universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Blockchain as a Service Solution for Ethereum Smart Contract Based Micro-Service Cloud Architecture“

verfasst von / submitted by

Zheng Li, BSc

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

Master of Science (MSc)

Wien, 2020 / Vienna, 2020

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 066 935
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Medieninformatik
Betreut von / Supervisor:	Univ.Prof.Dr. Wolfgang Klas
Mitbetreut von / Co-Supervisor:	Dr. Belal Abu Naim

Declaration of Originality

I hereby declare that except where specific reference is made to the work of others, the content of this thesis is original and has not been submitted in whole or in part for any other degree or qualification in this, or any other university. I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others.

13. Nov. 2020	Zheng Li
(Date)	(Signature)

Zusammenfassung

Blockchain ist eine neu entwickelte Technologie, die in den letzten Jahren erhebliche Beachtung gefunden hat. Die Industrie in verschiedenen Bereichen versucht nun, die Technologie zu nutzen, um von ihrer hochsicheren, transparenten und dezentralen Natur zu profitieren. Die drastischen einzigartigen Eigenschaften der Blockchain stellen jedoch sowohl die Hardware-Infrastruktur als auch die Software-Entwicklung vor große Herausforderungen für die Nutzung der Technologie. Das Konzept von "Blockchain-as-a-Service"(BaaS) konzentriert sich darauf, die Lösung anzubieten, mit der der technische Aufwand und die hohe Arbeitsbelastung für die Bereitstellung, Verwaltung und Wartung des Blockchain-Netzwerks in einer Cloud-Umgebung erhöht werden können. Diese Arbeit befasst sich mit den neuesten Technologien und Lösungen in den aktuellen verwandten Bereichen, diskutiert die mögliche Lösung zur Verbesserung der Benutzerfreundlichkeit der Blockchain-Technologie mit einem Cloud-Computing-Ansatz und präsentiert einen Prototyp, der für die Verwendung der Blockchain-Technologie im Kontext entwickelt wurde mit Microservice-Architektur in einer Cloud-Umgebung. Der Prototyp wird anhand der vordefinierten funktionalen Anforderungen, der Benutzererfahrung und der Systemleistung bewertet. Es werden weitere Verbesserungen und Erweiterungen vorgeschlagen, die auf den im gesamten Forschungsprozess gesammelten Erkenntnissen und Kenntnissen beruhen.

Abstract

Blockchain is a newly emerged technology that gained significant attention in recent years. Industry in various fields are now attempting to adopt the technology to benefit from its highly secure, transparent, decentralized nature. However, the blockchain's drastic unique characteristics present serious challenges in both hardware infrastructure and software development for utilizing the technology. The concept of "Blockchain-as-a-Service" (BaaS) focuses on offering the solution of lifting the technical overhead and the heavy workload of deploying, managing, and maintaining the blockchain network in a cloud environment. This work looks into the state-of-the-art technologies and solutions in the current related fields, discusses the possible solution to improve the usability of the blockchain technology with a cloud computing approach, and presents a prototype designed for utilizing blockchain technology in context with microservice architecture in a cloud environment. It evaluates the prototype based on the pre-defined functional requirements, user experience, and the system's performance. It proposes further improvements and extension based on the findings and knowledge gathered along the entire research process.

Contents

1 Motivation

The blockchain is a revolutionary technology that keeps gaining more and more attraction in both industry and people's everyday life. Raising earlier from the concept of cryptocurrency, namely the Bitcoin, this technology went through rapid growth and evolution in a short period, and the hype seems not getting colder anytime sooner.

Nowadays, we see a lot of implementation and utilisation of this brilliant concept like Ethereum and Hyperleger. Many of these newly emerged blockchain technologies focus mainly on a common goal together, that is to extend the usability of blockchain technology further into a broader and generalized field. Down to the basic, blockchain technology works as a decentralized database that stores transaction data in a chain of blocks in which adjacent blocks on the chain are interconnected and dependent. By using encryption and digital signatures, the data stored on a blockchain is highly secured and tamperproof. Outside users can interact with the blockchain via a personal account, which is protected by asymmetry encryption. All the changes a user made are permanently stored on the blockchain and never changed.

This highly secure, transparent, and decentralized nature makes the blockchain technology a perfect solution for use cases where data security and integrity are required, for example, in financial, health care, and so on.

With the rise of the concept like IoT, cloud computing has become widely used and well accepted in both the enterprise and consumer market. Cloud service providers like AWS, Microsoft Azure, and Google nowadays offer a wide range of cloud computing solutions and resource to allow both companies or individuals to deploy easily accessible cloud service in a simple fashion.

The beauty of cloud computing services is that it releases developers from the heavy workload of infrastructure configuration and management, allows them to focus on the implementation of the service itself. By utilising technology like Content Delivery Network (CDN) and Virtual Machine (VM), cloud computing can achieve high accessibility, reliability, and scalability. By migrating from traditional web services architecture to cloud computing, small businesses can start up any idea without high financial risk. Large companies can benefit from reducing network infrastructure management costs hugely while maintaining and improving the quality of the service.

Typically, in a cloud architecture, a sophisticated use case is divided into small independent running micro-services. It breaks down complicated business logic that is hard to implement and deploy, allowing developers to work cooperatively on a large scale. Because each micro-service only handles a tiny part independently, not only is the risk of a single point of failure significantly reduced, but network resources can also be efficiently distributed.

Although blockchain is the buzzword in recent years, the technology is still in its infancy. As we know, many current major platforms are still under development and facing dramatic changes in the future. A considerable challenge it faces is to generalize its usability in the broader field. On the other hand, with its distinct difference to the other technologies, the learning curve is much higher. Setting up a project with blockchain technology requires a lot of experience and expertise. In many cases, developers are designing project-specific architectures to utilize blockchain technology.

Currently, well-known cloud service providers like AWS and Microsoft are trying to combine the blockchain technology with the cloud computing service they offer. The idea is to provide a complete and easy-to-use solution for blockchain development and consumption. The advantage of combining these two technologies is that they are complementary in many aspects. To ensure the security of the service, a cloud service provider needs to invest a large amount of effort to protect user's data. Blockchain provides a simple yet elegant solution for data security and integrity. With a systemic cloud architect, we can reduce the complexity of developing a blockchain system and extend the functionality of the blockchain technology. Both developers and end-users will benefit from it.

1.1 Objectives

The primary challenge of this thesis is to provide an acceptable prototype solution that embeds the Blockchain technology in a cloud architecture and offer it to the end-users as a Blockchain-as-an-Service (BaaS) and allow users to create, deploy, and access micro-service architecture use cases, which are backed-up by Blockchain technology, without much technological overhead.

1.1.1 Specification of Functional Requirements

This section discusses the specific functional requirements of the prototype solution based on the objective of this thesis. Comprehensive and detailed architecture and design decisions are discussed in Section 3.4 and Section 4. On a grand level, the prototype solution should meet the following functional requirements.

– Cloud-based Blockchain management

  A user should be able to configure, deploy, and manage Blockchain instances that run in the cloud environment. These Blockchain instances serve as the foundation of the micro-service architecture applications that are created and consumed by end-users of the platform. The prototype should be a user-friendly solution that eliminates unnecessary technical details from the conventional blockchain setup and flats the steep learning curve of the Blockchain technology.

– Smart Contract IDE

  Although the primary focus of the prototype solution is not about providing users a state-of-the-art, full functional online smart contract IDE, users should be able to compose, compile, and at some level, debug smart contracts in an embedded environment. The prototype solution should offer a project-based code repository functionality as well to allow users to deploy and manage smart contracts with flexibility.

– Smart contract driven micro-services

  Micro-service architecture lives in the center of the concept. In a classical cloud architecture, database and traditional computing powers the micro-service. In comparison, the prototype solution should utilize smart contract that deployed on a blockchain instance as the back-end of the micro-service users created and have access to. The owner of the micro-service should be able to expose the service without spending excess effort or diving deep into the actions that take place behind the scene.

– Blockchain-based cloud data management

  One most prominent and commonly known disadvantage of the current blockchain technology is its lack of performance in terms of data storage and query. As data storage and query functionality plays a vital role in micro-service cloud architecture, the prototype solution must demonstrate its capability regarding handling data storage and query requests at a large scale and, meanwhile, reflects the highly secure, tamper-proof nature of the blockchain technology.

– Authentication and data privacy control

  Data transparency is a fundamental aspect of a blockchain system, but it also presents challenges in terms of personal privacy. We discuss this matter later in Section 4.3.11. In combination of a cloud architecture, the prototype solution must offer the possibility for end-users to take access control of their exposed micro-services, thus ensure their data privacy.

1.1.2 Focused research questions

Considering the functional requirements discussed in Section 1.1.1, I will focus on answering the following research questions in this thesis.

- How can we improve the usability and productivity of the Blockchain-as-an-Service (BaaS) solution? To which level can we simplify the whole blockchain application workflow?,
- How to associate the smart contract with micro-service architecture? What role can smart contracts, or generally speaking the Blockchain technology play in a cloud architecture?.
- How to improve the performance, availability, and scalability of the blockchain cloud solution, or what is the limitation that the solution face in the current state?
- What is the workflow for an end-user to create blockchain-based micro-services look like? How to handle the supporting requirements like data storage?

- In terms of security and privacy, how can user authentication be defined and handled in a BaaS context? What strategies can we take to protect users' privacy?

2 Background

In this section, we discuss the technologies that are chosen for implementing the prototype solution.

2.1 Micro-service oriented cloud architecture

The concept of micro-service was initially introduced as a software architecture to build large, replaceable, and maintainable systems. Generally speaking, micro-services are small, independently deployed service-oriented components that are loosely bounded in a correlated context. Micro-service should always be goal-oriented rather than solution-focused. By breaking down a complex system into small parts, developers can provide particular approaches for solving specific tasks. Thus avoid the problem of the system being too big and too complicated. Another design principle of micro-service is its replaceability. Rather than spending efforts and resources to maintain a service, replacing the component should always be the prioritised consideration for developers. [14]

Adapting to micro-service architecture provides many benefits. From developers' perspective, making system components individual and independent makes the team cooperation flexible and efficient. The choice of technology, language, and framework is no longer restricted. Different technologies, languages, and frameworks can co-exist in the same system and achieve solution optimisation. From the view of the whole system, a system can start small and grow large and complex at a steady pace. Developments can be run in parallel without much interference from other parts of the system. The system can upgrade or down-grade parts of its services gracefully at any given time to improve its robustness and minimises the cost of failure. [15]

2.2 Ethereum

Ethereum is an open-source, blockchain-based distributed computing platform. It was proposed in late 2013 by Vitalik Buterin and initially released in mid-2015. In the Ethereum paradigm, the Ethereum Virtual Machine (EVM) sits in a decentralized peer to peer network and processes transactions using public nodes in the network. In comparison to the transaction in other blockchain networks that serve as a decentralized ledger, the transactions in an Ethereum network can also be used for generalized computing. Generally speaking, Ethereum can be viewed as a transaction-based state machine. The starting point of the state is called the genesis state. As the state machine keeps on running, new transactions are executed and the state morphed consistently. During the transaction, the Ethereum state transition function carries out computation that modifies the chain to its next valid state, and arbitrary states between transactions are stored. All transactions are settled into blocks that are chained together with cryptographic hashes.

Ethereum supports both "proof-of-work" and "proof-of-stake" consensus algorithms. Under the "proof-of-work" protocol, network nodes invest a substantial amount of computing resources in completing transactions and get Ethers as the reward. This process is also referred to as mining and acts as a firm guarantee of data integrity. However, as the network grows, the difficulty of solving the mathematical puzzle increases, and the amount of transactions the network can handle per time unit is restricted. [16]

"Proof-of-stake" is a protocol that attempts to solve the throughput issue. It allows nodes to reach consensus without mining. Instead of generating new blocks competitively, the next node to create a new block on the chain is selected based on its stake. The stake of the node can vary based on the requirement. For instance, it can be the number of coins it holds or voting in a non-cryptocurrency scenario. "Proof-of-authority" is a specific case of "proof-of-stake". Instead of electing and changing stakeholder on the go, a validator's identity takes the role of the stake. Blocks are exclusively added by those who are officially determined as a validator.[17] The advantage of "proof-of-stake" protocol is that it increases the efficiency of transaction time and overall network consensus. It is particularly constructive in a private chain environment where authority needs to be assigned.

2.3 Blockchain as a Service (BaaS)

With the constant maturity of the blockchain technology, many transitional cloud service providers start to extend their cloud service solutions toward blockchain. One of the reasonable motivations is the growth of the IoT usage. By its nature, the secure and tamper-proof decentralized public ledger function provided by blockchain technology is an ideal solution for public infrastructure in fields such as energy, public security, and public health. However, because IoT devices are typically restricted in size and production costs, the computation power, data storage, and network capacity they can offer are usually limited. Running a blockchain network in such an environment is extremely difficult, and even if we leave out the consideration of the financial costs, we can not ignore the computational resources wasted to maintain such a project. [18]

Recently, to solve the problem of where should be blockchain be hosted, the concept of Blockchain-as-a-Service (BaaS) has emerged. It focuses on constructing a cloud-based solution for hosting blockchain instances to lift the obstacles the user faces in building and utilising blockchain technology. Users no longer have the burdensome task of managing the infrastructure for a blockchain project. Currently, service providers like AWS, Microsoft, and IBM offers BaaS with different approaches. Other providers like Google have announced their plan to join in the future. In Section 3, we discuss two well-known cloud service providers and their BaaS solutions in-depth, namely AWS Blockchain Template and Microsoft Azure Blockchain Workbench.

2.4 Docker and container technology

Docker aims to solve the "dependency hell" problem in modern applications. It is a Platform-as-a-Service (PaaS) solution that encapsulates a complete application runtime environment in a container. Under the hood, docker utilises the resource isolation feature of the Linux kernel like cgroups and namespaces to create isolated lightweight run time environments. It differs from the approach that is taken by other hypervisor-based virtualisation methods. Traditionally, hypervisor-based virtualisation runs either directly on the hardware(Xen) or as an additional software layer on a guest OS (VirtualBox). Both of these two types of virtualisation suffers from performance bottle-neck because they run a full-fledged operating system on top of another operating system and requires more resources like CPU, RAM, storage, and network bandwidth.

Containers, on the other hand, only take a protected portion of the Linux operating system. Each container then has its own isolated processing power, storage, and network resources. This approach provides significant benefits, as resources are more efficiently distributed and consumed. When the container stays in an idle state, it wastes almost nothing from the system resources. Furthermore, the cost of creating, running, and destroying containers is low.

Another great feature that docker offers is that it depends on the Advanced Multi-layered Unification Filesystem(AuFS) to deliver a copy-on-write experience. Advanced Multi-layered Unification Filesystem(AuFS) layers filesystem transparently in a stack. This allows the docker container to be composed of basis images, and basis images can be cross-referenced in different containers. [11]

Since its release, docker has been quickly adopted by both developer community and enterprise. Its use cases extends from professional development environments to large scale production environments. Especially in the cloud computing area, where performance, availability, and scalability is required, docker shows excellent potential.

2.5 Kubernetes and container-orchestration

Nowadays, more and more systems that rely on delivering service over the network via APIs face the challenge of high demand. Often in times, high availability and minimum failure tolerance are the critical driving force of a successful business. Kubernetes, an open-source container orchestration system developed by Google, gave the answer and provided the solution for building scalable, reliable distributed systems.

In traditional software systems, components are considered to be a mutable infrastructures. Developers use imperative approaches to bring the system to a desired state. Imperative means that the developers interact and change the system by giving descriptive instructions that bring the system from point A to point B. In contrast, in Kubernetes, software components and containers are treated as immutable assets of the system, and developers control the system primarily by declarative configuration. The declarative configuration approach generally allows the developer to control the system by describing the desired state of the system, and the heavy-duty of maintaining the system in the configured state is left to Kubernetes entirely. For example, if the developer tells the Kubernetes to run 3 instances of an application container in the cluster, Kubernetes runs exactly 3 replicas of that container. When replicas failed, Kubernetes starts up new replicas to the meet the configuration, oppositely, it kills running replicas if there are too many.

The Kubernetes approach provides several advantages over the traditional approach. First, software components can be divided into smaller units and kept isolated from other parts of the system. This not only prevents the situation where a single point of failure can bring the entire system down, but also divides the software components into much smaller chunks so that development teams can work more efficiently and flexibly. Second, the declarative nature of Kubernetes allows easy scaling for applications. Operators can make wise decisions and balances the operation costs based on the demands, starting low and gradually scale up while the business grows.

Additionally, because containers are immutable and can be replaced easily, developers can roll out new features and updates at a steady pace without bringing down the entire system, different versions of the system can co-exist without interference. In case of a failed upgrade, developers roll back to the old version and reverse all the changes. The system keeps on running without any downtime. [8]

3 Related Work

In this section, we take a look at some of the state-of-the-art researches regarding Blockchain-as-a-Service technology since the prototype is built on this concept. We also discuss and compare a few commercialized solutions that are currently offered on the market by some big players like Microsoft, Amazon, and IBM.

3.1 Blockchain as a service architecture design research

Many researchers in academia are exploring the possible architecture design of a Blockchain-as-a-Service focused cloud architecture nowadays. In work from Qinghua Lu et al. [10], the authors presented a unified and vendor-independent solution to address the scalability and security issues of blockchain-based applications. The solution consists of three categories of services, namely deployment as a service, design pattern as a service, and auxiliary-based application. Services in deployment as a service category allows users to configure and deploy customised blockchain network, monitor the status of the network, and deploy smart contracts on the blockchain. With an up and running blockchain network, users can utilize services in design pattern as a service category to manage data and smart contracts on the chain. Services in this category use concepts like off-chain data caching, smart contract design patterns, and data hashing and encryption to ensure both the accessibility and privacy of the on-chain data. Data integrity services like encryption key pairs management and file comparison are provided in the third category of auxiliary services.

NutBaaS [19] is another example of the Blockchain-as-a-Service architecture. The researchers proposed a four-layers design that places reliability and security at first and complements the shortcomings of the current BaaS platform. At ground level, the model is built on the resource layer, which provides cloud resource and infrastructure support for deploying blockchain networks with a variety of service providers, including Amazon Web Service, Microsoft Azure, and Alibaba Cloud. Above the resources layer lies the service layer in which blockchain essential services and advanced services like developer tools are implemented. It provides the users with the required functionalities for blockchain application development. By combining the functions offered by these two base layers, the third layer, the application layer, provides some abstracted applications to help create solutions faster according to business requirements. On top of that, the fourth layer, which is the business layer, explores business scenarios suitable for using blockchain technology.

3.2 Amazon Managed Blockchain

Amazon's "Amazon Managed Blockchain" solution is a fully managed service that gives the users ability to create and manage scalable blockchain networks. By the time of this thesis, the platform supports the Hyperledger Fabric blockchain framework and promise to extends the support for Ethereum in the future. The platform generally takes care of provisioning nodes, setting up the network, managing certificates and security, and scaling the network. User has the flexibility of managing network membership, inviting additional members from other AWS accounts, or add and remove new members by a voting API. Members of the network configure peer nodes that run the decentralized network and have the ability to scale up and add new nodes to increase the transaction processing power of the network. Communication within the blockchain network is secured by AWS Key Management Service, which is a component that manages user identities and issues enrolment certificates. Additionally, Amazon managed blockchain depend on Amazon QLDB technology to record immutable change logs and maintain the entire history of all transactions in the blockchain network, thus improving the reliability of the data.

3.3 IBM Blockchain Platform

The IBM Blockchain Platform provides a managed, full-stack enterprise-ready Blockchain-as-a-Service platform that aims at simplicity, flexibility, and reliability. Build on top of open-source technology like Red Hat OpenShift and Kubernetes, the solution is not vendor locked. Users have the flexibility of selecting the deployment platform on IBM Cloud or hybrid and multi-cloud environment. It uses Hyperledger Fabric as the core of the open-source component. To ensure the governance capabilities, the platform provides democratic management tools that allow members to manage the rules and policies of the decentralized network. Members must be known before joining the network, and new members can be dynamically added or removed. The system leverages protocols on the side of the blockchain network to ensure the validity of the transaction. This including authorizing client by initiation, validation by endorsers, confirmation of endorser response, and validation of the transaction by all network peers. Mechanisms like channels, private databases, and zero-knowledge proof technologies give a high protection of the data privacy on the network.

3.4 Microsoft Azure Blockchain Workbench

Microsoft offers "Azure Blockchain Workbench" as blockchain as a service solution in its whole cloud service ecosystem. Aside from the same fast and straightforward blockchain network deployment feature, the service is tightly packed with other Azure cloud services to create a hybrid environment. Smart contracts deployed on Azure Blockchain Workbench are more deeply controlled and managed by the workbench. To maintain the authentication and security of the blockchain application, Azure Blockchain Workbench depends on Azure Active Directory (Azure AD). It's a one-stop solution for the Azure cloud user to manage their user identities and create intelligence-driven access policies to secure resources. Not only is Azure AD the security gateway for all the other Azure cloud services that connected with the workbench, but it is also capable of providing access control down to each smart contract deployed on the workbench. Once a blockchain application is deployed, the workbench can generate a web-based client application automatically and provide an interface for users to interact with. Developers can also choose to expose the blockchain application via the RESTful gateway service API. To improve the performance of the blockchain application, the workbench keeps an off-chain SQL database and Azure Storage as the replica of the on-chain contract definition, configuration, and SQL-accessible data stored in contracts. Users can perform data queries by directly accessing this database, makes it easier to visualize and analyse the current state of the blockchain application. For application requires storage for large data, the workbench supports storing documents or other kinds of media content with blockchain business logic.

lize This section covers the software architectural design decisions of the prototype solution. In context of the functional requirements in Section 1.1.1, we discuss the structure of the prototype solution, suitable technologies and design patterns choices and take an in-depth view of the sub-components design of the system.

3.5 Overview

The prototype is a typical client-service architecture web-based application that offers user fast and straightforward blockchain-based micro-service functionality. At the front end, users interact with the system via a graphical user interface deployed as a web application, and the front end communicates with the backend via HTTP requests to invoke API that controls the whole system. In comparison to a typical data-oriented web service, deploy and maintaining blockchain instance in a cloud environment presents many challenges. To begin with, blockchain instances are basically sets of peer to peer network, to let them run on the cloud without interference with each other, suitable virtualisation and cluster management technologies are essential. Additionally, blockchain nodes consume a large amount of computing and storage resources, the performance and scalability of the prototype must be considered beforehand. Moreover, the control of users' access and interaction with the system, including with the blockchain instances they owned, is a crucial aspect that not to be neglected.

3.6 Technology and design pattern choices

3.6.1 VM vs Docker Container

As we discussed in Section 2.4, the main advantage that the container technology has against the hypervisor-based virtualisation is its fast deployment speed and high efficiency on resource usage. In our scenario, fast blockchain network nodes provision is the fundamental presupposition of a functional and user-friendly system. Although a hypervisor-based virtual machine may give more flexibility and control in terms of hardware specifications like computing power, RAM, and storage capacity, the isolated environment that a docker container provided is more than enough for a running blockchain node. Because a new NET namespace is instantiated separately from the Linux kernel NET namespace for each docker container created, the container has its own network stack, and the network stack is isolated and can not be seen from outside the container and from other containers. This simplifies the network configuration and prevent nodes interference in a virtual cluster setup. Most importantly, as nodes in a blockchain network have the same software framework stack, it is much faster to start up a node from a docker image due to the time freed from OS boot up and software installation. [7]

3.6.2 Kubernetes cluster for container orchestration

Due to the nature of the prototype and the choice of using docker containers as the base layer of the system, the prototype uses Kubernetes for container orchestration. By utilising the declarative configuration approach, we can create a system with high scalability. The decoupling of the hardware management and software deployment means more productive hardware resources utilisation, and simplified capacity extension. Kubernetes offers a flexible and highly customisable RESTful style API for managing in-cluster containers, which helps create and remove blockchain network on the fly.

To ensure continuous development, I used Skaffold as a helper in the development environment. Skaffold is a command-line tool that handles the workflow for building, pushing, and deploying Kubernetes-native applications.

3.6.3 Node.js and JavaScript

Node.js is a JavaScript runtime environment that executes JavaScript server-side code. Chrome's V8 JavaScript engine, which Node.js is built on, compiles JavaScript code directly to machine code before execution. As a result, applications run at high speed, despite that JavaScript is a scripting language. It is stable in building real-time network applications thanks to its event-based loop non-blocking execution. In spite of being a single-thread process, Node.js applications usually accomplish high throughput and low latency, especially in handling large amounts of requests simultaneously. A Node.js application can scale up both vertically by expanding resources in a single node and horizontally by adding additional nodes. Node.js, together with JavaScript, is the primary development language of choice for the prototype.

3.6.4 RESTful API design and Express.js

RESTful is a stateless client-server protocol for building service APIs. Each HTTP request encapsulates all the information required, and can be handled separately without the need of previous state. Each resource in a REST system is represented in the form of a unique URI, and is manipulated via action specified by HTTP request methods (POST for creation, GET for retrieving, PUT for update, and DELETE for removing). REST protocol separates the client and server-side development, improves the overall portability, visibility, and reliability of the system. There is no bound of format for information exchange, whether it's XML, JSON, or even plain text.

Express.js is a flexible and minimal framework for creating APIs in Node.js environment. The most significant advantage of the Express.js framework is its capability of fast application prototyping. In the prototype, the backend is a RESTful style API built with Express.js framework.

3.6.5 MongoDB for document-based database

MongoDB is a document-based, distributed database for general purposes. Data is stored in JSON-like documents and is retrieved as an object. By design, MongoDB suits well in cloud scenarios where high availability and scalability is required. Although it is not well known for data aggregation because of its lack of query support, the schema-less operation is well adapted for data structures that change over time. For a typical user-oriented web application, MongoDB is clearly the right candidate for the database.

3.6.6 Task Queue and Redis

Many actions in the prototype takes a certain amount of time to finish. Take the deployment of a blockchain network for example, and the system must first collect the configuration of the network, then send provision requests for each chain node and waiting for all the nodes to be online. Although Node.js support non-blocking execution and node scaling, it is a better practice to isolate these tasks away from the API server. A FIFO Task queue with a task agent pool is a suitable pattern here to deal with the problem. Whenever a user requests a certain task to the API server, API server publishes the task to the task queue, a subscribed task agent in the agent pool retrieves the task and execute it. In this way, the system can behave more responsive to the user and allows the developer to efficiently and reasonably distribute resources when scaling the system. To achieve this pattern, the prototype utilizes the Pub/Sub feature of the Redis database to implement the task queue.

3.6.7 React for Single-page Web Application

Currently, in the world of JavaScript web front-end development, there are many outstanding framework. React.js is one of them which gained the most popularity over the last few years due to its excellent performance in building Single-page applications (SPA). The term "Single-page Application", according to A. Mesbah and A. van Deursen, generally means "the single-page web interface is composed of individual components which can be updated/replaced independently, so that the entire page does not need to be reloaded on each user action". [13] Behind the scene, a single-page application relies on AJAX (Asynchronous JavaScript And XML) to communicate state changes with the server asynchronously, and achieves a faster response and better user experience. [12] The front-end of the prototype is a Single-page Application developed with React.js

3.7 System components

Six major components work together in the system, Web UI, API Gate, Database Gate, Transaction Gate, Chain Agent ,and Task Agent. Their relationship is shown in Figure 1.

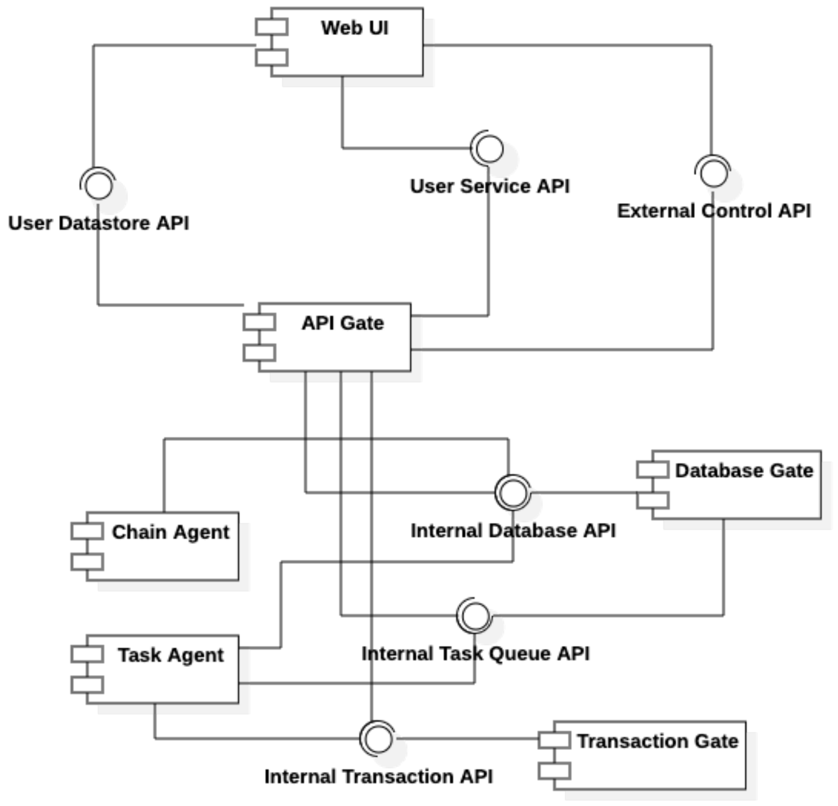


Figure 1: System Components of the Prototype BCCloud

Web UI is a single-page React.js application that acted as a front-end user interface of the system. It provides graphical user interfaces for interaction with the system. In the background, the Web UI communicates with API Gate via External Control API, User Service API, and External Control API implemented by the API Gate. For security concerns, communication between the WebUI and API Gate is governed by the authorisation policy of the system and isolated from other system components. Section 4.3.11 provides more detail on the authorisation and security control of the system

API Gate is the entry point of the backend system. It implements three access interfaces, namely the External Control API, the User Service API and the User Datastore API. The External Control API provides access point functionality for user account management and authentication, private blockchain instance deployment and management, project-based smart contract development and deployment, micro-service and datastore creation and management, and user task monitoring. The User Service API and The User Datastore API route requests to exist User Service and User Datastore that are created by users.

Database Gate is the gatekeeper for the MongoDB and Redis databases of the backend. It implements the Internal Database API for caching on-chain data and storing information of user accounts, private chain instances, smart contract project, micro-services, and datastore. It also provides the Internal Task Queue API for coordinating task execution in the system.

Transaction Gate offers the Internal Transaction API for chain transaction control in the system. Because process like contract deployment and contract method call are similar procedures for all blockchain network deployed in the system, a universal API that abstract these operations can lead to a compact design and ensure unified authorisation.

Task Agent subscribes to the Task Queue via the Internal Task Queue API and continuously monitoring for new tasks, whenever a task is published and acquired by a task agent, the agent executes the task and publish task process and result. The tasks carried out by Task Agent are usually time-consuming and doesn't require users attention to endure. For instance, deploying a private chain instance, compiling a smart contract, and deploying a datastore.

Chain Agent is a complementary component that sits alongside each private chain instance. It provides chain-data monitoring and on-chain data caching for User Datastores deployed on the chain instance. Through the event and log mechanism in Ethereum, the chain agent listens to chain transaction events and updates the caching status in the database once a transaction is mined. Further explanation of how Chain Agents work with Datastore can be found in Section 4.3.8.

3.8 Deployment View

Considering that Kubernetes provides an abstraction of the hardware infrastructure and enables developers to build, deploy and manage applications truly portable across environments [9] , in order to give a clear view of the deployment structure of the system, we focus here on describing the internal organization of the Kubernetes cluster.

3.8.1 Kubernetes basic concepts

Namespace creates virtual clusters in the same physical cluster. Objects within the same namespace must have a unique name, however, across the namespace, objects names do not interfere with others. For this reason, namespace is a practical way of dividing resources between users in the same cluster. [2]

Pods are the smallest execution unit in the Kubernetes cluster. It contains single or multiple application containers that work cooperatively as a unit that ideally can not be separated in terms of functionality. Each pod has a unique IP address, and containers inside all share the same network namespace. A set of shared volumes can be attached to a pod to provide persistent data storage for all containers in the pod. [3]

Deployments Although pods are the basic unit in the Kubernetes, it is often deployed with a controller to enable more control features like replication, failure recovery, and rollout. A deployment lets users describe the desired state of an application deployment, and matches the actual state to the desired state constantly at a controlled cycle. The most common use cases, for example, are application scaling and non-interrupt feature rollout. [1]

Service As pods in the cluster are dynamically created and destroyed, their IP address can not stay constant. Service exposes a set of pods by selector mechanism, decouples the binding of network address and network requests. [4]

3.8.2 Deployment View in Kubernetes Cluster

Figure 2 depicts the deployment view of the prototype Kubernetes cluster. System components are wrapped in Kubernetes Pods(round rectangle), their desired running state is described and encapsulated by the Kubernetes deployment(raised up rectangle box). Communications between components are depicted as solid black lines. User interact with the system from outside using a browser by sending request into the cluster. The ingress guards the entrance of the cluster and route the top-level domain traffic to WebUI Deployment and External Control API, User Service API and User Datastore API request to API Gate Deployment. Other components communicate internally and are isolated from outside the cluster. All system components except user-created chain instance components are deployed in the default Kubernetes namespace. For each user chain instance created by Task Agent, a unique namespace is created for the deployment to prevent interference.

Generally speaking, each deployment in the cluster can be scaled up and down by increasing or reducing the replicas count, as it can be seen from Task Agent Deployment. Furthermore, the component Chain Agent are bounded with the same pod in which the chain transaction node is deployed. It allows the Chain Agent to easily monitor the chain event via the shared network stack in the pod.

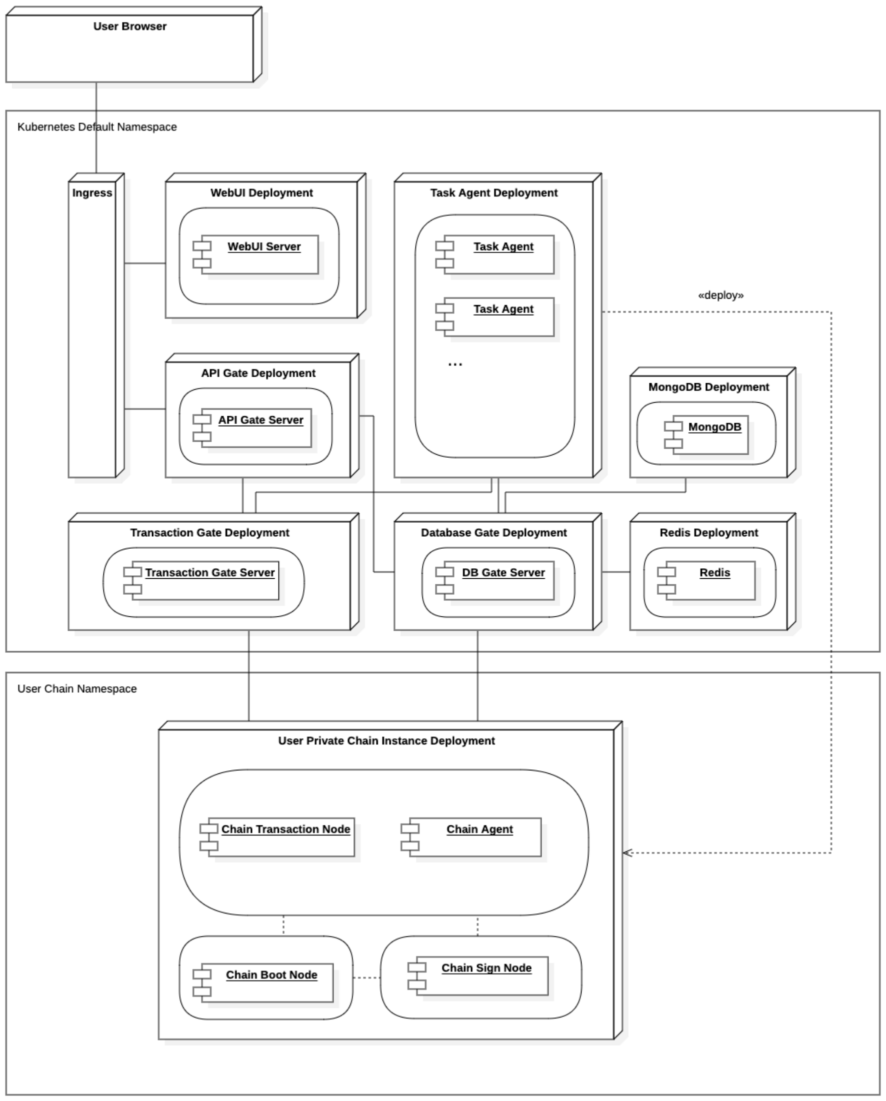


Figure 2: Deployment View of the Prototype BCCloud

4 Design and Implementation

4.1 Terminology

In this section we take a look at some of the basic terminologies and concepts used in the prototype.

– User and user account

  A user in the system is a person who has a registered account and performs interaction with his or her assets, namely private chain instances, projects, services, and datastores. The account encapsulated the user's identity information and authentication credentials. In the prototype system, a unique Ethereium user account is bounded with each user account in order to achieve a unified authentication mechanism and provide access control to the system.

– User chain

  User chain is the private blockchain instance configured and deployed by a user in the system and is the groundwork of other system features like user service and datastore. Because each instance is placed in a unique namespace in the Kubernetes cluster, a chain instance must have a unique name across the entire system.

– User project and project artifacts

  A user project is where a user develops, stores, and compiles smart contracts that are related to the same project. Compiled smart contracts are stored in the project as the project artifacts and can be later deployed on to a running user chain instance.

– User service

  A user service is the "micro-service" like feature of the prototype. In a word, it represents the entry point of the invocation call to a smart contract function on the blockchain. It handles the process by converting HTTP requests to blockchain transactions, thus allows the service consumers to interact with the services without knowing much details on the underlying smart contracts and chain instances. The user who created the service controls the access to the service via the consumers' identity, namely the Etherium account bounded by the user account.

– User datastore

  User datastore is a concept designed for improved data storage and query experience on blockchain. In a datastore, users can store data entries in a table like fashion. The data is saved on the chain and cached in a complementary database to ensure its integrity and offers faster query speed. Like user services, datastore implement the same access control the secure the data stored inside.

4.2 Use Cases

In this section, we discuss the use cases of the prototype by function group. Figure 3 shows the use case diagram of the prototype.

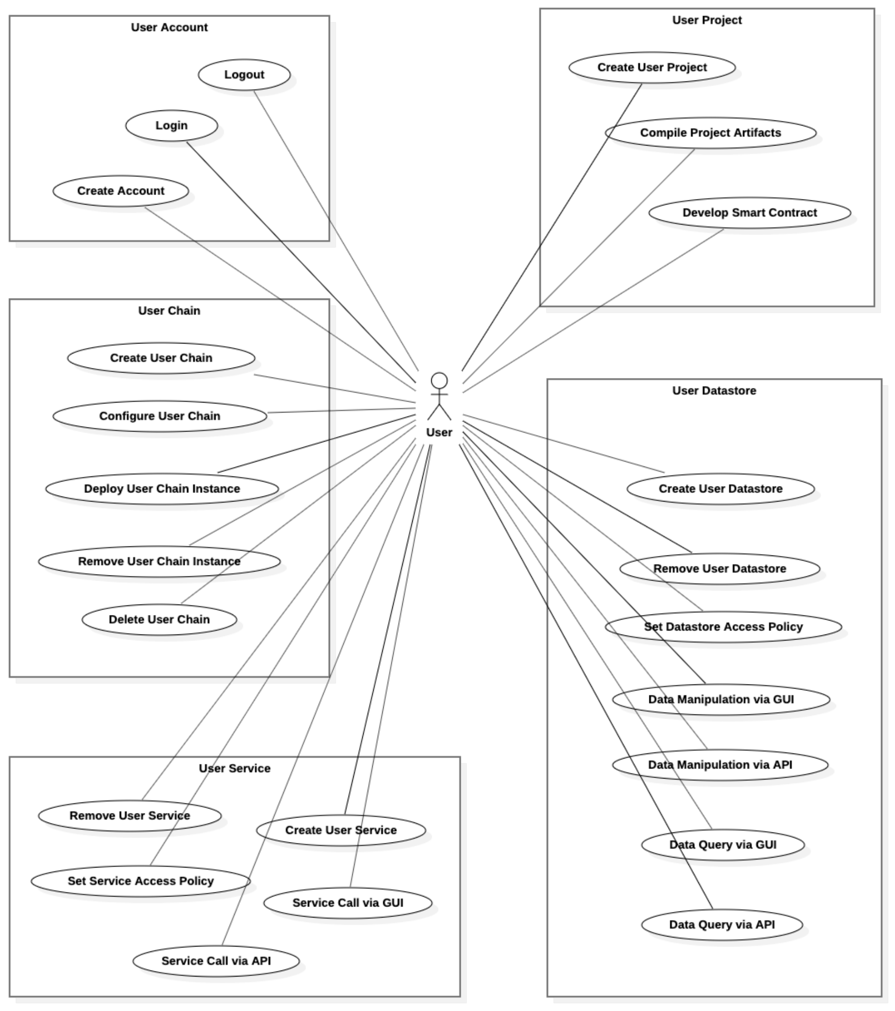


Figure 3: User Case Diagram of the Prototype BCCloud

4.2.1 User Account Use Cases

– Create Account

  A User must create an account before interacting with the system. For registration, the user needs to provide a valid ID (E-mail address) and a password to the system. The system generates account credentials, an Ethereum account with a public account address and a private sign key, and informs the user that the account is ready. User ID must be unique across the system.

– Login

  After the user creates an account, he or she needs to login to interact with the system. The user provides the user ID to retrieve the encrypted account credentials and decrypts it locally with the password. Then user authenticates with the server using the decrypted account credentials to finish the process and gain access to the system. At login, users have the option to stay logged in to skip this action in the future access, in which case, account credentials is preserved locally in the browser.

– Logout

  The user logs out to end the interaction with the system. The account credentials stored locally is then cleared no matter whether the user chooses to stay logged in or not. Afterward, further requests to the system is unauthorized due to the absence of the account credentials.

4.2.2 User Chain Use Cases

– Create User Chain

  The user can create private chains in the system. Each private chain has its unique name across the system, which is provided by the creator. Once created, the user chain is listed in the user's chain list and can be configured for deployment. Create a user chain does not create an actual chain instance, it only reserve the name of the chain in the system.

– Configure User Chain

  After creating a user chain, the user can set up the chain configuration, including its type, nodes, and parameters for mining activity. Chain's configuration can only be modified if there is no active instance deployed, or the previous instance is removed. Only after a chain is configured, the user can carry out further deployment. Access to the chain configuration is restricted to the chain creator only.

– Deploy User Chain Instance

  Once the user chain is configured, the user can start the deployment of the instance. The deployment task is firstly sent to the system task queue and then picked up and processed by a running task agent. During the deployment process, the task agent publishes task steps and logs. User has the option to monitor the process via a task console. Chain instance information is updated at the end of the task. Access to this function is restricted to the chain creator only.

– Remove User Chain Instance

  The user can remove the deployed running chain instance. Like deploying an instance, the un-deployment is handled by a task agent as well. The user is responsible for removing or shutting down user services and datastores that are deployed on to this instance. The chain configuration can be modified again after the task is finished. Access to the chain configuration is restricted to the chain creator only.

– Delete User Chain

  The User can delete a user chain with its configuration. The name of the user chain will be free for others once the chain is deleted. The pre-condition is that there is no deployed chain instance running. Access to this function is restricted to the chain creator only.

4.2.3 User Project Use Cases

– Create User Project

  The user can create a project for developing smart contracts. User only need to provide a name for the project. Afterward, the project will be listed in the user's project list. Access to the project is restricted to the creator only.

– Develop Smart Contract

  In a project, the user can create, edit, and delete smart contracts. The in-browser smart contract editor supports Solidity language high light and code hint. The user saves the smart contracts for compile into artifacts later. Smart contracts are only visible to the project owner.

– Compile Project Artifacts

  After the user saves the smart contract code, the contracts can be compiled and used as the project artifacts. The user can choose which version of the Solidity compiler to use, and the choice must be coherence with the code pragma. The compile task is sent to the task queue and executed by a task agent. The task agent publishes the artifacts in the project if the compile is successful. User then see the artifacts information like ABI, bytecodes in the artifact list. Otherwise, it stores the errors in the project. User can then see code error hints in the editor.

4.2.4 User Service Use Cases

– Create User Service

  The user can expose deployed smart contract function call as a micro-service by creating a user service. Pre-condition is a running user chain instance, and a smart contract with at least a callable method deployed successfully on that chain instance. For the creation, the user chooses the target chain instance, target smart contract on chain, and the method to invoke. Afterward, the service is listed in the user's service list. The service list is only visible to the user.

- Remove User Service

  The user can remove a running service. The service is not accessible once removed. Requests that are made before the removal of the service but still being processing in the system will not be terminated. The access to this function is restricted to the service creator.

- Set Service Access Policy

  The creator of a user service can define the access policy in two ways, a whitelist and a blacklist. If the whitelist is populated by at least one Ethereum account address, only the user accounts associated with those addresses have granted access to the service. On the other hand, if an Ethereum account address is on the blacklist, the user account associated with the address is forbidden from accessing the service.

- Service Call via GUI

  After the user service is created, the creator of the service can request the service in the GUI of the WebUI. The interface shows the required arguments of the call if there is any. The user sees the request result after the call is finished. Depends on the type of the call, its either returned data or transaction receipts.

- Service Call via API

  After the user service is created, users with the access to the service, which is defined and governed by the access policy of the service, can request the service via an exposed API point call. The API point and call format are visible to the creator of the service, and published by the creator. The requester sees the request result after the call is finished. Depends on the type of the call, its either returned data or transaction receipts.

4.2.5 User Datastore Use Cases

- Create User Datastore

  The user can create a datastore for storing data in a table like fashion. Precondition is a running user chain instance. At creation, the user provides a datastore name, type of the datastore, and the schema of the datastore(the columns). The creation task is handled by a task agent, and the datastore shows up in the user's datastore list after successful creation.

- Remove User Datastore

  The creator of the datastore can remove the datastore from the system. Afterward, the cached data will be deleted and further access to the datastore is not possible anymore.

– Set Datastore Access Policy

  The creator of a datastore can define the access policy in three ways, a read whitelist, a write whitelist, and a blacklist. If the read/write whitelist is populated by at least one Ethereum account address, only the user accounts associated with those addresses have granted read/write access to the service. On the other hand, if an Ethereum account address is on the blacklist, the user account associated with the address is forbidden from accessing the datastore.

– Data Manipulation via GUI

  Users with granted access, defined and governed by the access policy of the datastore, can create, edit, and revoke data entries in the GUI of the WebUI. The interface shows the state of the data, namely cached and mined.

– Data Manipulation via API

  Users with granted access, defined and governed by the access policy of the datastore, can create, edit, and revoke data entries via an exposed API endpoint.

– Data Query via GUI

  Users with granted access, defined and governed by the access policy of the datastore, can query the data by data position and column filters in the GUI of the WebUI. The interface shows the state of the data, namely cached and mined.

– Data Query via API

  Users with granted access, defined and governed by the access policy of the datastore, can query the data by data position and column filters via an exposed API endpoint.

4.3 System Features Design

4.3.1 User Registration

Users interacting with the system need to apply for an account to represent and authenticate the identity. The platform does not differentiate users from different roles, as all users are entitled to use all the functionality the platform provides. For the registration, the user needs to provide two parameters, namely a username and a password. The parameter username serves solely for authentication and is by no means used as a representation code of a user. For that purpose, the system generates a unique randomized user ID to represent a user across the system.

Each user account is associated with an Ethereum externally owned account(EOA), which consists of an account address and a private key. The account address, which takes the form of a long hexadecimal address, is essentially the matching public key of the private key as part of asymmetric key cryptography. [6] Although this public and private key pair is not stored directly on the Ethereum blockchain instances that the user created, similar in the Ethereum world, it is used for validating and verifying user interaction in the prototype system.

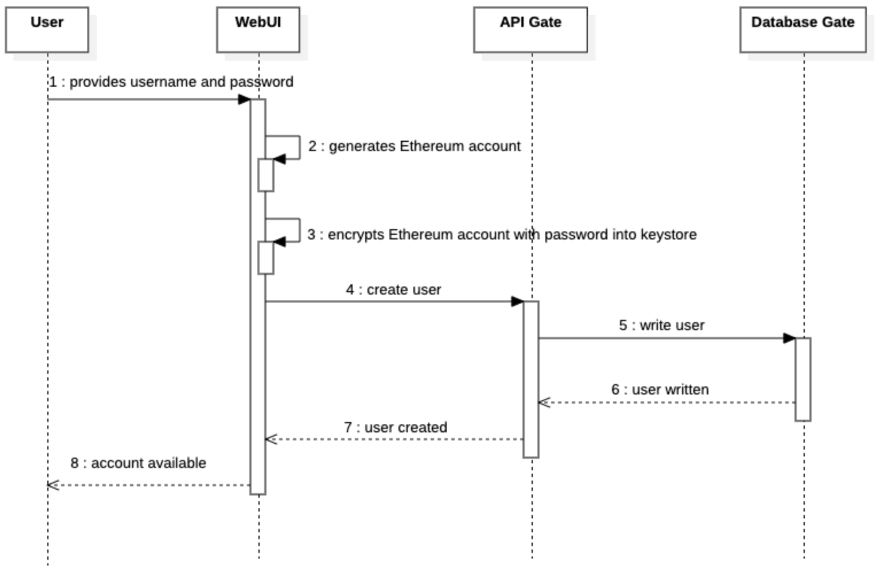

User
WebUI
API Gate
Database Gate
1 : provides username and password
2 : generates Ethereum account
3 : encrypts Ethereum account with password into keystore
4 : create user
5 : write user
6 : user written
7 : user created
8 : account available

Figure 4: Sequence Diagram - User Registration

Figure 4 depicts the process of user registration. At the beginning, the user provides a username and a password in the WebUI interface and requests to create an account in the system. To prepare for the registration, the WebUI first generates an Ethereum externally owned account(EOA) with the Web3.js framework function web3.eth.accounts.create() and then encrypts the generated private key with the password provided by the user by using the function web3.eth.account.encrypt(). The output of the encrypt function, which is called a keystore, is a JSON format object with the following structure.

```
{
  "version": 3,
  "id": "04e9bcbb-96fa-497b-94d1-14df4cd20af6",
  "address": "2c7536e3605d9c16a7a3d7b1898e529396a65c23
     ",
  "crypto": {
    "ciphertext": "a1c25da3ec...e906e6df24d251",
    "cipherparams": { "iv": "2885
       df2b63f7ef247d753c82fa20038a" },
    "cipher: "aes-128-ctr",
    "kdf: "scrypt",
    "kdfparams": {
      "dklen": 32,
      "salt": "4531b3c17...4807b2d216d022318ceee50be10
         ",
      "n": 262144,
      "r": 8,
      "p": 1
    },
    "mac": "b8b010fff3...7f7a9f1bd4e82a5dd35468fc7f6"
  }
}
```

WebUI then sends a request to the API Gate alone with the username, account address, and the keystore to create the user account. The API Gate processes the request by calling Database Gate to write the user account information and returning the generated user ID.

At this point, the user account is stored in the database as a document in the user collection, and has the following structure.

```
{
  "username": "",
  "accountAddr": "",
  "keystore": "",
  "projects": [],
  "chains": [],
  "tasks": [],
  "services": [],
  "datastores": []
}
```

Here, properties projects, chains, tasks, services, and datastores are array object for storing the related system entities of the user.

In the end, the WebUI informs the user that the account is created and ready for use.

4.3.2 User Login

After the registration, the user can then log into the platform with the username and password, Figure 5 shows the login process.

First, the user provides the username and password used in the registration process via the WebUI interface. The WebUI requests the user's account credentials from the API Gate by the username. The API Gate handles the request by reading the user's encrypted keystore from the Database Gate out, and requesting the Database Gate to refresh the token of this particular user. Here the token is stored in the Redis database under the key "auth:token:userId" where the "userID" is replaced by the user ID of the user, this token is used later in the process to verify the identity of the user.

After the WebUI receives the encrypted keystore, and the token from the API Gate, it decrypts the keystore using the password provided by the user. It is done by using the Web3.js framework function web3.eth.accounts.decrypt(). This function decrypts a keystore in JSON format and creates an Ethereum EOA that has the following format.

```
{
  "address": "0
     x2c7536E3605D9C16a7a3D7b1898e529396a65c23",
  "privateKey": "0x4c088...92ae468d01a3f362318",
  "signTransaction": function(tx){...},
  "sign": function(data){...},
  "encrypt": function(password){...}
}
```

At this point, WebUI acquired the public account address and the private key of the user. It can then sign the token by using web3.eth.account.sign() function. This Web3.js function signs arbitrary data with a private key and returns a signature object in following JSON structure.

```
{
  "message": "token string",
  "messageHash": "0x1da44b586eb07...056
     e7f47fbc6e58d86871655",
  "v": "0x1c",
  "r": "0xb91467e...aba02900b8979d43fe208a4a4f339f5fd
     ",
  "s": "0x6007e74cd8...8a5f5d4300f8e1a029",
  "signature": "0xb91467e570a6466aa...5
     d18a5f5d4300f8e1a0291c"
}
```

The signature object is then sent to API Gate along with the username again to the API Gate for user identity verification. API Gate infers the signing account address and the original token from the signature string by Web3.js function web3.eth.account.recover(). By comparing both the inferred address and the token with the data from Database Gate, the API gate can decide who is requesting login and verify the identity of the requestor. Consequently, the API Gate responds to the WebUI, who then informs the user with the result. By the end of a successful login, the user acquires the user ID, account address, and private key.

4.3.3 Tasks

As we discussed in Section 3.6.6, many of the time-consuming system activities are abstract into system tasks and are managed and executed by task queue and task agent. Many system components are involved from the creation of the task to the execution finish of the task. We first take a look at the task creation.

Figure 6 explains the process of the task creation. Usually, a task is initiated by the user in the WebUI interface, for instance, a chain instance deployment task. The WebUI collects the task parameter and sends task creation request to the API Gate along with the user's ID, the type and name of the task. There are six types of tasks that are defined in the prototype.

– CHAIN_INSTANCE_DEPLOY

  Deploy a user chain instance according to the configuration, required parameters: chain ID and user ID.

– CHAIN_INSTANCE_DELETE

  Delete a chain instance but keeps the chain configuration, required parameter: chain ID.

– CHAIN_DELETE

  Delete a chain entirely, including the chain instance and configuration. Remove it from the user's chain list, required parameters: chain ID, user ID.

– DATASTORE_DEPLOY

  Deploy a user datastore according to the configuration, required parameters: chain ID, user ID, name, type, columns(column name and data type).

– PROJECT_COMPILE

  Compile all smart contracts in a project, required parameters: project ID, user ID.

– PROJECT_ARTIFACT_DEPLOY

  Deploy a compiled smart contract to a running user chain instance, required parameters: project ID, artifact name, chain ID, contract constructor arguments, gas(optional), gas price(optional).

Then, the API Gate calls Database Gate API to store the task information into the task collection in the following document structure.

```
{
  "userId": "user id",
  "type": "type of the task",
  "name": "name of the task",
  "params": {},
  "status": "Created",
  logs: [
    { "timestamp": 139374736, type: 'INFO', message: '
        Task created.' }
  ]
}
```

The task ID, which is the document ID generated from MongoDB, is then published to the Redis task queue. The API Gate then updates status and the logs of the task before it sends back the task ID as the response to the WebUI request. The user can then monitor the running process of the task in the WebUI with the task ID.

Once the task is published to the task queue, one of the subscribed task agents gets the task and executes it. The activity of the task agent is described in Figure 7.

When started, an agent first sleeps a random period of time between 1 and 10 seconds before asking the queue for tasks. This prevents simultaneous task retrieval and request overflow from all the agents as the retrieval is implemented as an API call to the the Internal Task Queue API on Database Gate. The agent then executes the retrieved task based on the type and parameter of the task, publishing logs to the task along the way, and updating the task status subsequently. When there is no task retrieved or the agent finishes the task, it goes back to the random sleep state and keeps on the activity in loops

4.3.4 User Chain Deployment

The user chain is the building blocks for system features like user service and user datastore. The prototype supports the Ethereum private chain with Clique protocol. For exploration purposes, the number of chains a user can create is not restricted in the system.

To create a user chain, the user must provide a chain name. The name of the chain must be unique and never been used across the entire system. The reason being that each chain is deployed into a separate namespace in the Kubernetes cluster in order to isolate chain resources. It also makes the chain removal

Parameter	Description
type	The consensus protocol of the chain
sealerNodeCount	The number of sealer nodes in the chain network.
trasactionNodeCount	The number of transaction nodes (non-sealer) in the chain network.
gasPrice	Minimum gas price for mining a transaction.
gasLimit	Target gas ceiling for mined blocks.
gasTarget	Target gas floor for mined blocks.
txpoolPriceLimit	Minimum gas price limit to enforce for acceptance into the pool.

Table 1: User Chain Configuration Parameters

straightforward and less error-prone. The system stores user-created chain in the database in the chain collection with the following structure.

```
{
  "userId": "5eb51340e552560029c3b304",
  "name": "test-chain",
  "createdOn": 1588925263213,
  "status": "Deployed",
  "config": {
        type: "clique",
        sealerNodeCount: 1,
        transactionNodeCount: 2
        gasPrice: 0,
        gasLimit: 900000000,
        gasTarget: 800000000,
        txpoolPriceLimit: 0
  },
  "deployment": {
        "namespace": "test-chain",
        "createdOn": 1588925320628
  },
  "contracts": [],
  "services": [],
  "datastores": []
}
```

Under field "config" sits the chain configuration parameters provided by the user. Table 1 describes each parameter in detail.

The chain is only ready for deployment once the user finishes the configuration. The deployment task is handled by a task agent in a sequence depicted by Figure 8.

When the agent retrieves a chain instance deployment task, it first ready the chain configuration from the Database Gate. It then creates the namespace in the Kubernetes cluster, and all the related chain instance Kubernetes objects in later steps are then placed in this particular namespace.

To deploy a complete private chain network, the task agent needs to create the following resources objects.

– Sealer account for each sealer node

  This is the account which the sealer node used to perform mining. It is stored inside the namespace as a Secret object, and is available for access later when the sealer node starts up.

– Network configuration

  The network configuration is a JSON object that consists of a randomly chosen network ID, which is the network ID used for the peer-to-peer chain network, and the genesis block generated based on the chain configuration. This network configuration is stored inside the namespace as a ConfigMap object, and is available for access later when the chain nodes startup.

– Storage for each sealer node

  The storage of each sealer node is provisioned by a persistent volume claim(PVC). The persistent volume claim is a Kubernetes Object that abstracts storage from pods. The data stored in a PVC persists after pods attached to it are destroyed. In the case of a signer node crash, the chain data is not lost, and the operation keeps on once the restarted node reattach itself to the PVC. The system provisions no persistent storage for transaction nodes. When a transaction node fails, it restarts and re-sync the chain data with other nodes.

The agent continues the task once the resources are available. To deploy a chain network, the agent first deploys a bootnode for coordinate node discovery in the network. Then, according to the setup, the agent deploys all the signer nodes and transaction nodes.

Depends on whether the chain is configured to be exposed, after all nodes are online, the agent either patches the cluster ingress to allow external access to the chain or creates a master account for making transactions in the network.

4.3.5 User Project

In the platform, users develop and organize smart contracts in a user project. When the user creates a project, it is stored in the database in the following document structure.

```
{
  "userId": "5eb51340e552560029c3b304",
  "name": "test-project",
  "createdOn": 1588939932350,
  "files": {
        "Person.sol": "pragma solidity >=0.5.0;...."
  },
  "compilerVersion": "v0.5.0+commit.1d4f565a",
  "compileErrors": [],
  "artifacts": {}
 }
```

The contract source code written by the user is stored under the "files" property after the file name of the contract. Once all the source codes are written, the user can compile the source codes into project artifacts. The compile job is handle by the task agent with the project compile task. The process is described in Figure 9.

When a task agent picks up a project compile task from the task queue, the agent retrieves the source files and the selected compiler version of the project from the Database Gate. The source files and the compiler version are then sent to the Transaction Gate for a contract compile request.

The Transaction Gate loads the selected compiler, compiles the source codes into artifacts, and send the artifacts along with compile errors, if there is any, as the response to the agent. At last, the agent calls the Database Gate to update the project's artifacts and compiling errors. The artifacts, sorted after contract name, are stored under the "artifacts" properties of the project document in the following structure.

```
{
  "artifacts": {
        "Person": {
          "abi": [{...}, {...}, {...}],
          "devDoc": {...},
          "evm": {
                "assembly": "...",
                "bytecode": {...},
                "gasEstimates": {...},
                        ...
        },
                ...
  }
}
```

The ABI and bytecode are used later for contract deployment. The compile errors are stored under the property "compileErrors" in the following structure.

```
"compileErrors": [
  {
        "component": "general",
        "formattedMessage": "Person.sol:19:5:
            ParserError: Expected ';'...",
        "message": "Expected ';' but got '}'",
        "severity": "error",
        "sourceLocation": {
          "end": 316,
          "file": "Person.sol",
          "start": 315
        },
        "type": "ParserError"
  }
]
```

These compiler errors are then visiualized in the code editor for the user to correct.

4.3.6 Contract Deployment

Task agents deploy compiled project artifacts into contracts on chain instances. Figure 10 shows the deploy task procedure.

The task agent begins with retrieving the selected artifact, more precisely, the ABI and the bytecode of the artifact, from the Database Gate. It then reads the information of the target deploy chain instance to decide where to store the contract metadata later. Next, the agent requests the Transaction Gate to deploy the artifact to the target chain with the deployment arguments provided by the user.

For contract deployment, the Transaction Gate requires the master account of the chain instance to sign the transaction. After the contract deployment transaction is created and signed, Transaction Gate sends the signed transaction to the target chain network, and when the transaction is mined, it returns the transaction receipt.

Finally, the agent creates a contract document and attach it to the chain document and updates the database. The structure of the contract document is shown as follows.

```
{
    "chainId": "5eb51340e5525600w2c3b304",
    "name": "Person",
    "deployedOn": 1588939932350,
    "compilerVersion": "v0.5.0+commit.1d4f565a",
    "abi": [{...}, {...}, {...}],
    "receipt": {
                "status": true,
                "transactionHash": "0x9fc7...5836d8b",
                "transactionIndex": 0,
                "blockHash": "0xef95f2f1...
                   cbea9a2c4e133e34b46",
                "blockNumber": 3,
                "contractAddress": "0
                   x11f4d0A3c12e86B4b5F39B213F7E19D048276DAe
                   ",
                "cumulativeGasUsed": 314159,
                "gasUsed": 30234,
                "logs": [{
                // logs as returned by getPastLogs,
                   etc.
        }, ...]
        }
}
```

4.3.7 User Service

The user service feature is the micro-service solution of the platform. Generally speaking, a user service uses a deployed contract as the computing backend, and gives users straightforward access to the contract's function.

By design philosophy, a micro-service focuses on solving one single problem of a system. It is lightweight, easy to be replaced, and not mean to interference but work with other micro-services to create solutions for a complicated system. Ethereum smart contract shares some common characteristics. Due to the high computational cost of the blockchain mining process and the limit of gas consumption, the best practice of designing a smart contract is to keep the complexity low. Once a smart contract is deployed, it is not mean to be updated, as an alteration of the historical chain data is not possible. Furthermore, contracts can interact with other contracts on the same chain and work together coordinately.

For these reasons, the idea of using smart contract as the backend for a micro-service is attempting and promising, and is the core concept of the prototype.

The prerequisite for creating a user service is a contract deployed on the user chain instance. To create a user service based on a deployed contract, the user provides the name of the service and the contract methods to be invoked. The user service is stored in the "service" document collection in the database with the following structure.

```
{
        "serviceID": "5eb6add7e552560029c3b311",
        "name": "person_get_name",
        "createdOn": 1589030359277,
        "type": "common",
        "config": {
                "chainID": "5eb5134fe552560029c3b305",
                "contractID": "5
                    eb562dae552560029c3b310",
                "functionName": "getName"
        }
}
```

After creation, the user service is exposed via the API Gate under the API point "/api/service/:serviceId". When the service is called, the system invokes the backend contract call in the process describted in Figure 11.

The API Gate first route the request to the Transaction Gate, where all the contract call transactions are handled. The Transaction Gate retrieves the target contract data, including contract address and ABI, as well as the data of the chain where the contract is deployed. Based on the ABI of the contract and method to be invoked, the Transaction Gate exams whether or not the transaction requires to be signed. This is determined by the nature of the contract method. In a word, transactions that modify the state of the blockchain is required to be signed. As a result, if a contract method does not perform a purely "read-only" operation, the call must be signed.

At deployment, each chain that is not exposed has a master account bound to it. The master account is used for signing all sign required transactions in the chain. This approach moves the signing process from the client-side to the server-side, and provides a centralized control to the transaction request on the chain. Additionally, it allows arbitrary user to interact with the service, without that the user's Ethereum account being stored locally on the chain nodes. However, this leads to the loss of the initiator identity information because the same account signs all the transactions. The drawback can be encountered by the platform user identity verification mechanism described in Section 4.3.11.

Finally, after the transaction finishes, the Transaction Gate responses with either the transaction receipt or the call result, which is then returned to the caller by the API Gate.

4.3.8 User Datastore

In a typical blockchain environment, bulk operations like querying and filtering on discrete data are tricky. On the one hand, depending on the design of the smart contract, indexing and querying a large amount of structured data can pose severe challenges in both the algorithm design and storage capacity. Sophisticated indexing or querying algorithm usually bears a non-constant computational footprint, can cause transaction failure due to probable exceed in gas limit. Although the gas limit of a blockchain network can be tweaked to meet the demands, the computing overhead created by the blockchain mining process is not neglectable. On the other hand, chain data are replicated and stored on each chain node. Supplementary data generated for indexing causes storage waste on the chain node and dramatically increases the capacity requirement for chain nodes. Additionally, because that transaction mining is a time-consuming process, operations requested by users take a substantial amount of waiting time.

User Datastore is a special type of pre-defined user service that implements a data storage solution on the blockchain instance. Like a standard user service, user datastores are powered by smart contracts behind the scene. Each datastore has its table-like schema, which describes what and how the data are stored. Data operations from users are cached for data operation performance improvement. The Chain Agent component, which is deployed beside the transaction node, monitors the datastore activity via the chain event and updates the cache state when the data operation is finalized, i.e., the transaction is mined. In other words, the user datastore stores data in smart contract with a table-like structure and keeps off-chain replicas of the on-chain data to offers enhanced functionality.

Initially, the user configures the name, the datastore type, and the desired data schema. The data schema describes the columns and the type of data in each column. To deploy a user datastore, a task agent executes the "DATASTORE_DEPLOY" task, the procedure of the task is depicted in Figure 12.

After retrieving the namespace and the configuration of the target deploy chain, the Task Agent first requests the Database Gate to create a datastore document in the "datastore" collection. The datastore document is structured as follows.

```
{
        "name": "test-datastore",
        "type": "Datastore",
        "createdOn": 1588925345,
        "userId": "5eb51340e552560029c3b304",
        "chainId": "5eb5134fe552560029c3b305",
        "contract": "5eb513a3e552560029c3b309",
        "monitoring": true,
        "columns": {
                "personName": {
                        "columnIndex": 1,
                        "columnName": "personName",
                        "columnDataType": "string"
                }
        },
        "currentRowIndex": 0
 }
```

The Task Agent then compiles the chosen types of datastore contract and deploy it via the contract deployment API from the Transaction Gate. The deployed contract is then stored in the database like we discussed in Section 4.3.6, and the contract ID is written to the "contract" property of the datastore document to associate the contract and the datastore together. Afterward, the Task Agent publishes the datastore and waiting for the Chain Agent to start monitoring the chain events. As soon as the monitoring status is confirmed, the Task Agent sends transactions to the Transaction Gate to create all columns defined in the datastore schema. Eventually, when all the transaction is mined, the Task Agent binds the deployed user datastore to its creator and target deployment chain and creates its access policy.

4.3.9 User Datastore Data Operations

In a datastore contract, data entries are represented and organized by a data rows. Each data row is a mapping of a unique numeric row index to a data row struct. Inside each data row struct, the actual data is then stored in a mapping to the corresponding column index. Because Ethereum uses a "zero" state to represent non-initialized values, each data row struct contains boolean variables to indicate the existing state of the data. The contract implements an index-based interface for data operations like create, update and revoke, when invoked, the contract emits chain event that encapsulates the alteration to the on-chain data, including indexes, value, and block time.

An abstracted data operation process is depicted in Figure 13. When the API Gate receives a data operation request (create, write or revoke), it first caches the operation to the off-chain data at the corresponding data index. After the state of the off-chain data is altered to the desired state, the API Gate requests the Transaction Gate to execute all the contract method call transactions. The Transaction Gate sends the transactions into the chain network, and when the transaction is mined, the contract emits the chain events. The Chain Agent picks up the events and updates the cached data operation to finalize the state of the off-chain data.

The off-chain data cache organizes each data row as a document stored under the document collection named by the ID of the datastore. The following JSON object shows a typical data row cache structure.

```
{
  "indexID": "5eb513d4e552560029c3b30b",
  "rowIndex" : 0,
  "columns": {
    "personName": {
      "columnIndex": 0,
          "columnName": "personName",
          "columnDataType": "string",
          "history": [{
        "value": "John",
        "actor": "0
           x8f8c3ea438376A248DaA347850f841790eFA48a0
           ",
        t_cached": 1588925396,
        "t_bc": 1588925398,
          }],
      }
  },
  "revoked": {
          "actor": "0
              x8f8c3ea438376A248DaA347850f841790eFA48a0",
    "t_cached": 1588925396,
    "t_bc": 1588925398,
  }
}
```

All the modifications to the data are cached in the history property under each column. The "t_cached" and "t_bc" property are the cache operation timestamp and the transaction mining timestamp. The non-empty state of the "revoked" property represent that the data row has been revoked by a user. Once a data row is revoked, no further alteration can be done to the data row.

4.3.10 User Database Data Query

Usually, in Etherium smart contracts, due to the high cost of loop operation and the lack of data aggregation functionality, discrete data query in large quantity is hard to implement. By keeping the off-chain data replicas in the MongoDB database, we can achieve significant performance enhancement and flexibility for data query operations. The User Datastore API implemented in API Gate offers two types of data query mechanism in the system.

Position-based query allows users to query data based on the row index, users request with a starting row index, and the number of rows to retrieve.

Filter-based query allows users to define a query filter in the following format,

```
[columnName]: [filter value or filter operators]
```

where "columnName" is the name of the column the filter to be applied. The filter can either be a specific value, or a valid comparison operator supported by MongoDB(listed in Table 2). Filters can be aggregated by logical operators supported by MongoDB as well(listed in Table 3).

In both position-based and filter-based query, users retrieve the whole data row cache as the query result. In the case of the filter-based mechanism, data rows with historical values that fill the filter condition are included in the query result as well.

4.3.11 API Access Control

In the platform, the majority of the API access points are guarded by one of the following access control groups.

Operator	Description	Example
$gt	greater than	{"age": {"$gt": 18}}
$gte	greater than or equal	{"age": {"$gte": 18}}
$lt	less than	{"age": {"$lt": 18}}
$lte	less than or equal	{"age": {"$lte": 18}}
$in	matches values in an array	{"bloodType": {"$in": ["A", "B"]}}
$nin	matches values not in an array	{"bloodType": {"$nin": ["A", "B"]}}
$ne	not equal	{"gender": {"ne": "male"}}

Table 2: User Datastore Query Filter Comparsion Operators

Operator	Description	Format
$and	logical AND	{$and: [{f1}, {f2}, ...]}
$not	logical NOT	{$not: {f}}
$nor	logical NOR	{$nor: [{f1}, {f2}, ...]}
$or	logical OR	{$or: [{f1}, {f2}, ...]}

Table 3: User Datastore Query Filter Logical Operators

– Ownership Control Group

  Resources created by users, for instance, user chains, projects, services, datastores, and contracts, have the access and modify rights bound with the creator of the resource. In other words, only the owner of a particular resource can retrieve, adjust, and eliminate the resource.

– Security Rule Control Group

  User interactions with a published user service and user database are controlled by a set of security rules. Possible security rules are listed in Table 4.

The whitelist rules, when populated with the Ethereum EOAs of platform users, allow the governed interactions only from users on the list. The blacklist rule, on the opposite, blocks access from the users on the list.

Security rules are stored separately from the parent resource in the "access" document collection in the database in the following structure,

```
{
    "parentId": "",
```

Security Rule	Applicable Resource	Governed Interaction
read-whitelist	service and datastore	call, read
write-whitelist	datastore	write
blacklist	service and datastore	call, read, write

Table 4: Access Control Security Rules

```
    "readWhiteList": [],
    "writeWhiteList": [],
    "blackList": []
  }
```

,where the "parentId" property is the ID of the parent resource. As a part of a platform resource itself, it falls under the ownership policy control group, only the creator of a particular service or datastore can view, define and update the security rules.

The API Gate implements the platform access control, as it is the only entry point of all external user interactions. To cope with the access control, users need to submit to the identity verification protocol when communicating with the API Gate. In each API Gate request, the user must include "Token-Timestamp" and "Token-Signature" fields in the HTTP request header. The "Token-Timestamp" is simply the time, in the format of "YYYY-MM-DDTHH:MM:SSZ", when the request is generated at the client-side. The "Token-Signature" is the signature generated from the user by signing the "Token-Timestamp" with the private key of the Ethereum EOA.

The API Gate then verifies the identity of the user and authorizes the access in the process depicted in Figure 14.

First, the API Gate recovers the signer's account address with the timestamp and signature from the request headers and validate the timestamp against the current system time. A valid timestamp must not exceed the maximum tolerate time, which is defined as five minutes in the prototype. When the signer account is recovered successfully, and the timestamp of the request is not expired, the API Gate authorizes the request based on the access control group it belongs to. In the case of ownership control group, access is granted if the signer account is the same with the owner account. For security rules control group, API Gate reads the security rules and check the signer account against all the security rules.

4.4 Backend RESTful API

The RESTful backend API implemented by the API Gate can be divided by the system feature groups. Table 5 to table 12 describe each API endpoint in feature groups user, user chain, user project, user service, user datastore, contract, task, and external call of user service and user datastore. Details including the resource URI, HTTP call method, access control group, required parameters, and the response of each API endpoint. The API HTTP are routed by the cluster ingress uniformly by the "/api" pre-fix route. An example call to get a user's own profile takes the URL in the form of "/api/user/:userId". Requests with required parameters must use the "application/json" mime type and include the required parameters in the JSON body. Responses also use JSON as the returned data format.

Table 5: RESTful API - User

API Endpoint	/user/
Method	POST
Access Control	None
Description	Create a new user account.
Required Parameter	"username": The username of the account. "accountAddr": The Ethereum EOA address of the user. "keystore": The encrypted user keystore.
Response	"userId": User ID of the user created.
API Endpoint	/user/credential
Method	POST
Access Control	None
Description	Retrieve a user's login credential by username.
Required Parameter	"username": The username of the account.
Response	"keystore": The encrypted user keystore. "token": Login token for the user to sign with private key.
API Endpoint	/user/login
Method	POST
Access Control	None
Description	Login a user by the user's identity authentication.
Required Parameter	"username": The username of the account. "tokenSignature": The token from the /user/credential request signed by the user's private key.
Response	"userId": User ID of the user.
API Endpoint	/user/:userId
Method	GET
Access Control	None
Description	Retrieve a user's profile without the "accountAddr" and the "keystore".
Required Parameter	None.
Response	The profile of the user (see Section 4.3.1).

Table 6: RESTful API - User Chain

API Endpoint	/user/:userId/chain
Method	POST
Access Control	Ownership
Description	Create a new user chain.
Required Parameter	"name": The name of the chain.
Response	"chainId": Chain ID of the chain created.
API Endpoint	/user/:userId/chain/:chainId

Method	PUT
Access Control	Ownership
Description	Update the configuration of the chain.
Required Parameter	"config": The configuration of the chain(see Section 4.3.4) .
Response	Empty
API Endpoint	/user/:userId/chain/
Method	GET
Access Control	Ownership
Description	Retrieve the user's chains as a list.
Required Parameter	None
Response	A list of chain ID owned by the user.
API Endpoint	/user/:userId/chain/:chainId
Method	DELETE
Access Control	Ownership
Description	Delete a chain in the database, remove it from the user's chain list.
Required Parameter	None.
Response	"taskId": The ID of the task which performs the chain delete.
API Endpoint	/user/:userId/chain/:chainId/deployment
Method	POST
Access Control	Ownership
Description	Deploy a chain instance after its configuration.
Required Parameter	None.
Response	"taskId": The ID of the task which performs the chain instance deployment.
API Endpoint	/user/:userId/chain/:chainId/deployment
Method	DELETE
Access Control	Ownership
Description	Delete a running chain instance.
Required Parameter	None.
Response	"taskId": The ID of the task which performs the chain instance delete.
API Endpoint	/user/:userId/chain/:chainId/deployment/ artifact_deploy
Method	POST
Access Control	Ownership
Description	Deploy an artifact from a user project to the running chain instance.

Required Parameter	"projectId": The ID of the project which contains the artifact. "artifactName": The name of the artifact to be deployed. "args": The deployment arguments required for the artifact deployment.
Response	"taskId": The ID of the task which performs the artifact deployment.

Table 7: RESTful API - User Project

API Endpoint	/user/:userId/project
Method	POST
Access Control	Ownership
Description	Create a new user project.
Required Parameter	"name": The name of the project.
Response	"projectId": Project ID of the project created.
API Endpoint	/user/:userId/project
Method	GET
Access Control	Ownership
Description	Retrieve a user's project list.
Required Parameter	None
Response	A list of IDs of the user owned project.
API Endpoint	/user/:userId/project/:projectId
Method	GET
Access Control	Ownership
Description	Retrieve the information of a project by project ID.
Required Parameter	None
Response	The project information(see Section 4.3.5).
API Endpoint	/user/:userId/project/:projectId/files
Method	PUT
Access Control	Ownership
Description	Update the files in project.
Required Parameter	"files": The files to be updated(see Section 4.3.5).
Response	200.OK
API Endpoint	/user/:userId/project/:projectId/compilerVersion
Method	PUT
Access Control	Ownership
Description	Update the files in project.
Required Parameter	"compilerVersion": The version of compiler to be used.
Response	200.OK
API Endpoint	/user/:userId/project/:projectId/compile
Method	PUT

Access Control	Ownership
Description	Update the files in project.
Required Parameter	None
Response	"taskId": The ID of the task which performs the artifacts compile.

Table 8: RESTful API - User Service

API Endpoint	/user/:userId/service
Method	POST
Access Control	Ownership
Description	Create a new user service.
Required Parameter	"name": The name of the service. "type": The type of the service, current support "common". "config": The configuration of the service(see Section 4.3.7).
Response	"serviceId": The ID of the service created.
API Endpoint	/user/:userId/service
Method	GET
Access Control	Ownership
Description	Retrieve the user's service list.
Required Parameter	None
Response	A list of service IDs owned by the user
API Endpoint	/user/:userId/service/:serviceId
Method	GET
Access Control	Ownership
Description	Retrieve the information of the service by its ID.
Required Parameter	None
Response	Service information(see 4.3.7)
API Endpoint	/user/:userId/service/:serviceId
Method	DELETE
Access Control	Ownership
Description	Delete a service, remove it from the user's service list.
Required Parameter	None
Response	200.OK
API Endpoint	/user/:userId/service/:serviceId/access
Method	GET
Access Control	Ownership
Description	Retrieve a service's security rules.
Required Parameter	None
Response	The security rules of the serivce(see Section 4.3.11).
API Endpoint	/user/:userId/service/:serviceId/access/blacklist

Method	PUT
Access Control	Ownership
Description	Append a EOA address to the blacklist of the service.
Required Parameter	"actor": The EOA address to be appended to the blacklist.
Response	200.OK
API Endpoint	/user/:userId/service/:serviceId/access/blacklist/:actor
Method	Delete
Access Control	Ownership
Description	Remove a EOA address to the blacklist of the service.
Required Parameter	None
Response	200.OK
API Endpoint	/user/:userId/service/:serviceId/access/whitelist
Method	PUT
Access Control	Ownership
Description	Append a EOA address to the whitelist of the service.
Required Parameter	"actor": The EOA address to be appended to the whitelist.
Response	200.OK
API Endpoint	/user/:userId/service/:serviceId/access/whitelist/:actor
Method	DELETE
Access Control	Ownership
Description	Remove a EOA address to the whitelist of the service.
Required Parameter	None
Response	200.OK

Table 9: RESTful API - User Datastore

API Endpoint	/user/:userId/chain/:chainId/deployment/datastore
Method	POST
Access Control	Ownership
Description	Deploy a new datastore to a running chain instance.
Required Parameter	"name": The name of the datastore. "type": The type of the datastore, current support "Datastore". "columns": The schema of the datastore(See Section 4.3.8).
Response	"taskId": The ID of the task that handles the datastore deployment.
API Endpoint	/user/:userId/chain/:chainId/deployment/datastore
Method	GET
Access Control	Ownership

Description	Retrieve the list of datastore that is deployed to a running chain instance.
Required Parameter	None
Response	A list of IDs of the datastores deployed on the running chain instance.
API Endpoint	/user/:userId/chain/:chainId/deployment/datastore/ :datastoreId
Method	GET
Access Control	Ownership
Description	Retrieve information of the datastore.
Required Parameter	None
Response	The information of the datastore(see Section 4.3.8).
API Endpoint	/user/:userId/chain/:chainId/deployment/datastore/ :datastoreId
Method	DELETE
Access Control	Ownership
Description	Delete a datastore and remove it from user's datastore list
Required Parameter	None
Response	200.OK
API Endpoint	/user/:userId/datastore/
Method	GET
Access Control	Ownership
Description	Retrieve a user's datastore list
Required Parameter	None
Response	A list of ID of user's datastore
API Endpoint	/user/:userId/datastore/:datastoreId
Method	GET
Access Control	Ownership
Description	Retrieve information of the datastore.
Required Parameter	None
Response	The information of the datastore(see Section 4.3.8).
API Endpoint	/user/:userId/datastore/:datastoreId/access
Method	GET
Access Control	Ownership
Description	Retrieve the security rules of a datastore
Required Parameter	None
Response	The security rules of a datastore(see Section 4.3.11).
API Endpoint	/user/:userId/datastore/:datastoreId/access/ readWhiteList
Method	PUT
Access Control	Ownership
Description	Append a EOA to the read whitelist of the datastore.

Required Parameter	"actor": the EOA address to be appended.
Response	200.OK
API Endpoint	/user/:userId/datastore/:datastoreId/access/readWhiteList/:actor
Method	DELETE
Access Control	Ownership
Description	Remove a EOA from the read whitelist of the datastore.
Required Parameter	"actor": the EOA address to be removed.
Response	200.OK
API Endpoint	/user/:userId/datastore/:datastoreId/access/writeWhiteList
Method	PUT
Access Control	Ownership
Description	Append a EOA to the write whitelist of the datastore.
Required Parameter	"actor": the EOA address to be appended.
Response	200.OK
API Endpoint	/user/:userId/datastore/:datastoreId/access/writeWhiteList/:actor
Method	DELETE
Access Control	Ownership
Description	Remove a EOA from the write whitelist of the datastore.
Required Parameter	"actor": the EOA address to be removed.
Response	200.OK
API Endpoint	/user/:userId/datastore/:datastoreId/access/blacklist
Method	PUT
Access Control	Ownership
Description	Append a EOA to the blacklist of the datastore.
Required Parameter	"actor": the EOA address to be appended.
Response	200.OK
API Endpoint	/user/:userId/datastore/:datastoreId/access/blacklist/:actor
Method	DELETE
Access Control	Ownership
Description	Remove a EOA from the blacklist of the datastore.
Required Parameter	"actor": the EOA address to be removed.
Response	200.OK

Table 10: RESTful API - Contract

API Endpoint	/user/:userId/chain/:chainId/deployment/contracts
Method	GET
Access Control	Ownership

Description	Get the list of contracts deployed on the running chain instance.
Required Parameter	None
Response	A list of contract IDs on deployed on the running chain instance.
API Endpoint	/user/:userId/chain/:chainId/deployment/contracts/ :contractId
Method	GET
Access Control	Ownership
Description	Get contract information by ID.
Required Parameter	None
Response	The information of the contract(see Section 4.3.6)
API Endpoint	/user/:userId/chain/:chainId/deployment/contracts/ :contractId
Method	DELETE
Access Control	Ownership
Description	Delete the contract, remove it from the chain's contract list.
Required Parameter	None
Response	200.OK

Table 11: RESTful API - Task

API Endpoint	/user/:userId/task
Method	GET
Access Control	Ownership
Description	Get the list of tasks belongs to the user.
Required Parameter	None
Response	A list of task IDs that belongs to the user.
API Endpoint	/user/:userId/task/:taskId
Method	GET
Access Control	Ownership
Description	Get the information of the task.
Required Parameter	None
Response	The information of the task(see Section 4.3.3).

Table 12: RESTful API - User Service and Datastore External Call

API Endpoint	/service/:serviceId
Method	POST
Access Control	Security Rules
Description	Call the service by service ID.

Required Parameter	"option": The call option, including callArgs, gas, gasPrice. "callArgs": One of the options, the arguments for the contract method invocation if any is defined by the contract.
Response	"type": The type of call result data. Depends on the contract method, either the value or a transaction receipt. "data": The call result.
API Endpoint	/datastore/:contractId/row
Method	POST
Access Control	Security Rules
Description	Write a new row of data into datastore.
Required Parameter	"row": The row of data to insert in the form of $columnName: $columnIndex, $columnDataType, $datavalue.
Response	"rowIndex": The row index of the new data row created.
API Endpoint	/datastore/:contractId/data/:rowIndex
Method	DELETE
Access Control	Security Rules
Description	Revoke a row of data.
Required Parameter	None
Response	200.OK
API Endpoint	/datastore/:contractId/data/:rowIndex/ :columnIndex/dataValue
Method	PUT
Access Control	Security Rules
Description	Update a data field by its row and column index.
Required Parameter	"columnName": Name of the column. "columnDataType": Data type of the column. "dataValue": The actual value of the data.
Response	200.OK
API Endpoint	/datastore/:datastoreId/read
Method	PUT
Access Control	Security Rules
Description	Query data from datastore by datastore ID.
Required Parameter	"rowIndexSkip": Used together with retrieveCount, starting point of the row index to be retrieved. "retrieveCount": Used together with rowIndexSkip, the number of rows to be retrieved. "filters": The filter for data query, can not be used with "rowIndexSkip" and "retrieveCount" together(see Section 4.3.10).
Response	The query result.

4.5 Frontend Web Application

The frontend WebUI is an HTML single page application written with the React.js framework and served on a Node.js HTTP server. The cluster ingress routes root HTTP requests to the frontend WebUI server.

The entry point of the website is the login page(Figure 15). First-time users need to sign up for an account before login. At the login, users can choose to select the "Remember me" option to keep the login session and avoid login for the next visit. The user ID and the associated EOA account(including the private key) are stored in the browser local storage. Due to security reason, users should log out after each session if the website is accessed in a public environment, so that the data in the local browser is cleared.

The WebUI consists of five function areas, and users navigate via the left main function menu.

Dashboard (Figure 16) provides an overview of the user's profile and the resources of the user on the platform, including the number of chains, projects, services, datastores, and the task history. It is the first page the user is redirected to after a successful login.

My Chain is where users create, manage, and delete their user chains. Figure 17 is the detailed information view of a user chain named "test-chain". The view shows the status of the chain instance, deployed contracts and datastores on the chain instance. Users can deploy chain instance, create datastore on the chain instance in this view.

My Project helps users to develop smart contracts in Solidity language. Users create and edit their smart contracts in the in-browser IDE editor (Figure 18), compile the smart contracts into artifacts, and then deploy the artifact on to a running chain instance here. The smart contracts are stored under "FILES", and successfully compiled artifacts can be viewed under "ARTIFACTS".

My service is the user services created by the user across all the chain instances. The graphical interface (Figure 19) helps users to call the service and check the result. The exposed service API can be found here.

My datastore displays user datastore data in a table (Figure 16). Here, users create new data rows, modify the value of a data field, and revoke unwanted data rows.

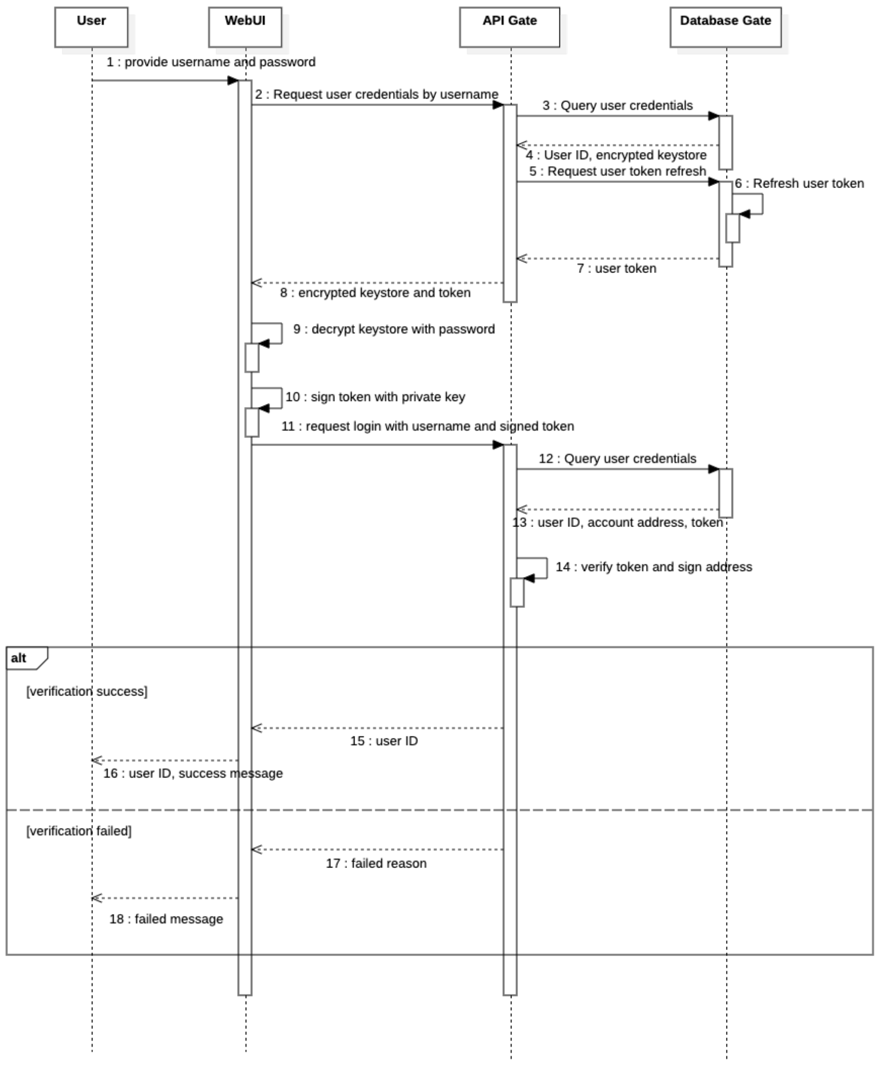

User
WebUI
API Gate
Database Gate
1 : provide username and password
2 : Request user credentials by username
3 : Query user credentials
4 : User ID, encrypted keystore
5 : Request user token refresh
6 : Refresh user token
7 : user token
8 : encrypted keystore and token
9 : decrypt keystore with password
10 : sign token with private key
11 : request login with username and signed token
12 : Query user credentials
13 : user ID, account address, token
14 : verify token and sign address
alt
[verification success]
15 : user ID
16 : user ID, success message
[verification failed]
17 : failed reason
18 : failed message

Figure 5: Sequence Diagram - User Login

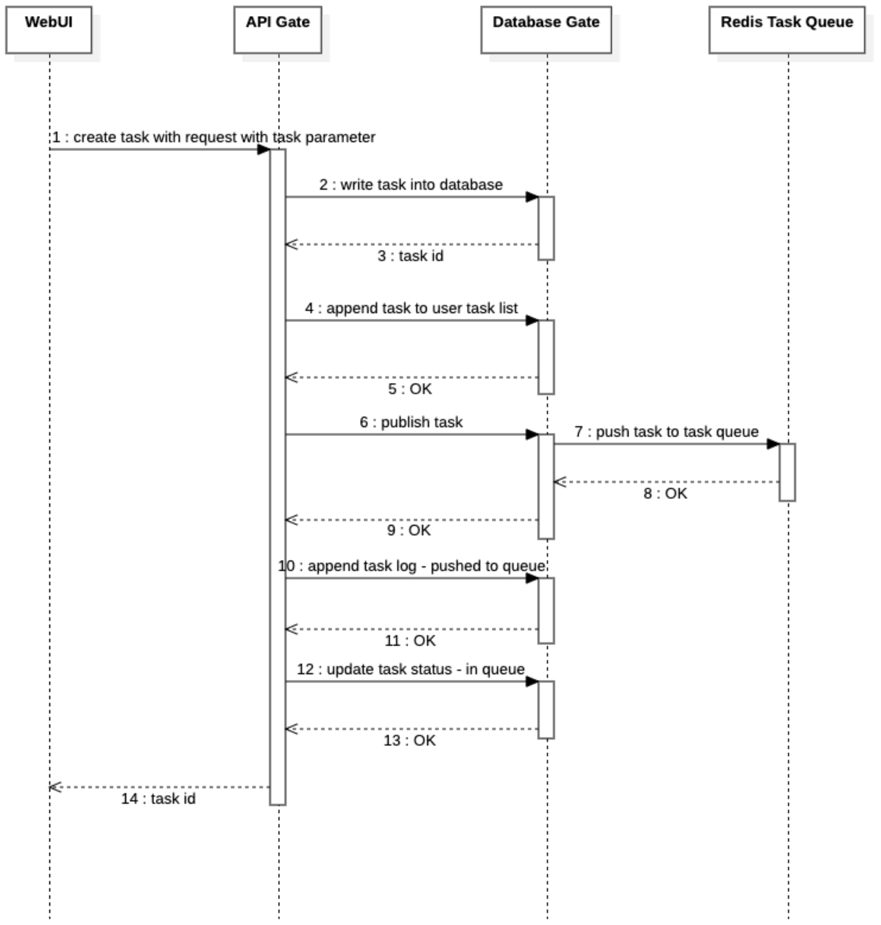

WebUI
API Gate
Database Gate
Redis Task Queue
1 : create task with request with task parameter
2 : write task into database
3 : task id
4 : append task to user task list
5 : OK
6 : publish task
7 : push task to task queue
8 : OK
9 : OK
10 : append task log - pushed to queue
11 : OK
12 : update task status - in queue
13 : OK
14 : task id

Figure 6: Sequence Diagram - Task Creation

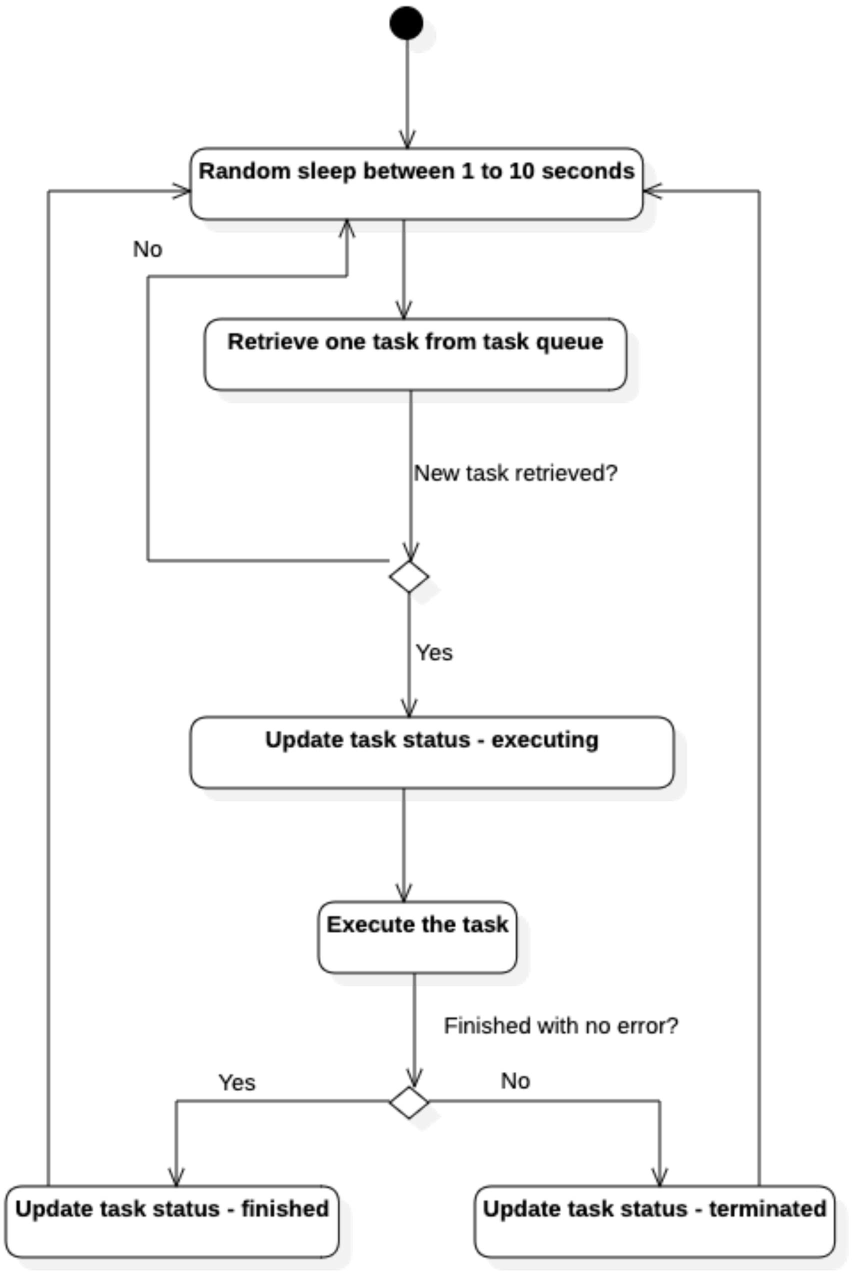

Random sleep between 1 to 10 seconds
No
Retrieve one task from task queue
New task retrieved?
Yes
Update task status - executing
Execute the task
Finished with no error?
Yes
No
Update task status - finished
Update task status - terminated

Figure 7: Activity Diagram: Task Agent

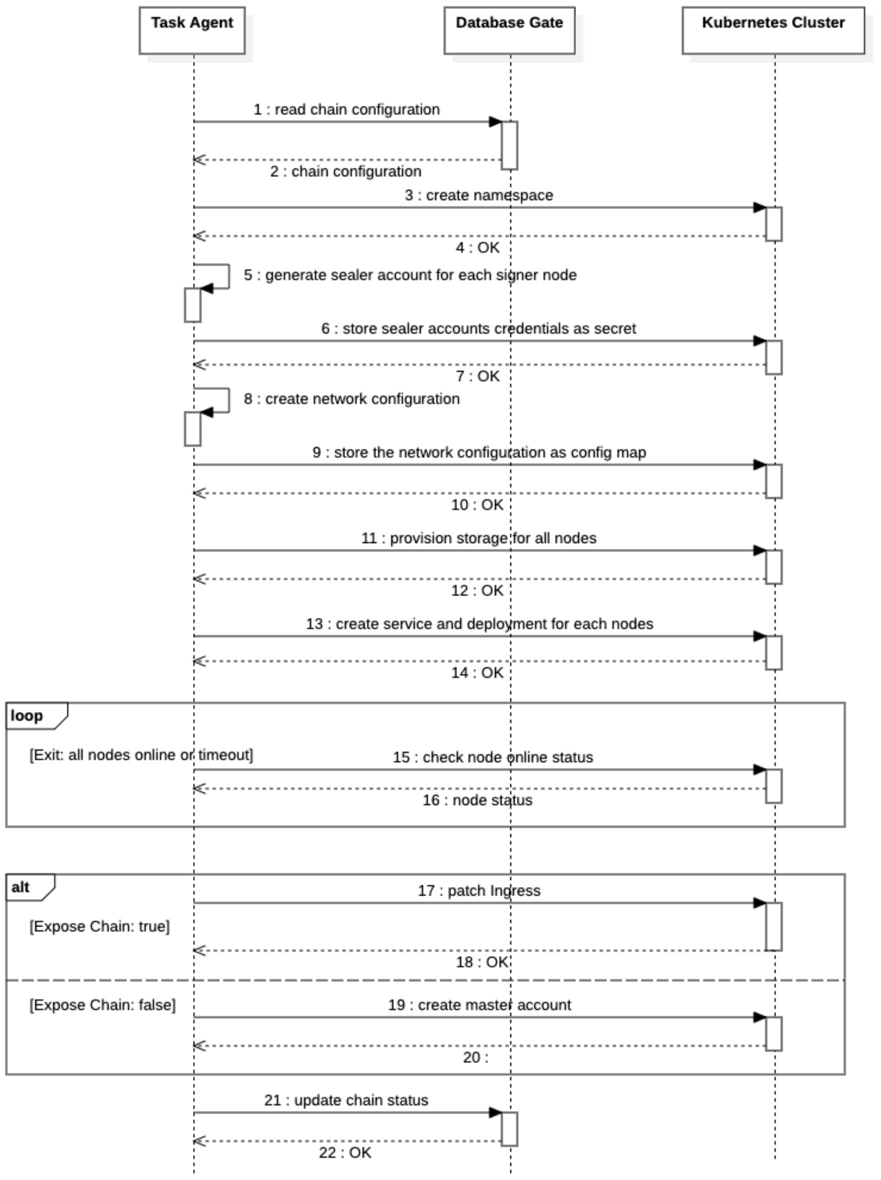

Task Agent
Database Gate
Kubernetes Cluster
1 : read chain configuration
2 : chain configuration
3 : create namespace
4 : OK
5 : generate sealer account for each signer node
6 : store sealer accounts credentials as secret
7 : OK
8 : create network configuration
9 : store the network configuration as config map
10 : OK
11 : provision storage for all nodes
12 : OK
13 : create service and deployment for each nodes
14 : OK
loop
[Exit: all nodes online or timeout]
15 : check node online status
16 : node status
alt
17 : patch Ingress
[Expose Chain: true]
18 : OK
[Expose Chain: false]
19 : create master account
20 :
21 : update chain status
22 : OK

Figure 8: Sequence Diagram - Chain Instance Deployment Task

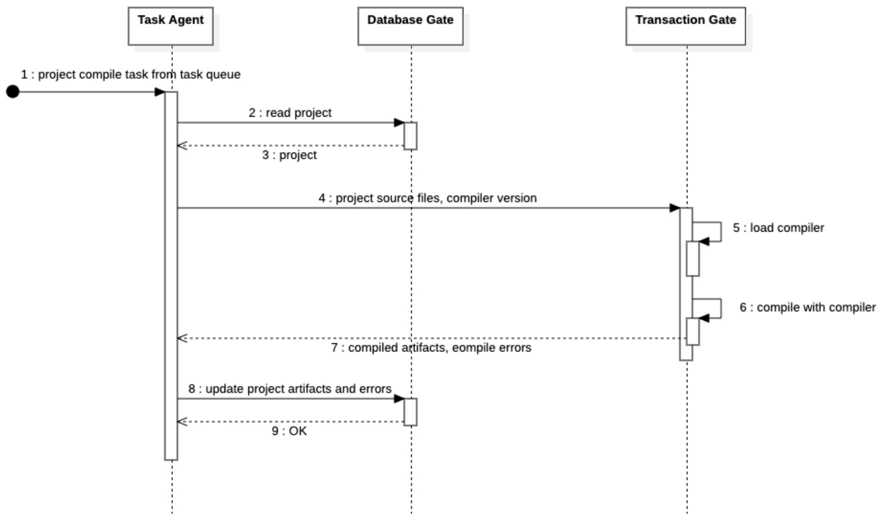

Task Agent
Database Gate
Transaction Gate
1 : project compile task from task queue
2 : read project
3 : project
4 : project source files, compiler version
5 : load compiler
6 : compile with compiler
7 : compiled artifacts, eompile errors
8 : update project artifacts and errors
9 : OK

Figure 9: Sequence Diagram - Project Compile Task

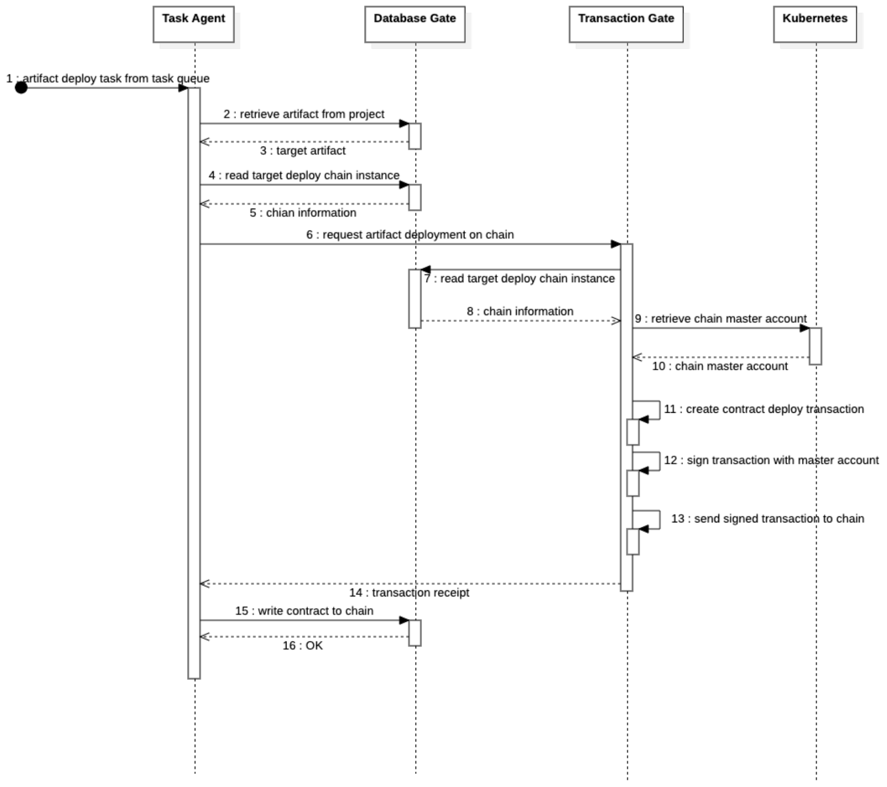

Task Agent
Database Gate
Transaction Gate
Kubernetes
1 : artifact deploy task from task queue
2 : retrieve artifact from project
3 : target artifact
4 : read target deploy chain instance
5 : chian information
6 : request artifact deployment on chain
7 : read target deploy chain instance
8 : chain information
9 : retrieve chain master account
10 : chain master account
11 : create contract deploy transaction
12 : sign transaction with master account
13 : send signed transaction to chain
14 : transaction receipt
15 : write contract to chain
16 : OK

Figure 10: Sequence Diagram - Project Artifact Deployment Task

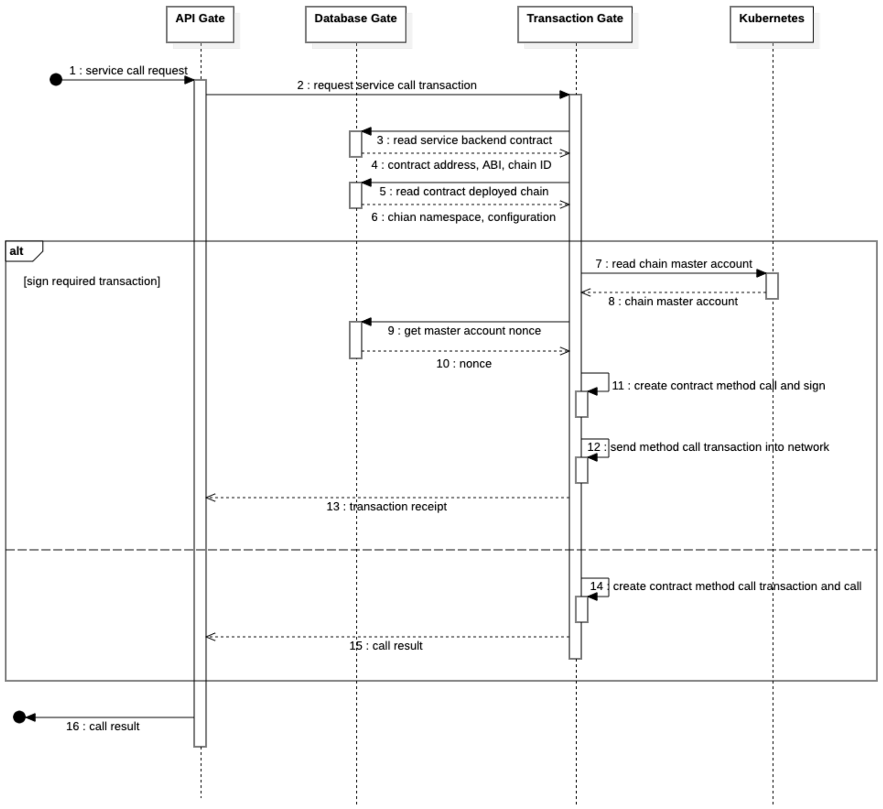

API Gate
Database Gate
Transaction Gate
Kubernetes
1 : service call request
2 : request service call transaction
3 : read service backend contract
4 : contract address, ABI, chain ID
5 : read contract deployed chain
6 : chian namespace, configuration
alt
[sign required transaction]
7 : read chain master account
8 : chain master account
9 : get master account nonce
10 : nonce
11 : create contract method call and sign
12 : send method call transaction into network
13 : transaction receipt
14 : create contract method call transaction and call
15 : call result
16 : call result

Figure 11: Sequence Diagram - User Service Call

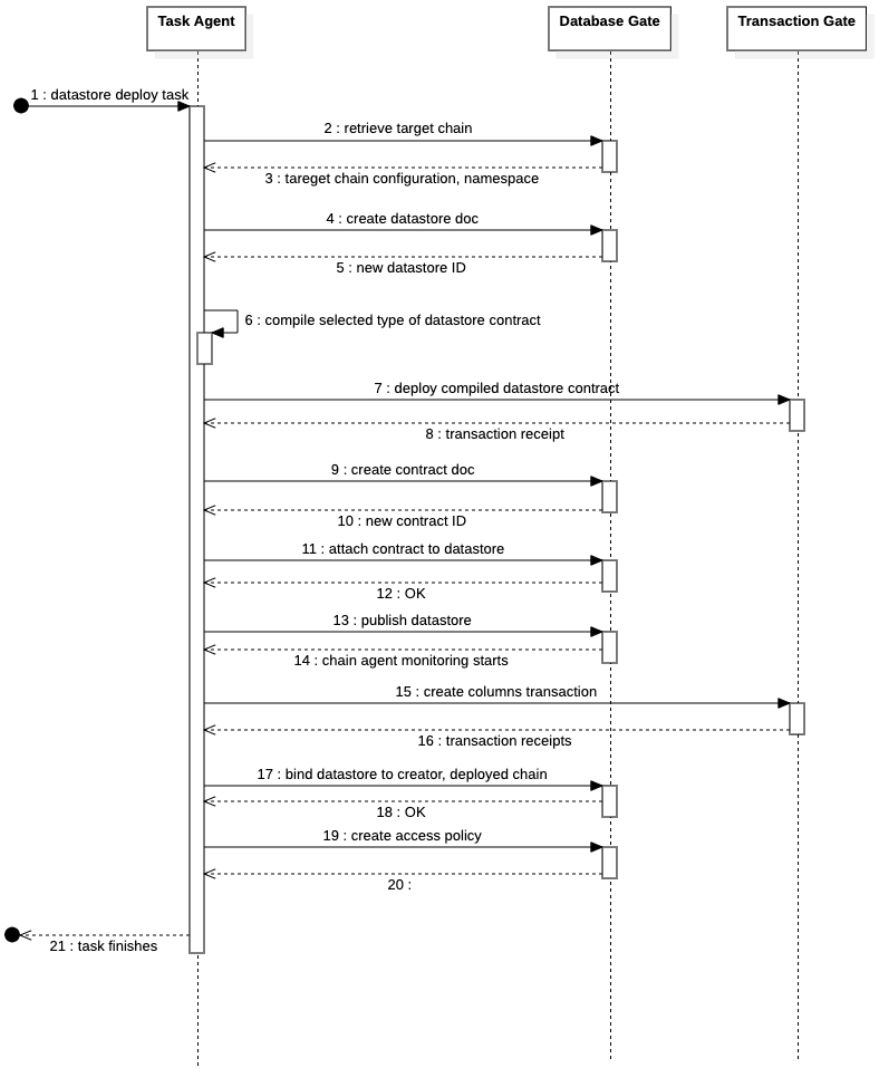

Task Agent
Database Gate
Transaction Gate
1 : datastore deploy task
2 : retrieve target chain
3 : tareget chain configuration, namespace
4 : create datastore doc
5 : new datastore ID
6 : compile selected type of datastore contract
7 : deploy compiled datastore contract
8 : transaction receipt
9 : create contract doc
10 : new contract ID
11 : attach contract to datastore
12 : OK
13 : publish datastore
14 : chain agent monitoring starts
15 : create columns transaction
16 : transaction receipts
17 : bind datastore to creator, deployed chain
18 : OK
19 : create access policy
20 :
21 : task finishes

Figure 12: Sequence Diagram - User Datastore Deployment

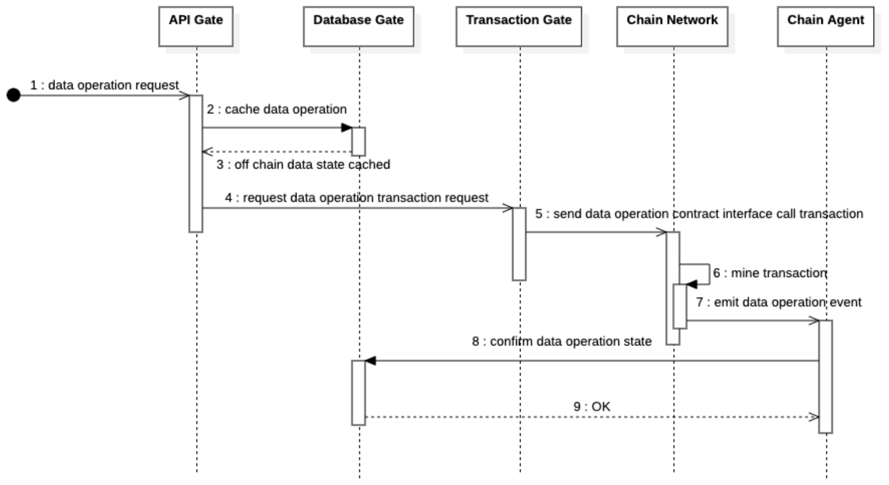

API Gate
Database Gate
Transaction Gate
Chain Network
Chain Agent
1 : data operation request
2 : cache data operation
3 : off chain data state cached
4 : request data operation transaction request
5 : send data operation contract interface call transaction
6 : mine transaction
7 : emit data operation event
8 : confirm data operation state
9 : OK

Figure 13: Sequence Diagram - User Datastore Data Operation

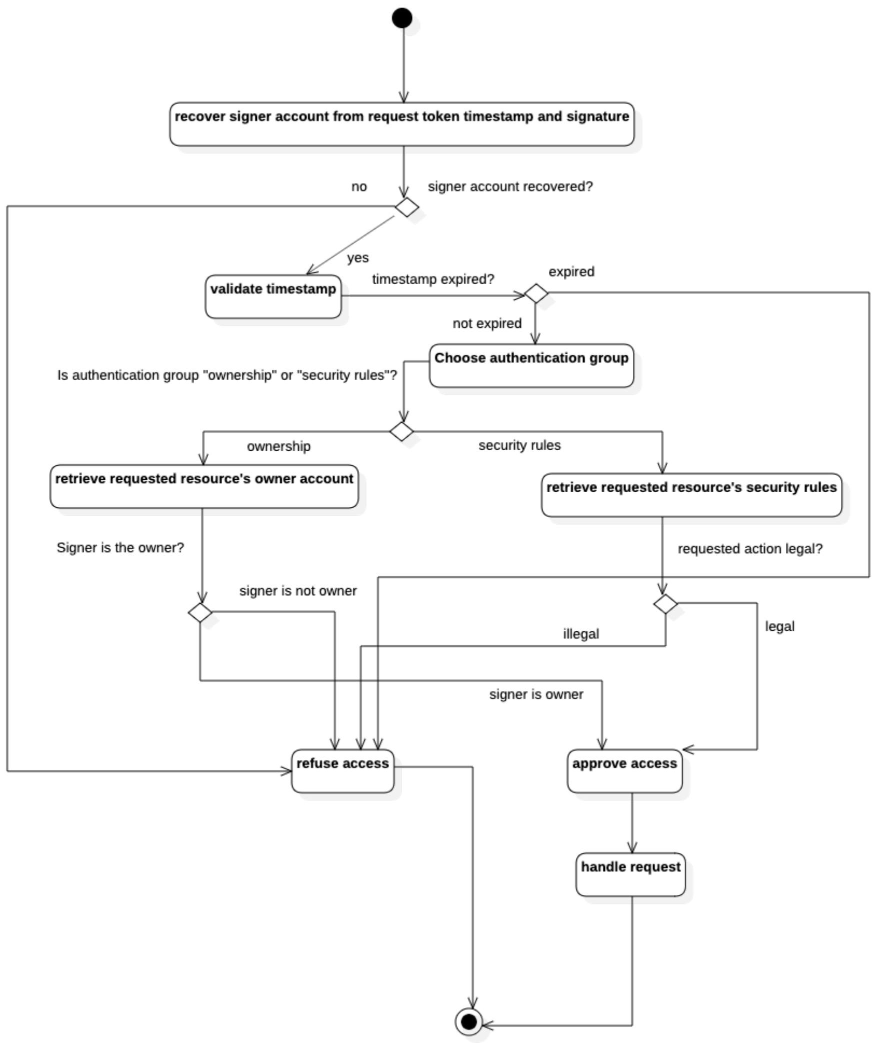

recover signer account from request token timestamp and signature
no
signer account recovered?
yes
validate timestamp
timestamp expired?
expired
not expired
Choose authentication group
Is authentication group "ownership" or "security rules"?
ownership
security rules
retrieve requested resource's owner account
retrieve requested resource's security rules
Signer is the owner?
requested action legal?
signer is not owner
illegal
legal
signer is owner
refuse access
approve access
handle request

Figure 14: Activity Diagram - API Access Control

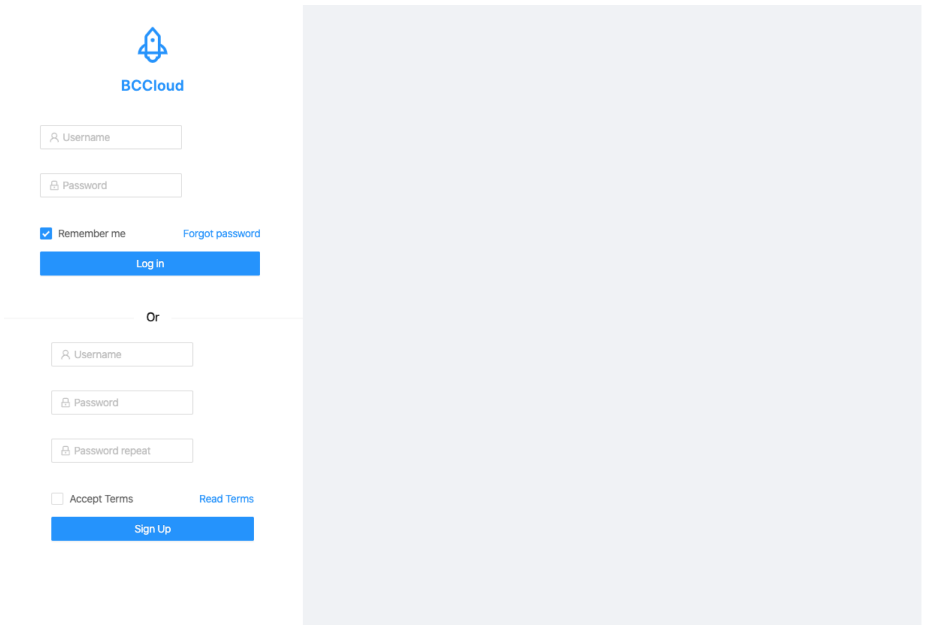
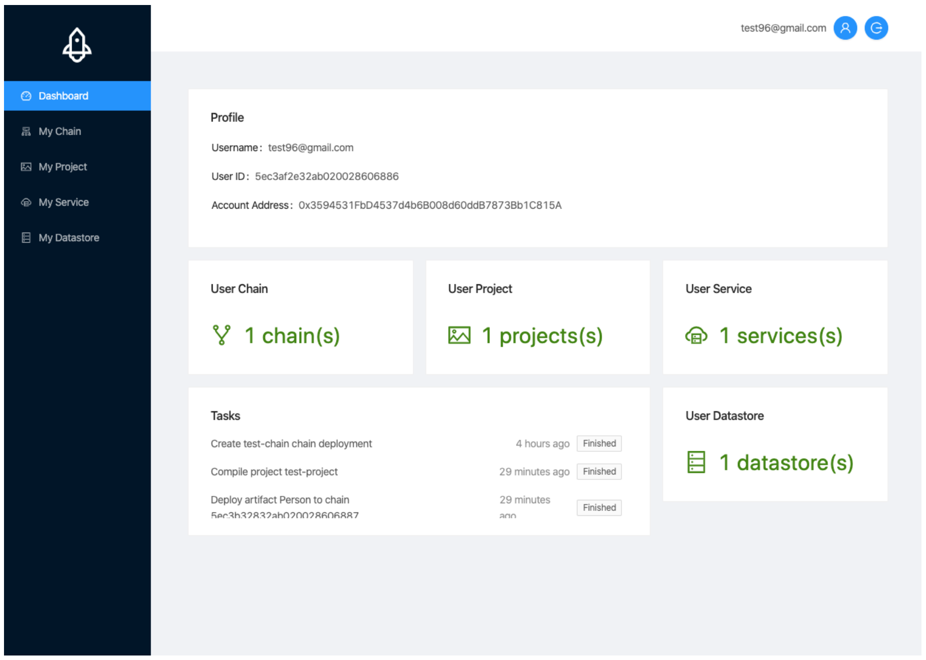


Figure 15: WebUI - Login



Profile

Username: test96@gmail.com

User ID: 5ec3af2e32ab020028606886

Account Address: 0x3594531FbD4537d4b6B008d60ddB7873Bb1C815A

User Chain: 1 chain(s)

User Project: 1 projects(s)

User Service: 1 services(s)

Tasks

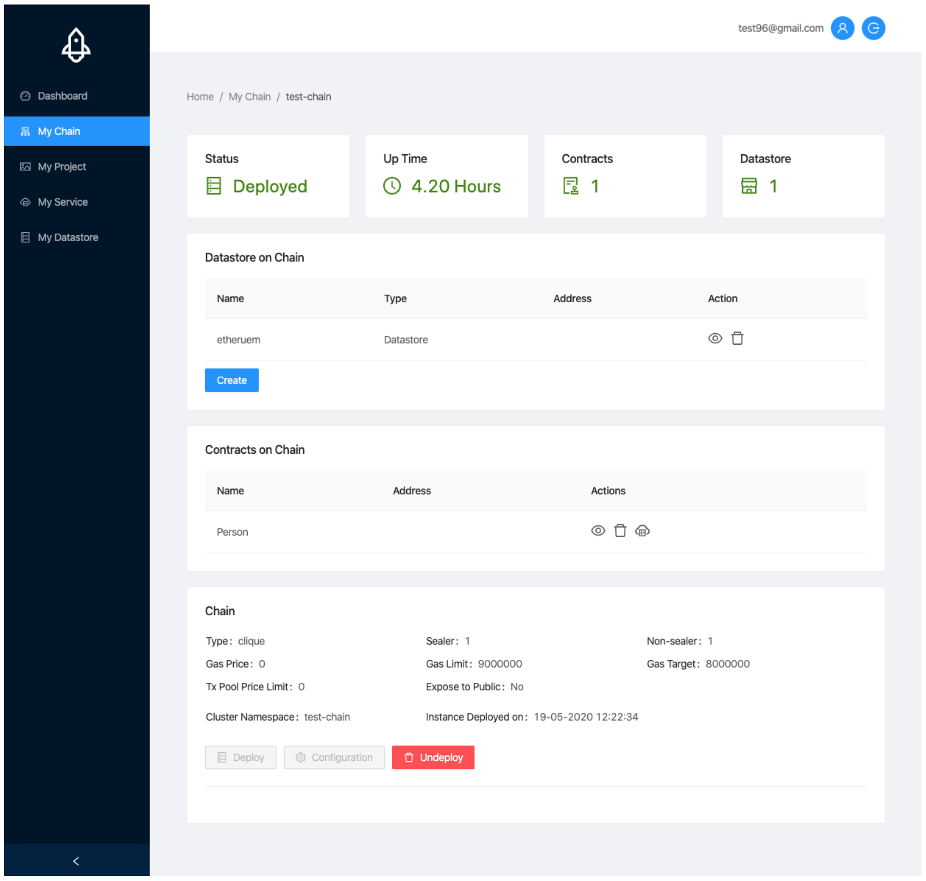
Create test-chain chain deployment — 4 hours ago — Finished

Compile project test-project — 29 minutes ago — Finished

Deploy artifact Person to chain 5ec3b32832ab020028606887 — 29 minutes ago — Finished

User Datastore: 1 datastore(s)

Figure 16: WebUI - Dashboard



Status	Up Time	Contracts	Datastore
Deployed	4.20 Hours	1	1

Datastore on Chain

Name	Type	Address	Action
etheruem	Datastore		

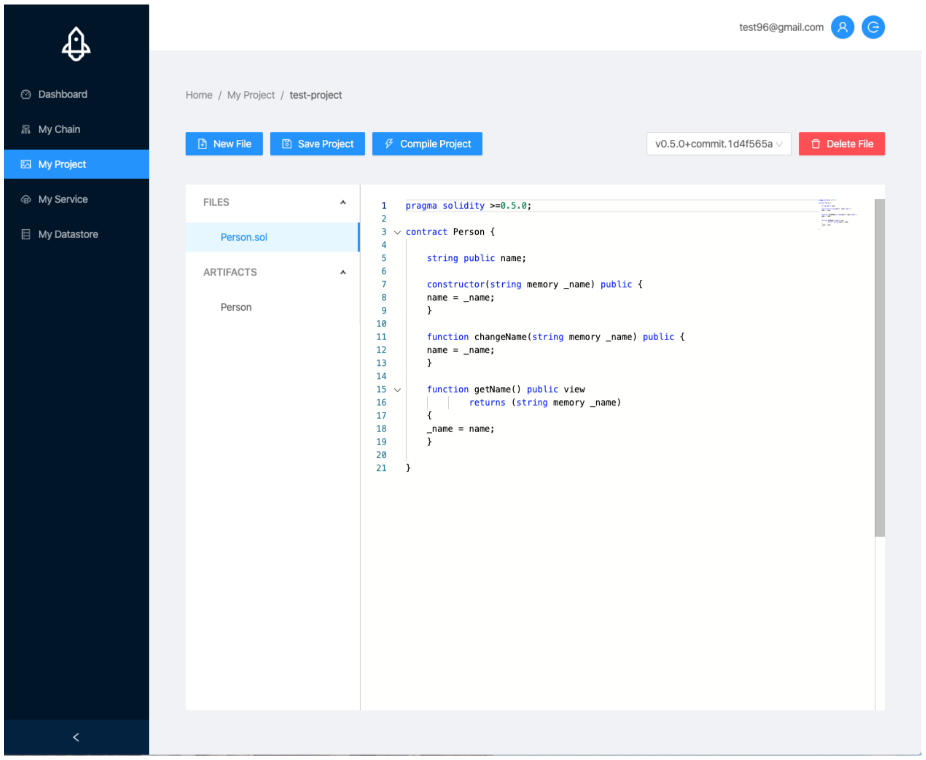
Contracts on Chain

Name	Address	Actions
Person		

Chain

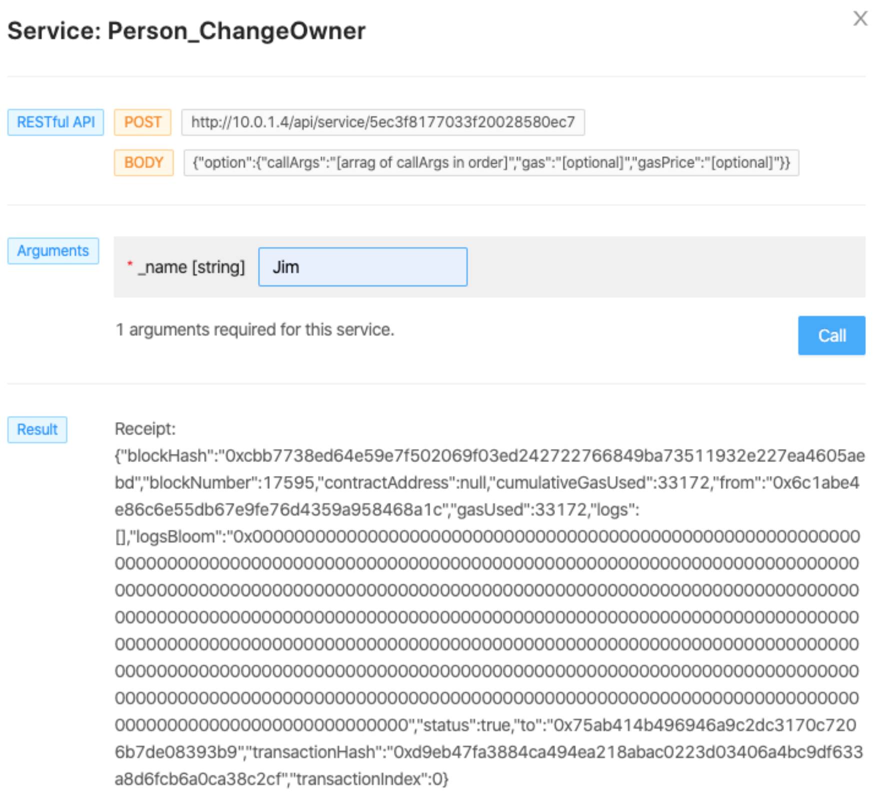
Type: clique	Sealer: 1	Non-sealer: 1
Gas Price: 0	Gas Limit: 9000000	Gas Target: 8000000
Tx Pool Price Limit: 0	Expose to Public: No	
Cluster Namespace: test-chain	Instance Deployed on: 19-05-2020 12:22:34	

Figure 17: WebUI - My Chain



```
pragma solidity >=0.5.0;

contract Person {

    string public name;

    constructor(string memory _name) public {
    name = _name;
    }

    function changeName(string memory _name) public {
    name = _name;
    }

    function getName() public view
            returns (string memory _name)
    {
    _name = name;
    }

}
```

Figure 18: WebUI - Project IDE Editor


Service: Person_ChangeOwner

RESTful API POST http://10.0.1.4/api/service/5ec3f8177033f20028580ec7

BODY {"option":{"callArgs":"[arrag of callArgs in order]","gas":"[optional]","gasPrice":"[optional]"}}

Arguments * _name [string] Jim

1 arguments required for this service.

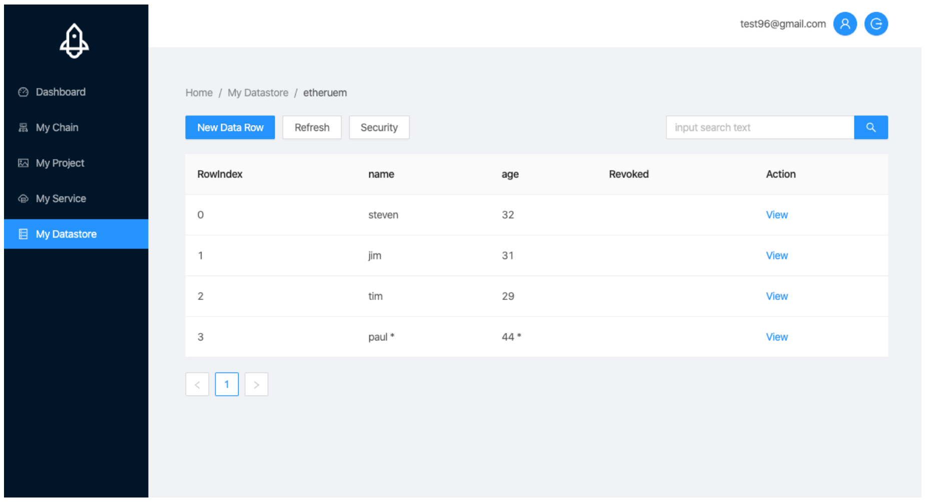
Call

Result

Receipt:

```
{"blockHash":"0xcbb7738ed64e59e7f502069f03ed242722766849ba73511932e227ea4605ae
bd","blockNumber":17595,"contractAddress":null,"cumulativeGasUsed":33172,"from":"0x6c1abe4
e86c6e55db67e9fe76d4359a958468a1c","gasUsed":33172,"logs":
[],"logsBloom":"0x0000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000","status":true,"to":"0x75ab414b496946a9c2dc3170c720
6b7de08393b9","transactionHash":"0xd9eb47fa3884ca494ea218abac0223d03406a4bc9df633
a8d6fcb6a0ca38c2cf","transactionIndex":0}
```

Figure 19: WebUI - User Serivce Call GUI



RowIndex	name	age	Revoked	Action
0	steven	32		View
1	jim	31		View
2	tim	29		View
3	paul *	44 *		View

Figure 20: WebUI - User Datastore Data Table

5 Evaluation

The evaluation of the prototype focuses mainly on two directions, namely the user experience and the performance of the prototype. In the user experience evaluation, a group of 10 people from IT background, in which 4 of them have extended blockchain knowledge and experience, are invited to a preliminary hands-on experiment with a follow-up feedback survey. Based on the feedback, we investigate the usability of the prototype and verify, from the user's aspect, if the prototype fulfills its design goal in terms of simplifying the overall workflow of the blockchain technology use case.For the performance test, the prototype is put through several designed tests. These test scenarios represent the system's primary functions and produce a measurement of the system performance with statistic. Based on the result, we evaluate the performance and analyse the potential bottleneck.

5.1 User Hands-on Experiment

At the beginning of the experiment, each tester is briefly introduced to the prototype and provided with a test manual and a guide throughout the test. The test manual can be found in Appendix. Leading by the test manual, the tester performs the following tasks with the prototype.

1. Register a user account and login to the platform.
2. Create a user chain and deploy the chain instance.
3. Write and compile a simple smart contract in a user project. Deploy it to the chain instance.
4. Create a user service that uses the deployed smart contract. Make requests to call the service in the WebUI graphical interface.
5. Create a user datastore, try out all the data operations.

The tasks cover the major functionalities of the prototype, also represent workflows that users typically perform. The tester should grasp an overall experience with the prototype and gain insight into what the prototype can offer.

5.2 Experiment Survey

To evaluate the usability, the testers are asked a series of questions directly after the experiment. The first ten questions are taken from the standard version of the System Usability Scale, namely,

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.

Table 13: User Experience SUS Questionary Result

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
Q1	4	5	5	4	5	5	3	4	5	5
Q2	1	1	1	1	1	1	1	1	1	1
Q3	5	5	5	5	5	5	4	5	5	5
Q4	3	2	4	2	1	2	3	1	1	2
Q5	5	5	5	5	5	5	5	5	5	5
Q6	1	2	1	1	1	1	1	2	1	1
Q7	5	5	5	5	5	5	5	5	5	5
Q8	2	1	1	1	1	1	1	1	1	1
Q9	4	5	5	5	5	5	4	4	5	5
Q10	4	1	4	4	1	4	5	2	1	3

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 a lot of things before I could get going with this system.

The System Usability Scale (SUS) is a simple, ten-item scale questionary giving a global view of subjective assessments of usability. For each item, tester gives score by choosing a scale from 1(represent strongly disagree) to 5(strongly agree). The SUS score is then calculated based on the score contribution of each item. Each item's score contribution ranges from 0 to 4. For items 1,3,5,7,and 9 the score contribution is the scale position minus 1. For items 2,4,6,8 and 10, the contribution is 5 minus the scale position. Multiply the sum of the scores by 2.5 to obtain the overall value of SUS. [5]

The survey result is listed in Table 13 and plotted in Figuren 21. The rows of the table represent 10 questions from the SUS questionary. The columns represent 10 testers, in which tester 2, 5, 8, and 9 are testers who have pre-knowledge and experience with blockchain technology. The prototype achieves an average total of 89 SUS scores in the experiment. We can see a clear polarization from question 10 between the tester groups with and with no pre-knowledge. However, the overall statistic is not dramatically influenced, which indicates that the prototype effectively reduced the complexity of the blockchain application use case.

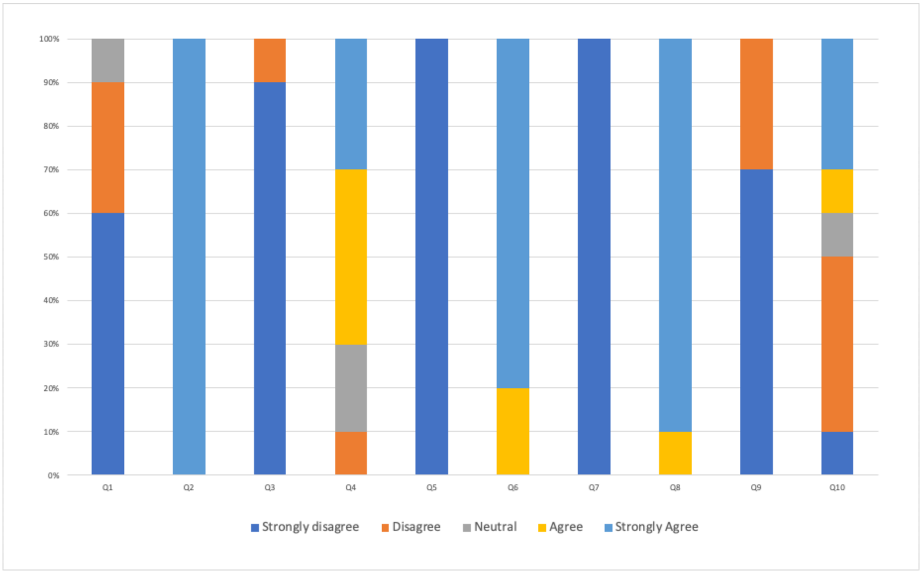


Figure 21: SUS Questionary Result

To further determine whether the prototype reaches its design goal of simplifying the overall workflow of the blockchain technology use case. The testers who already have pre-knowledge and experience with the blockchain technology are asked the following 4 additional questions.

11. The prototype provides a straightforward solution for you to create and deploy a private blockchain network without much technical overhead compared to the traditional workflow.

12. The smart contract development workflow offered by the prototype satisfies your needs as an online integrated IDE.

13. The user service you created is a meaningful way to utilize smart contract in a micro-service based cloud architecture.

14. The user datastore provides you a suitable solution for storing and querying data based on blockchain technology.

Table 14 shows the result of the extended questionary by testers with pre-knowledge and experience of the blockchain technology.

The data indicates that the testers are satisfied with the user experience, and the design goal is achieved.

5.3 System Performance Test

There are altogether 5 designed test scenarios that focus mainly on the performance of the user service and user datastore functions of the prototype. The test

Table 14: User Experience Addtional Questionary Result

	T2	T5	T8	T9
Q11	Strongly Agree	Strongly Agree	Agree	Strongly Agree
Q12	Agree	Neutral	Agree	Agree
Q13	Agree	Agree	Agree	Agree
Q14	Agree	Agree	Agree	Agree

Table 15: Service Performance Test Call Only (Unit in Milliseconds)

Round 1	Round 2	Round 3	Round 4	Round 5	Average
10338	10228	10210	10214	10217	10241.4

environment is a managed Kubernetes cluster on Microsoft Azure Cloud with 5 nodes of the standard F8s_v2 virtual machine, each of the virtual machine is equipped with 8 virtual CPUs and 16GiB of memory. The prototype is internally scaled to avoid bottleneck, the API Gate, Database Gate, Transaction Gate components each has 5 load-balancing replicas.

Service Performance Test (call only) is a test in which 5 rounds of 100 requests are sent to a user service created on a user chain instance with one transaction node and one signing node. The service invokes a transaction call with no mining in the chain instance required. In other words, a read-only function of a smart contract. The requests are dispatched sequentially in a 100 milli-second interval to overload the service capacity but avoid pertential request lost in the mean time. In each round, the time duration from sending the first request to all requests are answered is measured in millisecond. Table 15 listed the measured values and the average of the 5 rounds. The prototype achieved 10241.4 milliseconds per 100 requests on average. Based on the hardware of the test environment, the performance is good.

Service Performance Test (mining) has a similar test process and setup, except the service call invokes a transaction call with mining required in the chain instance. From the measurements in Table 16, we see that the average duration time increased to 12223 milliseconds due to the added mining process. However, this increase is foreseeable, and the overall performance did not drastically decrease.

Table 16: Service Performance Test Mining (in Milliseconds)

Round 1	Round 2	Round 3	Round 4	Round 5	Average
12256	12235	12218	12194	12212	12223

Service Performance Comparison Test performs 5 rounds of 50 requests described in Service Performance Test(call only and mining) to two chain instances, one with one transaction node one the two transaction nodes. In theory, because Kubernetes automatically balances the request loads between transaction nodes, the service on the chain instance with two transaction nodes has double transaction call handling throughput. Based on the data in Figure 22 and Figure 23, the performance improvement is limited in both case, around 10 to 20 milliseconds.

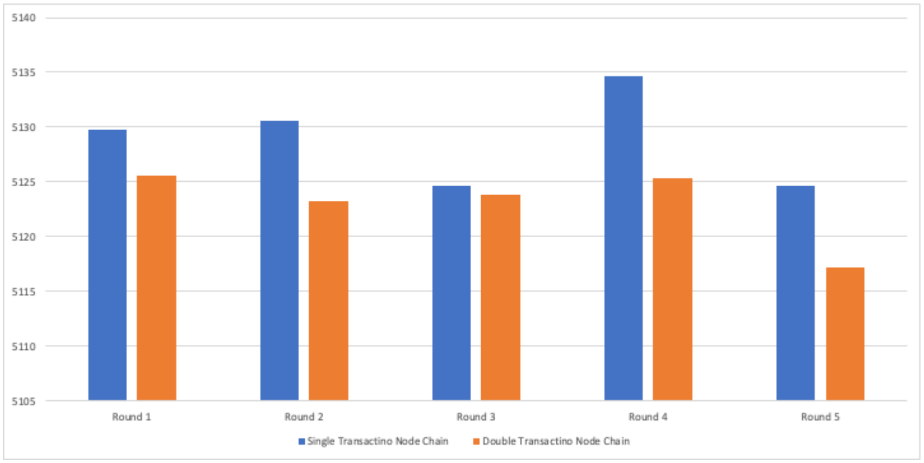


Figure 22: Service Performance Comparison Test Call Only (in Milliseconds)

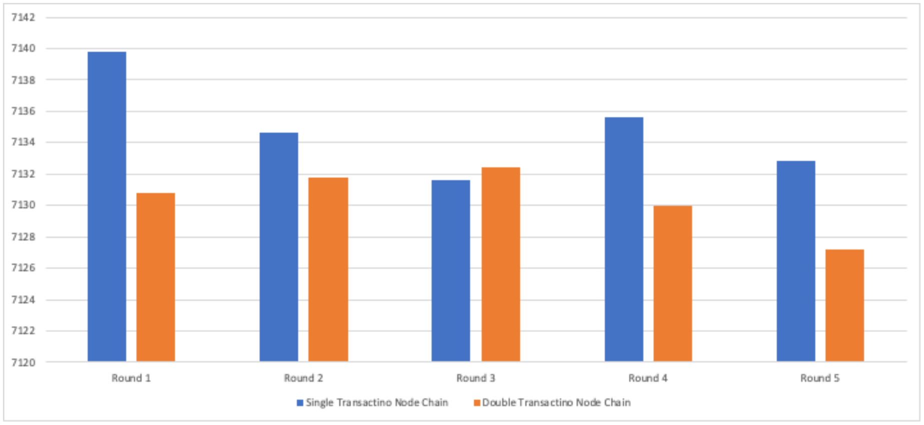


Figure 23: Service Performance Comparison Test Mining (in Milliseconds)

Datastore Read Test records the time duration for 100 read operations of 10 data rows from a datastore with two columns. The requests are dispatched in the same fashion as the service performance test. It took 101304 milliseconds for all the requests to finish. The performance is good, because only the off-chain data replica is involved in the read operation.

Datastore Write Test writes 100 data rows with two columns into a datastore. The requests are dispatched in the same fashion as the service performance test. The recorded timestamps are the test start, all writing request answered, and all transaction mined. It took 101359 millisecondes for all the write requests to finish, by which point, the changes are cached and visible to the user. Till all changes are confirmed, namely all mining completed, the system took another 11019 milliseconds.

6 Conclusion

This thesis aimed to create a Blockchain-as-an-Service(BaaS) solution that allows users to create, deploy, and access micro-service architecture use cases backed up by the blockchain technology. We introduced related background concept and technology, surveyed and reviewed the state-of-the-art approaches from current commercialized products. We implemented a prototype and evaluated it from both user experience and performance aspects as the concrete outcome of this thesis.

6.1 Summary of the prototype

- The technical overhead of blockchain technology is usually challenging but can be greatly reduced with abstraction and automation of the workflow. The prototype provided an effortless way for end-users to deploy blockchain instance with a few essential parameter settings and a few clicks.

- Smart contract has a close and meaningful connection to the micro-service architecture because of its lightweight and immutable nature. The user service feature of the prototype utilizes smart contracts as the backend for micro-services. This allows the micro-services benefits from the data security and integrity features from the blockchain technology.

- Combining the traditional blockchain technology with the Docker Container and the Kubernetes cluster orchestration technology, the prototype achieves good availability and scalability by dividing the system into functional components. Each component is individually scalable and kept at the desired state by Kubernetes. However, this does not solve the bottleneck created by the blockchain mining process.

- The prototype defined a clear workflow for the end-users from creating a blockchain network in the cloud, to deploy a public accessible smart contract-based micro-service. The user datastore combines the off-chain data cache concept with on-chain data storage, and offers an efficient solution for the end-users to store and query data with blockchain technology.

- In comparison to the traditional network authorization and authentication methods, the prototype takes advantage of the protogenic authentication mechanism from the Ethereum blockchain by associating the user profile with an EOA and produces several benefits. First, users' passwords are neither transmitted nor stored at the server-side. Second, users' communications with the system are authenticated by a non-constant signature signed by the EOA private key. Additionally, the EOA address allows users to identify and control access from other users in user service and datastore to ensure data privacy.

6.2 Limitation of the prototype

Regarding to the scope of this thesis and by the extended analyzation of the questions answered above, we formulatete following limitations of the prototype.

- Restricted blockchain model

  The prototype currently only supports the deployment of an Ethereum "proof-of-authority" private chain network with the clique protocol. Many configuration options of the peer-to-peer chain network are hidden from the end-users to provide a straightforward experience.

- Fixed and isolated data structure in datastore

  The user datastore only stores data in a table-like data structure. Despite the query filter, there is no support for cross datastore data aggregation and internal data exchange between user service and user datastore.

- No flexibility in the chain network after deployment

  Once a chain instance is deployed, the user has no way of re-configure the network setup(adding or removing sign node and transaction node) without un-deploy and re-deploy.

- The mining process bottleneck

  The mining process in the blockchain network causes delay to user service calls and user datastore operations. However, this can not be avoided or improved without significant technical updates from the Ethereum blockchain itself and is beyond the scope of this thesis.

7 Future Work

The thesis topic covers a wide range of research areas, and both the blockchain technology and concept of "Blockchain-as-a-Service" are undergoing a rapidly evolving process. The prototype achieved the design goal and covered all the functional requirements. However, from the limitation analysis and comparison with other state-of-the-art solutions, many improvements and extended features can be integrated.

- Extend the supported blockchain protocols

  Besides "proof-of-authority", Etheruem also supports the "proof-of-work" protocol. Although the "proof-of-work" protocol generally requires more computing resources and producing longer mining delay, it does offer a higher security level in terms of data integrity and can be more useful in many scenarios.

- Precise control of the blockchain instance

  The configuration of a blockchain network can be overwhelming, the prototype only exposed very few of them to reduce the complexity of the problem. However, in a real-world production environment, a more precise control over the configuration can fit more scenarios. Instead of masking them, conducting further research on exposing more configuration parameters is constructive to the project. Additionally, mechanisms to dynamically scale the network nodes in runtime can bring more flexibility to the end-users and fits the distributed nature of the blockchain network closely.

- User datastore in more data structures

  Provides more data structures gives users more versatility when organizing their data in the datastore. Traditional database technologies nowadays have different data structures that suit different requirements. Other than the table-based data structure, document-based and graph-based data structure received much attention as well. Exploring more blockchain-based data structures can elongate the application range of the user datastore.

- Internal data interface between user service and datastore

  Currently, the data stored in a user datastore is isolated from the user services and other user datastores in the same chain instance. In theory, data referencing a user service to one or multiple user datastores is possible as both utilize smart contracts as the backend. By implementing a universal internal data interface, end-users can create complex systems with multiple user services and datastore without handling the business logic outside the platform. Thus, it makes the prototype a complete standalone solution.

- Comprehensive user profile

  As for now, the user profile implemented in the prototype is simplistic. A complete implementation of the user profile with more user identity information can make system features like security rules of user services and user datastores more intuitive and, therefore, improve the overall user experience.

References

[1] Kubernetes Documentation - Deployments. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. [Online; accessed 04-May-2020].

[2] Kubernetes Documentation - Namespace. https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/. [Online; accessed 04-May-2020].

[3] Kubernetes Documentation - Pod Overview. https://kubernetes.io/docs/concepts/workloads/pods/pod/. [Online; accessed 04-May-2020].

[4] Kubernetes Documentation - Service. https://kubernetes.io/docs/concepts/services-networking/service/. [Online; accessed 04-May-2020].

[5] Brooke, J., et al. Sus-a quick and dirty usability scale. *Usability evaluation in industry 189*, 194 (1996), 4–7.

[6] Dannen, C. *Introducing Ethereum and Solidity*, vol. 1. Springer, 2017.

[7] Dua, R., Kohli, V., and Konduri, S. K. *Learning Docker Networking*. "Packt Publishing Ltd", 2016.

[8] Hightower, K., Burns, B., and Beda, J. *Kubernetes: up and running: dive into the future of infrastructure.* " O'Reilly Media, Inc.", 2017.

[9] Hightower, K., Burns, B., and Beda, J. *Kubernetes: up and running: dive into the future of infrastructure.* " O'Reilly Media, Inc.", 2017.

[10] Lu, Q., Xu, X., Liu, Y., Weber, I., Zhu, L., and Zhang, W. ubaas: A unified blockchain as a service platform. *Future Generation Computer Systems 101* (2019), 564–575.

[11] Merkel, D. Docker: lightweight linux containers for consistent development and deployment. *Linux journal 2014*, 239 (2014), 2.

[12] Mesbah, A., and Van Deursen, A. An architectural style for ajax. In *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA'07)* (2007), IEEE, pp. 9–9.

[13] Mesbah, A., and Van Deursen, A. Migrating multi-page web applications to single-page ajax interfaces. In *11th European Conference on Software Maintenance and Reengineering (CSMR'07)* (2007), IEEE, pp. 181–190.

[14] Nadareishvili, I., Mitra, R., McLarty, M., and Amundsen, M. *Microservice architecture: aligning principles, practices, and culture.* " O'Reilly Media, Inc.", 2016.

[15] Nadareishvili, I., Mitra, R., McLarty, M., and Amundsen, M. *Microservice architecture: aligning principles, practices, and culture.* " O'Reilly Media, Inc.", 2016.

[16] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. Tech. rep., Manubot, 2019.

[17] Network, P. Proof of authority: consensus model with identity at stake, 2017.

[18] Samaniego, M., and Deters, R. Blockchain as a service for iot. In *2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)* (2016), IEEE, pp. 433–436.

[19] Zheng, W., Zheng, Z., Chen, X., Dai, K., Li, P., and Chen, R. Nutbaas: A blockchain-as-a-service platform. *IEEE Access PP* (09 2019), 1–1.

A User Manual

The following user manual provides a short overview of the basic concept of the technologies behind the prototype and offers a description of the system functionalities and step-by-step usage instructions for end-users.

USER MANUAL

The following user manual provides a short overview of the basic concept of the technologies behind the prototype and offers a description of the system functionalities and step-by-step usage instructions for end-users.

INTRODUCTION

Micro-service architecture is a software architecture for building large, replaceable, and maintainable systems. A micro-service is a small, independent service-oriented component that is loosely bounded with other micro-services to create a complex system. Each micro-service is designed to be atomic and replaceable to avoid spending efforts and resources on service maintenance. Adapting to micro-service architecture gives the benefits of efficient team cooperation for the developers, and flexible and progressive development strategies for the company.

Ethereum is an open-source, blockchain-based distributed computing platform. It was proposed in late 2013 by Vitalik Buterin and initially released in mid-2015. Ethereum is an open-source, blockchain-based distributed computing platform. It was proposed in late 2013 by Vitalik Buterin and initially released in mid-2015. Ethereum supports both "proof-of-work" and proof of stake consensus algorithms. Under the proof of work protocol, network nodes invest a substantial amount of computing resources in the process of "mining" to completing transactions and get Ethers as the reward. "Proof-of-stake" allows nodes to reach consensus without mining, the next node to create a new block on the chain is selected based on its stake. "Proof-of-authority" is a specific case of "proof-of-stake", in which a validator's identity takes the role of the stake.

Blockchain-as-a-Service(BaaS) is a recently emerged concept that focuses on constructing a cloud-based solution for hosting blockchain instances to lift the obstacles the user faces in building and utilizing blockchain technology. Users no longer have the burdensome task of managing the infrastructure for a blockchain project. Currently, service providers like AWS, Microsoft, and IBM offers BaaS with different approaches.

The prototype is a solution that embeds the Blockchain technology in a cloud architecture and offer it to the end-users as a Blockchain-as-an-Service (BaaS) and allow users to create, deploy, and access micro-service architecture use cases, which are backed-up by the Blockchain technology, without much technological overhead.

PROTOTYPE FUNCTIONS AND USAGE INSTRUCTION

REGISTRATION AND LOGIN

To access the platform, registration is required to acquire a unique user profile representing the user's identity on the platform.

To register for a new account, at the registration part of the front page, type in a username and password and click "sign up" button. "Password repeat" and "Accept Terms" is no enforced in this prototype implementation.

After registration, type in the username and password and click login at the login part of the frontpage to login.

To avoid login next time, click "Remember Me". The user's account credentials, including the private key for authentication, will be stored in the browser's local storage until the user log out. Use it only in a private and trustable environment.

USER DASHBOARD

After successful login, the user is taken to the dashboard page. The page provides the user with an overview of the profile information and the resources the user created in the platform. The user can also monitor all the system tasks in the task list.

CREATE AND DEPLOY USER CHAIN

User chain is the private blockchain instance configured and deployed by a user in the system and is the groundwork of other system features like user service and datastore. Each user chain must have a unique name across the entire system.

To create a user chain:

- Go to "My Chain" in the left navigation menu, click "Create Chain".
- Type in a chain name in the left pop-up panel, and click "Create" .

The new user chain is then listed in "Chain List" table. View the user chain by clicking on the eye icon under "Action" column. The user is then taken to the chain view where chain information is displayed. The user can then configure the chain instance and deploy it.

To configure and deploy the user chain:

- Click "Configuration" button in the chain view, select chain type, sealer node count, and transaction node count and click "OK". Leave other parameters to its default if you are not familiar with the blockchain network setup.
- Click "Deploy" to initiate the deploy task. A task monitor will pop up and displaying the task status and log outputs. Once the task states is "Finished" close the pop-up.

The chain instance is now deployed.

WORKING WITH USER PROJECT

A user project is where a user develops, stores, and compiles smart contracts related to the same project. Compiled smart contracts are stored in the project as the project artifacts and can be later deployed on to a running user chain instance.

To Create a user project:

- Go to "My Project" in the left navigation menu, click "Create Project".
- Type in a project name in the left pop-up panel, and click "Create" .

The new project is then listed in "Project List" table. View the project by clicking on the eye icon under "Action" column.

In the project view, the user can create new files to write smart contracts in solidity. The code editor provides basic code high-lighting and a mini-map of the code. After saving the project, the smart contract's code can then be compiled into artifacts. The user can also specify the compiler version for the compile.

After the code is compiled, the compiled contracts are listed under the project artifacts, where information like the contract ABI and bytecode are displayed. If the compile encounters errors, the user will see error hints in the code editor's corresponding code position.

Once a project is successfully compiled and artifacts are generated, the user can deploy the artifact to a running user chain instance.

To deploy an artifact to a running chain instance:

- Select the artifact to be deployed from the project artifact list.
- Click "Deploy".
- Select the target deploy chain instance from the left panel, and provide contract constructor argument if any is required.
- Click "Confirm"

Once the deploy task is finished, the deployed contract can be found under the chain view under "My Chain".

CREATE AND ACCESS USER SERVICES

A user service is the "micro-service" like feature of the prototype. In a word, it represents the entry point of the invocation call to a smart contract function on the blockchain.

User services can be created from deployed contracts on running chain instances. It handles the process by converting HTTP requests to blockchain transactions, thus allows the service consumers to interact with the services without knowing much details on the underlying smart contracts and chain instances.

To create a user service:

- Navigate to the chain view of the chain on which the contract is deployed.

- Under the "Contracts on Chain" list, click the "Create Service" icon under the actions column.
- In the right panel opened, name the service, and select the target function of the contract to be invoked.
- Click "Confirm".
- Navigate to "My Service" from the left navigation menu.

The created service can be seen under the service list.

To call the service from the WebUI interface, click on the view icon under the action column. Provides any required arguments and click "Call" button. The result will be displayed once the call is finished. The RESTful API endpoint is published in the service view, and users can access the service via AJAX call outside the WebUI interface.

CREATE AND USE USER DATASTORE

User datastore is a concept designed for improved data storage and query experience on blockchain. In a datastore, users can store data entries in a table like fashion. The data is saved on the chain and cached in a complementary database to ensure its integrity and offers faster query speed. User datastore can be created from the chain view of a running chain instance.

To create a user datastore:

- Navigate to the chain view of the chain on which the datastore is deployed.
- Under the "Datastores on Chain" list, click the "Create" button.
- In the right panel opened, name the datastore, select the type.
- Adding columns to the datastore by giving column name and selecting column data type and click "Append Column".
- Click "Create".

Once the task is finished, the newly deployed datastore appears under the "Datastores on Chain" list. The user can then navigate to "My Datastore" in the left navigation menu to use the datastore in the datastore view.

In the datastore view, the user can create new data by data rows, newly created data(marked with a star after the data) is first cached in the off-chain data replica, and then confirmed(no star after the data) once mined on the blockchain.

By clicking the "View" link under the "Action" column, the user can have a detailed view of the data row and its modification history. Values are displayed in columns with additional information, including cached time, mining time, and the EOA address that triggered the action. The user can update the values of each column by giving the new value and click the "Update" button. The operation will be recorded in the data row history.

Data row can be revoked if it is never needed again. Revoking the data row does not delete it from the datastore, it is marked as revoked to mark the state of unused and prevent further changes.

Search filters allow the users to query the datastore with custom conditions. Single filter is written in the JSON format like

{[columnName]: [filter value or filter comparison operators]},

Aggregated filters are written in the JSON format like

{[Logical Operators]: [filter1, filter2, ...]},

The supported comparison operators and logical operators are listed in the following tables.

Comparsion Operator	Description	Example
$gt	Greater than	{"age": {"$gt": 18}}
$gte	Geater than or equal	{"age": {"$gte": 18}}
$lt	Less then	{"age": {"$lt": 18}}
$lte	Less than or equal	{"age": {"$lte": 18}}
$in	Matches values in an array	{"bloodType": {"$in": ["A", "B"]}}
$nin	Matches values not in an array	{"bloodType": {"$nin": ["A", "B"]}}
$ne	Not equal	{"gender": {"ne": "male"}}

Logical Operators	Description	Example
$and	logical AND	{$and: [{f1}, {f2}, ...]}
$not	logical NOT	{$not: {f}}
$nor	logical NOR	{$nor: [{f1}, {f2}, ...]}
$or	logical OR	{$or: [{f1}, {f2}, ...]}

B User Hands-on Experiment Guide

The following guide is provided to the participants for user experience experiment. The guide consisits of a foreword, a to-do list, a survey questionary, and an example smart contract code piece for participants with limited smart contract development experience.

USER HANDS-ON EXPERIMENT GUIDE

FOREWORD

Dear participants, for this experiment, we ask you to evaluation the user experience of our prototype. Please take some time and read the user manual provided to you first, and then perform the tasks listed in the To-Do list below. Finally, based on your user experience, please answer the 14 questions in the user experience survey section.

The experiment should take around 30 minutes, and if you have any questions during the experiment, we are glad to assist you. We appreciate your participation, and thank you for your opinions.

TO-DO LIST

- Register on the platform
- Log into the platform with the user account created
- Create and deploy a user chain instance. Due to the hardware limitation, we recommend no more one to three signer nodes and a maximum of four transaction nodes.
- Create a user project and develop some smart contract within the WebUI editor. If you don't have much experience in writing smart contracts, we provide an example code piece at the end of this guide.
- Create user services with the smart contract you deployed in the previous task. Try invoking the service inside the WebUI interface.
- Create a user datastore on the chain instance you deployed. You can define the columns of the datastore as you wish.
- Add some data rows into the datastore and observe the state of the data.
- Update some data in the datastore, and revoke some data rows, observe the state changes as well.
- Try the search filter function in the datastore, try out some filter operators described in the user manual.

USER EXPERIENCE SURVEY

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

☐ Strongly Disagree

☐ Disagree
☐ Neutral
☐ Agree
☐ Strongly Agree

2) I found the system unnecessarily complex.
☐ Strongly Disagree
☐ Disagree
☐ Neutral
☐ Agree
☐ Strongly Agree

3) I thought the system was easy to use.
☐ Strongly Disagree
☐ Disagree
☐ Neutral
☐ Agree
☐ Strongly Agree

4) I think that I would need the support of a technical person to be able to use this system.
☐ Strongly Disagree
☐ Disagree
☐ Neutral
☐ Agree
☐ Strongly Agree

5) I found the various functions in this system were well integrated.
☐ Strongly Disagree
☐ Disagree
☐ Neutral
☐ Agree
☐ Strongly Agree

6) I thought there was too much inconsistency in this system.
☐ Strongly Disagree
☐ Disagree
☐ Neutral
☐ Agree
☐ Strongly Agree

7) I would imagine that most people would learn to use this system very quickly.
☐ Strongly Disagree
☐ Disagree
☐ Neutral
☐ Agree

☐ Strongly Agree

8) I found the system very cumbersome to use.
☐ Strongly Disagree
☐ Disagree
☐ Neutral
☐ Agree
☐ Strongly Agree

9) I felt very confident using the system.
☐ Strongly Disagree
☐ Disagree
☐ Neutral
☐ Agree
☐ Strongly Agree

10) I needed to learn a lot of things before I could get going with this system.
☐ Strongly Disagree
☐ Disagree
☐ Neutral
☐ Agree
☐ Strongly Agree

11) The prototype provides a straightforward solution for you to create and deploy a private blockchain network without much technical overhead compared to the traditional workflow.
☐ Strongly Disagree
☐ Disagree
☐ Neutral
☐ Agree
☐ Strongly Agree

12) The smart contract development workflow offered by the prototype satisfies your needs as an online integrated IDE.
☐ Strongly Disagree
☐ Disagree
☐ Neutral
☐ Agree
☐ Strongly Agree

13) The user service you created is a meaningful way to utilize smart contract in a micro-service based cloud architecture.
☐ Strongly Disagree
☐ Disagree
☐ Neutral
☐ Agree

☐ Strongly Agree

14) The user datastore provides you a suitable solution for storing and querying data based on blockchain technology.

☐ Strongly Disagree
☐ Disagree
☐ Neutral
☐ Agree
☐ Strongly Agree

Example smart contract code

```
pragma solidity >=0.5.0;

contract Person {

   string public name;

   constructor(string memory _name) public {
           name = _name;
   }

   function changeName(string memory _name) public {
           name = _name;
   }

   function getName() public view
        returns (string memory _name)
   {
           _name = name;
   }

}
```

C Source Code

The GitLab repository of the prototype project is at https://git01lab.cs.univie.ac.at/thesis/0906101-zheng-li.git