service	False	True	Dev tools
Kaleido	False	True	-
Google Blockchain Node Engine	True	False	-

Table 2.3.: BaaS Providers Vendor Lock-In, multi-cloud / On-premise deployment and Other comparisons

- Simplify the development of blockchain applications by providing development tools.
- Secure the blockchain network and applications by providing design patterns for smart contracts, vulnerability checks, and visual monitoring.
- Provide services to enable infrastructure agnostic deployment, which means allowing the deployment of blockchain networks on-premise or in a multi-cloud fashion and preventing vendor lock-in.

Both uBaaS and NutBaaS work on the security aspects of the BaaS platform. The first did this by providing smart contract design patterns, and the second by integrating monitoring services and smart contract vulnerability checks. Both systems, including NBaaS, overlap by allowing users to deploy the blockchain nodes on-premise, on a single cloud, or in a multi-cloud environment. NBaaS achieves this by making the BaaS deployment a white box deployment using the deployable components approach. Other systems do this by implementing generic deployment services and container orchestration tools.

The other BaaS systems don't overlap, and each works on its specific area in BaaS. For instance, FSBaaS integrates different blockchain runtimes (fabric and lite) in a single node to enable their use simultaneously. On the other hand, FBaaS focuses on simplifying business logic implementation using FaaS functions.

2.4. Summary and Transition

Chapter 2, Related Work, started by giving a background overview of blockchain technology, covering its types, frameworks, consensus algorithms, and architecture.

Followed by offering a comprehensive overview of Blockchain-as-a-Service and the current leading platforms that provide BaaS. It compares them based on different factors and characteristics: blockchain types and frameworks, use cases, architecture, pricing, vendor lock-in, and multi-cloud/on-premise deployment ability. The comparison shows that all providers offer private blockchain types while most provide consortium blockchains.

Furthermore, Hyperledger Fabric is supported by almost all BaaS providers. Kaleido and Oracle Blockchain Platform offer the lowest price for hosting and managing a BaaS instance. The comparison also showed that not all top BaaS providers offer multi-cloud or on-premise deployments, negatively affecting blockchain's decentralization and resiliency characteristics.

Finally, it offers a comprehensive overview and compares the recent BaaS academic systems and their architecture. The comparison shows that these systems work on different areas that contribute to the adoption and utilization of BaaS systems, which are simplifying the blockchain network deployment and management process, maintaining the blockchain characteristics, supporting and guiding the blockchain developers by providing design patterns, and development tools and enabling agnostic deployment whether on-premises or multi-cloud deployment.

UBaaS, NutBaaS, and NBaaS overlap by allowing users to deploy the blockchain nodes on-premises, on a single cloud, or in a multi-cloud environment. However, NBaaS achieves this by making the BaaS deployment a white box deployment using the deployable components approach. FSBaaS integrates different blockchain runtimes in a single node to enable using them simultaneously. On the other hand, FBaaS focuses on simplifying business logic implementation using FaaS functions.

Based on the related work research, analysis, and comparison done between the various academic and commercial Blockchain-as-a-Service systems, we know that:

1. The BaaS academic systems that support multi-cloud blockchain deployment do not define an exact architecture and specification that clearly shows how to implement such a platform that supports multi-cloud and avoids vendor lock-in.

2. Many commercial BaaS providers do not support multi-cloud deployments, negatively affecting platform characteristics like decentralization, high availability, and security.

3. All the BaaS platforms that are currently in use are:
   a) Closed source.
   b) Commercial and cost a lot.

Therefore, this thesis will provide a BaaS architecture that focuses on providing a specification for a BaaS platform that supports multi-cloud and multi-region blockchain deployments. Besides that, a prototype will be implemented based on the defined specification, focusing on scalability, maintainability, high availability, and decentralization of the end solution.

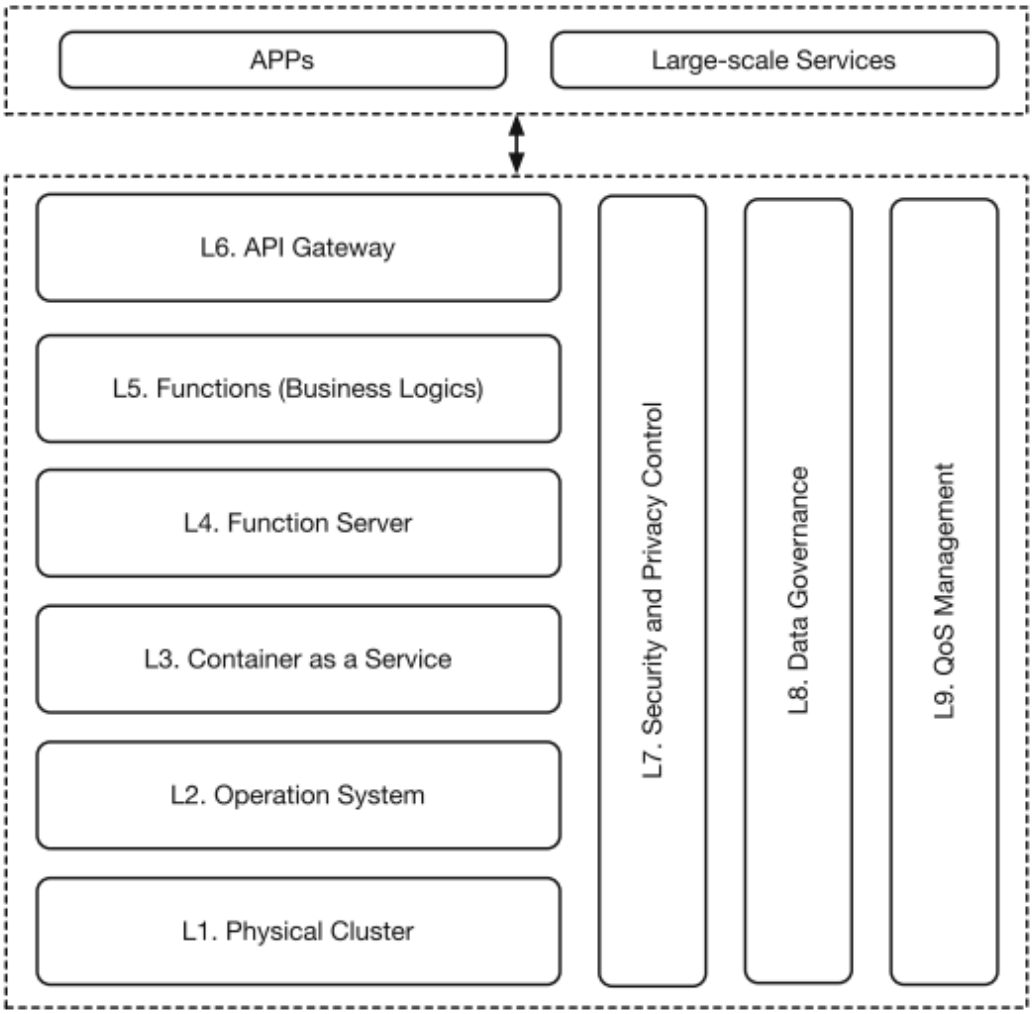

Figure 2.9.: FBaaS Architecture. Source: [7].

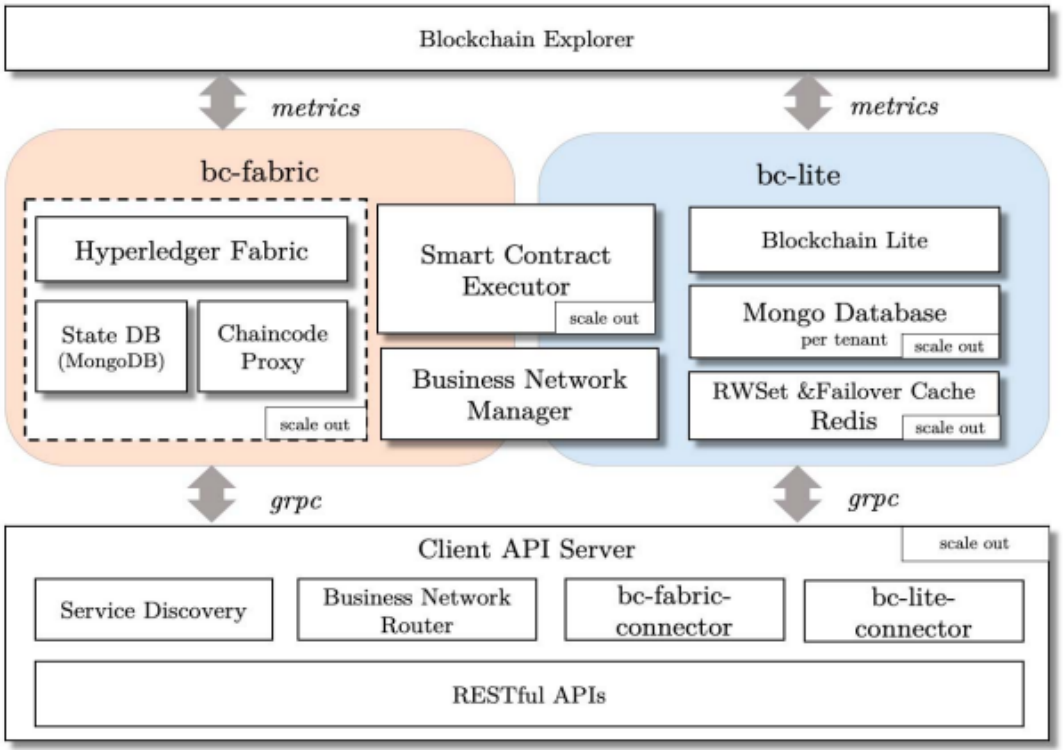

Figure 2.10.: FSBaaS Architecture. Source: [8].

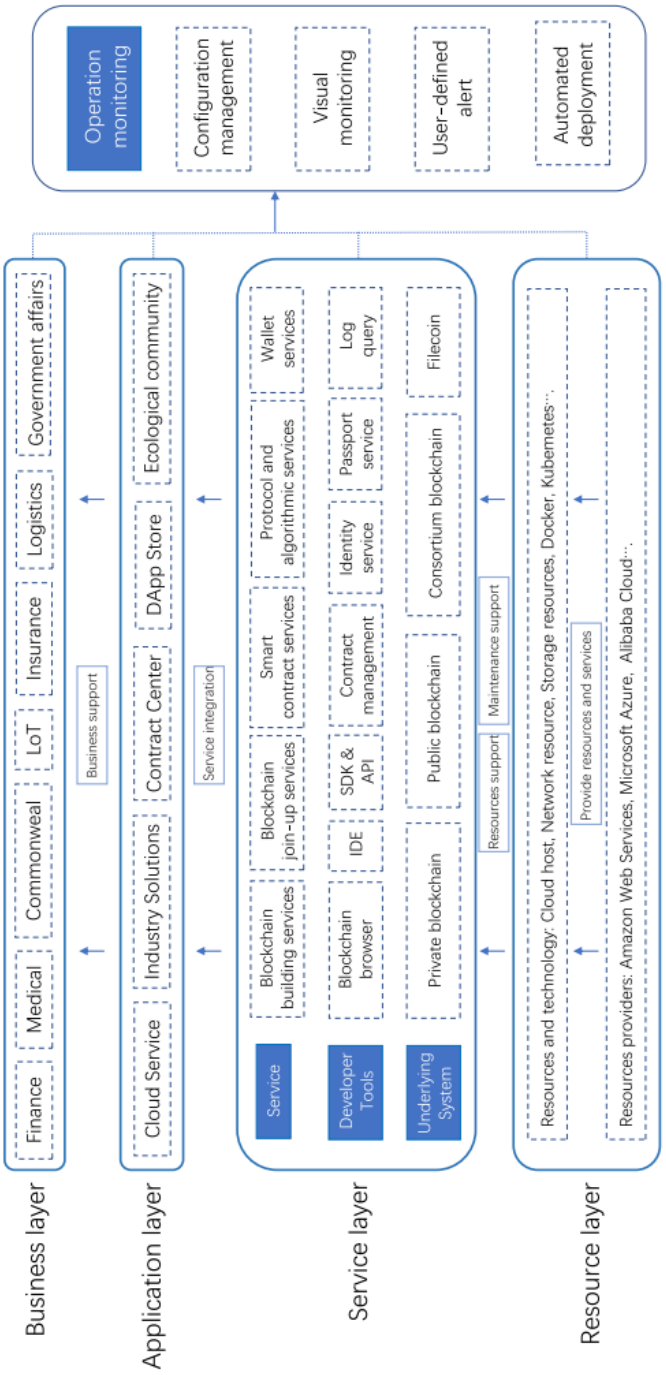

Figure 2.11.: NutBaaS Architecture. Source: [52].

3. Design

The presented design and specifications outlined in this chapter are done to realize a robust multi-cloud, multi-region Blockchain-as-a-Service platform. The design evolved through an iterative process, considering the challenges and issues mentioned in the Motivation and Related Work chapters. The design also considers all the defined functional and non-functional requirements and all the architectural design decisions that consider the different paths of achieving the end solution and select the most suitable among them.

3.1. Requirements

3.1.1. Functional Requirements

Functional Requirement: User Registration and Login

Requirement ID: FR-001
Description: Only authenticated users are permitted to access the platform. It must be possible for users to register and log into the platform by providing their credentials.
Priority: HIGH
Use Case: User Registration
Dependencies: NONE
Acceptance Criteria:

1. Users can register and log in to the platform
2. The platform is only accessible by authenticated users.
3. Once a user completes the login/registration process, he must be forwarded to the Home layout.

Functional Requirement: Cluster Management

Requirement ID: FR-002
Description: The platform must enable its users to create, configure, and delete clusters (a group of virtual machines that provide resources to deploy applications and software on them like Kubernetes) from various cloud providers so they can be later used to deploy blockchain networks and nodes on them.
Priority: HIGH
Use Case: Cluster Management
Acceptance Criteria:

1. The authenticated users can create clusters from different cloud providers. After the successful creation process, they should also be able to display them.
2. After the cluster creation process is completed, the system should enable the users to configure and delete the clusters they want.
3. The required input fields during cluster creation differ depending on the selected cloud provider.
4. Once a cluster is created, the users will receive a cluster creation notification. The notification will be emailed, and the cluster status will change on the dashboard to RUNNING.

Functional Requirement: Cluster Integration and Configuration

Requirement ID: FR-003
Description: The platform should allow the configuration and integration of clusters from different cloud providers. This feature enables users to create multi-cloud, multi-region blockchain networks.
Priority: HIGH
Use Case: Cluster Integration and Configuration
Dependencies: Successful implementation of the requirements: FR-001, FR-002.
Acceptance Criteria:

1. The authenticated users can integrate multiple clusters from different cloud providers.
2. The users can later integrate additional clusters into existing integrated clusters.
3. Once multiple clusters are integrated, a resilient, encrypted, and secure connection is established between those clusters that can be later utilized for communication between deployed blockchain nodes and applications distributed among these clusters.
4. The user is notified via email after the cluster integration process succeeds.
5. The platform should enable its users to view the clusters integration they created and their current status.
6. The platform should enable its users to delete clusters integration they previously created.

Functional Requirement: Cluster Monitoring

Requirement ID: FR-004
Description: The platform should enable observing the health status of each provisioned cluster.
Priority: MEDIUM

Use Case: Cluster Monitoring
Dependencies: Successful implementation of the requirements: FR-001, FR-002.
Acceptance Criteria:

1. The users can see their provisioned clusters' health status on the cluster management dashboard.

2. Once a cluster becomes unhealthy or unreachable, the user is notified via email.

Functional Requirement: Blockchain Management

Requirement ID: FR-005
Description: The system should support the configuration deployment and management of private and consortium blockchain types.
Priority: HIGH
Use Case: Blockchain Management
Dependencies: Successful implementation of the requirements: FR-001, FR-002, FR-003.
Acceptance Criteria:

1. The users can deploy and manage private and consortium blockchains on the previously provisioned clusters.

2. The system must enable configuring the deployment, whether to deploy the blockchain instance nodes and resources to a single cluster or distribute them among multiple integrated clusters to achieve multi-cloud, multi-region deployments.

3. When deploying a blockchain instance, the user can configure the blockchain network instance and its initial nodes. Furthermore, the user can select whether to deploy the blockchain nodes on a single or integrated cluster. In the case of integrated cluster deployment, the user can select a deployment policy for the nodes. The deployment policy can be either static when the user specifies whether a node is deployed on a specific cluster, or round-robin, when the nodes are distributed among the clusters in a round-robin fashion.

Functional Requirement: Blockchain Membership

Requirement ID: FR-006
Description: An authenticated platform user can only view a blockchain network instance when his account identifier owns a node. Furthermore, that user can only manage and configure a blockchain node that he owns.
Priority: HIGH
Use Case: Blockchain Management
Dependencies: Successful implementation of the requirements: FR-001, FR-002, FR-005.
Acceptance Criteria:

1. The system allows users to display only blockchain instances they are members of.
2. The system enables users to configure and manage their blockchain nodes.
3. The users receive email notifications when invited to a new blockchain network.

Functional Requirement: Blockchain Node Management

Requirement ID: FR-007
Description: The user can manage, configure, and interact with his node through the platform UI. Depending on the platform's supported blockchain framework, the user can view his node information, configure the node, manage his wallet, and sign transactions.
Priority: HIGH
Use Case: Blockchain Node Management
Dependencies: Successful implementation of the requirements: FR-001, FR-005.
Acceptance Criteria:

1. For each supported blockchain framework by the platform, the system must support all the operations offered by the blockchain framework into the platform.
2. In the case of the P2 blockchain framework, the user must be able to configure and manage his wallet, read the blockchain network information, and sign transactions.

3.1.2. Non-Functional Requirements

Non-Functional Requirement: Clusters Integration - Clusters Connectivity

Requirement ID: NFR-001
Description: The platform should enable cluster integration by supporting reliable, encrypted, and authenticated connectivity between multiple clusters.
Priority: High
Category: multi-cloud multi-region Support
Acceptance Criteria:

1. The connectivity between clusters must be successfully established whenever a user integrates a group of clusters.
2. The platform must enable its users to observe the status of the cluster integrations.

Non-Functional Requirement: Request Response Time

Requirement ID: NFR-002
Description: The platform should ensure that the response time for user requests and interactions is no more than 2 seconds.
Priority: HIGH
Category: Performance
Acceptance Criteria:

1. The AVG response time for user requests should be less than 2 seconds.
2. For cluster integration operations and blockchain node deployments, it is acceptable for some operations to take longer.

Non-Functional Requirement: Platform Components Design - Open-Closed Principle

Requirement ID: NFR-003
Description: The platform components must be implemented that allow the simple extension of the platform, whether by integrating a new cloud provider or supporting a new blockchain framework.
Priority: HIGH
Category: Maintainability and Reusability
Acceptance Criteria: The platform components are implemented in a maintainable and reusable way, enabling easy platform extension.

Non-Functional Requirement: Support at least 2 Cloud Providers

Requirement ID: NFR-004
Description: The platform should support cluster management and creation from at least two cloud providers.
Priority: HIGH
Category: multi-cloud, high availability, and decentralization
Acceptance Criteria:

1. The platform supports cluster management from multiple cloud providers.
2. The users can select different cloud providers for cluster creation and management.

Non-Functional Requirement: Clusters Monitoring

Requirement ID: NFR-005
Description: The managed clusters are monitored, and the user should be able to check their status and health through the platform.
Priority: HIGH
Category: Observability and Reliability
Acceptance Criteria:

1. The users can see their managed cluster's health and status through the platform.

Non-Functional Requirement: Web User Interface

Requirement ID: NFR-006
Description: The platform should support a web user interface that is easy to use and self-explanatory.
Priority: HIGH

Category: Usability
Acceptance Criteria:

1. Basic criteria for usability that should be met.
2. The system should allow users to navigate and use all functional requirements easily.

Non-Functional Requirement: Cluster Deployment Region

Requirement ID: NFR-007
Description: The platform should allow users to freely select the region where the managed clusters can be provisioned. This feature gives more flexibility and makes a multi-cloud Blockchain deployment highly available and decentralized.
Priority: HIGH
Category: Availability and Decentralization.
Acceptance Criteria:

1. Basic criteria for usability that should be met.
2. The system should allow users to easily navigate and use all functional requirements.

3.2. Design Approach and Overview

The platform is designed based on the given functional and non-functional requirements, considering the challenges and issues that must be addressed to achieve a reliable multi-cloud, multi-region Blockchain-as-a-Service. Furthermore, All the design and architectural design decisions are justified and discussed in the upcoming sections of this chapter.

3.2.1. Platform Components

As Figure 3.1 shows, the platform is designed based on the event-driven microservices architectural style and follows the microservices architectural design patterns.

The platform is composed of the following components:

1. **Identity and Access Manager:** Responsible for managing authentication and authorization of the user and supporting various features like MFA, SSO, OIDC OAuth2, SAML, and safely stored passwords. In the context of the implemented prototype, the open-source identity and access management tool Keycloak is used to manage access to the BaaS platform.
2. **Event Streaming Platform:** To enable reliable, scalable, asynchronous, and decoupled communication between the different microservices, an event streaming component is required to orchestrate the internal communication between different microservices. In the context of the implemented prototype, the open-source distributed event-streaming platform Apache Kafka is used.

3. **API Gateway:** The single entry point for all clients into the microservices, it can apply various cross-cutting concerns like, authentication, authorization, rate limiting, and request routing.

4. **Frontend User Interface:** A web application that provides a basic level of usability to interact with the platform and its services.

5. **BaaS Platform Microservices:** A set of loosely coupled and fine-grained services that implement all the Blockchain-as-a-Service platform.

3.2.2. Platform Microservices

As previously mentioned, the platform comprises multiple microservices, each covering a specific domain with a specific set of operations to orchestrate. The BaaS platform contains the following microservices:

1. **Cluster Management Service:** Responsible for all Kubernetes cluster Management tasks like creating, deleting, and configuring a cluster. It supports integration with multiple cloud providers through cloud providers programming language-specific SDKs.
   This microservice fulfills the requirements: FR-002.

2. **Cluster Integration Service:** Responsible for all cluster connectivity and integration tasks. Given a set of provisioned clusters, it established a resilient, encrypted, and authenticated connection between those Kubernetes clusters. This microservice fulfills the requirements: FR-003.

3. **Cluster Monitoring Service:** Responsible for observing the provisioned cluster's status and health. This microservice fulfills the requirements: FR-004.

4. **Notification and Alert Service:** Responsible for notifying the user whenever a specific event happens in the platform, like cluster creation or blockchain node invitation. The notification configurations can be changed depending on the user's preference. This microservice is responsible for fulfilling the requirements: FR-002, FR-003 and FR-005.

5. **Blockchain Management Service:** Responsible for blockchain network instance creation and management. This service schedules nodes and blockchain applications based on multiple deployment policies to be deployed on the provisioned cluster. This microservice is responsible for fulfilling the requirements: FR-005 and FR-006.

3.3. Architectural Design Decisions

3.3.1. Architectural Design Decision: Platform Architecture - Microservices

Decision ID: ADD-001

Context: As part of designing the BaaS Platform, we must decide on the architecture that meets the platform's scalability, reliability, and maintainability requirements.

Problem Statement: Choosing between microservices, monolithic, or serverless architectures.

Considered Options:

1. Microservices architecture
2. Monolithic architecture
3. Serverless architecture

Decision: The microservices architecture has been chosen, as it offers the required level of scalability for the platform. Besides that, it is highly available since all services are independently deployable and loosely coupled.

Benefits:

1. High scalability.
2. Loosely coupled services.
3. Independently deployable services.
4. Fine-grained control over service behavior.

Drawbacks:

1. Increased complexity.
2. Operational challenges.
3. Cost of communication.

Alternatives:

1. Monolithic architecture is simple and has a faster development cycle. However, it has limited scalability and is unsuitable for complex and diverse functional requirements. Furthermore, it does not meet the high availability requirements for a multi-cloud multi-region Blockchain-as-a-Service platform.
2. Serverless architecture offers high scalability and cost savings, but it is not suitable for complex workflows that may require orchestration. Furthermore, it does not offer complete control over the infrastructure.

Dependencies: Distribute the platform requirements between the services and end up with a set of microservices where each has its specific functionality.

References: [26, 32]

3.3.2. Architectural Design Decision: Event-Driven Communication - Apache Kafka

Decision ID: ADD-002

Context: To design the BaaS microservices, we must establish reliable, resilient, high throughput, and loosely coupled communication and data flow between the services.

Problem Statement: Choose between REST API and event-driven for inter-service communication.

Considered Options:

1. REST API Calls.
2. Event-Driven Communication - Apache Kafka.

Decision: The Event-Driven Communication - Apache Kafka is chosen for inter-service communication since this pattern offers scalability, loose coupling, and real-time data flow.

Benefits:

1. Loose coupling between services.
2. Highly scalable.
3. Real-time data flow and responsiveness.

Drawbacks:

1. Event management between the services.
2. Data consistency challenges.

Alternatives: REST API calls can lead to tight coupling between services. Synchronous calls can cascade effects during high-traffic periods. Error handling and retries become complex.

Dependencies: Understanding Kafka concepts and defining the topics.

References: [20, 3, 36]

3.3.3. Architectural Design Decision: Clusters Management

Decision ID: ADD-003

Context: To reliably manage Kubernetes clusters in the platform, we have to analyze and select the most suitable way of achieving the functionality, taking into count the maintainability, scalability, and availability. Furthermore, commercial, closed-source, or cloud provider-specific tools/solutions must be avoided to prevent vendor lock-in and achieve high decentralization and transparency of this platform.

Problem Statement: Choose between Infrastructure-as-Code solutions, directly using cloud provider SDK, or using existing open/source non-commercial Kubernetes cluster management tools to manage Kubernetes clusters.

Considered Options:

1. IaC - Terraform.
2. KubeSpray.
3. Rancher.
4. Cloud Provider SDK.

Decision: Use the cloud provider SDK offered by each provider that should be integrated into the BaaS Platform.

Benefits:
1. Complete control and flexibility over how the resources in each cloud provider are provisioned and managed.
2. No dependency or contract to any commercial or closed-source Kubernetes platform management tools and applications.
3. Open to integrate any cloud provider and on-premise SDK.

Drawbacks:
1. The BaaS platform should manage each Kubernetes cluster's state and keep it in sync with its actual state on the cloud provider.
2. Additional SDK is required to integrate a new cloud provider into the platform.

Alternatives:

1. IaC - Terraform: Infrastructure as Code manages and provides cloud infrastructure using code. Terraform is the most known and widely used IaC tool. Many IaC tools, including Terraform, provide a cloud development kit to enable using them in different programming languages. They offer state management of the provisioned infrastructure, easy-to-use declarative syntax, and reduce the complexity of the whole infrastructure management.
2. KubeSpray: It is an automation tool that manages production-ready Kubernetes clusters based on Kubeadm and Ansible. Despite being able to deploy clusters on various cloud providers, it is limited to specific cloud providers which negatively affects the BaaS platform's ability to integrate any additional cloud provider. Besides that, it does not offer any programming language SDKs.
3. Rancher: It is open-source for managing Kubernetes clusters with compelling features like Service Mesh Integration, IaC, and Multi-Cluster Management. It cannot be considered a solution for cluster management in the BaaS platform because some advanced features are only available in the enterprise version.

Dependencies: Select the cloud providers to integrate into the platform and their initial setup and SDKs.

References: [23, 15, 47, 13, 35]

3.3.4. Architectural Design Decision: Clusters Integration and Connectivity

Decision ID: ADD-004

Context: A resilient, encrypted, and authenticated cross-cluster connectivity must be established between the managed Kubernetes clusters by the platform to enable deployment of multi-cloud, multi-region blockchain nodes.

Problem Statement: Enabling cluster-to-cluster communication is essential for achieving multi-cloud blockchain applications. Once multiple clusters are connected, it should be possible for any blockchain nodes deployed on two different clusters to communicate securely as if they were in the same cluster. The communication must be encrypted, secure, and authenticated. There are various ways and technologies to enable cluster-to-cluster connectivity.

Considered Options:

1. Istio service mesh.
2. Cilium.
3. Rancher.
4. Custom Networking Implementation.

Decision: Use Istio service mesh to achieve cross-cluster connectivity.

Benefits:

1. Traffic Management and Load Balancing.
2. Observability and Monitoring: Istio integrates easily with monitoring tools like Prometheus.
3. Cross-Cluster Communication: Achieve the required cross-cluster communication for the platform seamlessly by configuring and deploying Istio manifests into the clusters.

Drawbacks: As Istio only offers some of the required features through its different programming language SDKs, some operations not supported by the SDK may be invoked by its CLI client, which introduces complexity and maintainability challenges.

Alternatives:

1. Custom Networking Implementation: Implement a gateway and proxy for each cluster to enable connectivity. The gateway and proxy can be configured with security, authentication, and all the required features for cross-cluster connectivity. However, this approach requires complex networking configurations and high effort to implement the required components on each cluster.
2. Cilium: A container network plugin offering multi-cluster service mesh. However, Istio has existed for several years and has matured regarding features and use cases. Besides that, istio has a vast community and is currently the leader in the service mesh space, which makes Cilcium a less suitable solution when compared to Istio.

3. Rancher: This option can be ignored, as it is justified in the ADD ADD-003 why the Rancher option is unsuitable for the platform.

Dependencies:

1. Setup Istio configurations, manifests, and components to deploy into each cluster whenever a user wants to leverage a multi-cluster, multi-region blockchain solution.

2. Automate the Istio setup process on a Kubernetes cluster to enable dynamic cross-cluster connectivity.

References: [35, 47, 22, 10]

3.3.5. Architectural Design Decision: OpenID Connect Authentication Protocol

Decision ID: ADD-005

Context: An effective authentication and authorization mechanism should be implemented to secure access to the BaaS platform based on the microservices architecture.

Problem Statement: Securing the BaaS platform is critical to controlling user access and privileges. Therefore, the different authentication mechanisms must be analyzed, and the most suitable one must be applied.

Considered Options:

1. OAuth 2.0.

2. JWT Authentication Only.

3. OpenID Connect (OIDC).

Decision: Implement OpenID Connect as an authentication and authorization mechanism for the platform.

Benefits:

1. API Gateway Compatibility: Integrating OIDC with an API Gateway enforces authentication at the platform's edge and acts as a single entry point.
2. OpenID Connect is a widely accepted standard protocol for authentication.
3. Centralized Identity Management.
4. Integration with External Identity Providers: This is an essential benefit as it allows users to authenticate using the identity provider they want supported by the platform.

Drawbacks:

1. Infrastructure Overhead: Setting up an OIDC server introduces additional effort to deploy, manage, and maintain an OIDC server like Keycloak.
2. Complexity: Integrating and configuring the OIDC server and its resource servers

introduces additional complexity.

Alternatives:

1. OAuth 2.0: It is a standard for delegated access and authorization and does not focus on authentication and identity management.
2. JWT Authentication Only: It is a token format that is a mechanism for transmitting data between parties in a trusted way. However, it focuses on authorization and lacks authentication and identity management.

Additional effort and complexity are required for implementing identity management and authentication flow in the case of OAuth2.0 and JWT Authentication.

Dependencies:

1. Setup and configure OIDC Server.
2. Integrate the API Gateway for OIDC authentication and each microservice for authorization.

References: [16, 39, 37]

3.3.6. Architectural Design Decision: Blockchain Resources Mangement

Decision ID: ADD-006

Context: In managing and deploying blockchain applications, whether on a single Kubernetes cluster or multi-cloud connected clusters, the platform must support configurable blockchain deployments and specifically offer deployment policies and strategies.

Problem Statement: Implementing multiple deployment policies and plans requires a very high level of control and flexibility. A deployment policy is simply a blockchain configuration that enables users to specify where and how the blockchain nodes are deployed and which clusters should exist or should not exist.
A deployment plan lets the user specify whether to deploy all the blockchain instance nodes into a single or multiple connected clusters.

Considered Options:

1. Helm SDK.
2. Kubernetes SDK.

Decision: Use Kubernetes SDK to implement the blockchain nodes and application deployment strategies and policies.

Benefits:

1. Fine-grained control: High control over Kubernetes resources and the ability to manipulate each resource programmatically.

2. Distributed Blockchain Nodes: Using this approach, we can define multiple deployment policies (high control and customization where each resource must get deployed and on which cluster in case of cross-cluster deployments.

Drawbacks:
Additional effort is required for programmatically implementing and realizing the deployment strategies and policiesd.

Alternatives:

1. Helm SDK: A tool for managing modules (charts) inside a Kubernetes cluster. Helm treats multiple resources as a single release, which makes it unsuitable for complete control over the nodes when the blockchain nodes are distributed among multiple clusters in the case of multi-cloud deployment.

Dependencies:

1. Define deployment strategies and policies.
2. Implement the specified strategies and policies using Kubernetes SDK.

References: [35, 25]

3.4. Components and Services Description/Design/Interfaces

This section will provide a deeper understanding and description of each component/microservice and clearly state how it will behave based on its provided interfaces/APIs.

Before diving into each microservice and its specific description and functionality, there are a set of standards that were applied when designing the microservices, which are:

1. REST -Representational State Transfer: An architectural style for designing web and networked applications, it is known for its statelessness, simplicity, scalability, and HTTP standardization by leveraging the existing HTTP protocol [20].
2. API-First Design: It is based on defining the set of APIs provided by a service as its core foundation. This approach provides many advantages like clarity of requirements, reduced duplication and redundancy, and client-server decoupling.
3. Pluggable Components: The components that each microservice communicates with are loosely coupled, which means components like Kafka cluster (for event-driven communication) and Keycloak identity and access management solution are loosely connected to these services, which makes switching to another technology in the future and simple a seamless process.
4. Leveraging Microservices Design Patterns: As the system is based on microservice architecture, different design patterns were applied in the architecture to address various challenges and ensure the platform's reliability and effectiveness [40].

a) Event-driven architecture: As discussed in the ADD ADD-002, event-driven communication is selected for inter- service communication since this pattern offers scalability, loose coupling, and real-time data flow.

b) API Gateway: Acts as the single entry point for the platform services by the client and other external systems, which makes it suitable for different cross-cutting tasks like authentication, routing, and rate limiting ADD-005.

c) Database Per Service: This pattern is widely applied when scalability and reduced data coupling are desired.

d) Containerization: To enable scalability and seamless cloud deployment process.

e) Single Responsibility Principle (SRP): Separate each microservice by its business capability, giving each of them a single responsibility to increase maintainability.

5. Layered Architecture Pattern [19]: Each microservice is designed based on this pattern, which separates it into several horizontal layers that function as a single unit. Each microservice is composed of three main layers following the Domain-Driver-Design Layered-Architecture approach:

   a) Application Layer: Accepts user commands and REST API calls, translate DTOs, apply authorization, and request validation.

   b) Domain Layer: responsible for the functionality of the business domain; it contains the domain services.

   c) Infrastructure Layer: responsible for managing the state of the microservice, communication for external services and components like event streaming platform, identity and access management component, cloud providers, Kubernetes clusters, and third-party external systems.

3.4.1. Cluster Management Microservice

The cluster management microservice is responsible for provisioning, deleting, and configuring Kubernetes clusters from different cloud providers. As discussed in the ADD ADD-003 and based on the FR FR-002, this service will use the cloud provider's specific SDK to be able to manage Kubernetes clusters on it. Furthermore, it must take care of the state of the managed and provisioned clusters in a database and act as a facade and control plane layer for the different supported cloud providers.

UML Class Diagram

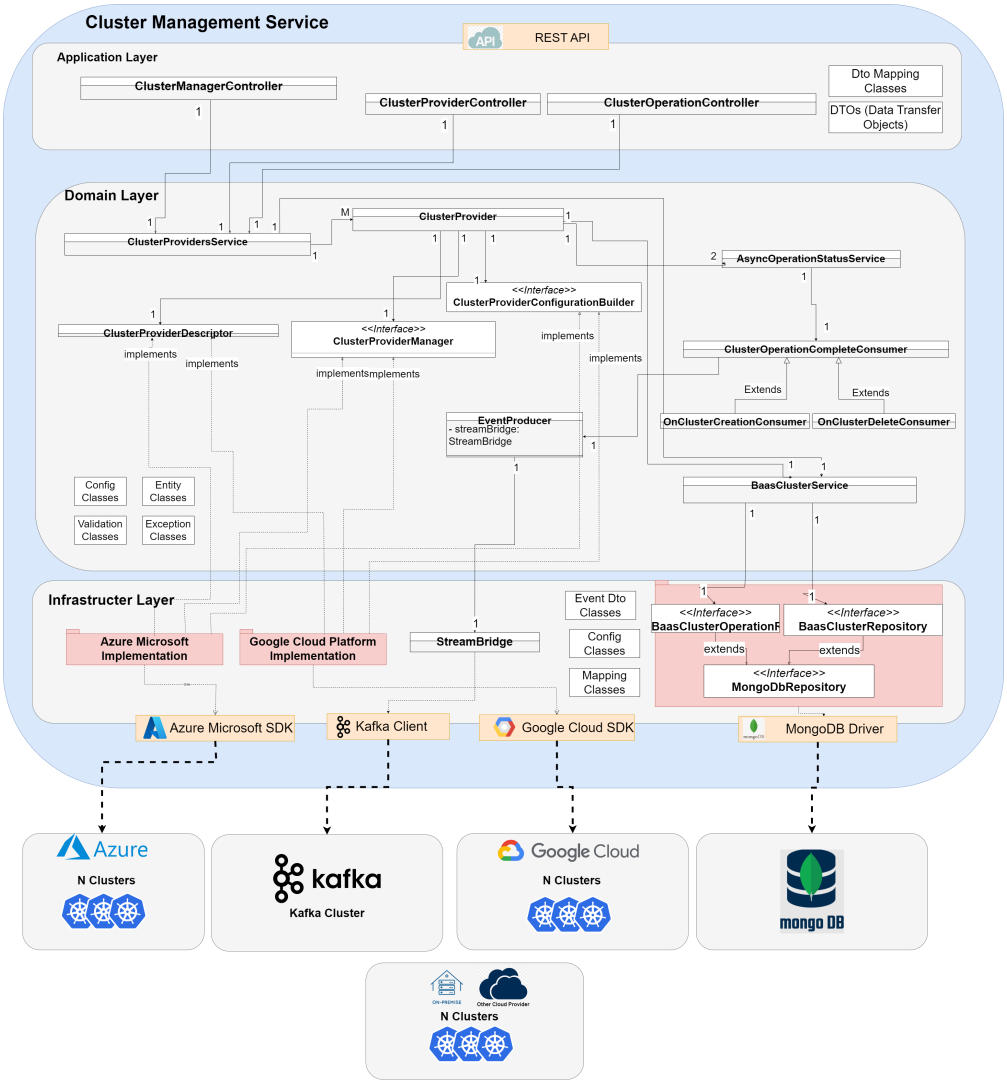

Figure 3.2.: Cluster Management Microservice - UML Class Diagram.

As Figure 3.2 shows, the cluster management service is composed of three layers:

1. Application Layer: It contains the controller classes that are responsible for receiving and handling REST API requests, including DTOs and DTO mapping classes. Furthermore, this layer handles the syntax validation and authorization of requests.

2. Domain Layer: It contains the interfaces and classes that together form a generic specification for managing different cloud providers' Kubernetes clusters.

3. Infrastructure Layer: We define concrete implementations for the specifications and interfaces provided in the domain layer for integrating Google Cloud and Microsoft Azure Cloud providers. Furthermore, this layer contains all the other classes and interfaces for managing the application's state using the NoSQL MongoDB database and producing events for their target topics in Apache Kafka for loosely coupled asynchronous inter-service communication.

API Reference

In the following, we describe each REST API endpoint the cluster management service provides and explain generically each endpoint input/output and its purpose. A detailed description of each endpoint's different parameters, payload, different responses, and authentication is present in the Git repository of this thesis in the cluster management service OpenApi manifest file.

- **GET /API/v1/providers**
  Description: *Retrieve a list of supported cloud providers.*
  Retrieve a list of supported Kubernetes cluster cloud providers, including their metadata (name, identifier, logo, meta-data) and configurations that can be applied by the user when provisioning a new cluster. For example, using Google Cloud provider, different regions and availability zones can be selected when provisioning a Kubernetes cluster.

- **POST /API/v1/providers/<providerIdentifier>/clusters**
  Description: *Provision a Kubernetes cluster on a specific cloud provider.*
  Create a Kubernetes cluster asynchronously on a cloud provider; the user submits the cluster name and configurations and returns an Operation DTO, which must be continuously polled once the provisioning operation is in DONE state.

- **DELETE /API/v1/providers/<providerIdentifier>/clusters/<clusterIdentifier>**
  Description: *Clear resources of a Kubernetes cluster on a specific cloud provider.*
  Deletes a Kubernetes cluster on a cloud provider by cluster identifier; this endpoint also returns an operation DTO that the user must continuously poll until the operation status is in DONE state.

- **GET /API/v1/clusters**
  Description: *Retrieve a list of all the Kubernetes clusters owned by a specific user on all the supported cloud providers.*
  Returns all the clusters the requesting user owns. These clusters can then be used for deploying blockchain applications and being integrated to enable cross-cluster multi-cloud communication.

- **GET /API/v1/providers/<providerIdentifier>/clusters/<clusterIdentifier>/operations**
  Description: *Retrieve all currently running operations on a Kubernetes cluster.*
  Returns all the operations currently being applied for a specific Kubernetes cluster

and their status. These operations are either CREATE Cluster, DELETE Cluster, or UPDATE Cluster operation.

- **GET /API/v1/providers/<providerIdentifier>/clusters/<clusterIdentifier>/operations/<operationIdentifier>**
  Description: *Retrieve a single operation for a Kubernetes cluster.*
  Returns the operation object for a Kubernetes cluster, which the user can use to poll the operation status after creating, deleting, or updating a Kubernetes cluster.

Cloud Providers Integration

The supported cloud providers by this service are selected based on three criteria:

1. Academic Licence: Most cloud providers offer an academic license. However, in most cloud providers, the academic license is limited and insufficient for the scope of the thesis.
2. Kubernetes Cluster: The cloud provider must support Kubernetes Service.
3. SDK: The cloud provider must offer an SDK that covers all the required operations to be done programmatically.

Therefore, the selected cloud providers are:

1. **Google Cloud Platform:** Offers GKE - Google Kubernetes Engine service, including the necessary SDKs, allowing multiple academic licenses [14].
2. **Microsoft Azure:** Offers Kubernetes service with an SDK that covers most of the required operations. Furthermore, access to the University of Vienna Azure portal was granted [4].

For each supported provider by this microservice, the user has to provide at least a **cluster name** that is unique among its account. Furthermore, the user must be able to choose a specific **region** or **availability zone** to deploy the Kubernetes cluster, which is an essential requirement to be able to give the platform the multi-region characteristic.

The microservice is designed considering maintainability and extendability characteristics, which means it can integrate any additional cloud provider or on-premise Kubernetes cluster with limited effort. Hence, integrating a cloud provider can be achieved by only implementing two interfaces and one abstract class without needing modification in any other place.

```
1  public interface ClusterProviderManager{
2
3      List<BaasCluster> getAllClusters();
4      List<BaasCluster> getAllClustersByAccountOwner(String accountOwner);
5      BaasCluster getCluster(GetClusterConfigurations getClusterConfigurations);
6      ClusterOperation provisionCluster(Configurations configurations, String ownerAccount);
7      ClusterOperation deleteCluster(DeleteClusterConfigurations deleteClusterConfigurations);
8      ClusterOperation clusterOperationStatus(GetOperationStatusConfigurations configurations);
9      Map<String,String> getClusterAuthData(GetClusterConfigurations getClusterConfigurations);
10 }
```

Listing 3.1: ClusterProviderManager

```java
public abstract class ClusterProviderDescriptor {

    private final String providerName;
    private final String providerIdentifier;
    private final String providerLogo;

    protected ClusterProviderDescriptor(String providerName, String providerIdentifier, String
    providerLogo) {
        this.providerName = providerName;
        this.providerIdentifier = providerIdentifier;
        this.providerLogo = providerLogo;
    }

    public abstract BaasClusterConfigurations getClusterConfigurations();
}
```

Listing 3.2: ClusterProviderDescriptor

```java
public interface ClusterProviderConfigurationBuilder {
    BaasClusterConfigurations getClusterConfigurations();

}
```

Listing 3.3: ClusterProviderConfigurationBuilder

The interface ClusterProviderManager shown in Listing 3.1 specifies creating and deleting operations that can be applied on a Kubernetes cluster resource using a cloud provider. Furthermore, it provides different options for querying existing Kubernetes clusters and their status.

The Abstract class ClusterProviderDescriptor shown in Listing 3.2 provides the required attributes and specification methods for describing a cloud provider through its metadata (name, identifier, logo, description) and configuration that can be applied to it by the client. The cloud provider configuration can be built by implementing the interface ClusterProviderConfigurationBuilder shown in Listing 3.3, which will be used by the implementation of the abstract class ClusterProviderDescriptor in Listing 3.2.

Extending the functionality and the operations in the future can be applied by introducing additional methods to the existing interfaces or introducing additional interfaces into the generic specification and implementing it by each supported cloud provider. For example, to support an automatic scheduled cluster upgrade, we could quickly introduce a new method in the interface 3.1 and implement it in each cloud provider.

3.4.2. Cluster Integration Microservice

As discussed in the ADD - Clusters Integration and Connectivity ADD-004 and based on the defined FR-Cluster Integration and Configuration FR-003, this service is responsible for integrating multiple Kubernetes clusters and establishing cross-cluster encrypted, secure and authenticated communication to enable the deployment and creation of multi-cloud multi-region blockchain applications.

UML Class Diagram

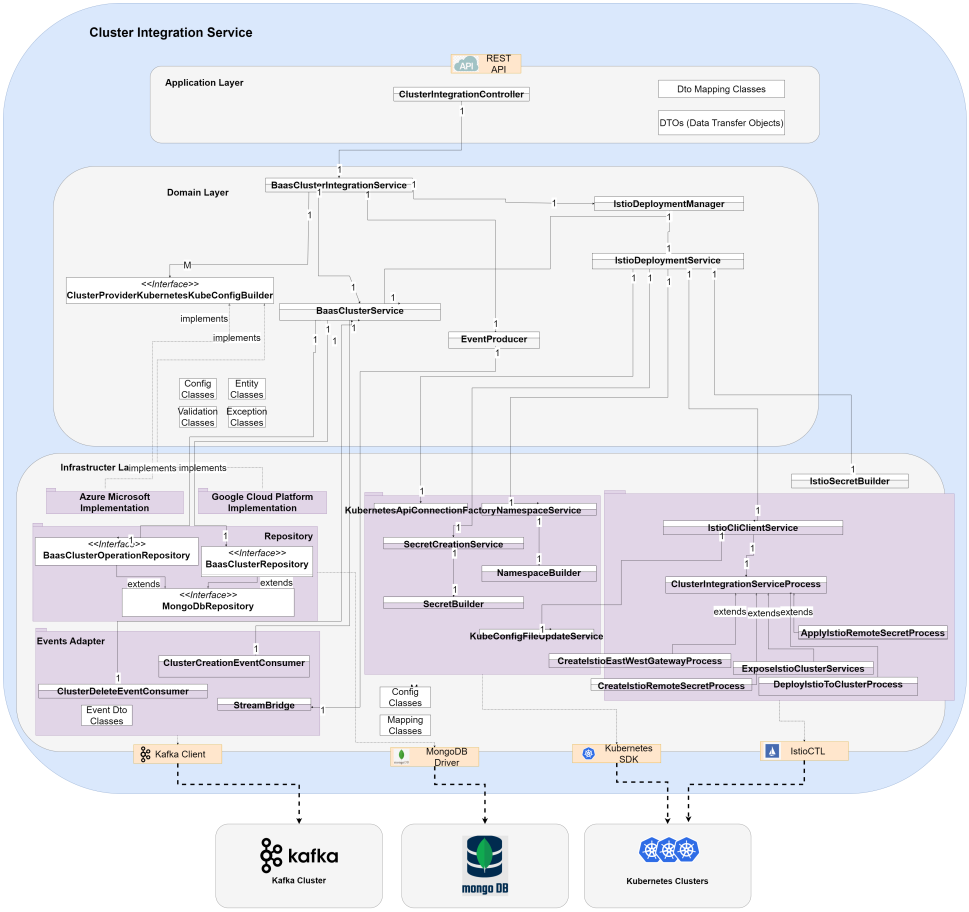

Figure 3.3.: Cluster Integration Microservice - UML Class Diagram.

As Figure 3.3 shows, the cluster integration service is composed of three layers:

1. Application Layer: It contains the controller classes that are responsible for receiving and handling REST API requests, including DTOs and DTO mapping classes. Furthermore, this layer handles the syntax validation and authorization of requests.

2. Domain Layer: It contains the interfaces and classes responsible for establishing and removing cross-cluster communication by installing the necessary components for each cluster to integrate. Furthermore, this layer defines interfaces each cloud provider must implement in the infrastructure layer. However, these interfaces are only used for constructing authentication credentials that can be used to access a Kubernetes cluster. Besides that, it defines specifications for managing Kubernetes

resources and cross-cluster communication resources that are implemented and extended in the infrastructure layer

3. Infrastructure Layer: Defines concrete implementations for the specification and interfaces provided in the domain layer for constructing Kubernetes cluster credentials and managing Kubernetes and Istioctl resources for cross-cluster communication.

API Reference

- **GET /API/v1/integrations**
  Description: *Retrieve all integrated clusters a specific user owns.*
  Returns all the integrated clusters a specific user owns, including their integration status. An integrated cluster is simply a set of connected Kubernetes clusters treated as a single resource by the platform to deploy blockchain applications.

- **POST /API/v1/integrations**
  Description: *Create an integrated clusters resource.*
  Creates a new integrated cluster resource by establishing connections between multiple Kubernetes clusters from various cloud providers. Other microservices can later use this resource to deploy multi-cloud multi-region applications.

- **DELETE /API/v1/integrations/<integrationIdentifier>**
  Description: *Remove existing integrated clusters resource*
  Remove existing integrated cluster resources by terminating the communication between all the included clusters and removing all the cross-cluster communication-related resources.

Cross-Cluster Communication - multi-cloud Connectivity

Based on the ADD - Clusters Integration and Connectivity ADD-004, a service mesh solution (Istio [22]) is applied to achieve cross-cluster connectivity and enable multi-cloud blockchain applications deployment.

To reach the point of being able to establish and terminate the desired cross-cluster communication dynamically, the different available resources, configurations, and deployment options of Istio Service Mesh must be analyzed.

Istio Network models

The Network model on Istio is offered in two options, The first one is **Single network** [22], which is appropriate for workload instances on the same network with non-overlapping IP addresses. The second option is **Multiple networks** [22], which is suitable for workload instances not in the same network and cannot directly reach each other. As the platform is meant to connect Kubernetes clusters across multiple cloud providers, which are, of course, not on the same network and may have overlapping IP addresses, the Multiple networks option was selected as the networking model for Istio in the context of cross-cluster communication.

Istio Control plane models
The Istio control plane maintains the communication configuration between the workload instances inside a service mesh [22]. A cluster owning its control plane is called **Primary Cluster** [22]. Meanwhile, in a Multi-Cluster deployment, a cluster that is using the other cluster's control plane that it is connected to is referred to as a **Remote Cluster** [22].

Following are the two different Control plane models:

1. **Primary-Remote:** Establish a multi-cluster service mesh by deploying a single control plane on the first cluster. Other clusters will not have their control plane and will use the one owned by the first cluster remotely, called Primary-Remote. This option is simple to apply with minimal configurations. However, it forms a massive single point of failure if the control plane fails. The cross-cluster communication is terminated, and all the blockchain nodes and applications across these clusters are affected [22].

2. **Multi-Primary:** Establish a multi-cluster service mesh by installing on each cluster its control plane. This option provides high availability, and if any control plane is down, only the resources inside it are affected. Furthermore, this option enables isolated service mesh configuration [22].

The BaaS platform needs to provide high availability and decentralization; therefore, **the option Multi-Primary was selected** [22] since it achieves high availability and prevents a single point of failure for the multi-cluster (multi-cloud) blockchain application and nodes by deploying a control plane for each Kubernetes cluster.

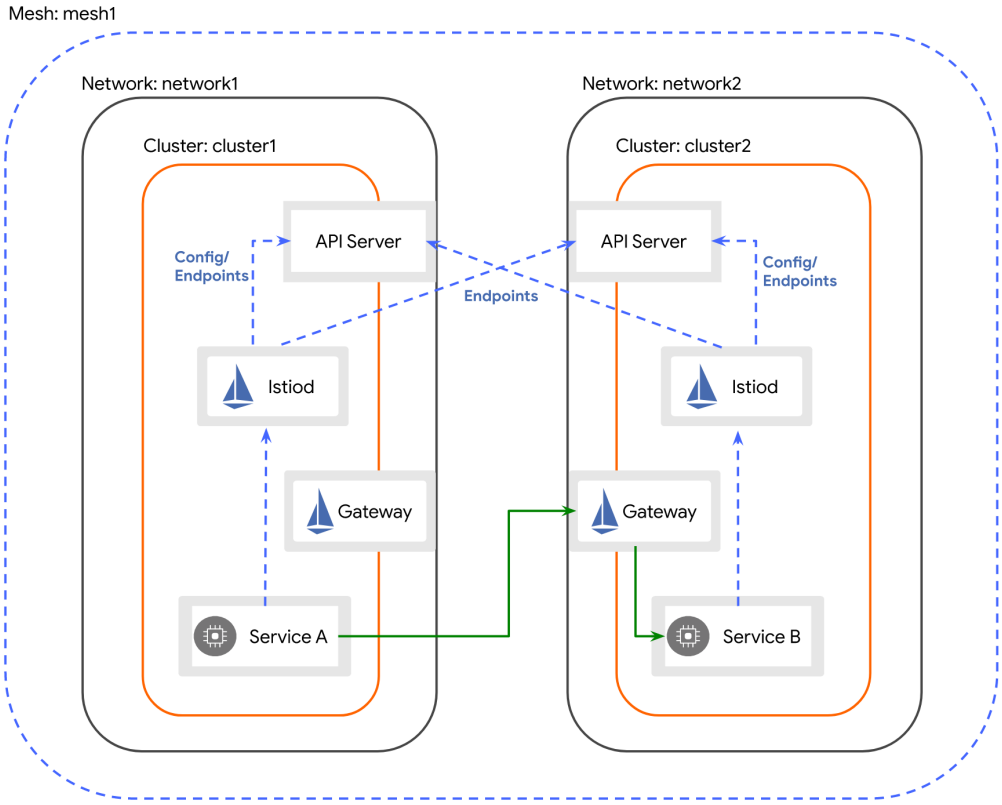

Figure 3.4.: Multi-Primary Multiple Networks Architecture. Source: [22]

Figure 3.4 shows the architecture we end with when applying the Istio Multi-Primary Multiple Networks deployment models. It comprises an Istio control plane and a dedicated east-west gateway on each cluster. The east-west gateway is responsible for receiving cross-cluster communication requests.

Identity and Trust Model

The previous Istio cross-cluster communication architecture must ensure an encrypted, authenticated, and secure connection. Therefore, the mTLS authentication protocol will be applied to achieve the desired secured service-to-service communication [22]. This protocol can be achieved by creating a certificate authority and using it to create and sign the certificates for each workload resource in the connected clusters to ensure that services authenticate against each other when communicating. Finally, Istio resources inside a mesh must be configured with mTLS, guaranteeing encrypted and authenticated cross-cluster communication [22].

Cross Cluster Communication between Google Cloud and Microsoft Azure

The supported cloud providers' environments must be correctly created and configured before starting with Kubernetes clusters provisioning and deploying cross-cluster communication-related resources.

Microsoft Azure [4]:

1. **Virtual Network:** is an isolated network within Microsoft Azure. Multiple Azure virtual networks must be created and configured to communicate with each other privately by enabling virtual network peering to enable multiple regions since each virtual network is bounded within a specific region.

2. **Virtual Network Components:** For each created virtual network, the **subnets**, **routing tables**, **network security groups**, and **IP address ranges** must be correctly configured to enable traffic coming from other cloud providers for multi-cloud cross-cluster communication.

3. **Azure service principal:** Create an identity to be used by the Blockchain-as-a-Service Platform to access the Azure resources and configure it with the required roles.

Google Cloud [14]:

1. **Virtual Private Cloud (VPC):** It is an isolated virtual network within the Google Cloud environment. A single VPC is sufficient as it is a global resource, which means it is not bound to any specific region.

2. **Subnets:** Since a subnet is bounded to a specific region, multiple subnets must be configured to enable multiple regions for Kubernetes cluster deployments.

3. **VPC Components:** Configure **routing tables**, **IP address ranges** and **firewall** to enable communication from other cloud providers.

4. **Service Account:** Create the required service account and configure its roles to enable the Blockchain-as-a-Service platform to authenticate against Google Cloud resources.

After applying the cloud providers' specific configurations and attaching the service principles and accounts to the microservices, we provision the Kubernetes clusters on both Google Cloud and Microsoft Azure using **Cluster Management Microservice**. Once the cluster is successfully deployed and active, we can deploy Istio on each cluster using cluster integration microservice to enable multi-cloud cross-cluster communication. Once Istio resources are also deployed, we end up with this architecture that enables multi-cloud, multi-region blockchain networks, and applications deployments.

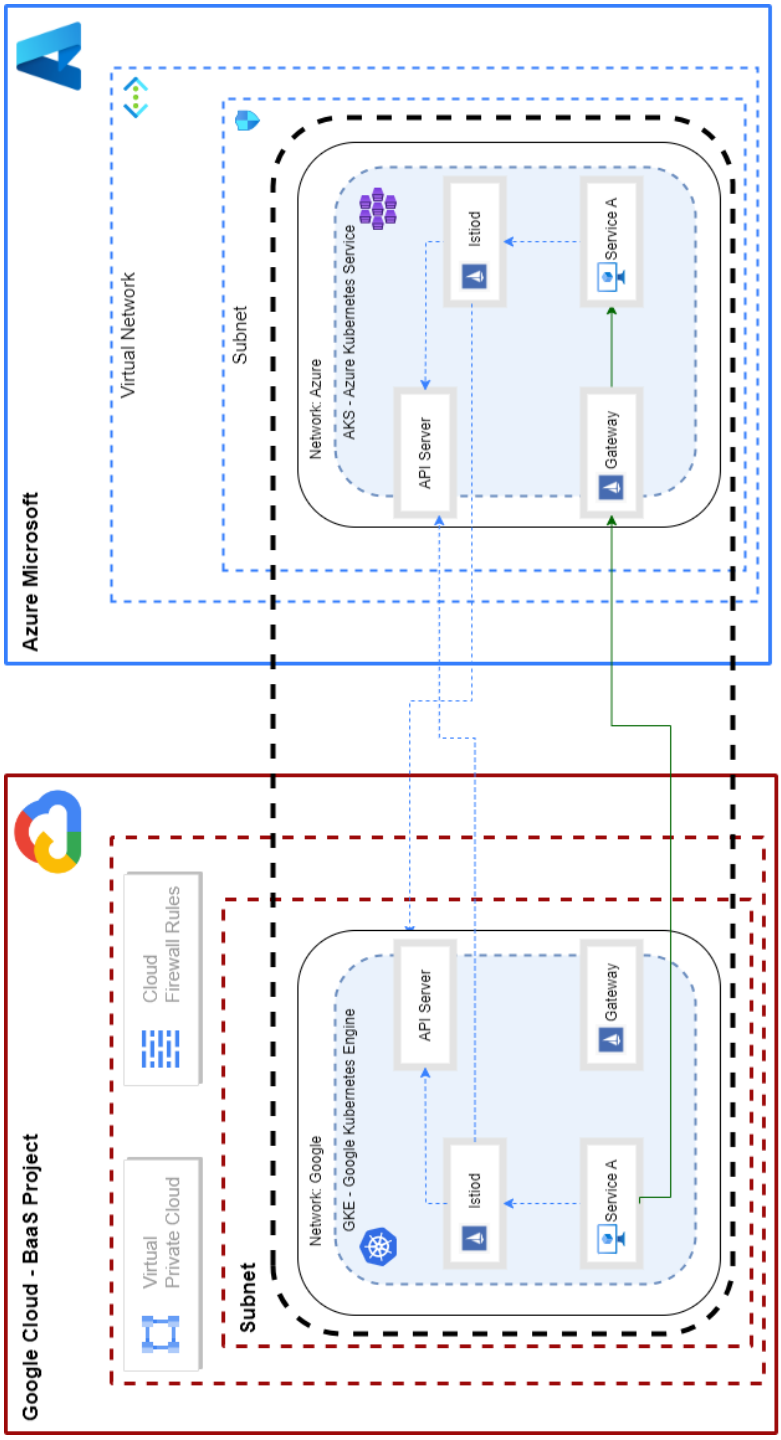

Figure 3.5.: Cluster Integration between Google Cloud and Microsoft Azure

Using the architecture shown in Figure 3.5, we can now establish multi-cloud, multi-

region encrypted, and authenticated communication between two Kubernetes clusters from different cloud providers. This solution will be managed as a resource in the BaaS platform and used to deploy blockchain network instances.

3.4.3. Blockchain Management Microservice

Based on the following decisions and requirements ADD-006, FR-005 and FR-006, this microservice will be responsible for creating and managing blockchain networks, applications, and nodes. This microservice supports **Custom Blockchain framework**, a self-implemented private and consortium blockchain framework. However, this service can be extended in the future to support more complex blockchain frameworks like Hyperledger Fabric or Private Ethereum for Enterprise.

This microservice depends on the **Cluster Management** and **Cluster Integration** microservices, which are responsible for provisioning and managing Kubernetes clusters and creating multi-cloud multi-region deployment environments for the blockchain network instances.

UML Class Diagram

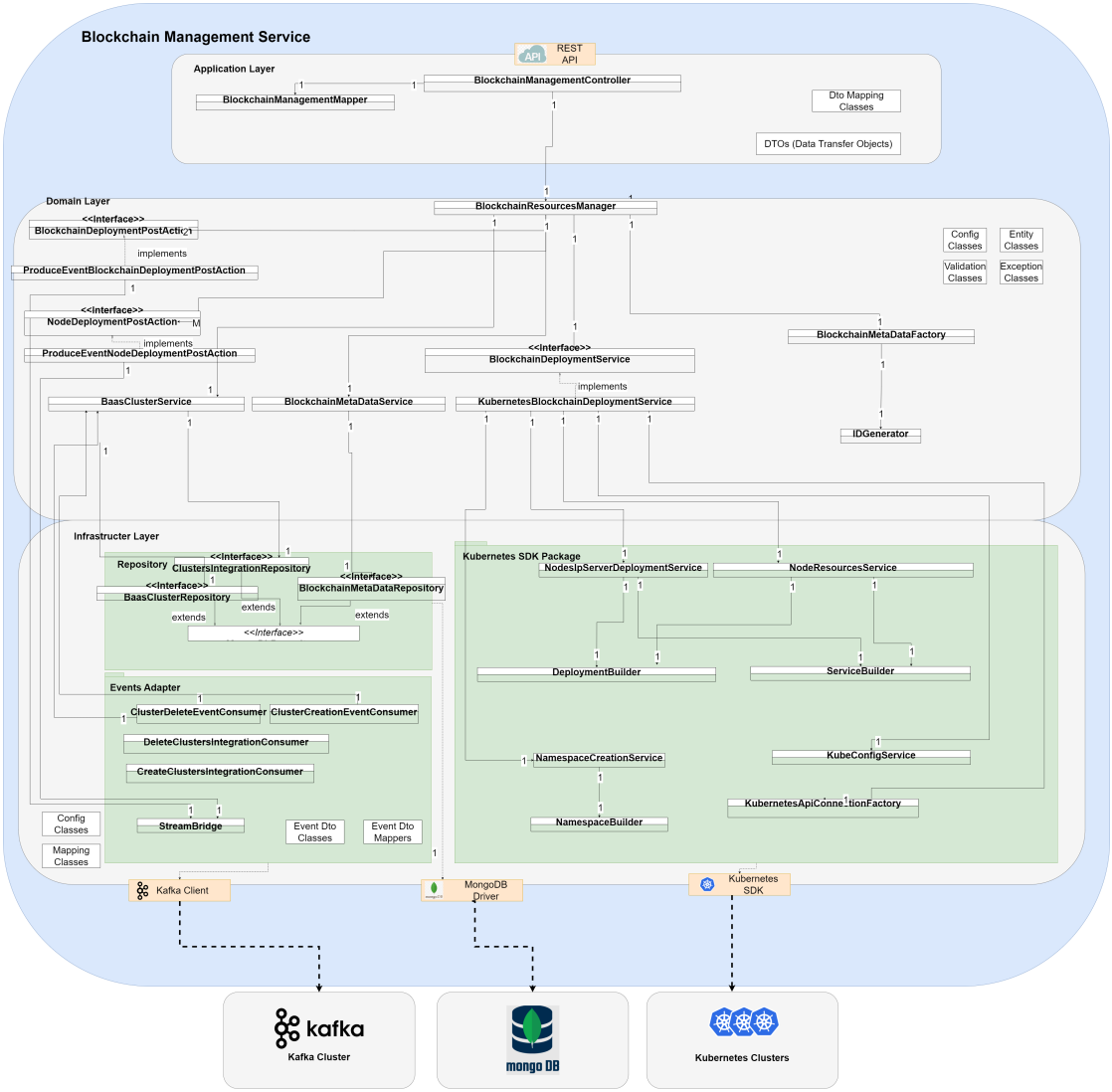

Figure 3.6.: Blockchain Management Microservice - UML Class Diagram.

As Figure 3.6 shows, the cluster integration service comprises three layers:

1. Application Layer: It contains the controller classes responsible for receiving and handling REST API requests.

2. Domain Layer: It contains the interfaces and classes responsible for creating, managing, and configuring blockchain network instances and their nodes and applications.

One of the core functionalities of this service is deploying a blockchain network either on a single Kubernetes cluster or on multiple connected Kubernetes clusters. This functionality is defined in the BlockchainDeploymentService interface and BlockchainResourceManager class. Furthermore, the domain classes and interfaces provide the ability to create a new node, append it to an existing blockchain network, and delete an existing node. Besides that, it allows the users to connect to their nodes and interact with them by providing the required data to connect to a blockchain node.

3. Infrastructure Layer: Defines concrete implementations for the specification and interfaces provided in the domain layer for constructing Kubernetes cluster credentials and managing Kubernetes resources on different clusters. Furthermore, it defines the classes for consuming and producing events to Apache Kafka and repositories to interact with the MongoDB database.

API Reference

- **GET /API/v1/blockchains**
  Description: *Retrieve all blockchain networks metadata that the calling user is involved in.*
  Returns all the blockchain network metadata for the blockchain that the calling user has already joined. A user is considered involved in a blockchain network when he owns a node.

- **POST /API/v1/blockchains**
  Description: *Create a new blockchain network instance.*
  Creates a new blockchain network with the specified number of initial nodes, including the required components for node discovery. The user specifies the number of initial nodes and who owns them. Furthermore, the user can specify if the blockchain must be deployed on a single Kubernetes cluster in SINGLE-CLOUD mode or distributed on multiple Kubernetes clusters, MULTI-CLOUD mode.

- **GET /API/v1/blockchains/<id>**
  Description: *Retrieve blockchain metadata by id*
  This endpoint returns the blockchain metadata for the user.

- **PATCH /API/v1/blockchains/<id>**
  Description: *Create a new blockchain node in an existing blockchain network instance*
  Given new node configurations like node name, owner account, email, and node type, a new node will be deployed in the existing blockchain network.

- **GET /API/v1/blockchains/<id>/nodes**
  Description: *Retrieve the blockchain node that a user owns in a blockchain network instance*

This endpoint returns the blockchain node with the defined ID the calling user owns.

- **GET /blockchains/<id>/nodes/<nodeId>/loadbalancer**
  Description: *Poll the required data to access a blockchain node created in a blockchain instance.*
  Once a new node is deployed in a blockchain network, it requires some time until its IP address is available and ready for serving user requests. This endpoint is used for actively polling the node metadata by the user until he or she gets valid access data for the node.

Supported Blockchain Framework

This microservice supports the Private and Consortium Blockchain Framework, described and documented in detail in the **Custom Blockchain Framework** section.
Extending this microservice to support additional blockchain frameworks like Hyperledger Fabric is possible; however, the services in the domain layer must be refactored and extended to provide the required support for additional blockchain applications.

Blockchain Deployment

This microservice offer different configurations when it comes to deploying blockchain application. The two essential configurations are **Deployment Plan** and **Deployment Policy**.

Deployment Plan:

1. **Single Cloud:** All blockchain resources will be deployed on a single Kubernetes cluster located on a single cloud provider, therefore called a single cloud deployment plan.

2. **multi-cloud:** The blockchain nodes and resources will be deployed in a multi-cloud, multi-region, and highly available fashion by distributing them among multiple Kubernetes clusters located on different cloud providers.
   Using this approach will prevent vendor lock-in and increase blockchain availability. If one Kubernetes cluster is down, then only the blockchain nodes and resources inside it are affected, and other nodes deployed in the rest of the Kubernetes clusters are unaffected.

Deployment Policy:

1. **Round-Robin:** In the case of a multi-cloud deployment plan, this deployment policy distributes the blockchain nodes in a round-robin fashion among the existing connected Kubernetes clusters. This approach is simpler as the user is not required to bind a specific blockchain node to a specific Kubernetes cluster.

2. **Manuel-Assignment:** In the case of a multi-cloud deployment plan, users must specify which cloud provider they prefer their blockchain node to get deployed by binding their node to a specific Kubernetes cluster. This policy is used in consortium blockchains where multiple enterprises are willing to deploy their nodes on different cloud providers.

   As the example Figure 3.7 shows, the node destination cluster to be deployed is decided based on the following formula:

   $$N \mod M = C$$

   N: The nodes count in the blockchain instance (excluding the current node that is being deployed).
   M: The number of the connected Kubernetes clusters.
   C: The target cluster index where the node is supposed to be deployed based on the round-robin formula.
   A new node must be scheduled to a cluster, based on the formula we have $N = 6, M = 4$, and therefore $6 \mod 4 = 2$, the cluster at index 2 is **Cluster 3**.

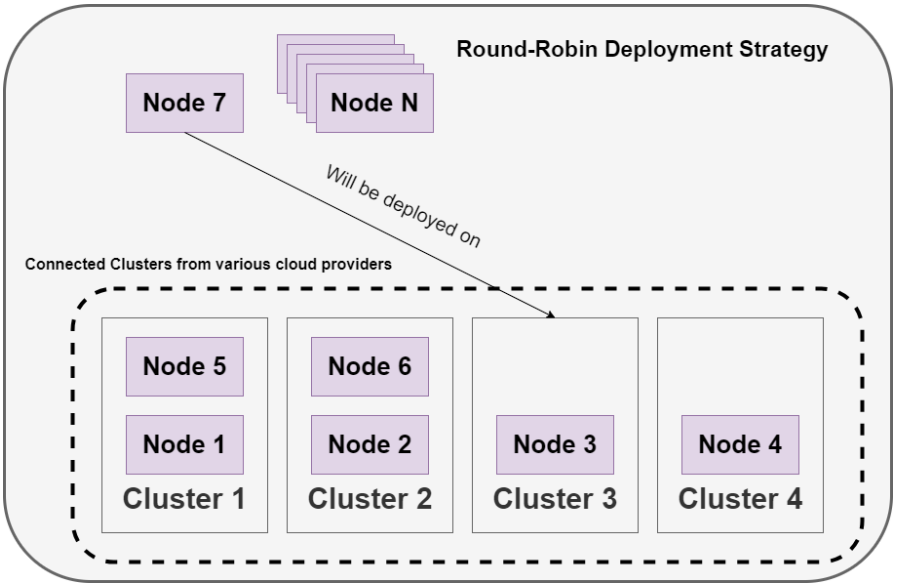

Figure 3.7.: Round-Robin Deployment Policy in Multi-Cloud Deployment Plan.

3.4.4. Notification Microservice

Based on the requirement FR-004, this microservice will be responsible for two functionalities: the first is to send notifications to users whenever a specific event occurs in the BaaS platform that they are involved in, and the second is to enable the users to configure

their notification preferences like enabling or disabling notifications for specific events categories.

UML Class Diagram

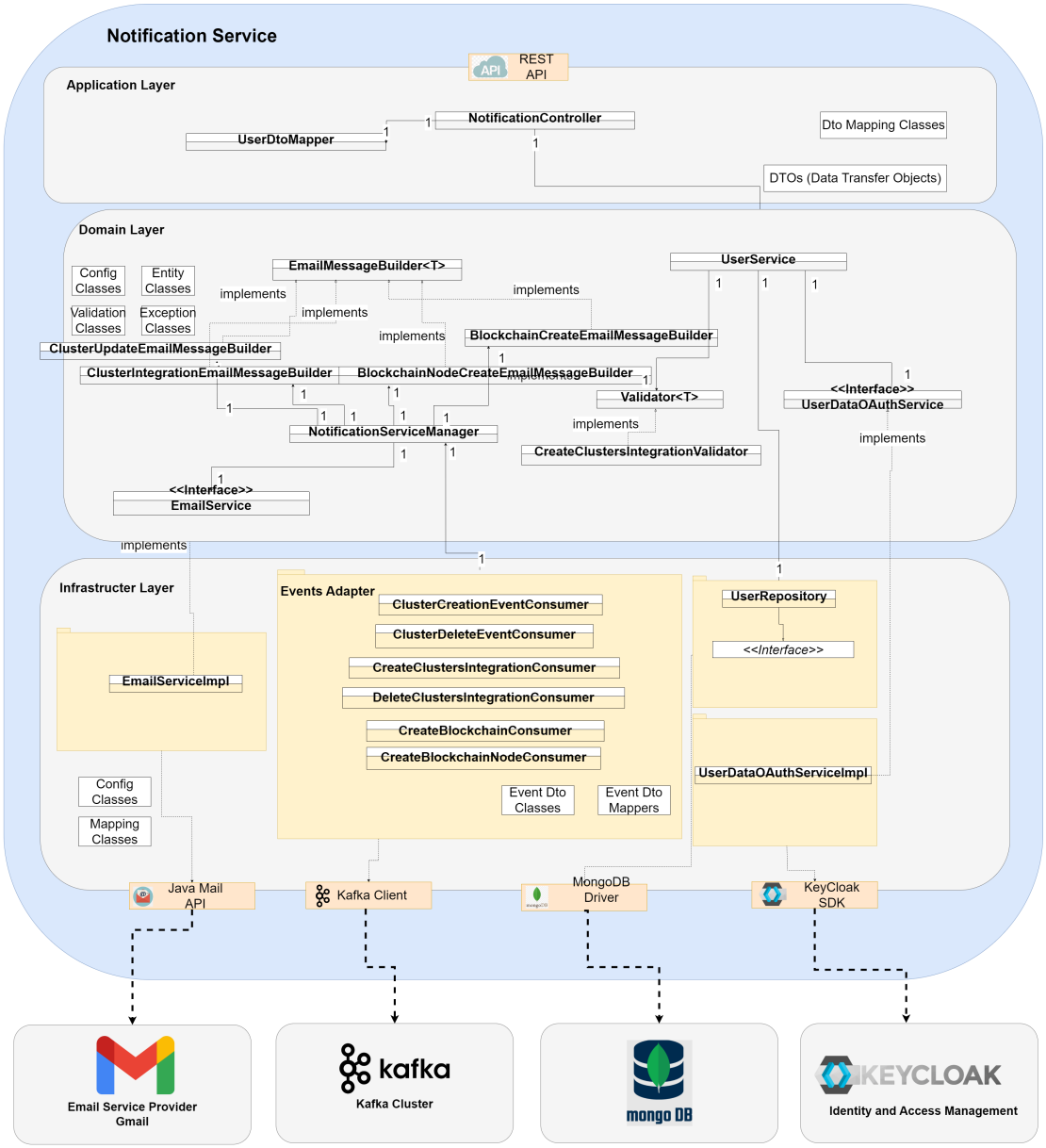

Figure 3.8.: Notification Microservice - UML Class Diagram.

As Figure 3.8 shows, the notification microservice is composed of three layers:

1. Application Layer: It contains the controller classes that are responsible for receiving and handling REST API requests, including DTOs and DTO mapping classes. Furthermore, this layer takes care of the syntax validation and authorization of requests.

2. Domain Layer: It contains the interfaces and classes responsible for managing user notification preferences, starting with the **UserService** class which handles notification preferences configuration requests. Furthermore, this layer defines the interface **UserDataOAuthService** to interact with the identity and access management component - Keycloak to sync the current platform users whenever a new user registers into the platform, meaning that this service only maintains user information that is necessary for notification management. Besides that, the class **NotificationServiceManager** class is responsible for sending notifications to users whenever the service consumes a new event.

3. Infrastructure Layer: Defines concrete implementations for the specification and interfaces provided in the domain layer for interacting with the identity and access management component, listening to Kafka events and interacting with the mail server component.

API Reference

- **GET /API/v1/notifications/users/<id>**
  Description: *Fetch the user current notification configurations.*
  Returns the current user notification configurations, which consist of categories of events that the user is interested in receiving notifications when they occur.

- **PUT /API/v1/notifications/users/<id>**
  Description: *Update the users current notification configurations.*
  Update the user's current notification configurations. For example, a user could send a request to enable notifications for blockchain node instantiation or cluster provisioning.

Notification Method

The notification service is configured using the email notification method; however, it can be extended to support additional notification methods like mobile SMS. The reason behind selecting this notification method is because it is the simplest method to implement and is sufficient for the scope of the thesis.

The configured mail server is Gmail SMTP Server since it is free to use when sending 500 emails per day or less, which is sufficient for the prototype and provides the required performance for the platform.

3.5. Inter-Service Communication

As discussed in the ADD-002, the microservices communicate with each other using the Event-Driven-Architecture [3] approach since it provides the highest scalability and decouples the services from each other.

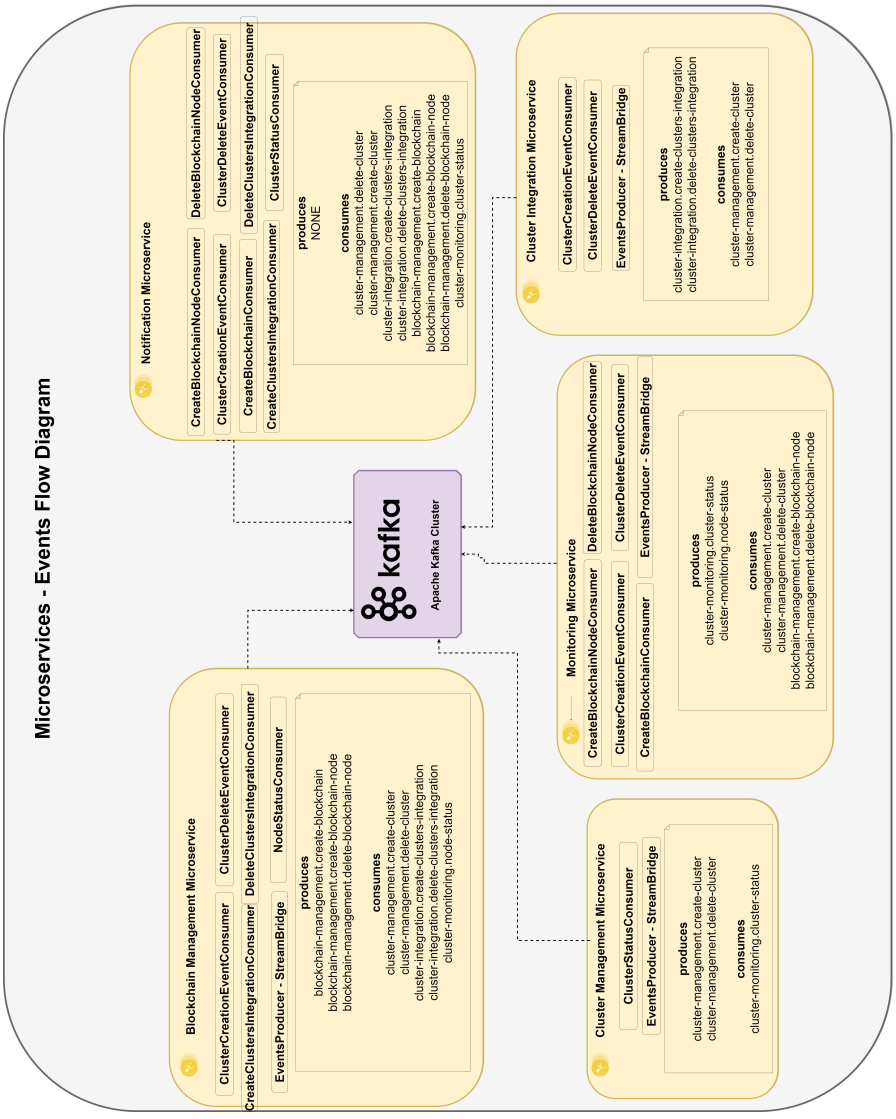

Figure 3.9.: BaaS Microservices Events Flow.

Figure 3.9 shows how the events are produced and consumed between each microservice in the platform [36].

3.6. BaaS Custom Blockchain Framework

On a high level, it is a private blockchain instance that is restricted to a single or multiple individual groups/businesses/organizations. Only authorized people can participate in it and view/create transactions. Those transactions are validated by nodes (validators) following the PoA - Proof-of-Authority consensus algorithm, which is based on the reputation/identity of a node.
This framework offers three types of nodes: wallet, validator, and full node.
Note: This blockchain framework will be used in the BaaS Platform. Its components and nodes will be dynamically managed and deployed on the cloud providers' Kubernetes clusters.

3.6.1. Structure

Blockchain Model

The Account model [55, 49] is used in this framework. The state in the blockchain is represented by the set of existing accounts and their state. The following accounts are supported:

1. An account with a public and private key used to create and sign transactions.

2. An account with a contract code specifying when a transaction/transaction must be created/confirmed (note: this feature is not implemented).

Blockchain Nodes

1. **Wallet Node:** Able to create and view transactions using its unique public and private keys, it does not validate blocks.

2. **Full Node:** Responsible for maintaining the blockchain state (blocks, accounts). It does not validate blocks and cannot create or sign transactions.

3. **Validator Node:** Wallet Node + Full Node also participates in the PoA consensus algorithm by verifying transactions once it is selected as a validator for the next block.

3.6.2. Consensus Algorithm

The used consensus algorithm is Proof-of-Authority - PoA [38]: Unlike other consensus algorithms like PoW or PoS, PoA is based on the reputation and identity of a node for validating transactions and blocks. A group of nodes will be selected, called validators, responsible for verifying pending transactions and accounts and adding them to the blockchain.

Advantages

- High transaction throughput.
- Fast block confirmation times.
- No need to perform computationally-intensive tasks like PoW.
- Well-suited for private or consortium blockchain networks.

Disadvantages

- More centralized than other PoW or PoS public blockchains.
- Security: The attacker can potentially control the network if a validator node is compromised.

Consensus Algorithm Description

A new block must be validated and created on a scheduled basis(every N seconds); this scheduled action is triggered simultaneously by all active validator nodes in the network. Only one validator will be randomly selected based on the selection strategy shown in the Listing 3.4; other validators will terminate the process since they are not selected and wait till the next block must be validated.
Once a validator is selected, it will create a new block, add the pending transactions and accounts, and broadcast it through the network.

Listing 3.4: PoA Consensus Algorithm - Validator Selection

Algorithm 1 PoA Consensus Algorithm - Validator Selection

```
validators ← sortedCurrentActiveValidatorsInNetwork
block ← previousValidatedBlock
previousValidator ← previousBlockValidator
N ← n
if block is not present then                    ▷ If its the first block being validated
    return validators[0]
end if
publicKeyAsInt ← hashAndConvertToInt(previousValidator.publicKey)
timeStampAsInt ← hashAndConvertToInt(block.timeStamp)
return (publicKeyAsInt+timeStampAsInt) mod validators.length
```

As the Listing 3.4 shows, each validator node fetches the current active validators list and sorts it ascendingly. After that, based on the previous block attributes (timestamp, validator public key), each validator node calculates a hash and converts it to an integer,

then applies the calculated integers modulo length of the current active validators list. This calculation gives us the selected validator for the next block, which is the same across all validator nodes that apply this calculation. In this way, we achieved a reliable and synchronized validator selection across all the validator nodes across the network.

3.6.3. Architecture

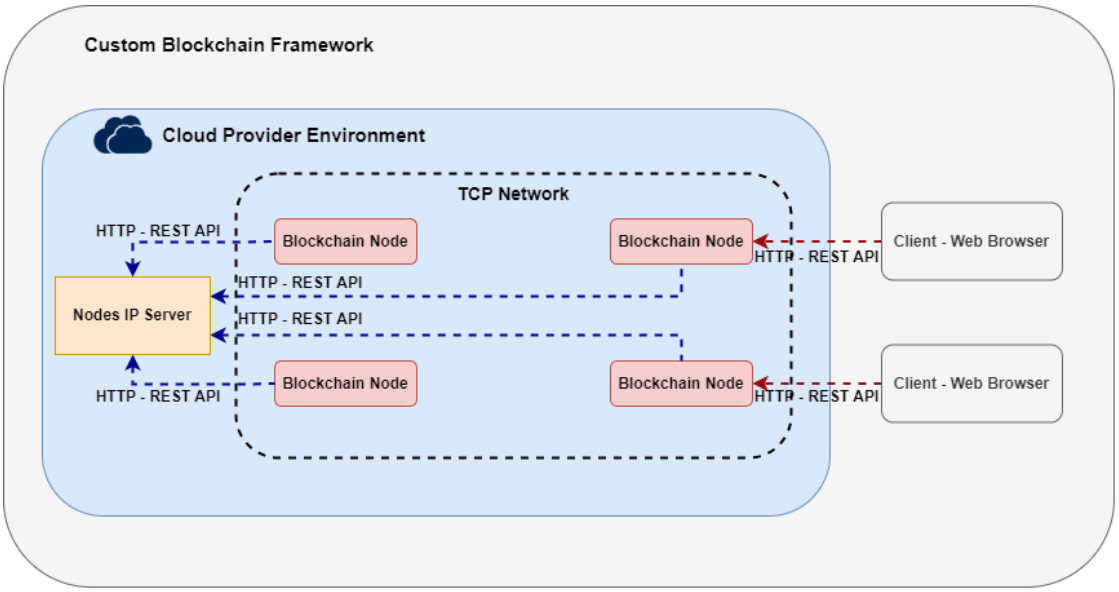

Figure 3.10.: Custom Blockchain Framework Architecture.

Figure 3.10 shows the components of the custom blockchain framework and how they communicate and interact with each other. The blockchain nodes communicate internally using TCP protocol with each other. They also communicate with the node's IP server initially to get the available nodes (peers) in the network using HTTP protocol.
The clients (validator and wallet node owners) can communicate with their nodes using REST API through the Web UI component.

3.6.4. Blockchain Entities Structure

Account Structure

1. Balance: Current account balance.
2. Public key: Used as an account identifier of a node and can also be used to verify if the sender's private key created and signed a specific transaction.
3. Private key: Used only by the node owner to sign transactions (stored privately in each node).
4. Username: A unique username that belongs to the node owner.

5. Nonce: Incremental number to avoid duplicate transactions; whenever a user creates a transaction, it gets incremented.

Transaction Structure

1. Transaction hash: The transaction identifier.
2. Sender public key.
3. Receiver public key.
4. Signature: The signed transaction by the sender's private key.
5. Sender nonce counter: The sender's nonce when applying this transaction.
6. Amount.
7. Timestamp: Transaction creation timestamp.

Block Structure

1. Block Hash: Block identifier.
2. Previous Block Hash: Previous block identifier.
3. Merkle Root: Calculated Merkle root for transactions verified in this block.
4. State Root: Calculated Merkle root for the state after applying the transactions in this block.
5. Validator Public Key: Validator identifier.
6. Height: Height of block (initially 0).
7. Pending Accounts: Pending accounts set.
8. Transactions: Pending transactions set.
9. Timestamp: Block creation timestamp.

Merkle Tree

In this blockchain framework, the Merkle tree is applied in two different scenarios:

1. **Transactions:** The transactions act as the leaves in each block, and every two elements (transactions) will be hashed together until we end up with a single hash value, the Merkle root. After constructing the Merkle tree and Merkle root, it is then possible for a node to ask if a transaction is confirmed to a block or not with simple payment verification and trust the response.

2. **Accounts and State Management:** The Merkle tree is also used in the context of accounts, which makes it easier for validators to verify the new calculated state (accounts) after executing the pending transactions.

3.6.5. Validation Rules

Transaction Validation Rules

A node only considers a transaction valid when more than **50 percent** of the nodes accept it. A node only accepts a transaction when the following rules are not violated:

- The Transaction hash must be correctly calculated.
- Amount is greater than zero.
- The transaction amount is equal to or less than the sender balance (including other pending transactions in each node's pending transactions pool).
- The Transaction nonce must be greater by one than the previously accepted or pending transaction.
- The receiver account must exist.
- The transaction signature must be successfully verified.

Block Validation Rules

- The Block hash must be correctly calculated and validated against the received block from other nodes.
- The Block height must be greater than the previous block height by one.
- The Calculated Merkle root must be verified against the received block.
- The calculated state root must be verified against the received block.
- The timestamp must be greater than the previously created block.
- The pending account's initial balance must be correct.

3.6.6. API Reference

Node User API Reference

This section describes the APIs the wallet and validator nodes offer when deployed on the cloud providers' Kubernetes clusters. It comprises a set of REST API endpoints that allow the users to use all the functionality provided by the blockchain framework.

- **GET /accounts/wallet-account-status**
  Description: *Fetch wallet account status.*
  Returns the wallet account data and can be used to poll the account status to check if it is confirmed into the blockchain (is appended into a block and transitioned from pending to confirmed status).

- **POST /accounts**
  Description: *Activate Account*
  Once a user gets an invitation into the blockchain, he/she must initially activate his/her account by calling this endpoint, which will schedule this account to be appended into the next block that will be validated and inserted into the blockchain. By invoking this endpoint, the user can activate his account.

- **POST /transactions**
  Description: *Create and Sign a new transaction.*
  Creates a new transaction that specifies the receiver, amount, and nonce.

- **GET /transactions**
  Description: *Retrieve all transactions in a blockchain.*
  Returns all the created and pending transactions in a blockchain

- **GET /accounts**
  Description: *Retrieve all activated accounts*
  Returns all activated accounts in a blockchain, which can be later used to reference them for sending transactions.

- **GET /blocks**
  Description: *Retrieve all blocks in a blockchain*
  Returns all blockchain block information, further information about transactions and accounts accepted in each block, can be queried using GET /accounts and GET /transactions endpoints.

Inter-Node Communication API Reference

This section describes the communication protocol between the blockchain nodes. The blockchain nodes communicate with each other using a Peer-to-peer TCP network that provides reliability and ensures message delivery.

Peer Discovery
It is implemented simply using an additional component **Nodes-IP-Server**, which is used as a bootstrap endpoint for any node. A node can register its IP: Port pair in this server, and on its startup, it receives all the available (registered) nodes in this server and ping them to validate their availability.

TCP Message Structure
-Message-Id: An identifier that uniquely identifies a TCP message.
-Endpoint: To let the receiver know which function should be invoked.
For example,/create-block, /create-transaction .
-Source Identifier: Represents the ip and port of the sender.
-Destination Identifier: Represents the ip and port of the receiver.
-Message Type: REQUEST, RESPONSE, or PING.

-Status Code: OK or ERROR.
-Error Message: Optional (only present when an error status code is returned.
-Payload: Payload of the request or the response in JSON.

TCP Endpoints (only for Inter-Node Communication)
1. /create-blocks: This is called by a validator once a new block is validated to send it to other network nodes.
2. /block-by-height: Get a block by height.
3. /blocks: Get all blocks.
4. /create-account: Invoked by the user to activate its wallet account.
5. /get-account: Get user account data.
6. /accounts: Get all accounts in a network.
7. /account-state: Get account state.
8. /get-transactions-by-account-id: Get transactions that a specific account is involved in.
9. /transactions-state: Get all transactions state.
10. /transaction-state: Get transaction state by transaction id.
11. /create-transaction: Create a transaction.
12. /get-validator-metadata: It is used by validators to synchronize the current active validators.
13. /validate-transaction: It is called by a wallet or validator node to verify a transaction in blocks using the Merkle root and Merkle path.

3.6.7. Custom Blockchain Framework using BaaS multi-cloud Architecture

Deploying this blockchain framework using the previously done multi-cloud environment decisions and setup would result in the following architecture shown in Figure 3.11.

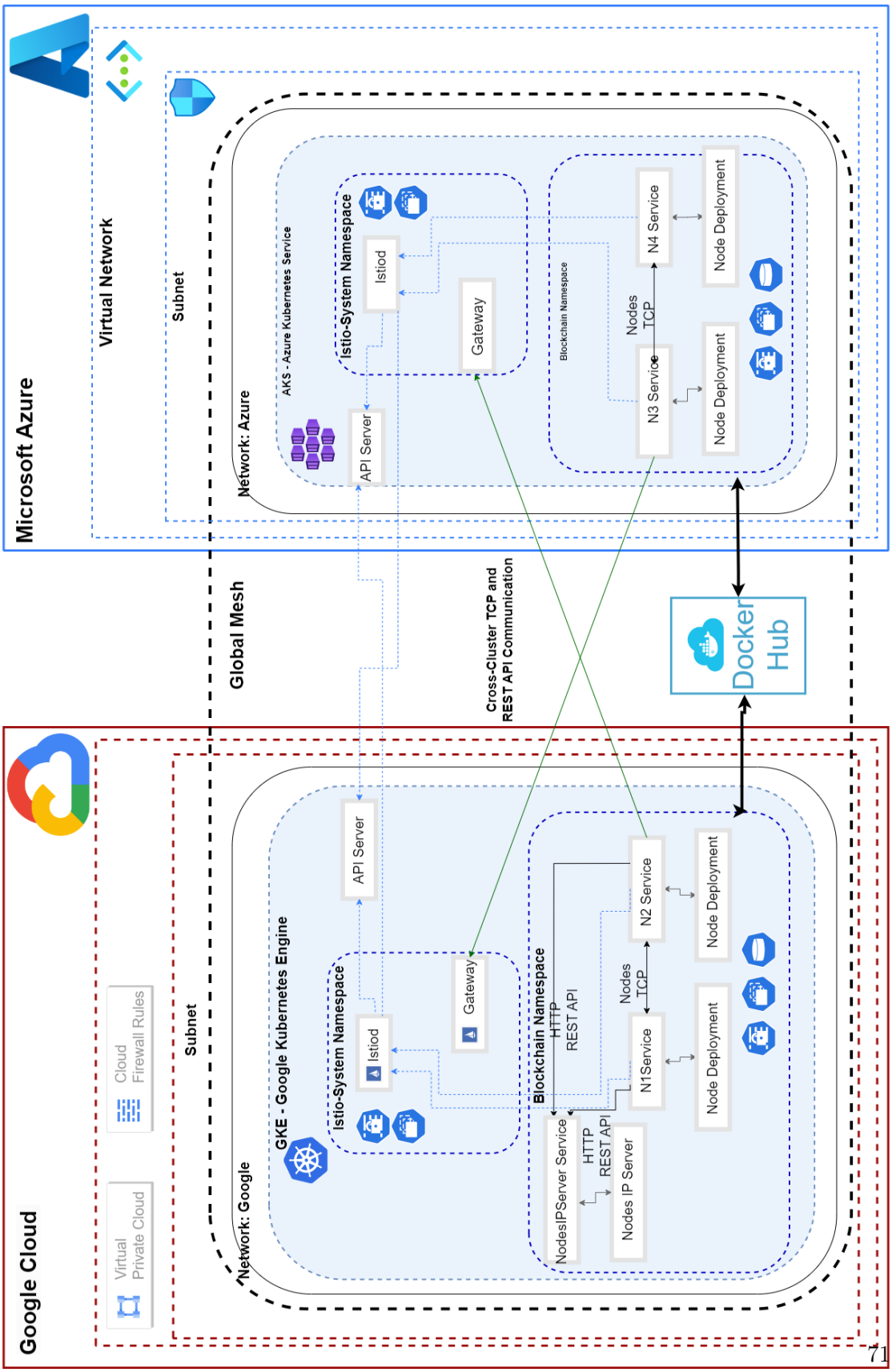

Figure 3.11.: Custom Blockchain Framework Multi-Cloud Architecture.

As Figure 3.11 shows, we deployed a blockchain instance in the multi-cloud environment. It is done by dynamically creating multiple Kubernetes resources on top of the prepared environment.
We create a **Kubernetes namespace** for the new blockchain with a generated ID as the namespace name. After that, we create the required **config maps and secrets** followed by creating a **deployment** for each node by pulling the blockchain framework image from **Docker Hub** and expose this deployment internally using a **Kubernetes service**. Then, we create a deployment and service Kubernetes objects for the Nodes-Ip-Server, also using Docker Hub to pull the Nodes-Ip-Server image. This component is required to enable peer discovery between each deployed node.
Finally, we expose the blockchain nodes to the clients by provisioning a **Google Cloud Load Balancer** for each node service. The inter-node TCP traffic and HTTP node-to-nodes-ip-server traffic are propagated authentically, encrypted, and securely between the two cloud environments.

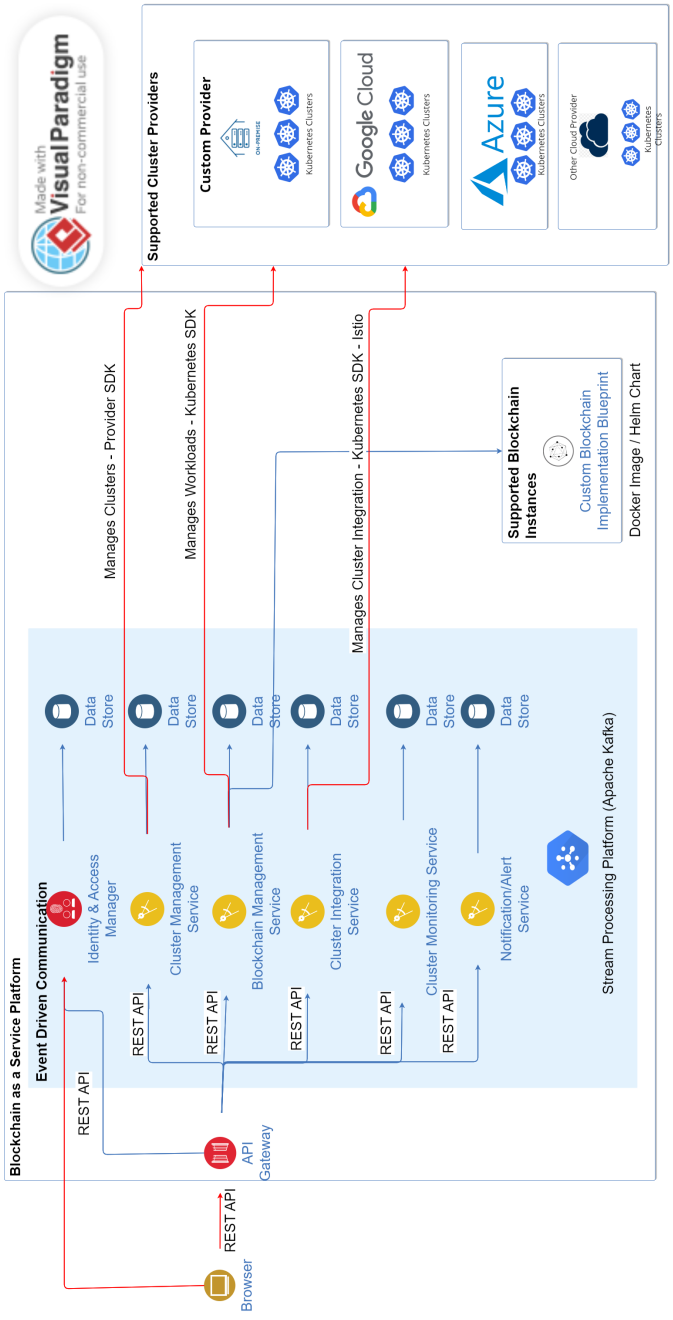

Figure 3.1.: BaaS Architecture Specification.

4. Implementation

This chapter gives an in-depth exploration of the technology stack used in the project, followed by going through the implementation details of the different microservices and components, and finally by providing a step-by-step guide to run the prototype.

4.1. Technology Stack

The technology stack outlined in Table 4.1 has been carefully chosen based on the project's functional and non-functional requirements, architectural design decisions, and our expertise in different technologies.

Table 4.1.: Project Components and Descriptions

Component/Feature	**Description**
Programming Language	Java 17, JavaScript, HTML5 and CSS.
Framework	Spring Boot 3.1.1 and Vue JS 3.
IDE	IntelliJ IDEA 2023 and Visual Studio Code.
Build Management Tool	Gradle 8.1.1 and NPM 9.8.1.
Source Code Management	Gitlab.
Documentation	JavaDoc.
Logging	SLF4J API.
Persistence	MongoDB 6.0.
Cloud Providers	Google Cloud and Microsoft Azure.
Containerization	Docker, Docker Hub and Kubernetes.
Network Communication Protocols	TCP, REST API.
Inter-Cluster Communication	Istio service mesh 1.18.
SDK	Docker, Kubernetes, Google Cloud, Azure, Keycloak, Apache Kafka, Java Mail API and IstioCTL.
API Gateway	Spring Cloud Gateway 2022.0.4.
Identity and Access Management	keycloak 22.0.1
Event Driven Architecture	Apache Kafka 2.8.1.
Kubernetes	1.27.3-gke.1700 and 1.26.6.

4.2. Implementation

This section will describe the important implementation details of each component and microservice on the BaaS platform.

4.2.1. API Gateway

The Spring Cloud Gateway framework is used as the API Gateway component for the BaaS platform, and It is based on Spring WebFlux, designed to support fully non-blocking reactive streams. It uses Netty as the built-in server to run reactive applications.
The gateway is used in the platform for various cross-cutting concerns.

Routing

The API Gateway effectively routes each received request by the client to its destination microservices based on different defined routing rules. These rules are simply predicates that either match a specific request or not. There are various predicates like path, header, and domain predicates to match a specific request to a specific microservice. Listing 4.1 demonstrates how the routes are defined for each microservice.

```
routes:
    - id: notification-service
      uri: ${NOTIFICATION_SERVICE_URL:http://localhost:8080}
      predicates:
        - Path=/api/v1/notifications/**
    - id: cluster-manager-service
      uri: ${CLUSTER_MANAGEMENT_SERVICE_URL:http://localhost:8080}
      predicates:
        - Path=/api/v1/providers/**,/api/v1/clusters/**
    - id: blockchain-management-service
      uri: ${BLOCKCHAIN_MANAGEMENT_SERVICE_URL:http://localhost:8080}
      predicates:
        - Path=/api/v1/blockchains/**,/blockchains/**
    - id: cluster-integration-service
      uri: ${CLUSTER_INTEGRATION_SERVICE_URL:http://localhost:8080}
      predicates:
        - Path=/api/v1/integrations/**
```

Listing 4.1: API Gateway Request Routing

Security

The API Gateway performs OpenID Connect Authentication; only authenticated requests can access the microservices. After correctly configuring the Keycloak Identity Provider, the Web UI, API gateway, and microservices must be correctly configured according to the applied configurations to the identity provider. The API Gateway will be responsible for handling authentication and each microservice is responsible for authorization handling.

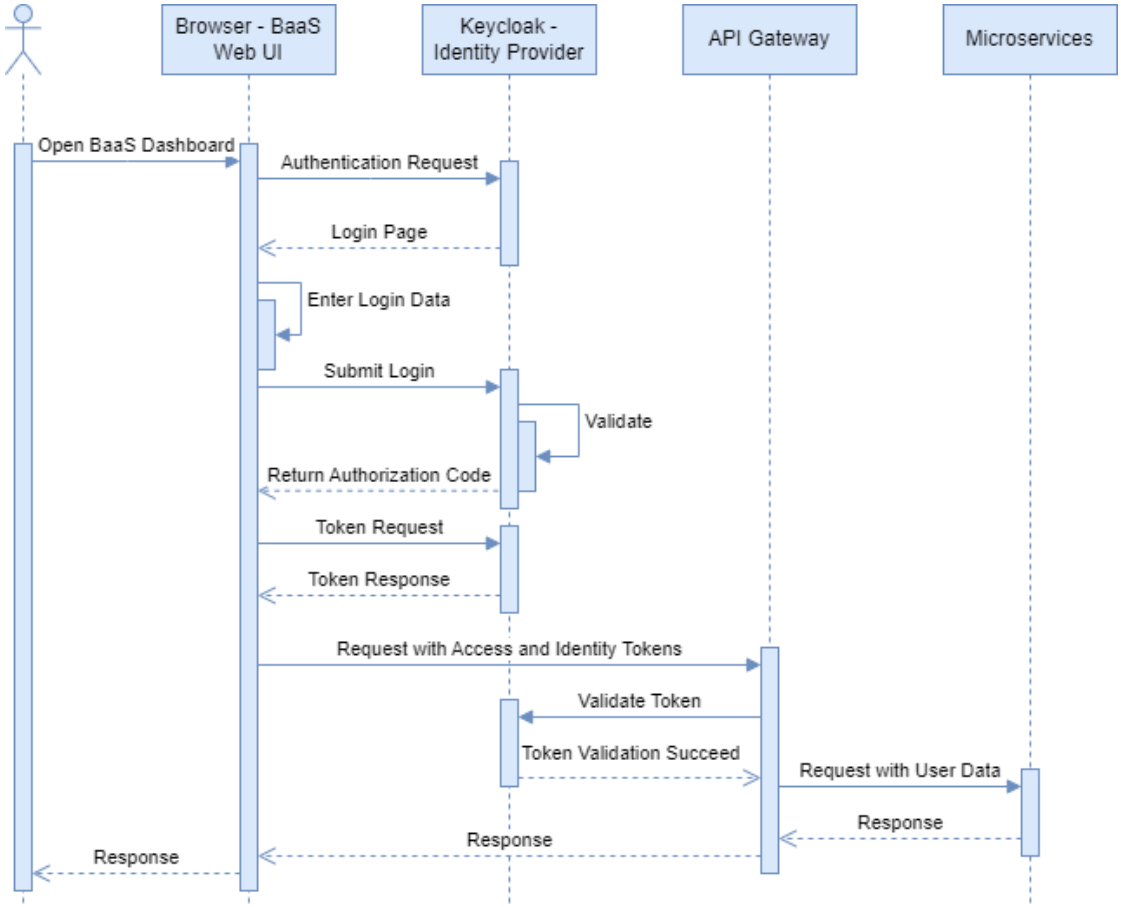

Figure 4.1.: BaaS Platform Successful Login Authentication Flow.

Figure 4.1 shows how the different components interact with each other when the users attempt to visit the BaaS platform dashboard page. Once the user enters the platform domain into the browser, he/she will get forwarded to Keycloak if no authentication session is present. The user enters his authentication data (username and password), submits his input, and returns to the BaaS platform. After that, the BaaS platform uses the received authorization code to request access, refresh, and ID tokens directly from Keycloak. Once those tokens are received, the user requests (to view the BaaS dashboard and profile) will be forwarded to the API Gateway components, including the access and identity tokens, which the API gateway against keycloak will validate. If the token is valid, the request will be forwarded to the correct microservice, which will then authorize the request and return the response back to the client through the API gateway.

Request And CORS Configurations

The API Gateway applies configuration regarding the requests configurations like the connection and response timeout. Furthermore, it configures cross-origin resource sharing, which is required for local development and cloud deployment.

4.2.2. Identity And Access Management Component

Keycloak is integrated into the platform by simply deploying it as a docker image; once deployed, we configure it by creating a realm for managing users, applications, and OAuth clients. After that, we configure a client with the required attributes like ID, protocol, redirect URLs, login/registration pages, secrets, and other attributes.
Finally, the components that interact with Keycloak, which are the client (frontend application) and API Gateway, will be configured with the previously defined client configurations.

4.2.3. Event Driven Communication - Apache Kafka

As discussed in the design chapter, an event streaming platform is utilized for event-driven communication between the BaaS microservices. Which is deployed using a docker image and automatically registering the topics into the broker using the producers' microservices. Almost all of the microservices are producers and consumers at the same time.
Spring Cloud Stream, a framework for enabling loosely coupled event-driven communication between scalable microservices, is used in each microservice. Spring Cloud Stream offers several advantages; one is making the event streaming platform a pluggable component into the microservices using a binders layer, which eliminates the effort to write code for a specific broker and simplifies the migration process to another event streaming platform.

4.2.4. Data Store

NoSQL - MongoDB document database technology is used as the data store for each microservice due to its flexibility. It does not require a schema definition, and it is known for its high scalability and high read/write throughput.

In the BaaS platform, each microservice is supposed to have its own data store following the database per service pattern in a microservices architecture. It is deployed by setting up the MongoDB image, volume, and networking configurations.

4.2.5. Microservices

Clean Code Standards

All the implemented microservices follow the clean code standards regarding the code directory and file structure. The code is also formatted using the IntelliJ IDE automatic code formatting.

Besides that, all the properties used in each microservice are dynamically injected, and no hard coding is present. The naming convention is done following the Java standard naming convention rules defined by Oracle.

To ensure delivering high-quality code, the implemented microservices follow **SOLID Design principles**, **Do not repeat your self - DRY**, **You are not going to need it - YAGNI**, **Separation of Concerns - SoC**, **The Law of Demeter - LoD**, **Composition Over Inheritance**, **Law of Closure**, and **KISS** software engineering principles.

Shared Dependencies and Requirements

- **MongoDB:** Each microservice requires a running NoSQL MongoDB component for managing the persistence layer (Note: it is also possible to have the same MongoDB server used for multiple microservices).
- **Apacke Kafka:** All microservices require a running Apache Kafka Cluster that handles inter-service communication as asynchronous events.
- **API Gateway:** All microservices require a running API Gateway component, which is the entry point for the BaaS Microservices.

API-first - REST API

The microservices API layer is implemented based on an API-first approach using OpenAPI Specification, a format for describing and documenting APIs in different formats.
For each microservice, an OpenApi specification file in JSON is defined, including all the rest of the endpoints, responses, parameters, payloads, and response codes. Listing 4.2 shows an example endpoint defined in the Cluster Management Microservice OpenApi file.

```
openapi: 3.0.1
info:
  title: OpenAPI definition
  version: v0
servers:
  - url: http://localhost:8080
    description: Cluster Management Microservice
paths:
  /api/v1/providers:
    get:
      tags:
        - cluster-provider-controller
      operationId: getCurrentSupportedProviders
      responses:
        "200":
          description: OK
          content:
            'application/json':
              schema:
                $ref: '#/components/schemas/ClusterProvidersDescriptionDto'
```

Listing 4.2: Cluster Management Microservice OpenAPI - GET /providers endpoint

Using the OpenAPI build management plugin, the REST API contract interfaces and DTOs can be generated in each microservice, which later on can be defined with a concrete

implementation.
Furthermore, request validation for payload and parameters can be defined in the OpenAPI manifest file. However, it is limited for syntax validation like non-nullable fields and specific input regex.

Cluster Management Microservice

The main functionality of this microservice is managing and provisioning Kubernetes clusters from various cloud providers. As discussed in the design chapter, this is implemented following the SOLID principles by defining interfaces and an abstract class each new cloud provider must implement that the platform is willing to support.

```
1  package at.ac.univie.baas.clustermanager.provider.specification;
2
3  public class ClusterProvidersService {
4
5      private final Set<ClusterProvider> clusterProviders;
6      private final BaasClusterService baasClusterService;
7      private final CreateClusterValidator createClusterValidator;
8      private final DeleteClusterValidator deleteClusterValidator;
9
10     public List<BaasClusterProviderDescription> getProvidersDescription(){
11         return this.clusterProviders.stream()
12                 .map(ClusterProvider::getProviderDescriptor)
13                 .map(ClusterProviderDescriptor::getClusterProviderDescription)
14                 .toList();
15     }
16
17     private ClusterProvider getProviderByIdentifier(String identifier){
18         return this.clusterProviders.stream()
19                 .filter(provider -> provider.clusterProviderIdentifierEqualsTo(identifier))
20                 .findAny()
21                 .orElseThrow(() -> new BaasClusterManagerException(String.format("The cluster
       provider with ID: %s doesnt exist",identifier),HttpStatus.BAD_REQUEST));
22     }
23
24     public List<BaasCluster> getClustersByAccountOwner(String accountOwner){
25         return this.baasClusterService.findAllClusters()
26                 .stream()
27                 .filter(cluster -> cluster.getOwnerAccount().equals(accountOwner))
28                 .filter(cluster -> !cluster.getIsDeleted())
29                 .toList();
30     }
31
32     public ClusterOperation createCluster(CreateClusterConfigurations
       createClusterConfigurations, String ownerAccount){
33         final String formattedClusterName = BaasCluster.buildFormattedClusterName(
       createClusterConfigurations.getClusterName(),ownerAccount);
34         createClusterConfigurations.setClusterName(formattedClusterName);
35         this.validate(this.createClusterValidator,createClusterConfigurations,ownerAccount);
36         ClusterProvider clusterProvider = this.getProviderByIdentifier(
       createClusterConfigurations.getClusterProviderIdentifier().toString());
37         try{
38             return clusterProvider.createCluster(createClusterConfigurations,ownerAccount);
39         }catch (Exception e){
40             throw new BaasClusterManagerException(e.getMessage(), HttpStatus.
       SERVICE_UNAVAILABLE);
41         }
42     }
43 }
```

Listing 4.3: ClusterProvidersService Specification Class

Listing 4.3 shows some methods of the class that provide a generic code to manage and provision Kubernetes clusters. Each cloud provider is defined in its own **ClusterProvider** instance, which contains the interfaces and classes and must be implemented and extended using their SDK. Each cloud provider environment must be configured with the required IAM credentials and roles, which will also be added to this microservice.

Dependencies and Requirements

- **All the shared dependencies mentioned at the beginning of this chapter.**
- **Google Cloud [14]:** A **Service Principle** account must be created with the required GKE roles, which can be imported into the application as **key.json** file. Furthermore, all the required cloud components must be correctly configured (creating VPC, subnets, and allowing public traffic through the firewall for cross-cluster communication) before using the created service principle. The **PROJECT IDENTIFIER** and **SERVICE ACCOUNT FILE NAME** environment variables must also be defined to locate and use the service account correctly.
- **Microsoft Azure [4]:** A new **App Registration** must be created. After that, the **SUBSCRIPTION ID**, **TENANT ID**, **CLIENT SECRET**, and **CLIENT ID** environment variables must be injected into the docker image of the application for correct usage of the App registrations. All the required Azure resources must be correctly created and configured: VNets, Subnets, and Network Security Groups.

Cluster Integration Microservice

As the main functionality of this service is to establish connectivity between multiple Kubernetes clusters and based on the ADD-004 which states that Istio Service Mesh [22] - IstioCTL will be used to configure and deploy the required cross-cluster connectivity resources into each Kubernetes [35] cluster, this section will give a deeper look into the implementation of the Istio Client.

```
1  package at.ac.univie.baas.clusterintegration.istio.process;
2
3  @Slf4j
4  public abstract class ClusterIntegrationServiceProcess {
5
6      protected static final String KUBECTL = "kubectl";
7      protected static final String ISTIOCTL = "istioctl";
8      private final Long timeoutInSeconds;
9
10     public abstract String run(String... args);
11
12     protected String abstractRunProcessAndReturnOutput(String... commands){
13         ProcessBuilder processBuilder = new ProcessBuilder(commands);
14         processBuilder.redirectErrorStream(true);
15         Process getOSUsernameProcess = null;
16         boolean hasTerminated = false;
17         String processOutput = "";
18         try {
19             getOSUsernameProcess = processBuilder.start();
20             hasTerminated = getOSUsernameProcess.waitFor(this.timeoutInSeconds, TimeUnit.
       SECONDS);
21             processOutput = this.readProcessOutput(getOSUsernameProcess.getInputStream());
22         } catch (Exception e) {
23             log.error(processOutput);
24             throw new RuntimeException(e);
25         }
26         if(!hasTerminated){
27             throw new RuntimeException("internal process execution error - timeout occurred");
28         }else if(!this.processTerminatedSuccessfully(getOSUsernameProcess.exitValue())){
29             throw new RuntimeException(String.format("internal process execution error -
       process terminated with the error code %s, %s",getOSUsernameProcess.exitValue(),
       processOutput));
30         }else{
31             return processOutput;
```

```
32          }
33      }
34  }
```

Listing 4.4: ClusterIntegrationServiceProcess Abstract Class

Since Istio doesn't offer any SDK and IstioCTL is a command-line client that offers the desired functionality, Listing 4.4 shows a generic process abstract class that multiple fine-grained Istio processes can extend, each process has its own specific code and functionality that will be executed. For example, Listing 4.5 shows the CreateIstioEastWestGatewayProcess class, which is responsible for creating a process that invokes the IstioCTL to create the east-west gateway that enables receiving cross-cluster requests. All the error and exception cases are handled in the service layer, which invokes the set of istio processes. All the required process dependencies are installed during the image build time of the microservice, which can be seen in the **Dockerfile** of this microservice.

```
1  package at.ac.univie.baas.clusterintegration.istio.process;
2
3  public class CreateIstioEastWestGatewayProcess extends ClusterIntegrationServiceProcess{
4
5      private static final String ISTIO_CTL_PATH_FORMAT = System.getenv("HOME").concat("/.
       istioctl/bin/istioctl");
6      private final String osUsername;
7      private final IstioFindAndReplaceService istioFindAndReplaceService;
8
9      @Override
10     public String run(String... args) {
11         try {
12             String patchedIstioGatewayFileName = this.istioFindAndReplaceService.
       findAndReplaceNetworkIdForEastWestGatewayManifest(networkIdentifier);
13             String[] COMMANDS = new String[]{
14                     ISTIO_CTL_PATH_FORMAT,
15                     "install",
16                     "-y",
17                     "-f",
18                     patchedIstioGatewayFileName
19             };
20             return this.abstractRunProcessAndReturnOutput(COMMANDS);
21         } catch (IOException e) {
22             throw new RuntimeException(e);
23         }
24     }
25 }
```

Listing 4.5: CreateIstioEastWestGatewayProcess Class

Kubernetes

As Istio requires creating Kubernetes resources (namespace and secrets) [35] before applying connectivity resources, the Kubernetes SDK is used the same as it is described in the **Blockchain Management Microservice Implementation Description**.

Dependencies and Requirements

- **All the shared dependencies mentioned at the beginning of this chapter.**
- **Command Line Clients:** google-cloud-sdk, Kubectl, and Istioctl must be installed during the docker image build time. these clients are required for correctly istioctl cross-cluster resources setup.
- **Root CA:** A root certificate authority must be configured and created. After that, these CA files must be added to the resource folder, and their name must

be configured in the properties.yaml file. This certificate authority is required for mTLS authentication between the cluster's resources communicating with each other.

- **Google Cloud Service Account:** As described in the cluster-management microservice implementation description, the same configurations must be applied for the correct usage of the Google Cloud Platform.

Blockchain Management Microservice

This microservice main functionality is managing and deploying blockchain frameworks on a single or multiple cloud providers Kubernetes clusters, which is discussed in the ADD-003 and in the blockchain-management microservice design chapter.

```
1  package at.ac.univie.baas.blockchainmanagementservice.service;
2
3  public class BlockchainResourcesManager {
4
5      private final BlockchainMetaDataFactory blockchainMetaDataFactory;
6      private final BlockchainDeploymentService blockchainDeploymentService;
7      private final BlockchainMetaDataService blockchainMetaDataService;
8      private final List<NodeDeploymentPostAction> nodeDeploymentPostActions;
9      private final BlockchainDeploymentPostAction blockchainDeploymentPostAction;
10     private final BaasClusterService baasClusterService;
11     private final DeployBlockchainNetworkValidator deployBlockchainNetworkValidator;
12
13
14     public BlockchainMetaData deployNewPrivateBlockchain(String ownerAccount,
       BlockchainDeploymentRequestDto blockchainDeploymentRequestDto){
15         BlockchainMetaData blockchainMetaData = this.blockchainMetaDataFactory.
       buildBlockchainMetaData(blockchainDeploymentRequestDto);
16         this.deployBlockchainNetworkValidator.validate(this.deployBlockchainNetworkValidator,
       blockchainMetaData,ownerAccount);
17         BlockchainMetaData persistedBlockchainMetaData = this.blockchainMetaDataService.save(
       blockchainMetaData);
18         List<BaasCluster> clusters = this.findClustersForScheduledDeployment(
       persistedBlockchainMetaData);
19         try{
20             BlockchainMetaData patchedBlockchainMetaData = this.blockchainDeploymentService.
       scheduleBlockchainDeployment(persistedBlockchainMetaData,clusters);
21             this.blockchainMetaDataService.save(patchedBlockchainMetaData);
22             patchedBlockchainMetaData.getBlockchainNodes().forEach(node -> this.
       nodeDeploymentPostActions.forEach(postAction -> postAction.apply(node)));
23             this.blockchainDeploymentPostAction.apply(patchedBlockchainMetaData);
24             return patchedBlockchainMetaData;
25         }catch (Exception e){
26             e.printStackTrace();
27             throw new RuntimeException(e);
28         }
29     }
30
31     private List<BaasCluster> findClustersForScheduledDeployment(BlockchainMetaData
       persistedBlockchainMetaData) {
32         if(persistedBlockchainMetaData.getDeploymentModel() == DeploymentModel.SINGLE_CLOUD){
33             return List.of(this.baasClusterService.findClusterByIdentifier(
       persistedBlockchainMetaData.getDeploymentEntityIdentifier()));
34         }else{
35             return this.baasClusterService.findClustersIntegrationByIdentifier(
       persistedBlockchainMetaData.getDeploymentEntityIdentifier())
36                     .orElseThrow(() -> new RuntimeException("ClustersIntegration Not found"))
37                     .getIntegratedCluster();
38         }
39     }
40
41     public Optional<BlockchainMetaData> getBlockchainMetaDataById(String blockchainId) {
42         return this.blockchainMetaDataService.findById(blockchainId);
43     }
44
45 }
```

Listing 4.6: BlockchainResourcesManager Class

The class BlockchainResourcesManager shown in the Listing 4.6 acts as the entry point for the offered functionality by this microservice. The listing shows the methods **deployNewPrivateBlockchain** and **getBlockchainMetaDataById** responsible for creating a new blockchain instance, including its initial nodes using the **Custom Blockchain Framework** that is described in the design chapter and querying an existing blockchain by its ID.

```
1  package at.ac.univie.baas.blockchainmanagementservice.k8sapi.services;
2
3
4  public interface BlockchainDeploymentService {
5
6      BlockchainMetaData scheduleBlockchainDeployment(BlockchainMetaData blockchainMetaData, List
       <BaasCluster> baasClusters);
7      void deployNewNodeIntoBlockchain(String namespace, BlockchainNodeMetaData metaData,
       BaasCluster baasCluster, String nodeIpServerBaseUrl);
8      Optional<String> pollNodeServiceLoadBalancerIp(BlockchainNodeMetaData metaData, BaasCluster
         baasCluster);
9  }
```

Listing 4.7: BlockchainDeploymentService Interface

Listing 4.7 shows the interface that defines the specification for a blockchain instance provisioning, blockchain node provisioning, and pulling nodes status (load balancer IP). This interface is implemented with the help of Kubernetes SDK, and it is utilized to build fine-grained service classes that generate the creation and management process of Kubernetes resources.

```
1  package at.ac.univie.baas.blockchainmanagementservice.k8sapi.services;
2
3  public class KubernetesBlockchainDeploymentService implements BlockchainDeploymentService{
4
5      private final NamespaceCreationService namespaceCreationService;
6      private final NodeResourcesService nodeResourcesService;
7      private final NodesIpServerDeploymentService nodesIpServerDeploymentService;
8      private final KubeConfigService kubeConfigService;
9      private final KubernetesApiConnectionFactory kubernetesApiConnectionFactory;
10
11
12     @Override
13     public void deployNewNodeIntoBlockchain(String namespace, BlockchainNodeMetaData
       blockchainNodeMetaData,BaasCluster baasCluster, String nodeIpServerBaseUrl) {
14         try {
15             String kubeConfigFileContent = this.kubeConfigService.
       generateKubeConfigFileContentFor(baasCluster.getProvider(),baasCluster.getClusterAuthData
       ());
16             CoreV1Api coreV1Api = this.kubernetesApiConnectionFactory.getCoreVApiInstance(
       kubeConfigFileContent);
17             AppsV1Api appsV1Api = this.kubernetesApiConnectionFactory.getAppsV1ApiInstance(
       kubeConfigFileContent);
18             this.nodeResourcesService.createNodeResources(blockchainNodeMetaData,namespace,
       coreV1Api,appsV1Api,nodeIpServerBaseUrl);
19         }catch (ApiException e){
20             e.printStackTrace();
21             System.out.println(e.getResponseBody());
22         }
23     }
24
25     @Override
26     public Optional<String> pollNodeServiceLoadBalancerIp(BlockchainNodeMetaData
       blockchainNodeMetaData,BaasCluster baasCluster) {
27         try {
28             String kubeConfigFileContent = this.kubeConfigService.
       generateKubeConfigFileContentFor(baasCluster.getProvider(),baasCluster.getClusterAuthData
       ());
29             CoreV1Api coreV1Api = this.kubernetesApiConnectionFactory.getCoreVApiInstance(
       kubeConfigFileContent);
30             return this.nodeResourcesService.pollNodeServiceLoadBalancerIp(
       blockchainNodeMetaData,coreV1Api);
31         } catch (ApiException e) {
32             e.printStackTrace();
33         }
34         return Optional.empty();
35     }
```

```
36
37      private Map<String, String> createNodesDeploymentPlan(BlockchainMetaData blockchainMetaData
        , List<BaasCluster> clusters) {
38          List<BlockchainNodeMetaData> blockchainNodeMetaDataList = blockchainMetaData.
        getBlockchainNodes().stream().toList();
39          return IntStream.range(0,blockchainMetaData.getBlockchainNodes().size())
40                  .boxed()
41                  .map(index -> new AbstractMap.SimpleEntry<>(blockchainNodeMetaDataList.get(
        index).getNodeId(),clusters.get(index % clusters.size()).getId()))
42                  .collect(Collectors.toMap(AbstractMap.SimpleEntry::getKey, AbstractMap.
        SimpleEntry::getValue));
43      }
44
45
46  }
```

Listing 4.8: KubernetesBlockchainDeploymentService Class

The class KubernetesBlockchainDeploymentService shown in the Listing 4.8 implements the interface on Listing 4.7, it define its methods using multiple generic templated services that are shown in the listing: **NamespaceCreationService**, **NodeResourcesService**, **NodesIpServerDeploymentService**, **KubeConfigService** and **KubernetesApiConnectionFactory**.
The method **createNodesDeploymentPlan** contains the implementation of the **Round-Robin Deployment Strategy**.

Dependencies and Requirements

- **All the shared dependencies mentioned at the beginning of this chapter.**
- **Google Cloud Service Account [14]:** As described in the Cluster Management microservice Implementation Description, the same configurations must be applied for the correct usage of the Google Cloud Platform. The reason for not requiring an app registration account for Azure is that Google Cloud Kubernetes clusters require **gcloud command-line** authentication to be able to use each cluster **Kubectl** file; meanwhile, Azure doesn't have any tools requirements for Kubectl authentication.

Notification Microservice

This service manages user notification preferences and notifies users when a specific event occurs in the Platform.

```
1  package at.ac.univie.baas.notificationservice.service;
2
3  public class NotificationServiceManager {
4
5
6      private final ClusterUpdateEmailMessageBuilder clusterUpdateEmailMessageBuilder;
7      private final ClusterIntegrationEmailMessageBuilder clusterIntegrationEmailMessageBuilder;
8      private final BlockchainCreateEmailMessageBuilder blockchainCreateEmailMessageBuilder;
9      private final BlockchainNodeCreateEmailMessageBuilder
       blockchainNodeCreateEmailMessageBuilder;
10     private final EmailService emailService;
11     private final UserService userService;
12
13     public void onClusterCreation(BaasCluster baasCluster){
14         User user = this.userService.findByUsername(baasCluster.getOwnerAccount());
15         if(!user.isEnableNotificationsOnClusterEvents()){
16             return;
17         }
18         Email email = this.clusterUpdateEmailMessageBuilder.buildEmail(baasCluster, Constants.
       EMAIL_EVENT_NAME_CLUSTER_CREATE);
```

```
19          this.emailService.sendSimpleMail(user.getEmail(),email.getTitle(),email.getBody());
20      }
21
22      public void onClusterIntegration(String clustersIntegrationIdentifier, String ownerAccount,
          String integrationMessage) {
23          User user = this.userService.findByUsername(ownerAccount);
24          if(!user.isEnableNotificationsOnIntegrationEvents()){
25              return;
26          }
27          Email email = this.clusterIntegrationEmailMessageBuilder.buildEmail(
        clustersIntegrationIdentifier,ownerAccount,integrationMessage);
28          this.emailService.sendSimpleMail(user.getEmail(),email.getTitle(),email.getBody());
29
30      }
31
32      public void onCreateBlockchainMetaData(BlockchainMetaData blockchainMetaData) {
33          User user = this.userService.findByUsername(blockchainMetaData.getOwnerAccount());
34          if(!user.isEnableNotificationsOnWorkloadsEvents()){
35              return;
36          }
37          Email email = this.blockchainCreateEmailMessageBuilder.buildEmail(blockchainMetaData);
38          this.emailService.sendSimpleMail(user.getEmail(),email.getTitle(),email.getBody());
39      }
40  }
```

Listing 4.9: KubernetesBlockchainDeploymentService Class

Listing 4.9 shows the class NotificationServiceManager, which acts as the entry point for the notification functionality for all the Apache Kafka events consumers defined in this class. It will send an email notification to the user whenever a new event is consumed, and the user preferences include notifications for that event. This functionality is achieved using the custom-defined email builder classes and the Java Mail API working behind the used **spring boot mail** library.

Keycloak SDK

```
1  package at.ac.univie.baas.notificationservice.service;
2
3  public class UserDataOAuthServiceImpl implements UserDataOAuthService{
4
5      private final Keycloak keycloak;
6      private final String realm;
7
8      @Override
9      public Optional<UserOAuthDto> getUserDataByUsername(String username) {
10         return this.keycloak.realm(this.realm)
11                 .users()
12                 .searchByUsername(username, true)
13                 .stream()
14                 .map(userRepresentation -> UserOAuthDto.builder()
15                         .id(userRepresentation.getId())
16                         .email(userRepresentation.getEmail())
17                         .username(userRepresentation.getUsername())
18                         .build())
19                 .findAny();
20     }
21 }
```

Listing 4.10: KubernetesBlockchainDeploymentService Class

This microservices synchronizes the created user data from the **Identity and Access Management** component using the interface **UserDataOAuthService** and its implementation**UserDataOAuthServiceImpl** shows in Listing 4.10, which returns the required user data (email, username, and identifier) for notification management.

Dependencies and Requirements

- **All the shared dependencies mentioned at the beginning of this chapter.**
- **Command Line Clients:** google-cloud-sdk, Kubectl, and IstioCTL must be installed during the docker image build time. these clients are required for correctly IstioCTL cross-cluster resources set up.

Monitoring Microservice

The service is responsible for periodically checking the Kubernetes clusters' health status and producing an event for other microservices about the latest status of a specific cluster. Depending on their responsibilities and boundaries, other microservices can take specific actions on a failing Kubernetes cluster.

```
1  package at.ac.univie.baas.clustermonitoring.service;
2
3  @Service
4  @Slf4j
5  public class ScheduledMonitoringService {
6
7
8      @Scheduled(fixedRateString = "${monitoring.rate}", timeUnit = TimeUnit.MINUTES)
9      public void scheduleFixedDelayMonitoring() {
10         System.out.println("ScheduleFixedDelayMonitoring started...");
11         var baasClustersIdAuthDataMap = this.baasClusterService.findAllClusters().stream()
12                 .map(cluster -> new AbstractMap.SimpleEntry<>(cluster, this.kubeConfigService.
       generateKubeConfigFileContentFor(cluster.getProvider(), cluster.getClusterAuthData())))
13                 .collect(Collectors.toMap(AbstractMap.SimpleEntry::getKey, AbstractMap.
       SimpleEntry::getValue));
14
15         for (var entry : baasClustersIdAuthDataMap.entrySet()) {
16             this.monitorCluster(entry);
17         }
18
19     }
20 }
```

Listing 4.11: ScheduledMonitoringService Class

Listing 4.11 shows the approach taken by this service, which schedules a monitoring request that invokes a set of classes and methods that send health check requests to each cluster and produce Kafka events regarding the cluster status.

4.3. Running The Prototype

4.3.1. Cloud Providers Prerequisite

All the supported cloud providers must be correctly configured, which are **Google Cloud** and **Microsoft Azure**.

Google Cloud

1. Create a Google Cloud account, either activate billing or create the account with the limited 3-month academic license.
2. Use the existing default project with its already created VPC, subnets, and other networking components.

3. Activate **Kubernetes Engine API** in the default project.
4. In the **IAM And ADMIN** page, create a service account and key credentials for it. then add to the service account the following Roles:
   a) Compute Admin
   b) Kubernetes Engine Admin
   c) Service Account User
5. Enable external traffic in the VPC firewall by creating a new high-priority firewall rule that allows public IP Addresses to reach the resources inside the VPC, The firewall rule must have the following properties
   a) Type: Ingress
   b) Targets: Apply to all
   c) Action: Allow
   d) Priority: 1000

Required output from this step: Google cloud project identifier and the service account key.json file.

Microsoft Azure

1. Create an Azure account, either activate billing or create the account with the limited 100-dollar academic license.
2. Create a subscription (any name) and a resource group named **test**.
3. Create an App registration account with a secret for enabling the BaaS platform to access the Azure platform using the created account.

Required output from this step: Subscription ID, Tenant ID, Client ID, and Client Secret.

4.3.2. Local Setup Prerequisites

To be able to run the platform locally, you need to fulfill the following prerequisites:

1. Install: Docker version 20.10.17 and Docker Compose version v2.6.1 or above.
2. Internet Connection.
3. Java 17

To start the BaaS platform, navigate to the **implementation** where the **docker-compose** is located. The docker-compose file contains the following components and microservices:

1. Event Streaming Platform - Apache Kafka
2. Data Store - NoSQL MongoDB
3. Identity and Access Management - Keycloak
4. API Gateway Component
5. Cluster Management Microservice
6. Cluster Integration Microservice
7. Blockchain Management Microservice
8. Monitoring Microservice
9. Notification Microservice
10. Frontend Server

Before starting the docker-compose file, update the defined environment variables with the output of the **Cloud Providers Prerequisite** step.

Azure environment variables:

1. AZURE-SUBSCRIPTION-ID
2. AZURE-TENANT-ID
3. AZURE-CLIENT-SECRET
4. AZURE-CLIENT-ID

Google Cloud environment variables:

1. GOOGLE-PROJECT-IDENTIFIER
2. GOOGLE-KEY-FILE-PATH: key.json
3. GOOGLE-SERVICE-ACCOUNT

NOTE: The **key.json** file must be copied and placed in the following places:

- implementation/blockchainmanagementservice directory
- implementation/clusterintegrationservice directory
- implementation/clustermonitoring directory
- implementation/clustermanager/src/main/resources directory

After that, run the command **docker compose up**, which will take 4-7 minutes to create the images, and run the containers and start up the Blockchain-as-a-Service platform.

Once all the components and microservices are running, Keycloak must be first configured before being able to be used as an Identity and Access Management component for the platform. The following steps show in detail how to configure Keycloak:

1. Navigate to the Keycloak dashboard on **localhost:8086**.
2. Login using the default credentials: Username: **admin** and password: **admin**.
3. Create a new realm with the name: **baas**
4. Navigate to realm settings, then select the login configuration tab and enable user registration.
5. Navigate to Clients, then create a new client with the ID: **baas**, set * for Valid Redirect URLs, Valid post logout redirect URIs, and Web Origins.

Finally, the BaaS platform is accessible on **http://localhost:8080**.

5. Evaluation and Discussion

The platform is evaluated based on a user experiment following the System Usability Scale - SUS [6], a ten-item scale that gives a subjective usability assessment. The users were given a user manual and asked to perform a set of tasks on the platform that covered most of the platform functionality. After performing these tasks, the users were asked to answer the System Usability Scale ten-item survey, which they ranked from Strongly Disagree (1) and Strongly Agree (5) [6]. Furthermore, the users were asked to answer three more questions that focused on the functionality perspective of the platform.

After that, the results of the SUS and custom-defined items were collected and calculated by applying calculations on each item and then summing all of the 10 items' calculations results. Items 1,3,5,7 and 9 are calculated by subtracting 1 from the scale position, Items 2,4,6,8 and 10 are calculated by subtracting the scale position from 5, and then the SUS overall value is obtained by multiplying the sum of the scores with 2.5 [6]. Furthermore, the results of the custom-defined questions were separately calculated by averaging the user's answers for all of the questions.
Regarding the user's background, they all have a good knowledge of information technology, and 3 have already participated in blockchain-related projects.

The platform functionality and usability were improved based on the conducted user experiments and feedback.

Besides that, a Non-Functional Requirements evaluation was done based on different scenarios and use cases, focusing on the **Decentralization**, **High Availability**, **preventing Vendor Lock-in**, **Multi-Cloud** and **Performance**.

5.1. User Experiment

5.1.1. System Usability Scale - Tasks

The users were asked to perform these tasks in the order they were defined with the support of the user manual that helps them in each task. the tasks that the users we asked to do cover most of the BaaS platform functionality:

1. Register into the platform.
2. Sign in.
3. Create two Kubernetes clusters on two different cloud providers.
4. Integrate the successfully created Kubernetes clusters.

5. View the current Kubernetes clusters and clusters integration.

6. Switch to the Custom Blockchain Framework and deploy a new blockchain instance.

7. Activate your wallet account in the created blockchain instance.

8. Create a transaction.

9. View transactions and blocks state.

5.1.2. System Usability Scale - Survey

As discussed, this survey is composed of the defined standard System Usability Scale items and three additional functional-related items [6]:

1. I think that I would like to use this system frequently.

2. I found the system unnecessarily complex.

3. I thought the system was easy to use.

4. I think that I would need the support of a technical person to be able to use this system.

5. I found the various functions in this system were well integrated.

6. I thought there was too much inconsistency in this system.

7. I would imagine that most people would learn to use this system very quickly.

8. I found the system very cumbersome to use.

9. I felt very confident using the system.

10. I needed to learn many things before I could get going with this system.

11. No technical issues or problems occurred when doing each task.

12. The platform behavior matches the expected results in the user manual.

13. The platform maintains a consistent user interface and behavior throughout the different tasks.

5.1.3. System Usability Scale - Results

Table 5.1.: System Usability Scale (SUS) Results

User	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
U1	5	1	5	1	5	2	5	1	5	2
U2	5	1	4	2	5	1	5	1	4	1
U3	4	1	5	1	5	1	5	1	5	1
U4	4	2	4	1	5	1	4	1	4	2
U5	5	1	4	2	5	1	5	2	4	2
U6	5	1	5	1	4	2	5	2	4	3
U7	4	2	5	2	5	1	4	1	5	2

Table 5.1 shows the plain results of the users (rows) evaluating each item/question (column) on a scale from 1 to 5 where 1 stands for Strongly disagree and 5 stands for Strongly agree.

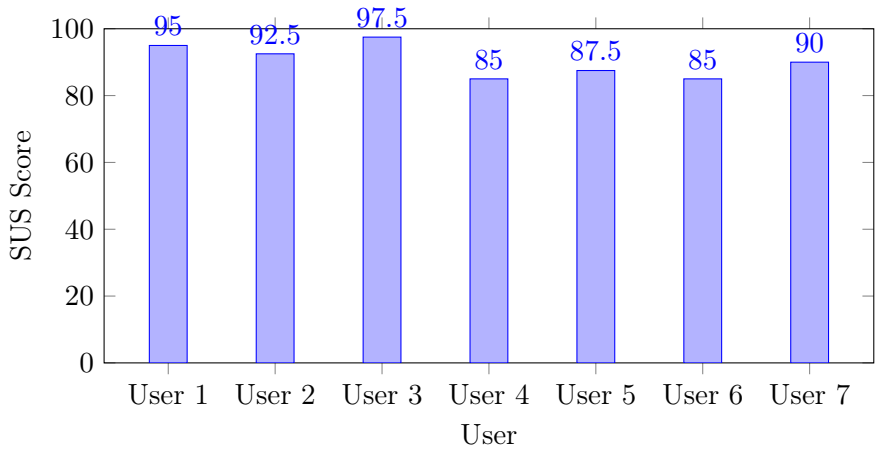

Figure 5.1.: System Usability Scale (SUS) Scores by User

Figure 5.1 demonstrates the SUS score for each user by aggregating the results of each item and applying the standard calculations defined by the System Usability Scale standards. This gives an overall SUS average of **90.3**, which is clearly above the defined SUS average, which is **68** [6].

5.1.4. Funtional-Related Items Results

Table 5.2.: Functional Items Results

User	Q11	Q12	Q13
U1	5	5	5
U2	5	5	5
U3	5	5	5
U4	5	5	3
U5	5	5	4
U6	5	5	5
U7	5	5	4

Table 5.2 shows the results of the functional-related items in the survey. Overall, most of the users confirmed the correct and seamless functionality of the platform.

5.1.5. Overall Results

The SUS results exceeded the SUS average of **68**, and the BaaS platform fulfilled all the functional requirements for the results of the custom-defined question.

The users also provided feedback regarding the prototype functionality and usability:

1. Add expected waiting time for requests requiring a long time, like provisioning a Kubernetes cluster.
2. Support asynchronous processing for cluster integration requests.
3. Explicitly provide successful processing messages when interacting with a blockchain node.

The received feedback was discussed with the users and applied to the prototype.

5.2. Non-Functional Requirements Evaluation

Improving the decentralization and high availability of the BaaS platform blockchain instances using the multi-cloud approach introduces different challenges and disadvantages that the platform must take care of:

1. **Complexity:** It is a complex process to manage the different clusters and their connectivity compared to managing a single Kubernetes cluster and deploying the blockchain instances on it.
2. **Resilience and Fault Tolerance:** The platform must act on a failing Kubernetes cluster by migrating the workloads deployed on it into other connected Kubernetes clusters.

3. **Observability:** The platform must enable detailed observability for each Kubernetes cluster and each workload/blockchain node deployed on it.

5.2.1. Decentralization

A primary non-functional requirement of the BaaS platform is the decentralization and high availability of the deployed blockchain instances. Having a blockchain instance with **N** blockchain nodes deployed across **M** Kubernetes clusters in a **Round-Robin Deployment Strategy** means:
Each cluster has $\frac{N}{M}$ blockchain nodes deployed on it, which means the blockchain nodes are decentralized by $\frac{1}{M}$ across each cluster.

This demonstrates achieving a BaaS platform that hosts blockchain instances while prioritizing and improving decentralization characteristics by not hosting all blockchain nodes on a single cloud provider's Kubernetes cluster.

5.2.2. High Availability

Since the BaaS platform uses **Google Cloud** and **Microsoft Azure** as cloud providers:

- The Google Kubernetes Engine Service Level Agreement for Zonal Cluster has a Monthly Uptime Percentage of **99.5 Percent**, which means a yearly downtime of **43.8 Hours**.
- The Azure Kubernetes Service Level Agreement for AKS Clusters that do not use Azure Availability Zone. has a Monthly Uptime Percentage of **99.95 Percent**, which means a yearly downtime of **4.38 Hours**.

The results below can be obtained when using at least **3** connected Kubernetes clusters (2 on Google Cloud and 1 on Microsoft Azure) with a blockchain instance with **6** nodes.

Two Google Cloud Kubernetes Cluster Availability

The probability of **2** Kubernetes clusters (2 Google Cloud Clusters) going down (4 blockchain nodes go down out of **6**) is calculated as:

$$\text{Probability of failing both events} = Q_1 \times Q_2$$

where

$$Q_1 = 1 - P_1 = 1 - 0.995 = 0.005$$

and

$$Q_2 = 1 - P_2 = 1 - 0.995 = 0.005$$

Therefore,

$$\text{Probability of both Kubernetes clusters failing} = 0.005 \times 0.005 = 0.000025$$

which means the **availability** of both clusters increases (the nodes deployed on a specific cluster) to $100 - 0.0025 =$**99.9975 Percent.**

One Azure Kubernetes Cluster and Two Google Cloud Kubernetes Cluster Availability

This availability can be calculated in the same demonstrated way in the previous section, which gives us the following results:

The probability of all 3 clusters failing is reduced to **0.00000125 Percent**, which means the global availability of the 3 Kubernetes cluster that hosts the blockchain instance increases to: **99.99999875 Percent (which is approximately less than 3 minutes of downtime in a year) .**

Increasing the availability of a blockchain instance deployed (the nodes deployed on a specific cluster) on 3 different Kubernetes clusters to 99.99999875 percent proves the high availability characteristic that the BaaS platform offers in comparison to a single cloud, single cluster blockchain instance deployments.

5.2.3. Multi-Cloud And Vendor Lock-In

Since the BaaS platform allows its users to freely select the suitable cloud provider and the region/zone where they are willing to deploy their blockchain instances and resources, this explicitly means that the platform prevents vendor lock-in and provides the required level of flexibility for the user to host their resources on the cloud provider they want.

5.2.4. Performance

The platform performance was experimented with and tested by applying load tests on various endpoints and services to verify that the platform can support the anticipated number of concurrent users.

Test Environment

- Test Tool: Postman - Load Tester/Collection Runner and cURL
- Environment: Development/Local Deployment
- Network Conditions: Stable
- Concurrent requests count: 100

Client Request With Database Operation

Performance Metrics

- Total Requests: 100
- Total Errors: 0
- Average Response Time: 20 ms
- Minimum Response Time: 12 ms
- Maximum Response Time: 47 ms

Client Request With Cloud Provider SDK Read Request

Performance Metrics

- Total Requests: 100
- Total Errors: 0
- Average Response Time: 391 ms
- Minimum Response Time: 300 ms
- Maximum Response Time: 536 ms

Client Request With Cloud Provider SDK Create Request

Performance Metrics

- Total Requests: 10
- Total Errors: 0
- Average Response Time: 423 ms
- Minimum Response Time: 344 ms
- Maximum Response Time: 556 ms

Multi-Cloud Connectivity

Performance Metrics

- Total Requests: 100
- Total Errors: 0
- Average Response Time: 85 ms
- Minimum Response Time: 82 ms
- Maximum Response Time: 91 ms

As the load test result shows, the BaaS platform successfully processed **100** concurrent requests for different scenarios and use cases, verifying the ability to handle the anticipated expected user number of the prototype.

6. Future Work and Conclusions

6.1. Future Work

Achieving the desired multi-cloud BaaS platform design and prototype and fulfilling all the functional and non-functional requirements was challenging and full of complex milestones. However, there are still many milestones to bring this prototype into a production-ready Blockchain-as-a-Service platform:

1. **Cloud Providers:** Currently, the platform is limited to Google Cloud and Microsoft Azure cloud providers. integrating additional cloud providers improves the platform overall **decentralization** and **vendor lock-in**.

2. **Blockchain Framework:** The platform is currently only offering **Custom Blockchain Framework**. However, introducing additional blockchain frameworks like **Ethereum** and **Hyperledger Fabric** and providing the required functionality from the deployment of the resources to providing developer tools like smart contracts IDEs would make this platform a real alternative against the current commercial BaaS platforms.

3. **Clusters Connectivity/Integration Process Optimizations:** Despite achieving the functional and non-functional requirements regarding clusters integration and connectivity, the process can be further improved and optimized in future research by:

   a) Writing a custom SDK for the used Istio Service Mesh to achieve cross-cluster connectivity rather than depending on the **IstioCTL**.

   b) Optimize the provisioning of cluster connectivity resources, which is doing very well when tested. However, the performance is reduced when tested against many requests.

4. **On-Premise:** To further increase the platform's flexibility and vendor lock-in properties, it would be essential in future work and research to consider allowing users to import their own on-premise Kubernetes Clusters.

5. **Resiliency And Automatic Failover:** In the current architecture, the blockchain nodes can be distributed across multiple clusters, and in case one cluster fails, the nodes would be online again once the cluster has restarted. A new component or microservice can be introduced that automatically migrates all the deployed resources on a Kubernetes cluster to another active cluster.

6. **Commerciality:** To use this platform as a commercial product, additional microservices and components are required for billing and invoicing management, cost tracking, and payment services.

7. **Identity and Access Management:** The platform depends on the open source identity and access management component Keycloak, which currently only offers username and password-based registration and login. It would bring a huge benefit and attract a big group of users to the platform when configuring Keycloak to support external identity providers like **Azure Active Directory** or **Google**

8. **User Interface:** Since the current UI is only sufficient for successfully completing the functional requirements, it is essential to create a new UI component designed and implemented to meet the expected professional level of UI/UX.

6.2. Conclusion

In conclusion, this thesis provides a comprehensive overview of Blockchain-as-a-Service and the current leading platforms that provide BaaS technology. It compares them based on factors and characteristics like blockchain types and frameworks, use cases, architecture, pricing, vendor lock-in, and multi-cloud/on-premise deployment ability. The comparison shows that many top BaaS providers like Google and AWS don't offer multi-cloud or on-premise blockchain framework deployments, negatively affecting blockchain's decentralization, high availability, and resiliency characteristics.

The thesis also provides a comprehensive overview and compares the recent BaaS academic systems and their architecture. The comparison shows that these systems work on different areas that contribute to the adoption and utilization of BaaS systems, which are simplifying the blockchain network deployment and management process, maintaining blockchain characteristics, supporting and guiding the blockchain developers by providing design patterns and development tools and enabling agnostic deployment whether on-premises or multi-cloud deployment. The comparison shows that NBaaS, uBaaS, and NutBaaS consider the multi-cloud characteristic by enabling the deployment of blockchain nodes On-Premises, on a single cloud, or in a multi-cloud environment. However, these academic system lacks a concrete specification of a multi-cloud blockchain architecture and only discuss it on a very high-level approach.

Therefore, this thesis presents a specification and prototype implementation of a Blockchain-as-a-Service platform, emphasizing the facilitation of multi-cloud blockchain framework deployments through the utilization of entirely open-source technology and microservices architecture. The specification, underlying concepts, and architectural design decisions have been justified and elaborated on, and the prototype's functional and non-functional requirements and usability have been evaluated based on user experiments, formal mathematical proof, and performance tests, which demonstrate significant enhancements in the core attributes of this platform, specifically decentralization, high availability, and non-vender lock-in.

Finally, this thesis discusses the future work and potential areas for further exploration toward realizing a production-ready, open-source, multi-cloud, and multi-region BaaS platform that hopefully impacts future research and contributes to the landscape of blockchain technology.

Bibliography

[1] Elli Androulaki, Artem Barger, and Vita Bortnikov. Hyperledger fabric: a distributed operating system for permissioned blockchains. 2018.

[2] Amazon Web Services (AWS). Amazon managed blockchain. 2020.

[3] Amazon Web Services (AWS). Event-driven architecture. 2023.

[4] Microsoft Azure. Azure kubernetes service (aks). 2023.

[5] Muhammad Nasir Mumtaz Bhutta, Amir A. Khwaja, Adnan Nadeem, and Hafiz Farooq Ah. A survey on blockchain technology: Evolution, architecture and security. 2021.

[6] J. B. Brooke. Sus: A 'quick and dirty' usability scale. 1996.

[7] Huan Chen and Liang-Jie Zhang. Fbaas: Functional blockchain as a service. 2018.

[8] Jingxiao; X. Sean Chen, Yaoliang; Gu. A full-spectrum blockchain-as-a-service for business collaboration. 2019.

[9] Lin Chen, Lei Xu, and Weidong Shi. On security analysis ofproof-of-elapsed-time (poet). 2017.

[10] Cilium. Cilium. 2023.

[11] Alibaba Cloud. Alibaba cloud blockchain-as-a-service. 2020.

[12] Google Cloud. Google blockchain node engine. 10 2022.

[13] Google Cloud. Cloud sdk. 2023.

[14] Google Cloud. Google kubernetes engine (gke). 2023.

[15] Kubernetes community. kubespray. 2023.

[16] Ralf Küsters Daniel Fett and Guido Schmitz. The web sso standard openid connect: In-depth formal security analysis and security guidelines. 2017.

[17] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. 2015.

[18] Derar Eleyan, Samer Muneer Alshurafa, and Amna Eleyan. A survey paper on blockchain as a service platforms. 10 2021.

[19] Eric Evans. Domain-driven design: Tackling complexity in the heart of software. 2023.

[20] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures. 2000.

[21] Linux Foundation and Hyperledger Sawtooth Community. Hyperledger sawtooth. 2018.

[22] IBM Google and Lyft. Istio. 2023.

[23] HashiCorp. Terraform. 2023.

[24] Red Hat. Openshift. 2020.

[25] Helm. Kubernetes - helm. 2023.

[26] Sinem Guney Hulya Vural, Murat Koyuncu. A systematic literature review on microservices.

[27] IBM. Ibm blockchain platform. 2020.

[28] IBM. Ibm - blockchain benifits. 2021.

[29] IBM. Ibm blockchain platform - technical overview. 04 2022.

[30] Alsunaidi Shikah J, Alhaidari, and Fahd A. A survey of consensus algorithms for blockchain technology. 2019.

[31] Markus Jakobsson, Tom Leighton, Silvio Micali, and Michael Szydlo. Fractal merkle tree representation and traversal. 2003.

[32] ardalis JeremyLikness and erjain. Serverless apps: Architecture, patterns, and azure implementation. 2023.

[33] Kaleido. Kaleido blockchain-as-a-service. 2021.

[34] Kostis Karantias, Aggelos Kiayias, and Dionysis Zindros. Proof-of-burn. 2019.

[35] Kubernetes. Kubernetes. 2023.

[36] Apache Software Foundation LinkedIn. Apache kafka. 2023.

[37] J. Bradley M. Jones and N. Sakimura. Json web token (jwt). 2015.

[38] Mpyana Mwamba Merlec Md. Mainul Islam and Hoh Peter. A comparative analysis of proof-of-authority consensus algorithms: Aura vs clique. 2022.

[39] Microsoft. The oauth 2.0 authorization framework. 2012.

[40] Microsoft. Design patterns for microservices. 2023.

[41] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[42] Tyron Ncube, Nomusa Dlodlo, and Alfredo Terzoli. Private blockchain networks: A solution for data privacy. 2019.

[43] Oracle Cloud Infrastructure (OCI). Oracle blockchain platform service. 2021.

[44] Diego Ongaro and John Ousterhout. Raft. 2021.

[45] Md Mehedi Hassan Onik and Mahdi H. Miraz. Performance analytical comparison of blockchain-as-aservice (baas) platforms. 2020.

[46] Weishan Zhang Qinghua Lu, Xiwei Xu Liming Zhu. ubaas: A unified blockchain-as-a-service platform.

[47] Ranchor. Enterprise kubernetes management - ranchor. 2023.

[48] Jie Song, Mohammed Alkubati, Yubin Bao, and Ge Yu. Research advances on blockchain-as-a-service: architectures, applications and challenges. 08 2022.

[49] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. 2017.

[50] Min Xu, Xingtong Chen, and Gang Kou. A systematic review of blockchain. 2019.

[51] Wei Yao, Junyi Ye, and Renita Murimi Guiling Wang. A survey on consortium blockchain consensus. 2021.

[52] Weilin Zheng, Zibin Zheng, Xiangping Chen, Kemian Dai, and Peishan Li. Nutbaas: A blockchain-as-a-service platform. 2019.

[53] Zibin Zheng, Hong-Ning Dai, and Shaoan Xie. An overview of blockchain technology: Architecture, consensus, and future trends. 06 2017.

[54] Jinqing Yang Xianghua Lin Zhitao Wan, Minqiang Cai. A novel blockchain as a service paradigm. 2018.

[55] Hong-Ning Dai Zibin Zheng, Shaoan Xie. An overview of blockchain technology: Architecture, consensus, and future trends. 2017.

A. Appendix

A.1. Source Code

The git repository on `https://git01lab.cs.univie.ac.at/thesis/2023ss/01529155_ismail_alhamzeh` contains all the source code, documentation, and figures created for this thesis.

A.2. Figures

This section contains a detailed version of the UML class diagram figures that describe each microservice designed in this thesis.

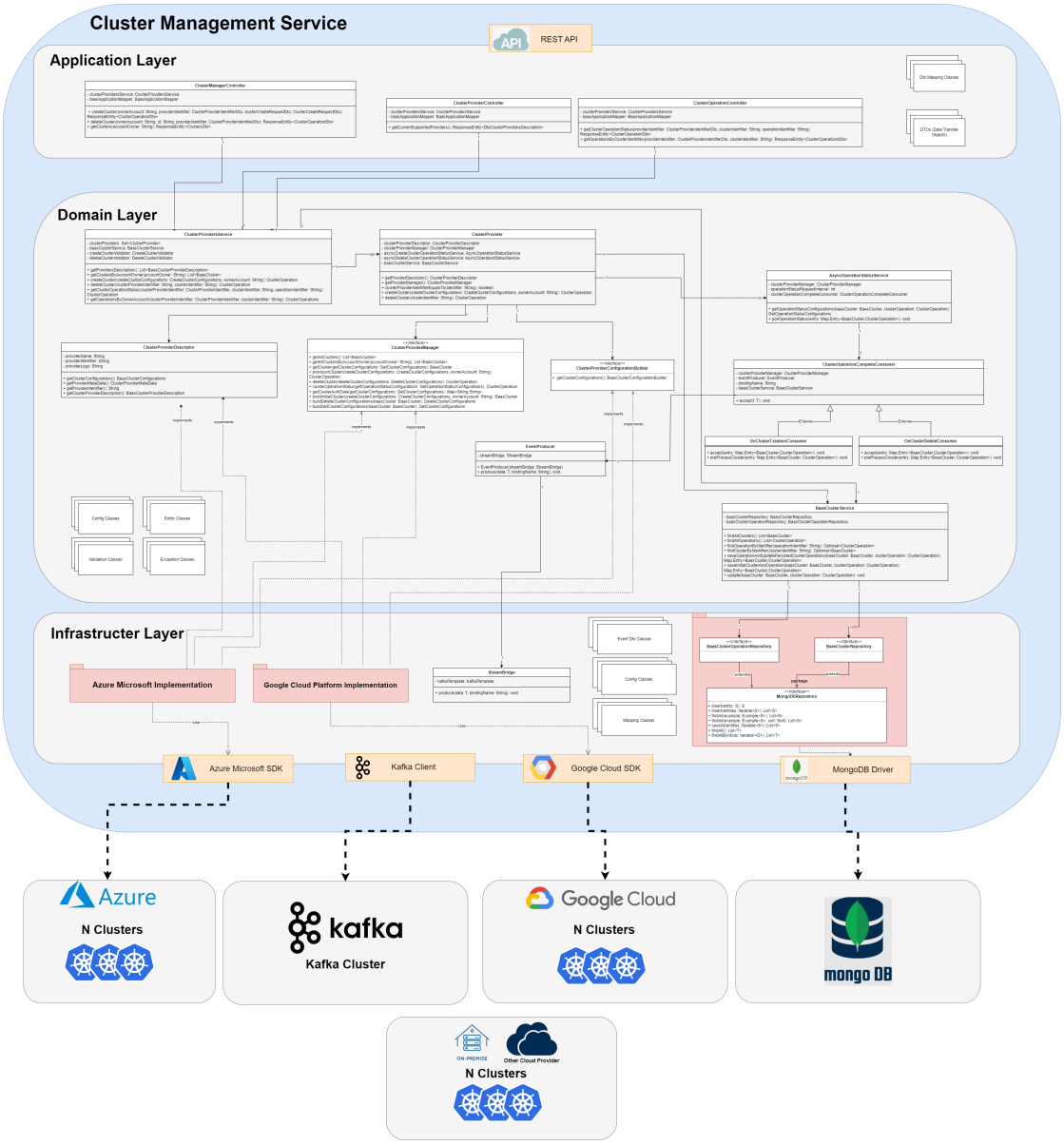

Figure A.1.: Cluster Management Microservice - UML Class Diagram.

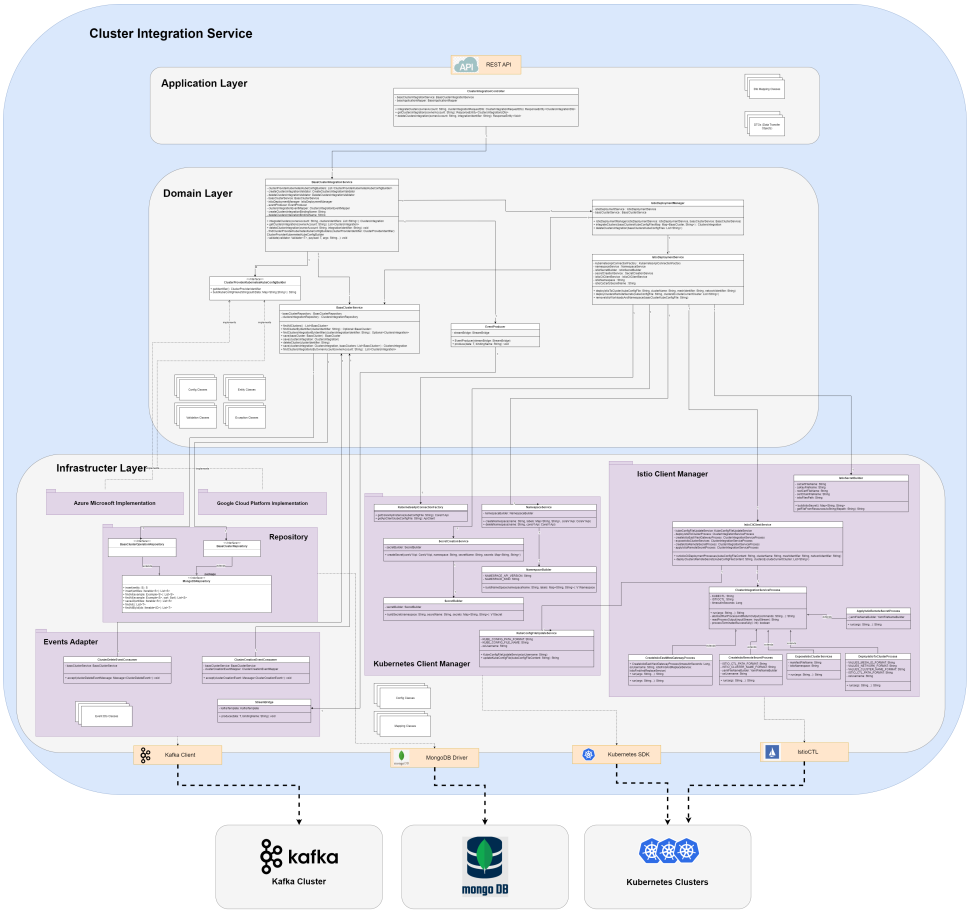

Figure A.2.: Cluster Integration Microservice - UML Class Diagram.

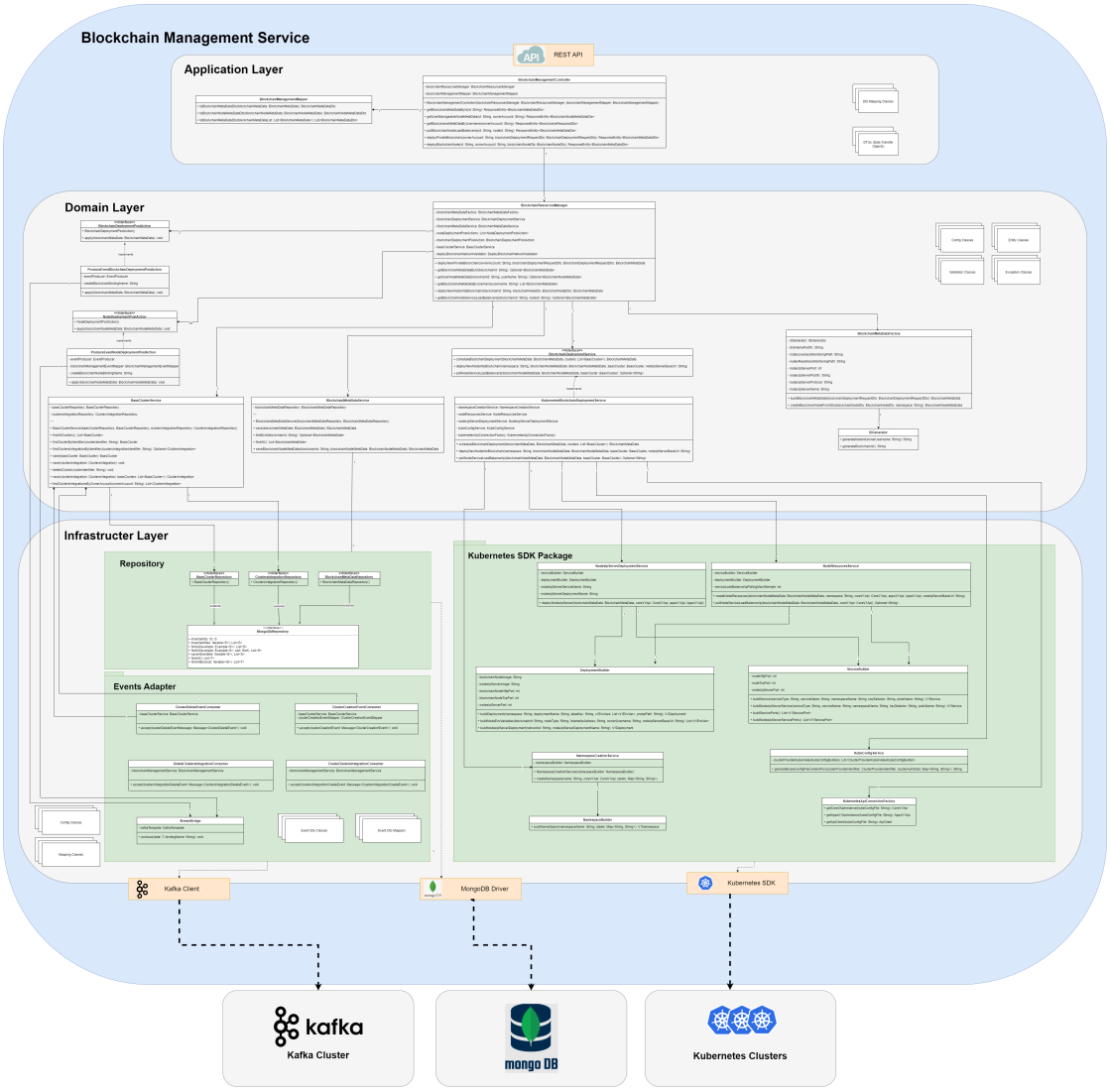

Figure A.3.: Blockchain Management Microservice - UML Class Diagram.

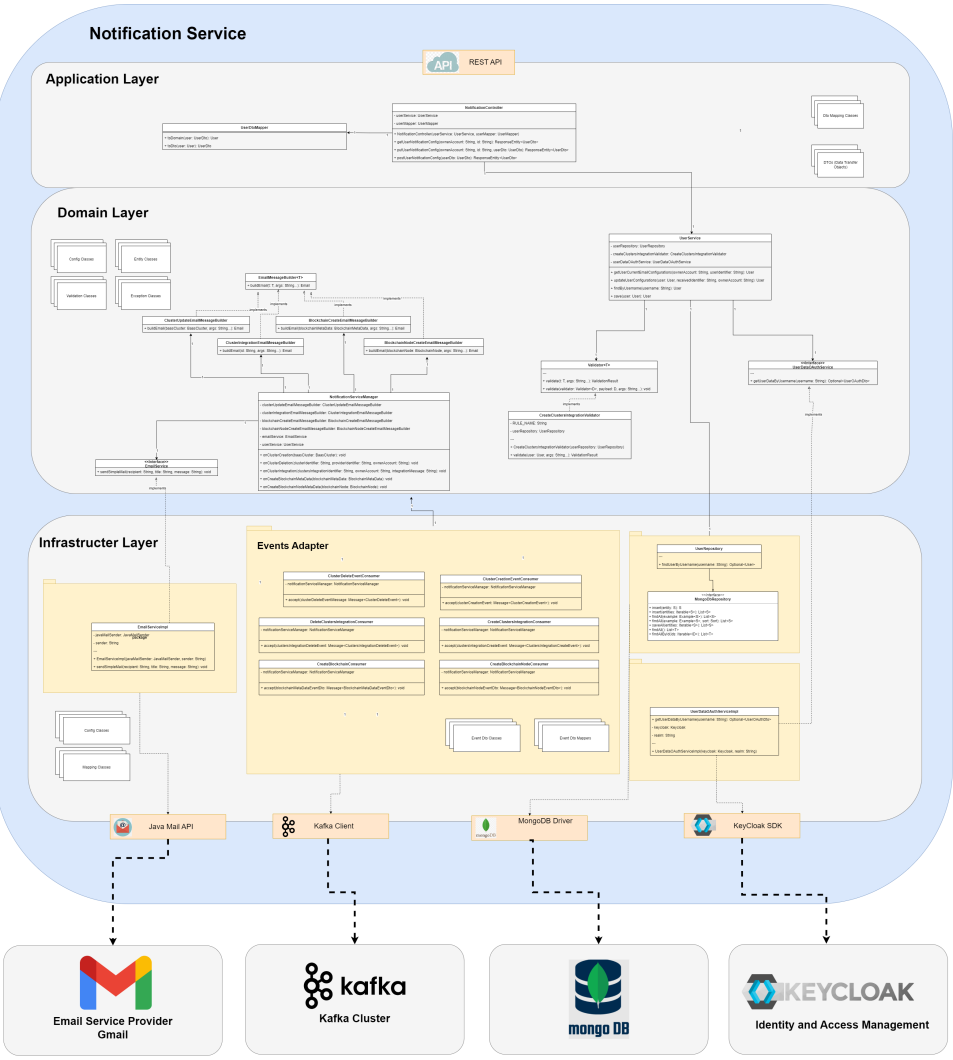

Figure A.4.: Notification Microservice - UML Class Diagram.