A MIDDLEWARE FOR INTEGRATION OF BLOCKCHAIN SMART CONTRACTS WITH ZEEBE PROCESS ENGINE AS TASK IMPLEMENTATIONS

By:

Gasan Rzaev

s3553213

Supervisor:

Prof. Dimka Karastoyanova

August 31, 2023

Abstract

Blockchain technology has attracted a lot of attention as an innovative breakthrough that has the potential to revolutionize several industries, including finance, healthcare, logistics, and more. The usage of smart contracts, which are self-executing programs that automate transactions when a set of predetermined conditions are met, is one of the main characteristics of blockchain technology. Smart contracts have the potential to lower transaction costs, get rid of intermediaries, and improve accountability and transparency. Another crucial area of focus where smart contracts can bring benefits is business process management (BPM), specifically workflow automation.

Workflow automation is a method of automating repetitive and manual tasks involved in business processes. By using smart contracts, it is possible to create a secure and transparent system that can streamline these processes and eliminate the need for a third-party entity. This can ensure traceability, correctness of process execution, and its immutability. Therefore, this thesis aims to explore the potential of blockchain-based workflow automation and its impact on business process management, by means of implementing and testing middleware with functionality to interact with smart contracts.

The questions that this research is aiming to answer are the following: “Are blockchains' smart contracts a beneficial solution for workflow automation in business process management systems?”; “How flexible and scalable can the integration of middleware between blockchains' smart contracts and business process management be?”. The main objective of the project is to build a middleware in answer to these questions.

Contents

List of Figures

Listings

1 Introduction

Business Process Management (BPM) is the discipline that focuses on optimizing and improving organizational processes to enhance efficiency and align with strategic goals. To gain a competitive advantage, it needs methodical planning, implementation, monitoring, and continual improvement. BPM and workflow automation are closely related because workflow automation uses digital tools and technology to streamline and carry out business operations effectively while BPM provides the structure and strategy for doing so. Organizations may achieve higher levels of productivity, cost effectiveness, and agility in today's changing business environment thanks to the integration of BPM and workflow automation.

Workflow automation is a technological approach that substitutes automated digital tools and systems for human, repetitive, and paper-based tasks inside a business process. The goal is to improve efficiency, consistency, and productivity in carrying out complex processes by streamlining the flow of tasks, information, and data while minimizing human intervention and potential errors.

There are many different tools and applications that can be used for workflow automation. This research focuses on Camunda Platform 8. For the purpose of running process instances, Camunda Platform 8 makes use of the Zeebe process engine. Using Camunda Platform 8, users may model their workflows in BPMN, distribute them, and carry them out on the Zeebe engine.

In the realm of computer science, blockchain stands as a decentralized and tamper-proof digital ledger, utilizing cryptographic techniques and consensus algorithms to securely record transactions and enable a myriad of decentralized applications. The decentralized applications deployed on blockchain networks make use of smart contracts, which are programs that execute themselves when certain predetermined conditions are met. The significance of smart contracts and blockchain lies in providing transparency, immutability, and trustworthiness, offering numerous benefits such as enhanced data security, reduced reliance on intermediaries, and the potential for innovative and efficient solutions across various industries.

In this paper we provide a brief introduction on blockchain and smart contract, as well as workflow automation. Then, we go over the related literature and existing research in Section 3. In Section 4 we discuss the technologies used in this project, and Section 5 will show the implementation of the middleware for smart contracts and our workflow automation engine, and show example workflows.

2 Background Information

This section serves the purpose of briefly introducing the reader to the technological concepts used in this project and their properties, with a specific focus on preparing the reader for the following sections on '*Technologies Used*' and '*Implementation*'.

2.1 Workflow automation

Workflow automation is a methodology through which businesses structure their processes to optimize customer service and achieve mission-critical objectives. This approach drives the design of business process workflows, aiming to enhance efficiency and streamline operations [1]. At the core of workflow automation lies the concept of a process engine. A process engine acts as a software application that efficiently manages the processing, storage, and distribution of data and information related to business processes. Notably, one of the key functions of process engines is tracking the progress and completion of process instances, enabling users to gain insights into the status of running and finished processes [2].

2.1.1 Process engine

A process engine is an instrumental software application responsible for overseeing the processing, storage, and dissemination of data pertinent to business processes. This vital component plays a crucial role in workflow automation by facilitating the tracking of process instances [2]. Through this tracking mechanism, users gain visibility into active and completed processes, enabling them to monitor progress and identify bottlenecks. In conjunction with the BPM methodology, the use of process engines ensures that business processes are structured to maximize efficiency and align with organizational objectives. By leveraging process engines, organizations can achieve heightened agility, productivity, and customer-centricity in their operations, ultimately contributing to the attainment of critical business goals [1].

2.2 Blockchain

A blockchain is a database of transactions that is updated and shared across many computers in a network. Every time a new set of transactions is added, its called a "block" - hence the name blockchain. Public blockchains like Ethereum allow anyone to add, but not remove, data [3].

2.2.1 Ethereum and Ethereum Virtual Machine

A choice of the blockchain for the project played significant role in development of the software. Using the Ethereum example, rather than Bitcoin, offers a broader perspective due to Ethereum's focus on decentralized applications through smart contracts executed by means of its virtual machine. Additionally, Ethereum's transition from proof-of-work (PoW) to proof-of-stake (PoS) consensus presents valuable insights into the evolution of blockchain technology, particularly in terms of energy efficiency, scalability, and security.

The term consensus mechanism refers to the entire stack of protocols, incentives and ideas that allow a network of nodes to agree on the state of a blockchain [4]. Proof-of-Work (PoW) is a consensus mechanism where miners compete to solve complex puzzles, requiring significant computational power. The first miner to find the solution gets to add a block and is rewarded with coins. Proof-of-Stake (PoS) is an alternative mechanism where validators are chosen based on the number of coins they hold and stake as collateral. Validators create blocks and are rewarded based on their stakes.

The Ethereum Virtual Machine (EVM) is a Turing-complete virtual machine at the core of the Ethereum blockchain. It executes smart contracts and decentralized applications by processing bytecode instructions. Operating on a distributed network, the EVM ensures secure and deterministic execution of contracts across the Ethereum ecosystem, enabling the creation of diverse and innovative applications. It is not only sandboxed but actually completely isolated, which means that code running inside the EVM has no access to network, filesystem or other processes. Smart contracts even have limited access to other smart contracts [5].

Ethereum's state is a large data structure which holds not only all accounts and balances, but a machine state, which can change from block to block according to a pre-defined set of rules, and which can execute arbitrary machine code. The specific rules of changing state from block to block are defined by the EVM [6].

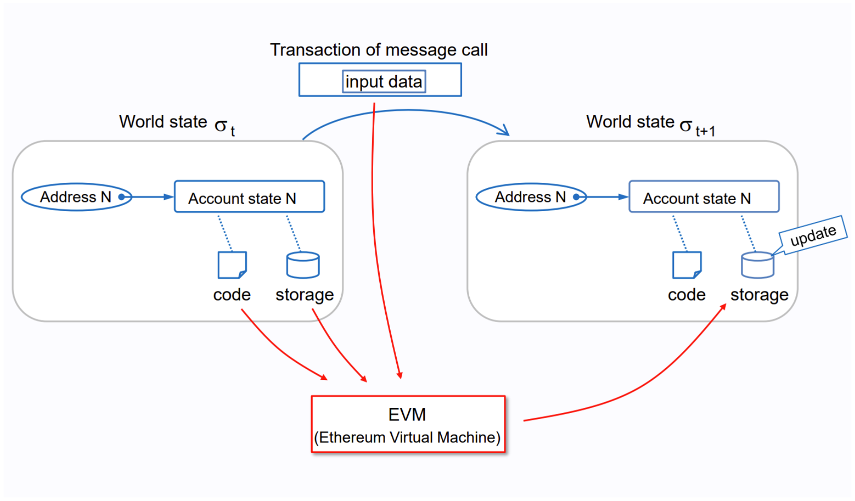


Figure 1: Ethereum EVM illustrated [6]

2.2.2 Scalability issue (Layer 2 and sidechain solutions)

The escalating demand for blockchain technology has highlighted a critical challenge: the inherent trade-off between decentralization and scalability. As blockchain networks strive to maintain their decentralized nature, the performance limitations of their consensus mechanisms become increasingly apparent, hindering their ability to handle a high volume of transactions efficiently. In response to this challenge, Layer 2 solutions have emerged as a promising avenue to address the scalability issue while preserving the core principles of blockchain technology. These solutions offer innovative approaches that enable off-chain transaction processing, while still leveraging the security and finality of the underlying blockchain. In this section, we delve into the realm of Layer 2 solutions, exploring their mechanisms, benefits, and implications for enhancing blockchain scalability without compromising decentralization and security.

Layer 2 blockchain solutions involve building supplementary protocols or networks on top of an existing blockchain to enable off-chain transaction processing and data computation. It is easy to understand that the main concept of all Layer 2 solutions is that of lightening the blockchain, in order to help in scaling up. The consequence of this idea is not only higher transaction speed, but also (in general) lower fees (which is a direct consequence of increased TPS): this means that L2 solutions are appropriate places for micropayments to be performed [7].

The following are the different types of high-level Layer 2 solutions [8]:

- Rollups: Rollups perform transaction execution outside layer 1 and then the data is posted to layer 1 where consensus is reached. As transaction data is included in layer 1 blocks, this allows rollups to be secured by native Ethereum security.

- State channels: State channels utilize multisig contracts to enable participants to transact quickly and freely off-chain, then settle finality with Mainnet. This minimizes network congestion, fees, and delays. The two types of channels are currently state channels and payment channels.

Additionally, there are scalability solutions that depend on their own security and consensus mechanism from the main blockchain. Those solutions are called sidechains. A sidechain is an independent EVM-compatible blockchain which runs in parallel to Mainnet. These are compatible with Ethereum via two-way bridges, and run under their own chosen rules of consensus, and block parameters [8]. Unlike layer 2 scaling solutions, sidechains do not post state changes and transaction data back to Ethereum Mainnet [9].

2.3 Smart contracts

The code in Ethereum contracts is written in a low-level, stack-based bytecode language, referred to as "Ethereum virtual machine code" or "EVM code". The code consists of a series of bytes, where each byte represents an operation. The operations have access to three types of space in which to store data [3]:

- The **stack**, a last-in-first-out container to which 32-byte values can be pushed and popped
- **Memory**, an infinitely expandable byte array
- The contract's long-term **storage**, a key/value store where keys and values are both 32 bytes. Unlike stack and memory, which reset after computation ends, storage persists for the long term.

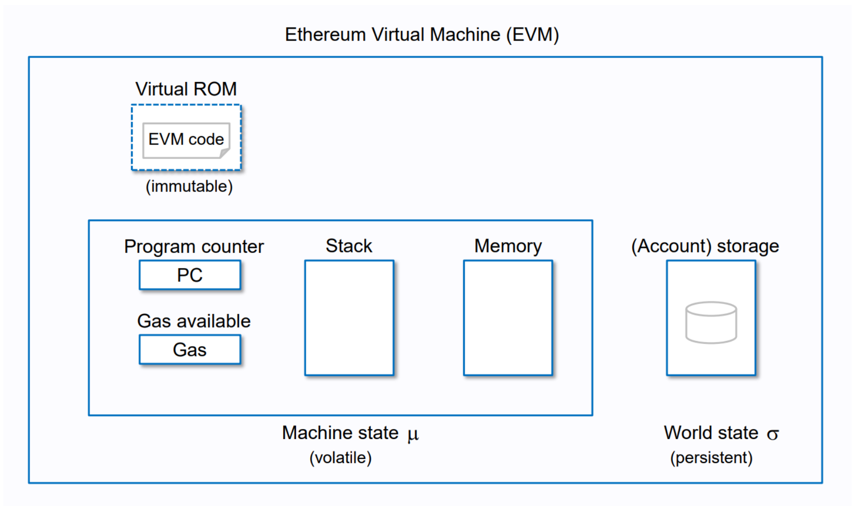


Figure 2: EVM simple stack-based architecture [6]

There are several widely used programming languages for smart contracts. In this paper we are going to consider 2 of them, namely *Solidity* and *Vyper*. Both languages are compiled into bytecode (EVM code) that is stored in the blockchain.

2.3.1 Solidity

Solidity is an object-oriented, high-level language for implementing smart contracts. Solidity is a curly-bracket language that has been influenced and inspired by several well-known programming languages. Solidity is most profoundly influenced by C++, but also borrowed concepts from languages like Python, JavaScript, and others. Solidity is statically typed, supports inheritance, libraries, and complex user-defined types, among other features [5].

2.3.2 Vyper

Vyper is a contract-oriented, pythonic programming language that targets the Ethereum Virtual Machine (EVM). The main three principles and goals of this language are the following [10]:

- Security: It should be possible and natural to build secure smart-contracts in Vyper.
- Language and compiler simplicity: The language and the compiler implementation should strive to be simple.

- Auditability: Vyper code should be maximally human-readable. Furthermore, it should be maximally difficult to write misleading code. Simplicity for the reader is more important than simplicity for the writer, and simplicity for readers with low prior experience with Vyper (and low prior experience with programming in general) is particularly important.

. For these reasons it could be considered as a more intuitive and secure solution, however, it brings following limitations [10]:

- No modifiers
- No class ingeritance
- No recursive calling or infinite loops
- No function or operator overloading

2.3.3 Oracle Problem

In the context of blockchain implementations, the oracle problem refers to the challenge of obtaining external, real-world data and securely integrating it into the blockchain ecosystem. Blockchains are inherently self-contained and deterministic, meaning that their consensus mechanisms rely on information within the blockchain itself to validate transactions and execute smart contracts.

Nick Szabo, who popularized the term "smart contract", gave as an example a smart contract that enforces car loan payments. If the owner of the car fails to make a timely payment, a smart contract could programmatically revoke physical access and return control of the car to the bank. As Szabo's example shows, however, the most compelling applications of smart contracts such as financial instruments—additionally require access to data about real-world state and events [11].

This is where the oracle problem arises. Oracles are entities or services that fetch external data and provide it to smart contracts within the blockchain. The issue lies in ensuring the accuracy, security, and reliability of the data being fed into the blockchain, as it directly affects the integrity of smart contracts and the outcomes of decentralized applications. If the data provided by oracles is inaccurate or manipulated, it can lead to incorrect smart contract execution and undesirable outcomes.

The lack of trustworthy oracles is often referred to as critical obstacle to the evolution of Ethereum and decentralized smart contracts in general [11].

The following is an example structure of an oracle framework. The framework in figure 3 is to be read from left to right, following the natural flow of data from external sources to the blockchain [12].

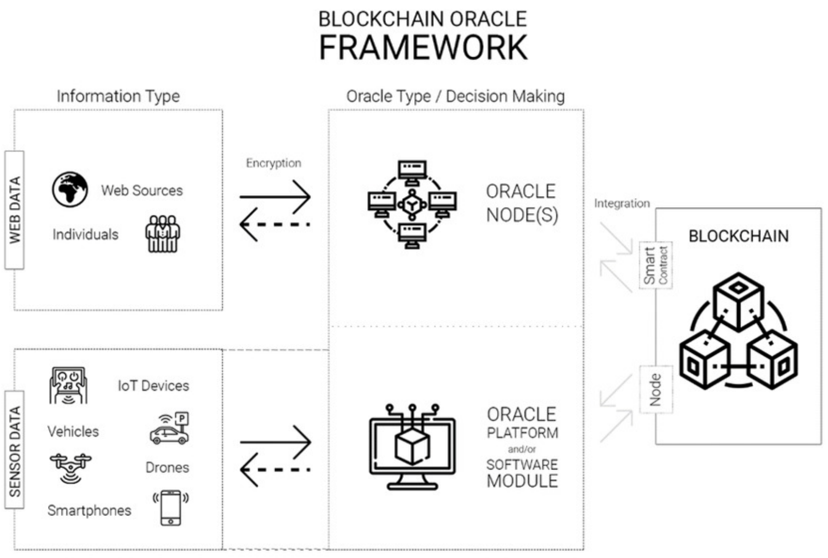


Figure 3: A graphical representation of blockchain oracle framework [12]

3 Related Work

We conducted an extensive review of existing literature and practical implementations that intersect the realms of workflow automation and blockchain smart contracts. The primary objective was to comprehensively understand the contemporary landscape of research pertinent to our subject matter. This section provides a consolidated overview of the literature and applications deemed significant within this context.

Security is one of the less developed and stable parts of the blockchain smart contracts, since there are many vulnerabilities found even in the compilers of the languages. An example of such vulnerability is Vyper compiler, that made smart contracts vulnerable to reentrancy attacks [13]. Most smart contracts need audits to be used safely by users of decentralized applications (Dapps). To address this issue and help inexperienced developers deploy contracts and use them for business process management and specifically inter-organizational process collaboration, Xiong et al. [14], have proposed a solution that generates smart contracts using Business Process Model and Notation (BPMN) models, commonly used for defining process models. The mentioned proposal mainly focuses on message communication, since the BPMN models are translated first into Communicating Sequential Programs (CSP#) and only then into smart contracts. This is done so that unverified BPMN models are not used for smart contract generation, hence creating deadlocks or other vulnerable implementations of smart contracts [14].

Another tool that tackles previously mentioned issue by automating the generation of Solidity smart contract code is called Lorikeet [15]. Lorikeet consists of the following components:

- **BPMN translator**, that translates BPMN diagrams into Solidity code.
- **Registry generator**, that generates registry data schema in Solidity code.
- **Blockchain trigger**, that handles deployment and interaction with blockchain smart contracts.

These components of Lorikeet are designed as micro-services and deployed in individual Docker containers. Lorikeet, basically, generates a set of smart contracts from a business process and uses them to manage the business process. This makes Lorikeet a good tool for managing business processes on-chain [15]. The following figure (Figure: 4) is a visual representation of Lorikeet's architecture.

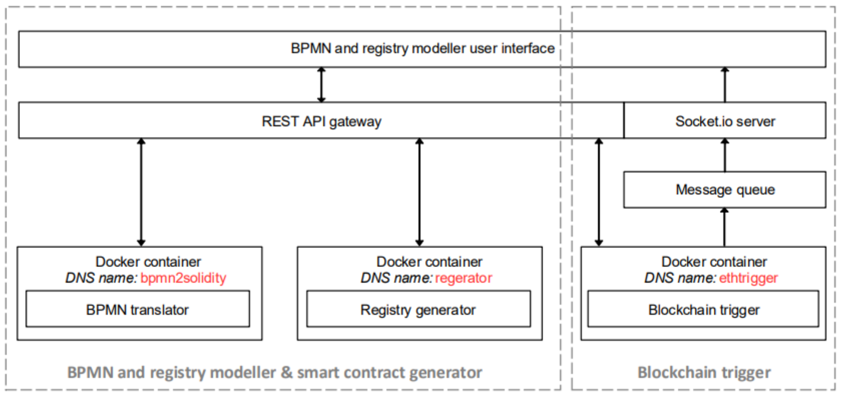


Figure 4: Lorikeet Architecture

A tool with similar functionality that has also been mentioned in [15], is called Caterpillar. Caterpillar is BPMN execution engine on Ethereum. The objective of Caterpillar is to offer users a platform to create blockchain applications to ensure that collaborative business processes that are created using the BPMN notation are carried out appropriately. To accomplish this, Caterpillar also converts BPMN models into smart contracts. In contrast to other applications and literature, Caterpillar does not rely on message exchanges between participants to provide coordination. It is assumed that they solely rely on the blockchain for coordination. Additionally, Caterpillar keeps track of each process instance's and its subprocesses' status on the blockchain [16]. The following figure (Figure 5) is a visual representation of Caterpillar's architecture.

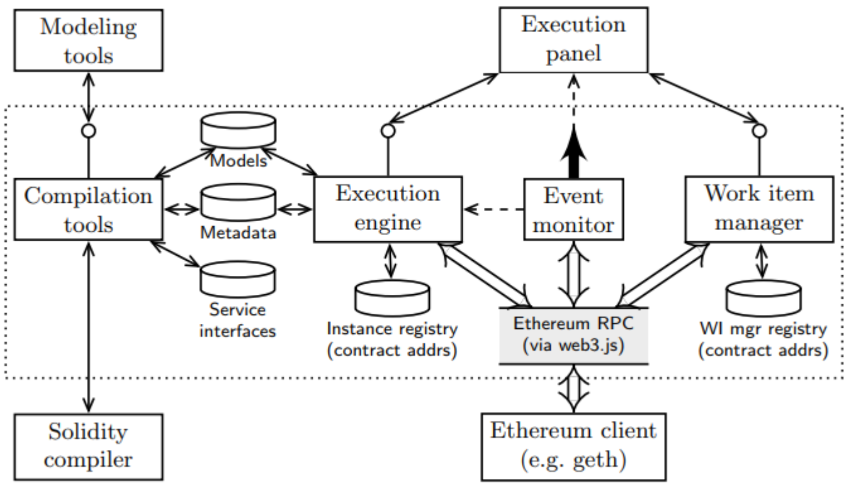


Figure 5: Caterpillar Architecture [16]

Moreover, alternative architecture proposal to tackle the connection between the paradigms of business process management and blockchain smart contracts, has been suggested by Emre Ozaras [17], where the architecture involved message passing between different components using different interface such as:

- **Zeebe gRPC** for communication with workers
- **JSON-RPC** for communication with blockchain smart contracts

A visual representation of the suggested architecture can be found in the following Figure 6.

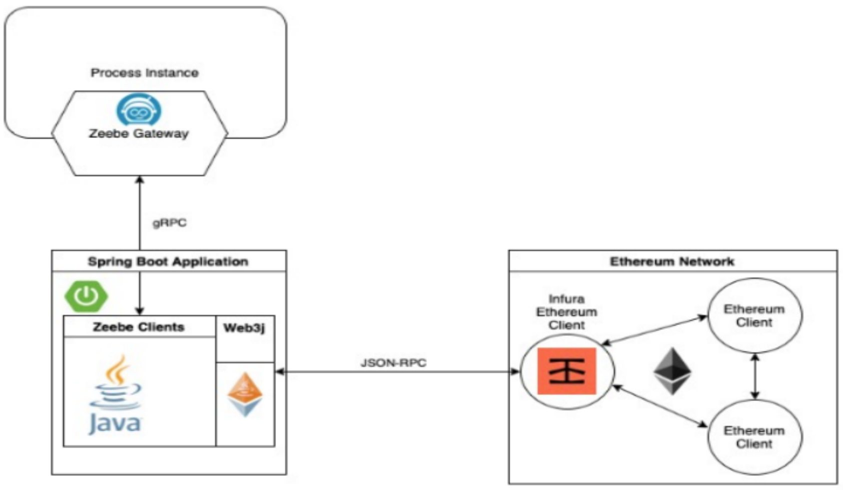


Figure 6: Architecture of a solution proposed by Emre Ozaras [17]

In the suggested architecture, all of the business logic is written in Java, and for the creation of a new instance of a new type of workflow it is required to write a smart contract for it, possibly using one of the previously mentioned tools.

Additionally, Emre Ozaras [18], has suggested a use of Service Oriented Architecture (SOA) for a more modular and generic solution. This architecture involves a set of services and an Enterprise Service Bus (ESB). The visual representation of the architecture can be seen in the Figure 7.

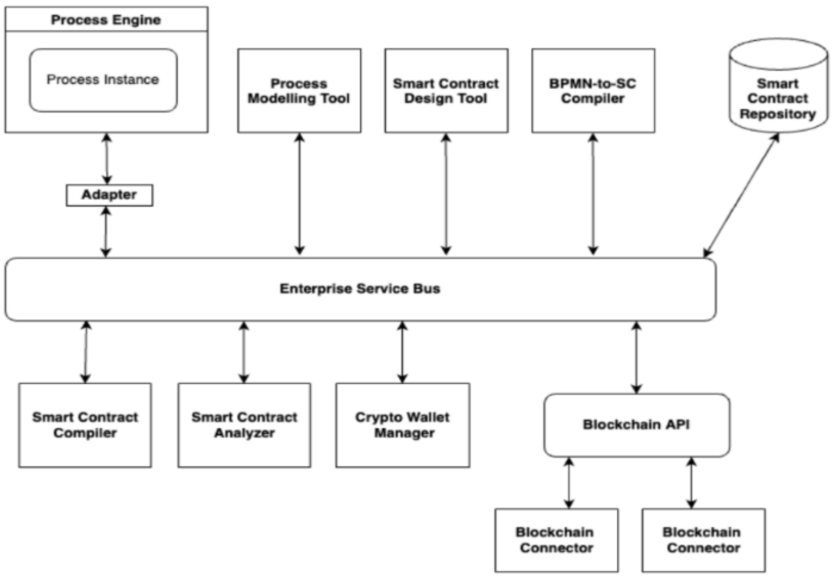


Figure 7: Proposal for a more modular and generic solution [18]

Finally, in both studies of Emre Ozaras [17] [18], it has been acknowledged that there are other possible solutions where there exists a logical separation between type level data and instance level data with a link between them [18].

4 Technologies Used

4.1 Camunda Platform 8

Camunda Platform 8 is an open-source process automation and workflow management software that provides tools for designing, modeling, executing, and monitoring business processes. It allows organizations to define their processes using BPMN (Business Process Model and Notation) diagrams and deploy them for automated execution. Camunda Platform 8 offers features like process execution orchestration, task management, integration with external systems, and real-time monitoring, making it a comprehensive solution for optimizing and automating business workflows.

The following Figure 8 is a visual representation of Zeebe architecture provided by Camunda Platform 8 documentation [19]

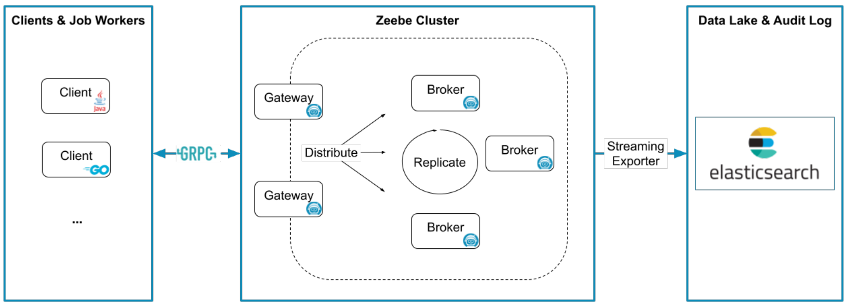


Figure 8: Zeebe's Architecture [19]

4.1.1 Camunda Zeebe Broker

Camunda Broker is a component of the Camunda Platform that serves as the central messaging system for communication between workflow instances and external service tasks. It enables decoupling the execution of process instances from the execution of service tasks, making the architecture more scalable and resilient. Camunda Broker ensures reliable message delivery and supports features like task retries, timeouts, and dead-letter queues, enhancing the overall robustness of workflow executions.

4.1.2 Camunda Zeebe Operate

Camunda Operate is a monitoring and operational management tool for Camunda Platform workflows. It provides a user-friendly interface for tracking the execution status of workflow instances, visualizing process instances, and identifying bottlenecks or issues in real-time.

4.1.3 PyZeebe

PyZeebe is a Python gRPC client library that provides an interface for interacting with the Zeebe process engine. PyZeebe allows developers to connect to Zeebe instances through a Zeebe Gateway, deploy BPMN workflows, start and manage process instances, and handle task interactions within workflows using Python code. It simplifies the integration of Zeebe-based workflow automation into Python applications.

4.2 Web3.py

Web3.py is a Python library that provides a convenient interface for interacting with the Ethereum blockchain. It allows developers to interact with Ethereum nodes, send transactions, query blockchain data, and interact with smart contracts using Python code.

4.3 Truffle

Truffle is a blockchain development environment that is used for testing, deploying, and debugging of smart contracts. This framework works exclusively for smart contract for EVM based blockchains and can be configured to deploy smart contracts on all EVM-based blockchains.

Ganache is a part of the Truffle framework that allows a user to run a local blockchain for testing of deployment and behaviour of deployed smart contracts.

4.4 IPFS

InterPlanetary File System or shortly IPFS is a decentralized distributed file system that uses peer-to-peer technology to store and uniquely identify content that allows hosts of each node to store and share information.

The following Figure 9 shows comparison of client-server model and peer-to-peer model of networks.

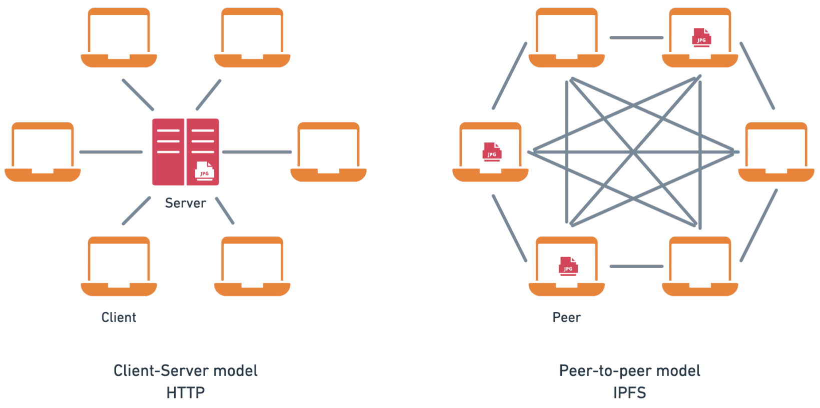


Figure 9: Comparison of Client-Server Model using HTTP and Peer-to-peer model using IPFS [20]

4.4.1 How it works?

- When you add a file to IPFS, your file is split into smaller chunks, cryptographically hashed, and given a unique fingerprint called a content identifier (CID). This CID acts as a permanent record of your file as it exists at that point in time.

- When other nodes look up your file, they ask their peer nodes who's storing the content referenced by the file's CID. When they view or download your file, they cache a copy — and become another provider of your content until their cache is cleared.

- A node can pin content in order to keep (and provide) it forever, or discard content it hasn't used in a while to save space. This means each node in the network stores only content it is interested in, plus some indexing information that helps figure out which node is storing what.

- If you add a new version of your file to IPFS, its cryptographic hash is different, and so it gets a new CID. This means files stored on IPFS are resistant to tampering and censorship — any changes to a file don't overwrite the original, and common chunks across files can be reused in order to minimize storage costs.

5 Implementation

In this section we will describe and show proposed solution architecture of a middleware divided into several components. Due to high complexity of the model we decided to take modularized approach in order to ensure scalability, maintainability, and flexibility. The proposed solution architecture consists of several interconnected components that collectively enable the integration of blockchain-based smart contracts with the Camunda Platform 8 for workflow automation. The code snippets presented for each module in the following sections may not always encompass the full functional logic, as including such comprehensive details would result in excessive length.

5.1 Smart contracts

The smart contracts provided in this section are deployed from a specific configured address on blockchain that holds authority over functions that change the state variables of the contracts, meaning that other wallet addresses will not be able to corrupt the data of workflow types and their instances. Moreover, the data stored in the smart contracts are available to any address in the blockchain.

5.1.1 Workflow registry contract

Workflow registry contract manages and maintains a registry of different workflow types and their corresponding smart contracts. It provides functions for registering, updating, and deleting workflow types, as well as retrieving the address of a specific workflow's smart contract.

The following Listing 1 provides the smart contract implementation for Workflow Registry

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract WorkflowRegistry {
    struct WorkflowType {
        string name;
        address smartContract;
    }

    mapping(string => WorkflowType) private workflowTypes;

    event WorkflowTypeRegistered(string name, address smartContract);
    event WorkflowTypeUpdated(string name, address oldSmartContract, address newSmartContract);
    event WorkflowTypeDeleted(string name, address smartContract);

    modifier onlyOwner() {
        require(msg.sender == owner, "Only␣the␣contract␣owner␣can␣call␣this␣function");
        _;
    }

    address private owner;

    constructor() {
        owner = msg.sender;
    }

    function registerWorkflowType(string calldata name, address smartContract) external onlyOwner {
        require(smartContract != address(0), "Invalid␣smart␣contract␣address");
        require(bytes(name).length > 0, "Invalid␣workflow␣name");
        require(workflowTypes[name].smartContract == address(0), "Workflow␣type␣already␣registered");

        workflowTypes[name] = WorkflowType(name, smartContract);
        emit WorkflowTypeRegistered(name, smartContract);
    }

    function updateWorkflowType(string calldata name, address newSmartContract) external onlyOwner {
        require(bytes(name).length > 0, "Invalid␣workflow␣name");
        require(newSmartContract != address(0), "Invalid␣smart␣contract␣address");
        require(workflowTypes[name].smartContract != address(0), "Workflow␣type␣not␣registered");

        address oldSmartContract = workflowTypes[name].smartContract;
        workflowTypes[name].smartContract = newSmartContract;
        emit WorkflowTypeUpdated(name, oldSmartContract, newSmartContract);
    }

    function deleteWorkflowType(string calldata name) external onlyOwner {
        require(bytes(name).length > 0, "Invalid␣workflow␣name");
        require(workflowTypes[name].smartContract != address(0), "Workflow␣type␣not␣registered");

        address smartContract = workflowTypes[name].smartContract;
        delete workflowTypes[name];
        emit WorkflowTypeDeleted(name, smartContract);
    }

    function getSmartContractAddress(string calldata name) external view returns (address) {
        return workflowTypes[name].smartContract;
    }
}
```

Listing 1: Workflow registry contract

5.1.2 Workflow instance contract

Workflow instance contract manages and tracks the data about instances of a specific workflow type. It stores data about the BPMN model used and the task implementation by means of storing IPFS hash of the directory with the files. Additionally, this contract maps the event log and process status to process instance identifier.

The following Listing 2 provides the smart contract implementation for Workflow Instance

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract WorkflowInstance {
    address public owner;
    string public ipfsHash;
    mapping(uint256 => string) public eventLog;
    mapping(uint256 => string) private processStatus;

    modifier onlyOwner() {
        require(msg.sender == owner, "Only the owner can call this function"
            );
        _;
    }

    constructor() {
        owner = msg.sender;
    }

    function setIpfsHash(string memory _hash) public onlyOwner {
        ipfsHash = _hash;
    }

    function addEventLog(uint256 _processId, string memory _eventLogHash)
        public onlyOwner {
        eventLog[_processId] = _eventLogHash;
    }

    function setProcessStatus(uint256 _processId, string memory _status)
        public onlyOwner {
        processStatus[_processId] = _status;
    }

    function getProcessStatus(uint256 _processId) public view returns (
        string memory) {
        return processStatus[_processId];
    }
}
```

Listing 2: Workflow instance contract

5.2 Middleware

Here include general explanation of middleware (with diagrams).

5.2.1 Core Client

This class combines Zeebe client and IPFS client into one class and carries functionality of data uploading and data downloading from IPFS using provided content identifiers (hash) and, additionally, it has functionality of process deployment and instantiation as well as creation of dynamic workers that inherit task handlers from specified files.

In the following Listing 3 we can see Core Client class definition with constructor and all methods excluding logic withing each method.

```python
from pyzeebe import ZeebeClient as PyZeebeClient
from zeebe_channel import Channel
from init_worker import init_worker
import ipfshttpclient
import asyncio
import os
import importlib.util
import types
from registry_oracle import RegistryOracle
from instance_oracle import InstanceOracle
import contract_data_config


class CoreClient:
    def __init__(self, process_dir_name, hostname=None, port=None):
        self.process_dir_name = os.path.abspath(
            f"src/processes/{process_dir_name}"
        )
        self.channel = Channel(hostname=hostname, port=port).channel
        self.zeebe_client = PyZeebeClient(self.channel)

        try:
            self.ipfs_client = ipfshttpclient.connect()
        except Exception as e:
            self.ipfs_client = None
            print(f"Error:␣Couldn’t␣connect␣to␣the␣IPFS␣HTTP␣client:\n{e}\n\
                n")

        self.instance_oracle = InstanceOracle() # Oracle config data
        self.registry_oracle = RegistryOracle() # Oracle config data

    async def deploy_bpmn(self, bpmn_file_path):
        """
        Deploys a process from a BPMN model (.bpmn file)
        """

    async def create_task_instance(self, bpmn_process_id, variables=None):
        """
        Runs a process instance
        """

    async def create_and_run_worker(self, task_handler_file_name):
        """
        Creates a worker for the process instance and runs it until it
            completes all tasks
        """

    def add_task_handlers_to_worker(class_instance, file_path):
        """
        Adds all task handlers to the worker instance from a specified path
        """

    def persist_to_ipfs(self):
        """
        Adds a directory "src/processes/{Client.process_dir_name}" to IPFS
            and returns a hash
        """

    def download_from_ipfs(self, ipfs_hash):
        """
        Downloads directory with files from IPFS and store it in the
            directory of the "src/processes/{Client.process_dir_name}"
            folder
        """
```

Listing 3: Core Client class

5.2.2 Web3 utility

Web3 utility is a set of classes that provide us with an interface of connecting to blockchain of a choice using an HTTP provider, and interaction with smart contract providing its address and ABI (Application Binary Interface)

In the following Listing 4 5 we can see definition of two interface classes

```python
from web3.middleware import geth_poa_middleware
from web3 import Web3


class Web3Connector:
    def __init__(self, provider_url: str):
        self.provider_url = provider_url
        self.connect_to_client()

    def connect_to_client(self):
        try:
            self.client = Web3(Web3.HTTPProvider(self.provider_url))
            self.client.middleware_onion.inject(geth_poa_middleware, layer
                =0)
        except Exception:
            raise Exception("Couldn't␣connect␣to:␣" + self.provider_url)

    def is_connected(self):
        return self.client.isConnected()

    def get_client(self):
        return self.client
```

Listing 4: Web3Connector class

```python
import json
from web3 import Web3


class Web3Contract:
    def __init__(self, client: Web3, contract_address: str, abi: json):
        self.client = client
        self.contract_address = Web3.toChecksumAddress(contract_address)
        self.abi = abi
        self.contract = self.client.eth.contract(
            address=self.contract_address, abi=self.abi
        )

    def get_contract(self, pool_address=None, abi=None):
        if pool_address == None and abi == None:
            return self.contract
        else:
            return self.client.eth.contract(address=pool_address, abi=abi)

    def has_function(self, function_name):
        if self.contract.find_functions_by_name(function_name) == []:
            return False
        else:
            return True

    def read_call(self, function_name, params=[], contract_address=None):
        '''
        Performs read contract method call and returns the output
        '''

    def write_call(self, function_name, params=[], private_key=None,
        caller_address=None, contract_address=None):
        '''
        Performs write contract method call and returns the output
        '''
```

Listing 5: Web3Contract class

5.2.3 Oracle

Oracle class serves as a connector to interact with smart contracts deployed on a blockchain network. Upon initialization, it establishes a connection to the specified smart contract using the Web3 utility of the project.

In the following Listing 6 we can see Oracle class definition with constructor and all methods excluding logic withing each method.

```python
from web3_utils.web3_connector import Web3Connector
from web3_utils.web3_contract import Web3Contract


class Oracle:
    def __init__(self, address, abi, provider_url):
        self.address = address
        self.abi = abi
        self.provider_url = provider_url
        self.contract = self.connect_to_oracle_contract()

    def connect_to_oracle_contract(self):
        client = Web3Connector(provider_url=self.provider_url).get_client()
        contract = Web3Contract(client, self.address, self.abi)

        return contract
```

Listing 6: Oracle class

5.2.4 Instance Oracle

Instance Oracle class is a class that inherits Oracle class and is responsible of interacting with Workflow Instance smart contract and provides us with functionality to call any methods from the contract

```python
from web3_utils.web3_connector import Web3Connector
from web3_utils.web3_contract import Web3Contract
from oracle import Oracle

class InstanceOracle(Oracle):
    def __init__(self, address, abi, provider_url):
        super().__init__(address, abi, provider_url)

    def get_ipfs_hash(self):
    # Function logic

    def get_process_event_log_hash(self, process_id):
    # Function logic

    def set_process_event_log_hash(self, process_id, event_log_hash,
        private_key, caller_address):
    # Function logic

    def set_ipfs_hash(self, ipfs_hash, private_key, caller_address):
    # Function logic

    def set_process_status(self, process_id, status, private_key,
        caller_address):
    # Function logic
```

Listing 7: Instance Oracle class

5.2.5 Registry Oracle

Registry Oracle class is a class that inherits Oracle class and is responsible of interacting with Workflow Registry smart contract and provides us with functionality to call any methods from the contract

```python
from web3_utils.web3_connector import Web3Connector
from web3_utils.web3_contract import Web3Contract
from oracle import Oracle

class RegistryOracle(Oracle):
    def __init__(self, address, abi, provider_url):
        super().__init__(address, abi, provider_url)

    def workflow_type_exists(self, workflow_type):
        # Function logic

    def register_workflow_type(self, name, smart_contract_address,
       private_key, caller_address):
        # Function logic

    def update_workflow_type(self, name, new_smart_contract_address,
       private_key, caller_address):
        # Function logic

    def delete_workflow_type(self, name, private_key, caller_address):
        # Function logic

    def get_address_of_instance(self, name):
        # Function logic
```

Listing 8: Registry Oracle class

5.2.6 Dynamic Zeebe Worker

Dynamic Zeebe worker is a class that dynamically imports task handling modules and wraps them with the Zeebe worker decorator from PyZeebe that lets it log the events of the task execution.

```python
import asyncio
import importlib.util
from init_worker import worker

def get_dynamic_worker(worker_path, worker_name):
    # Load the module using the path
    module_name = worker_name
    spec = importlib.util.spec_from_file_location(module_name, worker_path +
        f"/{module_name}")
    my_module = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(my_module)

    MyWorker = my_module.Worker
    class DynamicWorker(MyWorker):
        async def run(self):
            await worker.work()

    return DynamicWorker
```

Listing 9: Dynamic Zeebe worker

```python
    from pyzeebe import ZeebeWorker

    worker = None

    def init_worker(channel):
        global worker
        worker = ZeebeWorker(channel)
```

Listing 10: Zeebe worker from PyZeebe

5.2.7 Task Handler

Task handler is a module that consists of task implementation for specified task types of certain workflow instances. An example of a task handler can be seen in the following Listing 11

```python
    from init_worker import worker
    from pyzeebe import Job


    class Worker:
        def decorator(job: Job):
            with open("event_log.txt", "a") as log_file:
                log_file.write(str(job) + "\n")
            return job

        @worker.task(task_type="example_task_type", after=[decorator])
        async def task_handler(taskData):
            return {"output": f"Example␣output␣using:␣{taskData}"}
```

Listing 11: Example task handler implementation

5.3 Solution Architecture

In this section we will provide a diagram of the proposed solution architecture implemented in this project with the use cases of the core component communicating with its components.

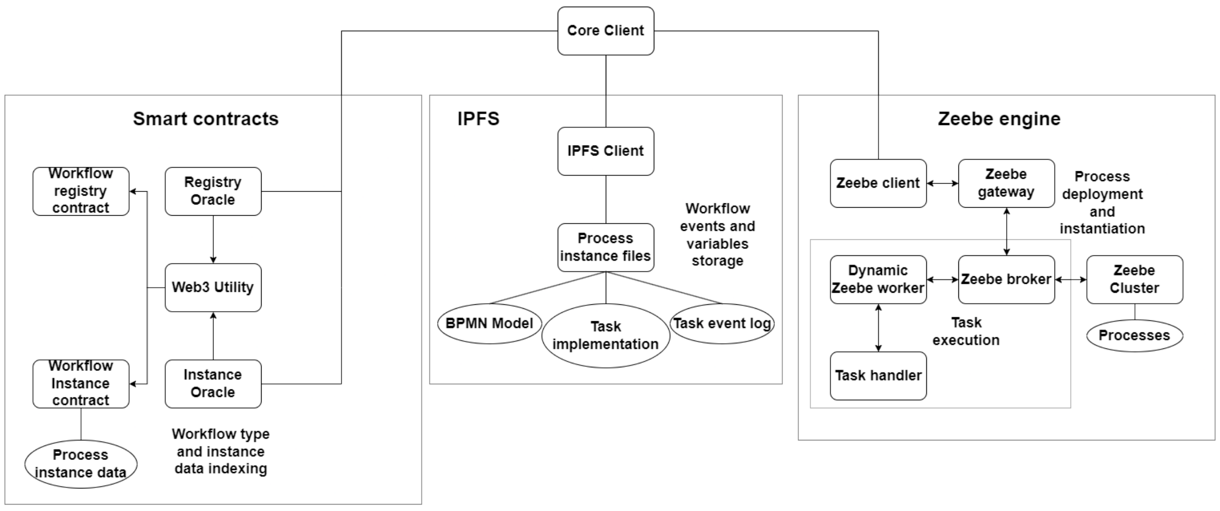


Figure 10: Solution Architecture

5.3.1 Use Case: Core module interaction with IPFS client

In the following sequence diagrams we present the process of uploading and downloading data in and from the IPFS network.

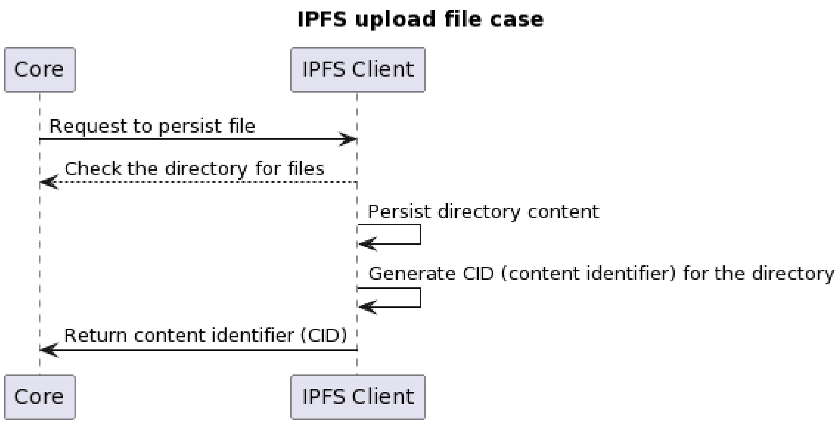

IPFS upload file case
Core
IPFS Client
Request to persist file
Check the directory for files
Persist directory content
Generate CID (content identifier) for the directory
Return content identifier (CID)
Core
IPFS Client

Figure 11: Core uploading into IPFS

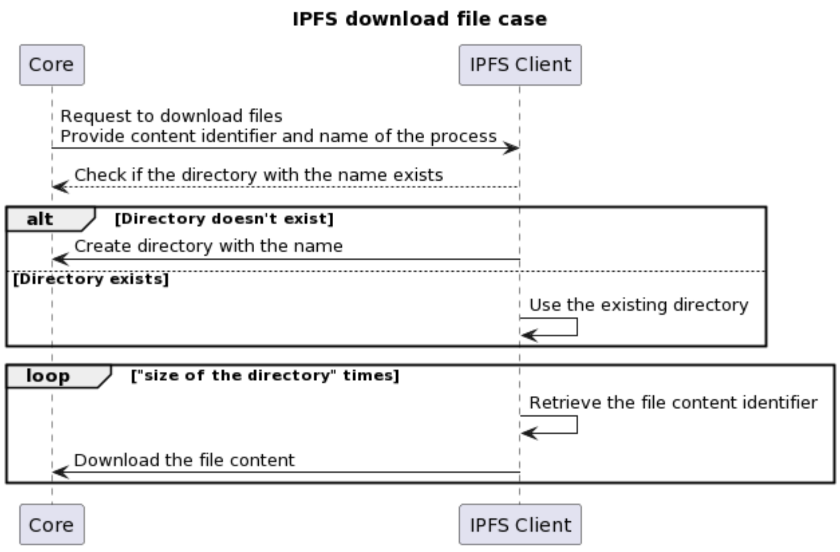

IPFS download file case
Core
IPFS Client
Request to download files
Provide content identifier and name of the process
Check if the directory with the name exists
alt
[Directory doesn't exist]
Create directory with the name
[Directory exists]
Use the existing directory
loop
["size of the directory" times]
Retrieve the file content identifier
Download the file content
Core
IPFS Client

Figure 12: Core downloading from IPFS

5.3.2 Use Case: Process instance deployment

In the following sequence diagrams we present the process of process deployment, where the core module of the middleware first fetches all the necessary data from the blockchain module and then deploys the process using Zeebe module

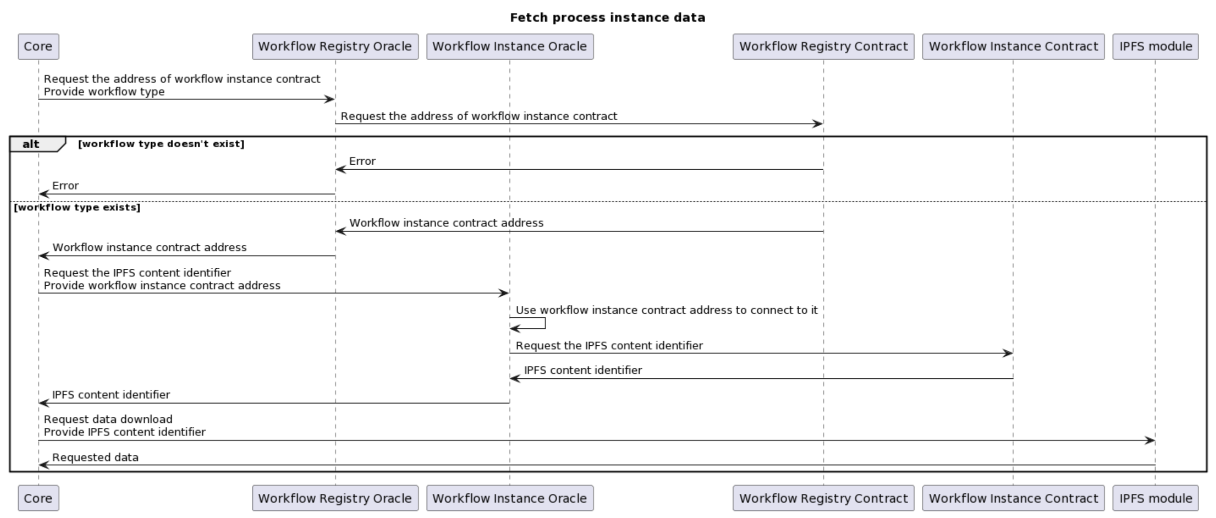

Fetch process instance data
Core
Workflow Registry Oracle
Workflow Instance Oracle
Workflow Registry Contract
Workflow Instance Contract
IPFS module
Request the address of workflow instance contract
Provide workflow type
Request the address of workflow instance contract
alt [workflow type doesn't exist]
Error
Error
[workflow type exists]
Workflow instance contract address
Workflow instance contract address
Request the IPFS content identifier
Provide workflow instance contract address
Use workflow instance contract address to connect to it
Request the IPFS content identifier
IPFS content identifier
IPFS content identifier
Request data download
Provide IPFS content identifier
Requested data

Figure 13: Fetch process instance sequence

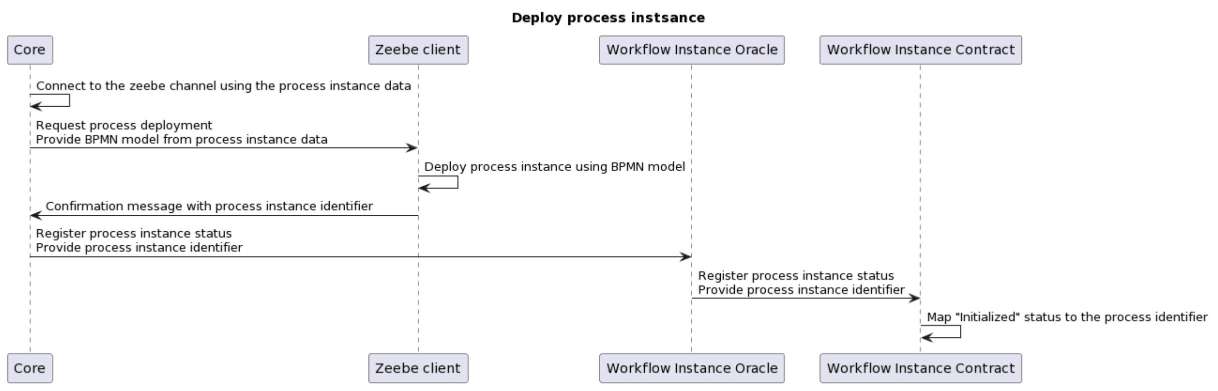

Deploy process instsance
Core
Zeebe client
Workflow Instance Oracle
Workflow Instance Contract
Connect to the zeebe channel using the process instance data
Request process deployment
Provide BPMN model from process instance data
Deploy process instance using BPMN model
Confirmation message with process instance identifier
Register process instance status
Provide process instance identifier
Register process instance status
Provide process instance identifier
Map "Initialized" status to the process identifier

Figure 14: Deploy process instance sequence

5.3.3 Use Case: Process instance execution

In the following sequence diagram we present the process of process instance execution, where the core module uses the deployed process instance and process instance data for task handling and then creates and runs the worker for the instance. Additionally, during the execution of each task the core component logs each step of the process and persists everything to the blokchain module through IPFS module.

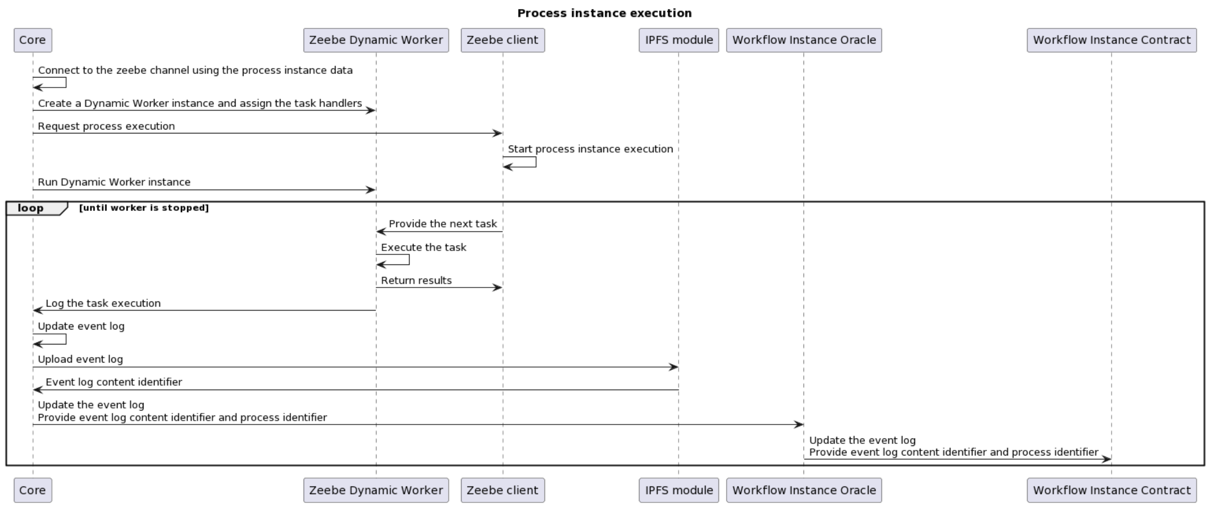

Process instance execution
Core
Zeebe Dynamic Worker
Zeebe client
IPFS module
Workflow Instance Oracle
Workflow Instance Contract
Connect to the zeebe channel using the process instance data
Create a Dynamic Worker instance and assign the task handlers
Request process execution
Start process instance execution
Run Dynamic Worker instance
loop
[until worker is stopped]
Provide the next task
Execute the task
Return results
Log the task execution
Update event log
Upload event log
Event log content identifier
Update the event log
Provide event log content identifier and process identifier
Update the event log
Provide event log content identifier and process identifier

Figure 15: Process instance execution

5.3.4 Use Case: Registration of workflow type

In the following sequence diagram we present the process of registration of a new workflow type.

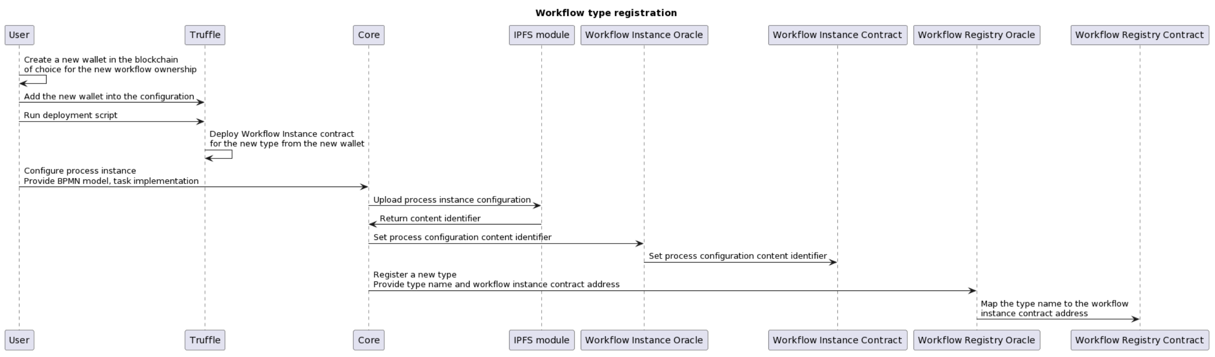

Workflow type registration
User
Truffle
Core
IPFS module
Workflow Instance Oracle
Workflow Instance Contract
Workflow Registry Oracle
Workflow Registry Contract
Create a new wallet in the blockchain
of choice for the new workflow ownership
Add the new wallet into the configuration
Run deployment script
Deploy Workflow Instance contract
for the new type from the new wallet
Configure process instance
Provide BPMN model, task implementation
Upload process instance configuration
Return content identifier
Set process configuration content identifier
Set process configuration content identifier
Register a new type
Provide type name and workflow instance contract address
Map the type name to the workflow
instance contract address

Figure 16: Workflow type registraion

5.4 Evaluation

In this section we will go through an example workflow in order to provide a proof of concept for our proposed solution.

5.4.1 Example: Employee Onboarding Workflow

Employee Onboarding is a crucial process for successfully integrating new employees into an organization. Automating this process brings numerous benefits, including efficiency, consistency, compliance, engagement, and cost savings. Given its impact on retention, productivity, and company culture, onboarding is an essential practice for all companies aiming to attract, retain, and develop their workforce.

The process flow can be explained in the following steps:

- Start Event: Gather essential data about the new employee, including their name, surname, date of birth, years of experience, profession, position, and level of education.

- Register a company Email:

  - Using the provided name and surname, generate a company email address in the format Example: `name_surname@company.com`.

- Assign Authorization Level:
  - Depending on the employee's position (Junior, Mid-level, Senior), assign an appropriate authorization level (3, 2, 1) for access to company resources.

- Provide Courses to Follow:
  - Based on the employee's profession (Accountant or Software Developer), recommend relevant courses for them to follow.

- Provide Software License:
  - Depending on both profession and position, grant appropriate software licenses:
    * Accountants: Receive licenses for tools like Microsoft Excel, PowerBI, or NetSuite.
    * Software Developers: Receive licenses for tools like Visual Studio, JetBrains, or Jenkins.

- Compute Salary:
  - Considering the employee's profession, position, level of education, and years of experience, calculate a salary using a specific formula. The outcome reflects the unique attributes of the employee's profile.

The following Figure 17 is a BPMN diagram used for the above-mentioned workflow.

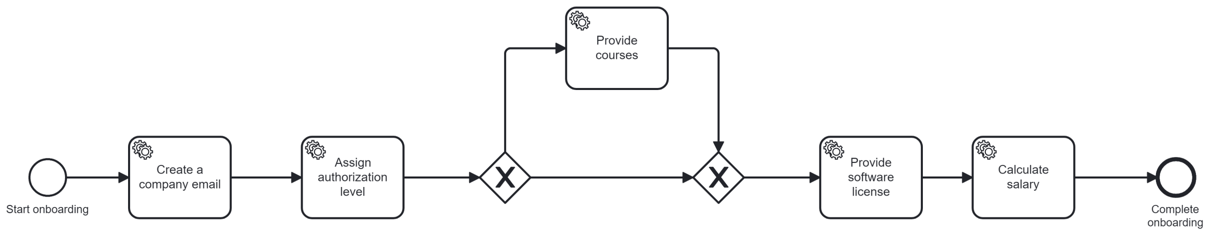


Figure 17: BPMN Model of Employee Onboarding process

With the following task implementation logic (Listing 12):

```python
from init_worker import worker
from pyzeebe import Job


class Worker:
    def decorator(job: Job):
        with open("event_log.txt", "a") as log_file:
            log_file.write(str(job) + "\n")
        return job

    @worker.task(task_type="create_email", after=[decorator])
    async def create_email(first_name, last_name):
        return {"email": f"{first_name.lower()}_{last_name.lower()}@organization
            .org"}

    @worker.task(task_type="assign_auth_level", after=[decorator])
    async def assign_auth_level(position_level):
        position_auth_level_map = {
            "Junior": 1,
            "Mid-Level": 2,
            "Senior": 3,
        }
        if position_level not in position_auth_level_map:
            return {"auth_level": 0}
        return {"auth_level": position_auth_level_map[position_level]}

    @worker.task(task_type="course_provision", after=[decorator])
    async def course_provision(profession):
        profession_course_map = {
            "Software␣Engineer": "Introduction␣to␣C++",
            "Accountant": "Finance␣micro-Masters␣from␣MIT"
        }
        if profession not in profession_course_map:
            return {"course": "No␣course␣found"}
        return {"course": f"{profession_course_map[profession]}"}

    @worker.task(task_type="soft_license_provision", after=[decorator])
    async def soft_license_provision(profession, position):
        soft_license_map = {
            "Software␣Engineer": {
                "Junior": "Visual␣Studio",
                "Mid-Level": "JetBrains",
                "Senior": "Jenkins"
            },
            "Accountant": {
                "Junior": "Microsoft␣Excel",
                "Mid-level": "Microsoft␣PowerBI",
                "Senior": "NetSuite",
            },
        }
        if profession not in soft_license_map or position not in
            soft_license_map[profession]:
            return {"soft_license": "No␣course␣found"}
        return {"soft_license": f"{soft_license_map[profession][position]}"}

    @worker.task(task_type="calculate_salary", after=[decorator])
    async def calculate_salary(level_of_education, position_level,
        year_of_experience, profession):
        # Example calculation using provided data
        result = 0
        return {"salary": f"{result}"}
```

Listing 12: Task implementation

5.4.2 Smart contracts deployment

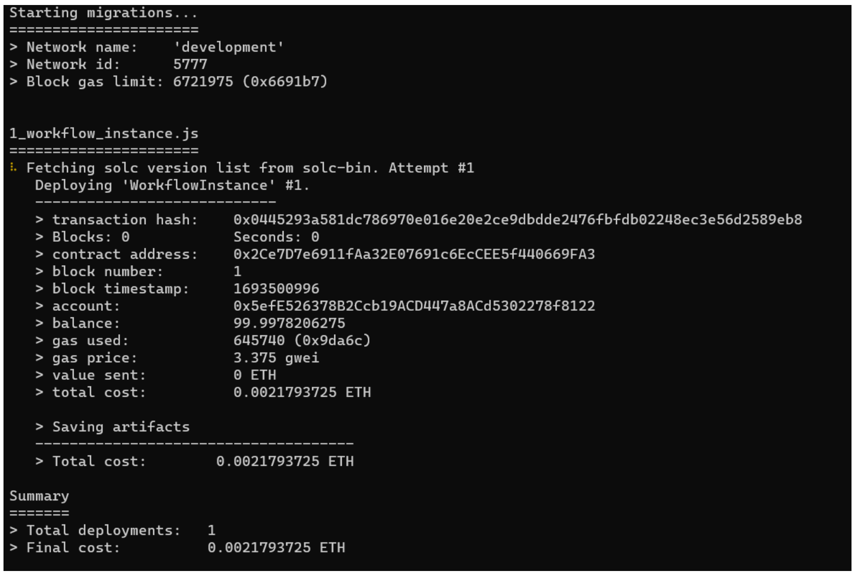
For deploying smart contracts we use Truffle into the Ganache local blockchain network with the configuration from Listing 13.

```
require('dotenv').config();
const { MNEMONIC, PROJECT_ID } = process.env;

const HDWalletProvider = require('@truffle/hdwallet-provider');

module.exports = {
    development: {
    host: "127.0.0.1",      // Localhost (default: none)
    port: 7545,             // Standard Ethereum port (default: none)
    network_id: 5777,        // Any network (default: none)
    },
},

compilers: {
    solc: {
      version: "0.8.0",
    }
},
```

Listing 13: Truffle configuration

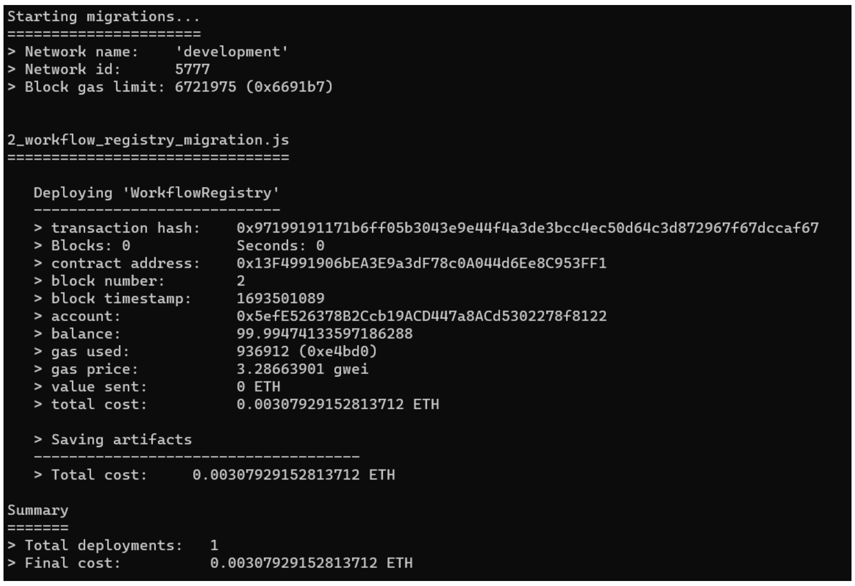
After which we deploy smart contracts using the following command (in PowerShell):

```
$ truffle migrate --network development
```



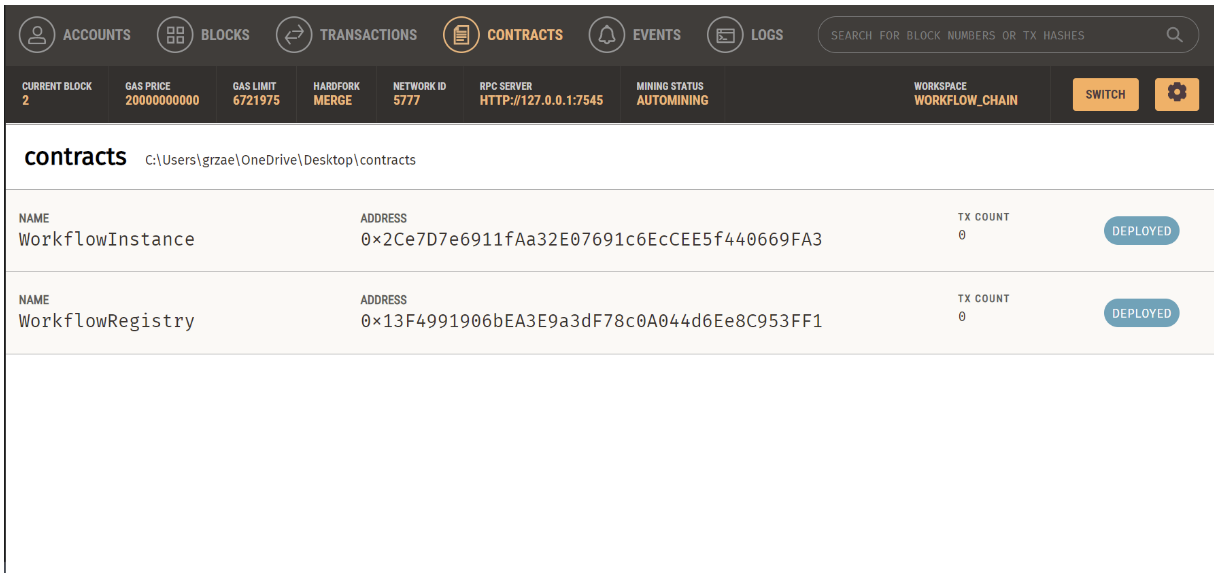
```
Starting migrations...
======================
> Network name:    'development'
> Network id:      5777
> Block gas limit: 6721975 (0x6691b7)


1_workflow_instance.js
======================
⠧ Fetching solc version list from solc-bin. Attempt #1
   Deploying 'WorkflowInstance' #1.
   ----------------------------
   > transaction hash:    0x0445293a581dc786970e016e20e2ce9dbdde2476fbfdb02248ec3e56d2589eb8
   > Blocks: 0            Seconds: 0
   > contract address:    0x2Ce7D7e6911fAa32E07691c6EcCEE5f440669FA3
   > block number:        1
   > block timestamp:     1693500996
   > account:             0x5efE526378B2Ccb19ACD447a8ACd5302278f8122
   > balance:             99.9978206275
   > gas used:            645740 (0x9da6c)
   > gas price:           3.375 gwei
   > value sent:          0 ETH
   > total cost:          0.0021793725 ETH

   > Saving artifacts
   -------------------------------------
   > Total cost:        0.0021793725 ETH

Summary
=======
> Total deployments:   1
> Final cost:          0.0021793725 ETH
```

Figure 18: Workflow Instance contract successful migration



```
Starting migrations...
======================
> Network name:    'development'
> Network id:      5777
> Block gas limit: 6721975 (0x6691b7)


2_workflow_registry_migration.js
================================

   Deploying 'WorkflowRegistry'
   ----------------------------
   > transaction hash:    0x97199191171b6ff05b3043e9e44f4a3de3bcc4ec50d64c3d872967f67dccaf67
   > Blocks: 0            Seconds: 0
   > contract address:    0x13F4991906bEA3E9a3dF78c0A044d6Ee8C953FF1
   > block number:        2
   > block timestamp:     1693501089
   > account:             0x5efE526378B2Ccb19ACD447a8ACd5302278f8122
   > balance:             99.99474133597186288
   > gas used:            936912 (0xe4bd0)
   > gas price:           3.28663901 gwei
   > value sent:          0 ETH
   > total cost:          0.00307929152813712 ETH

   > Saving artifacts
   -------------------------------------
   > Total cost:     0.00307929152813712 ETH

Summary
=======
> Total deployments:   1
> Final cost:          0.00307929152813712 ETH
```

Figure 19: Workflow Registry contract successful migration



CURRENT BLOCK 2 | GAS PRICE 20000000000 | GAS LIMIT 6721975 | HARDFORK MERGE | NETWORK ID 5777 | RPC SERVER HTTP://127.0.0.1:7545 | MINING STATUS AUTOMINING | WORKSPACE WORKFLOW_CHAIN

contracts C:\Users\grzae\OneDrive\Desktop\contracts

NAME	ADDRESS	TX COUNT	
WorkflowInstance	0x2Ce7D7e6911fAa32E07691c6EcCEE5f440669FA3	0	DEPLOYED
WorkflowRegistry	0x13F4991906bEA3E9a3dF78c0A044d6Ee8C953FF1	0	DEPLOYED

Figure 20: Ganache deployed contracts confirmation

5.4.3 Zeebe set-up

For local testing we run Zeebe with Elasticsearch Exporter that is needed for Camunda Operate.

The following command runs the Elasticsearch from it's installed directory:

```
$ ./bin/elasticsearch
```

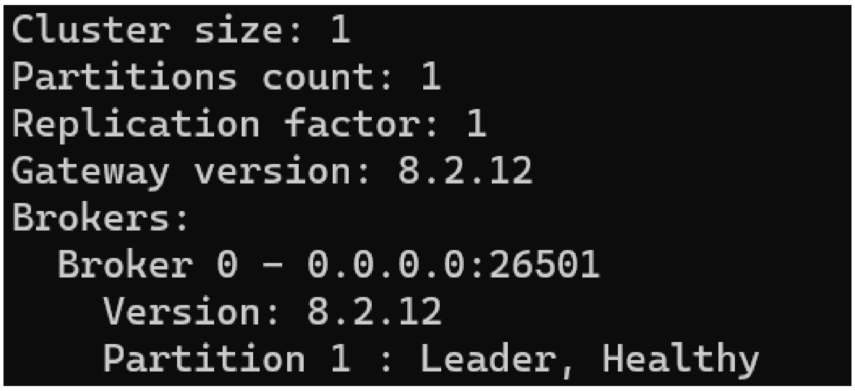
After which we can run Zeebe with the following command from the Zeebe's installed directory:

```
$ ZEEBE_BROKER_EXPORTERS_ELASTICSEARCH_CLASSNAME=io.camunda.zeebe.exporter.ElasticsearchExporter
    ./bin/broker
```

Once Zeebe broker is running, we can check it by running the following command:

```
$ ./bin/zbctl --insecure status
```

Which should return the response from the Figure 21 (depending on network configuration)



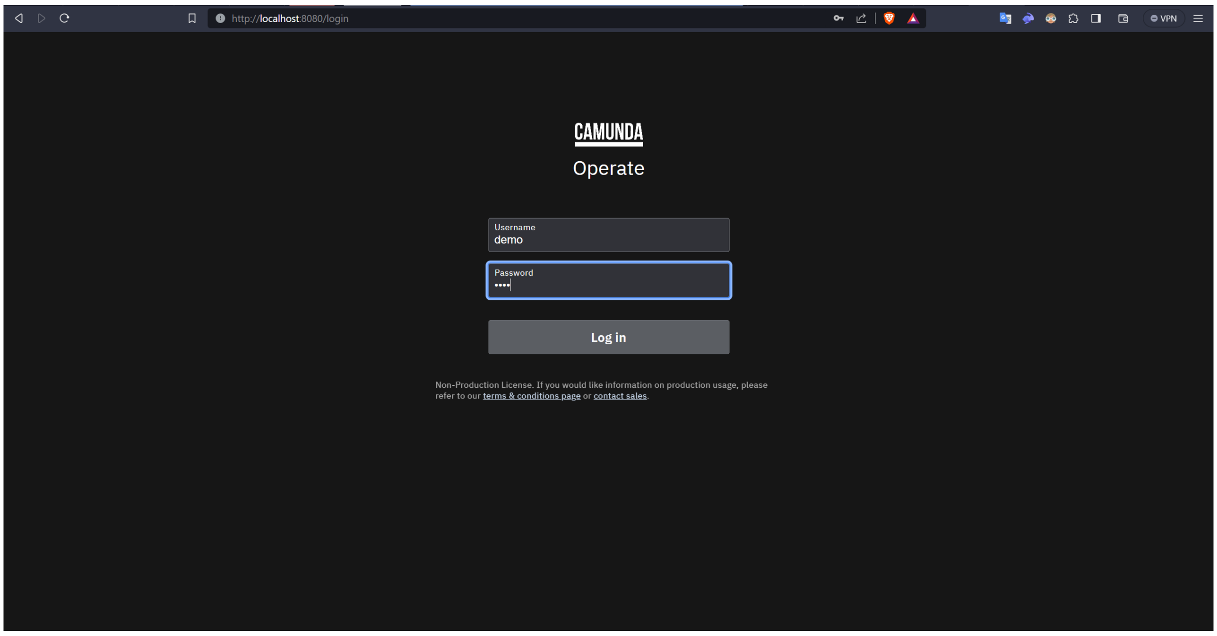
```
Cluster size: 1
Partitions count: 1
Replication factor: 1
Gateway version: 8.2.12
Brokers:
  Broker 0 - 0.0.0.0:26501
    Version: 8.2.12
    Partition 1 : Leader, Healthy
```

Figure 21: Zeebe broker status

In order to run the Camunda Operate we run the following command in the directory where it is installed:

```
$ ./bin/operate
```

Depending on the network configuration we can use the host and the port to access the interface and the login page as shown in the Figure 22. After using default credentials ("demo" as both username and password) we can access the dashboard where we can monitor all process instances. The dashboard is shown in the Figure 23.



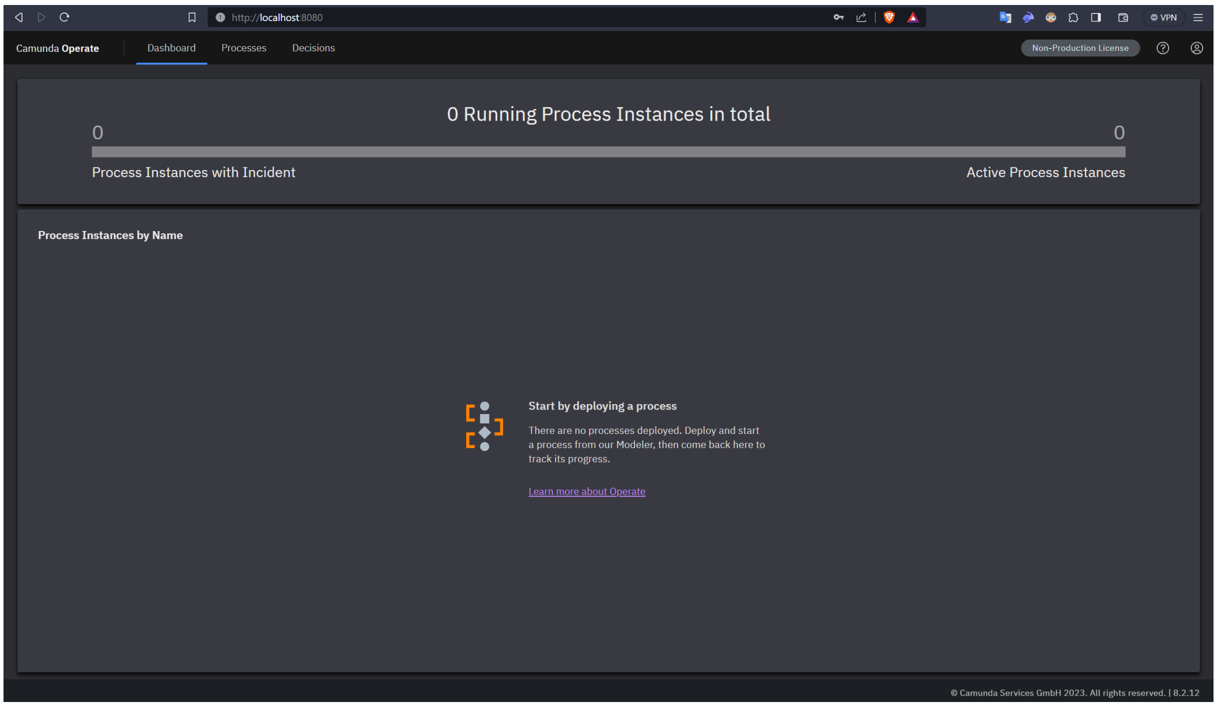
http://localhost:8080/login

Username: demo

Password: ••••

Non-Production License. If you would like information on production usage, please refer to our terms & conditions page or contact sales.

Figure 22: Camunda Operate interface login page



0 Running Process Instances in total

0 Process Instances with Incident

0 Active Process Instances

Start by deploying a process

There are no processes deployed. Deploy and start a process from our Modeler, then come back here to track its progress.

© Camunda Services GmbH 2023. All rights reserved. | 8.2.12

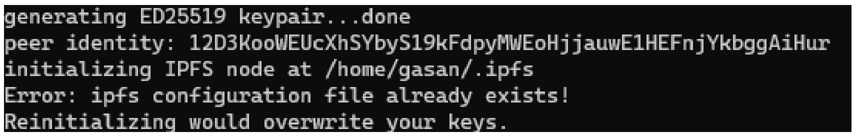
Figure 23: Camunda Operate Dashboard

5.4.4 IPFS Upload

For launching IPFS node we need to run the following command after installation:

```
$ ipfs init
```

The Figure 24 shows the expected response after initializing an IPFS node.



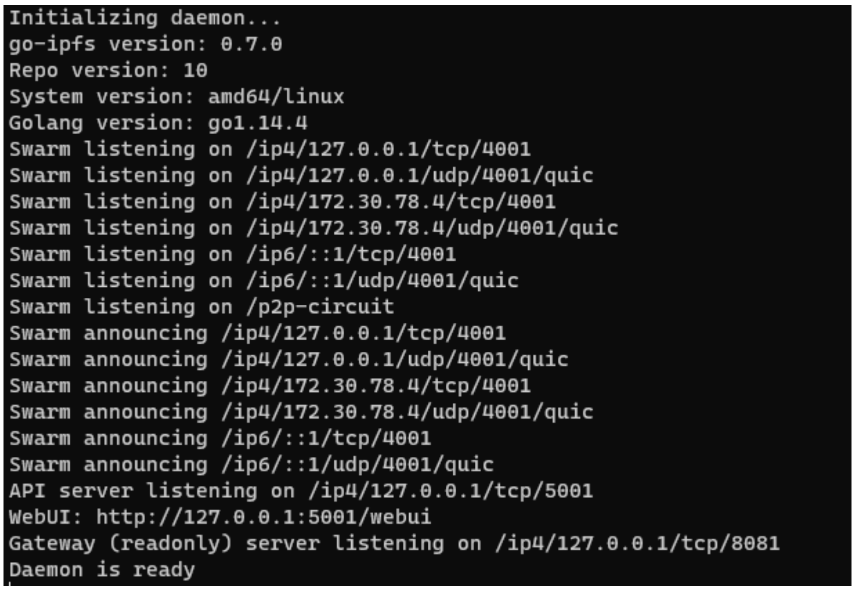
```
generating ED25519 keypair...done
peer identity: 12D3KooWEUcXhSYbyS19kFdpyMWEoHjjauwE1HEFnjYkbggAiHur
initializing IPFS node at /home/gasan/.ipfs
Error: ipfs configuration file already exists!
Reinitializing would overwrite your keys.
```

Figure 24: Initializing IPFS node

After initialization of IPFS node we can run the IPFS daemon that will allow us to connect to the IPFS network using the following command:

```
$ ipfs daemon
```

The Figure 25 shows the expected response after initializing an IPFS daemon.



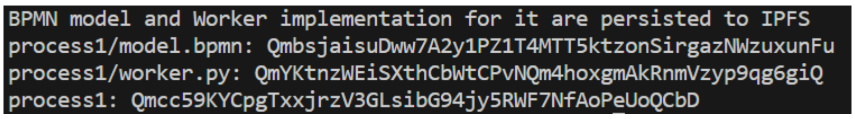
```
Initializing daemon...
go-ipfs version: 0.7.0
Repo version: 10
System version: amd64/linux
Golang version: go1.14.4
Swarm listening on /ip4/127.0.0.1/tcp/4001
Swarm listening on /ip4/127.0.0.1/udp/4001/quic
Swarm listening on /ip4/172.30.78.4/tcp/4001
Swarm listening on /ip4/172.30.78.4/udp/4001/quic
Swarm listening on /ip6/::1/tcp/4001
Swarm listening on /ip6/::1/udp/4001/quic
Swarm listening on /p2p-circuit
Swarm announcing /ip4/127.0.0.1/tcp/4001
Swarm announcing /ip4/127.0.0.1/udp/4001/quic
Swarm announcing /ip4/172.30.78.4/tcp/4001
Swarm announcing /ip4/172.30.78.4/udp/4001/quic
Swarm announcing /ip6/::1/tcp/4001
Swarm announcing /ip6/::1/udp/4001/quic
API server listening on /ip4/127.0.0.1/tcp/5001
WebUI: http://127.0.0.1:5001/webui
Gateway (readonly) server listening on /ip4/127.0.0.1/tcp/8081
Daemon is ready
```

Figure 25: Initializing IPFS daemon

After setting up IPFS node and the daemon, we can run our core client function *persist_to_ipfs*(). The testing code is provided in the Listing 14 and its response with content identifiers in the Figure 26

```python
async def test_ipfs_deployment():
    # Try to run the IPFS Daemon and use an example BPMN model to persist it
    my_client = CoreClient(
        process_dir_name="process1", hostname="localhost", port=26500
    )

    ipfs_hash = my_client.persist_to_ipfs()
    return ipfs_hash
```

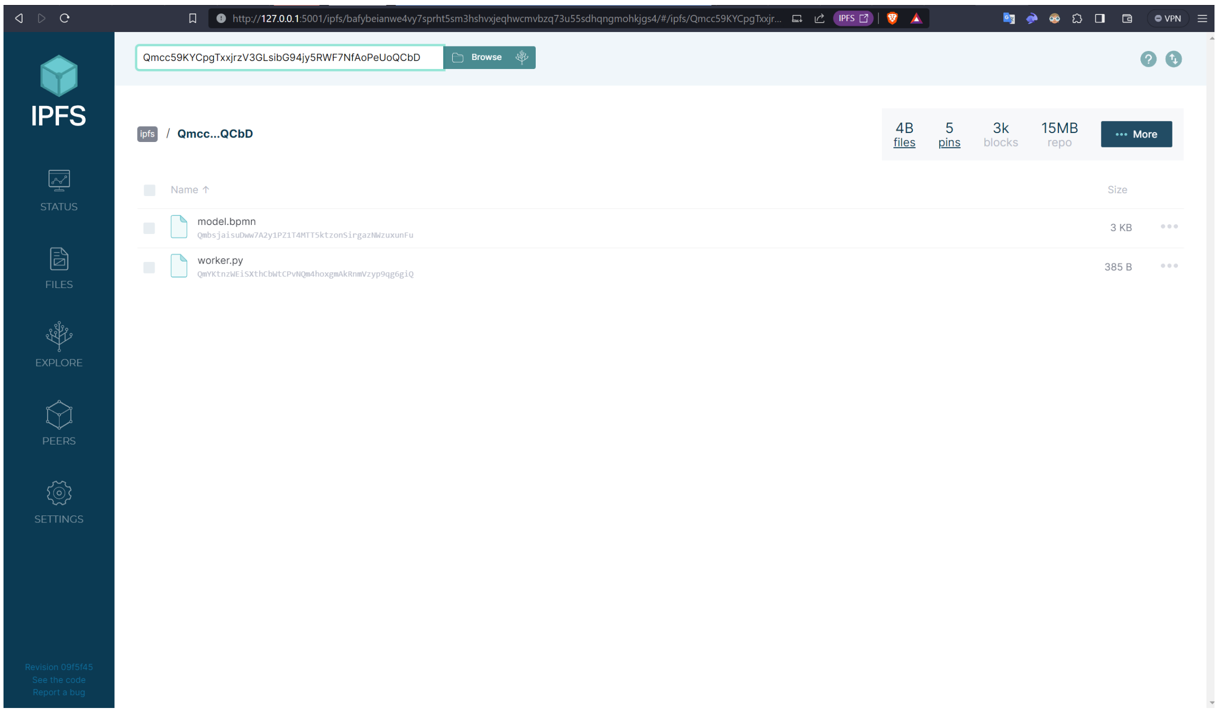
Listing 14: IPFS upload testing code



```
BPMN model and Worker implementation for it are persisted to IPFS
process1/model.bpmn: QmbsjaisuDww7A2y1PZ1T4MTT5ktzonSirgazNWzuxunFu
process1/worker.py: QmYKtnzWEiSXthCbWtCPvNQm4hoxgmAkRnmVzyp9qg6giQ
process1: Qmcc59KYCpgTxxjrzV3GLsibG94jy5RWF7NfAoPeUoQCbD
```

Figure 26: Response of the IPFS upload test

In order to check that the files were uploaded we can use the content identifier of the directory in the IPFS webUI that was provided when we initialized the daemon and search for the content of the directory uploaded. The Figure 27 shows the result of the content identifier search using webUI of the IPFS daemon.



http://127.0.0.1:5001/ipfs/bafybeianwe4vy7sprht5sm3hshvxjeqhwcmvbzq73u55sdhqngmohkjgs4/#/ipfs/Qmcc59KYCpgTxxjr...
IPFS
STATUS
FILES
EXPLORE
PEERS
SETTINGS
Browse
More

Qmcc59KYCpgTxxjrzV3GLsibG94jy5RWF7NfAoPeUoQCbD

ipfs / Qmcc...QCbD

4B files · 5 pins · 3k blocks · 15MB repo

Name ↑	Size
model.bpmn QmbsjaisuDww7A2y1PZ1T4MTT5ktzonSirgazNWzuxunFu	3 KB
worker.py QmYKtnzWEiSXthCbWtCPvNQm4hoxgmAkRnmVzyp9qg6giQ	385 B

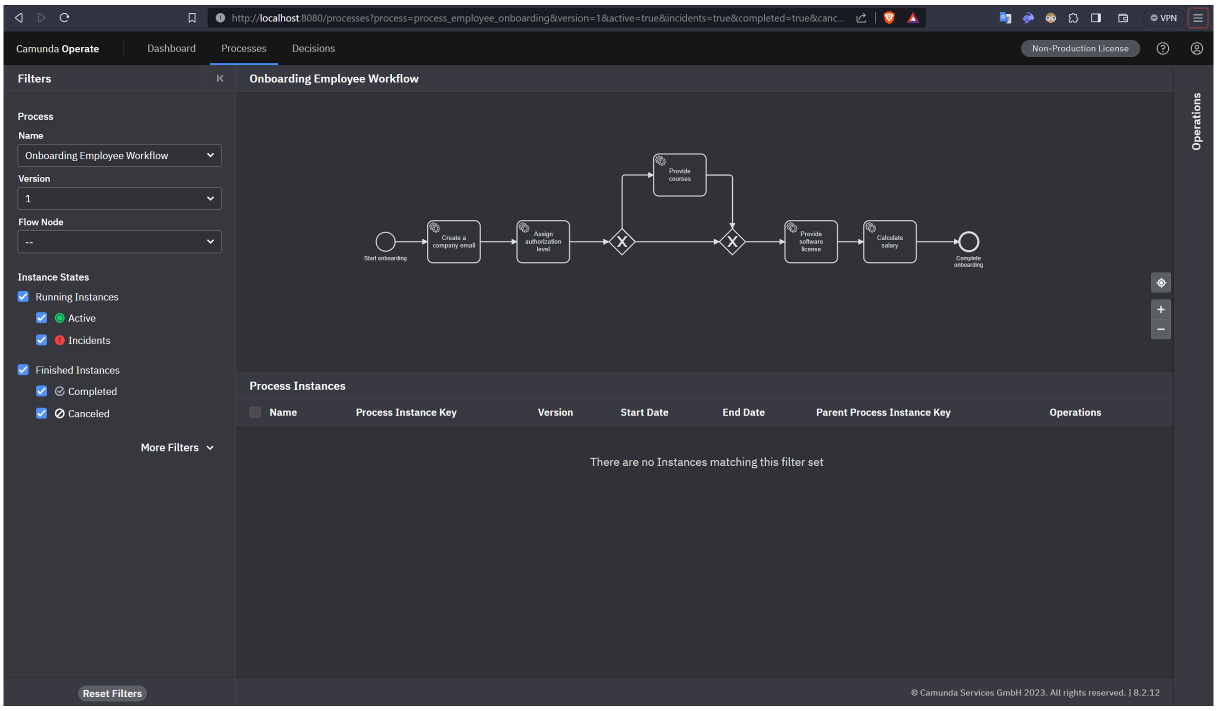
Figure 27: IPFS webUI search for uploaded directory

5.4.5 Zeebe process deployment and instantiation

After preparing a BPMN model for the process and setting up zeebe client we need to deploy the process using our core client. For process deployment testing we use the code from the Listing 15, and for checking the deployment we can use Camunda Operate interface from the Figure 28.

```
async def test_zeebe_deployment():
    # Try to run the zeebe process and use an example BPMN model
    my_client = CoreClient(
        process_dir_name="process1", hostname="localhost", port=26500
    )

    bpmn_path = os.path.abspath(my_client.process_dir_name + "/model.bpmn")
    await my_client.deploy_bpmn(bpmn_file_path=bpmn_path)
```

Listing 15: Process deployment testing code


http://localhost:8080/processes?process=process_employee_onboarding&version=1&active=true&incidents=true&completed=true&canc...
Camunda Operate
Dashboard
Processes
Decisions
Non-Production License
Operations
Start onboarding
Create a company email
Assign authorization level
Provide courses
Provide software license
Calculate salary
Complete onboarding
More Filters
Reset Filters
© Camunda Services GmbH 2023. All rights reserved. | 8.2.12

Filters

Process

Name: Onboarding Employee Workflow

Version: 1

Flow Node: --

Instance States

- [✓] Running Instances
  - [✓] Active
  - [✓] Incidents
- [✓] Finished Instances
  - [✓] Completed
  - [✓] Canceled

Onboarding Employee Workflow

Process Instances

Name	Process Instance Key	Version	Start Date	End Date	Parent Process Instance Key	Operations

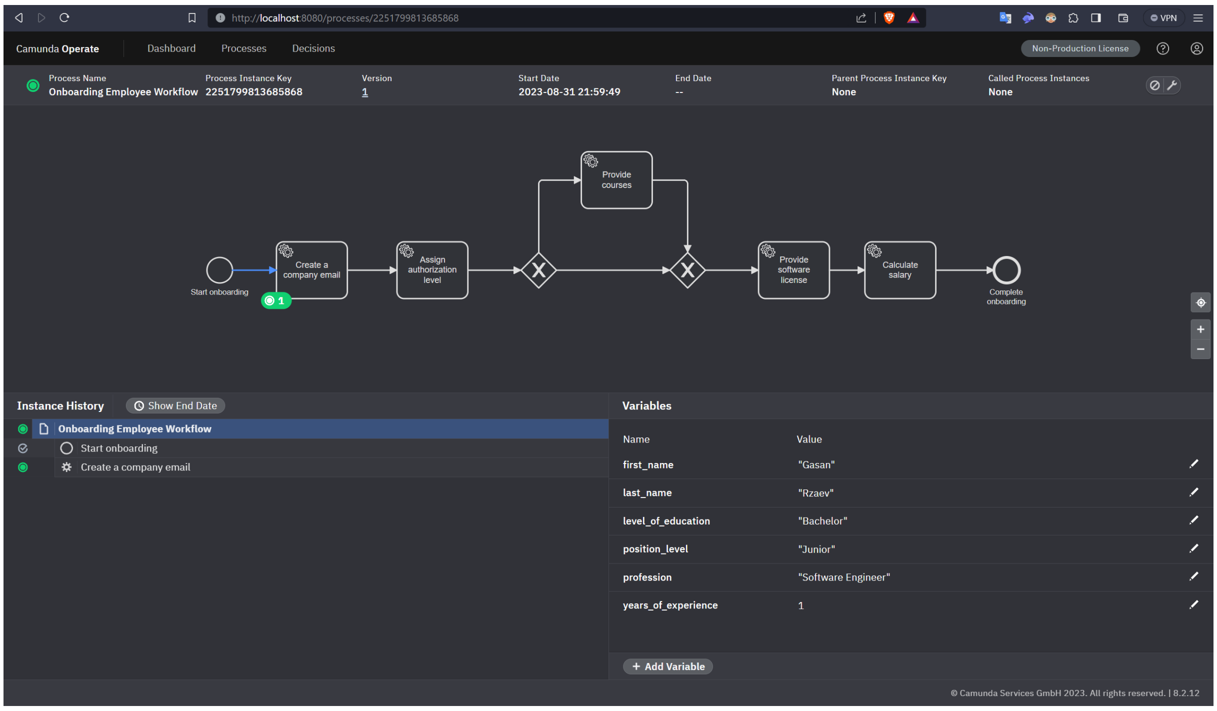
There are no Instances matching this filter set

Figure 28: Camunda Operate deployed process result

After deploying the process we can create an instance using the code from the Listing 16 and to see the result of instantiation we can use Camunda Operate in the Figure 29.

```
async def test_creating_process_instance():
    my_client = CoreClient(
        process_dir_name="process1", hostname="localhost", port=26500
    )

    input_variables = {
        "first_name": "Gasan",
        "last_name": "Rzaev",
        "position_level": "Junior",
        "profession": "Software␣Engineer",
        "level_of_education": "Bachelor",
        "years_of_experience": 1,
    }

    instance = await my_client.create_task_instance(
        bpmn_process_id="process_employee_onboarding", variables=input_variables
    )

    print(instance)
```

Listing 16: Process instantiation testing code



Process Name	Process Instance Key	Version	Start Date	End Date	Parent Process Instance Key	Called Process Instances
Onboarding Employee Workflow	2251799813685868	1	2023-08-31 21:59:49	--	None	None

Instance History

- Onboarding Employee Workflow
  - Start onboarding
  - Create a company email

Variables

Name	Value
first_name	"Gasan"
last_name	"Rzaev"
level_of_education	"Bachelor"
position_level	"Junior"
profession	"Software Engineer"
years_of_experience	1

© Camunda Services GmbH 2023. All rights reserved. | 8.2.12

Figure 29: Camunda Operate process instantiation result

5.4.6 Process execution

Using the following code snippet we can run the Dynamic Task worker which will use the task handlers provided previously to run and complete the tasks, and log everything in the event log.

```
async def test_running_workers():
    # Try to run a Zeebe worker to Handle the Task of the Process Instances
    my_client = CoreClient(
        process_dir_name="process1", hostname="localhost", port=26500
    )

    await my_client.create_and_run_worker("worker.py")
```

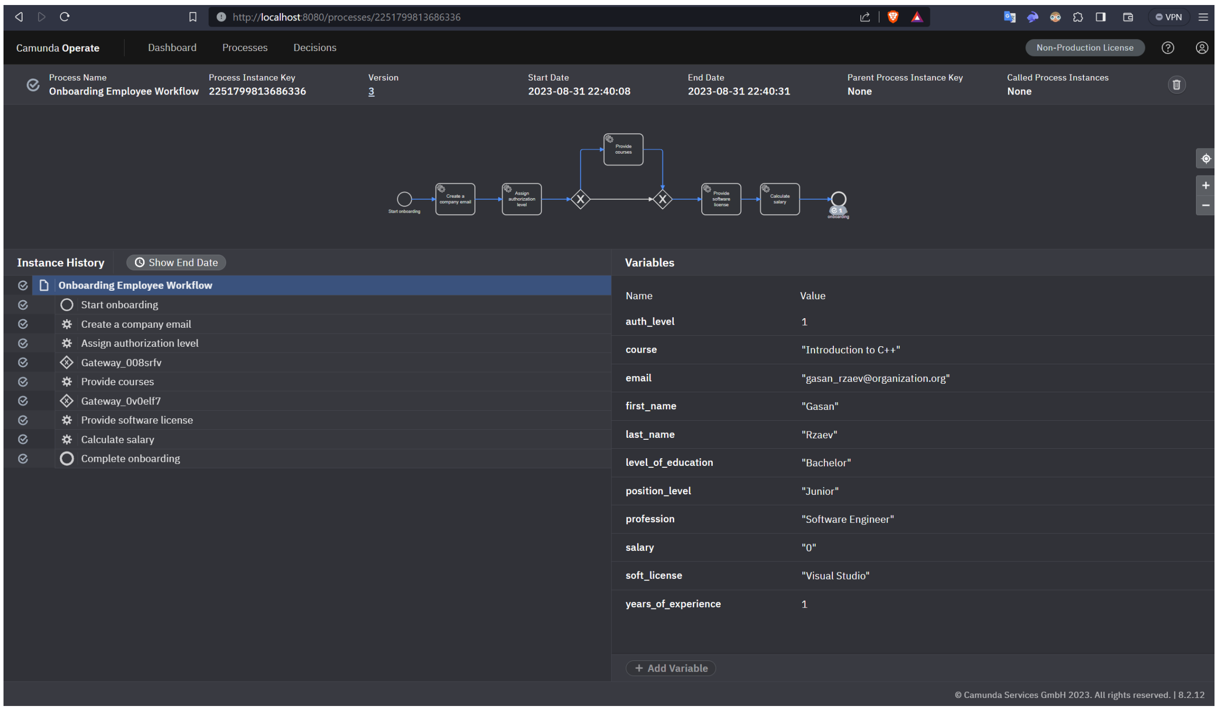
After running the execution we have appended an event log contents of which can be seen in the Listing ??

```
{'jobKey': 2251799813686349, 'taskType': 'create_email', 'processInstanceKey': 2251799813686336, 'bpmnProcessId': 'process_employee_onboarding', 'processDefinitionVersion': 3, 'processDefinitionKey': 2251799813686022, 'elementId': 'Activity_01r6nmf', 'elementInstanceKey': 2251799813686346, 'customHeaders': {}, 'worker': 'Gasan', 'retries': 3, 'deadline': 1693514441594, 'variables': {'last_name': 'Rzaev', 'first_name': 'Gasan', 'email': 'gasan_rzaev@organization.org'}}
{'jobKey': 2251799813686362, 'taskType': 'assign_auth_level', 'processInstanceKey': 2251799813686336, 'bpmnProcessId': 'process_employee_onboarding', 'processDefinitionVersion': 3, 'processDefinitionKey': 2251799813686022, 'elementId': 'Activity_1ovseyd', 'elementInstanceKey': 2251799813686360, 'customHeaders': {}, 'worker': 'Gasan', 'retries': 3, 'deadline': 1693514441644, 'variables': {'position_level': 'Junior', 'auth_level': 1}}
{'jobKey': 2251799813686373, 'taskType': 'course_provision', 'processInstanceKey': 2251799813686336, 'bpmnProcessId': 'process_employee_onboarding', 'processDefinitionVersion': 3, 'processDefinitionKey': 2251799813686022, 'elementId': 'Activity_06gnke6', 'elementInstanceKey': 2251799813686371, 'customHeaders': {}, 'worker': 'Gasan', 'retries': 3, 'deadline': 1693514441685, 'variables': {'profession': 'Software␣Engineer', 'course': 'Introduction␣to␣C++'}}
{'jobKey': 2251799813686385, 'taskType': 'soft_license_provision', 'processInstanceKey': 2251799813686336, 'bpmnProcessId': 'process_employee_onboarding', 'processDefinitionVersion': 3, 'processDefinitionKey': 2251799813686022, 'elementId': 'Activity_0kckiv0', 'elementInstanceKey': 2251799813686382, 'customHeaders': {}, 'worker': 'Gasan', 'retries': 3, 'deadline': 1693514441712, 'variables': {'profession': 'Software␣Engineer', 'position_level': 'Junior', 'soft_license': 'Visual␣Studio'}}
{'jobKey': 2251799813686397, 'taskType': 'calculate_salary', 'processInstanceKey': 2251799813686336, 'bpmnProcessId': 'process_employee_onboarding', 'processDefinitionVersion': 3, 'processDefinitionKey': 2251799813686022, 'elementId': 'Activity_0bgu5gm', 'elementInstanceKey': 2251799813686392, 'customHeaders': {}, 'worker': 'Gasan', 'retries': 3, 'deadline': 1693514441736, 'variables': {'profession': 'Software␣Engineer', 'position_level': 'Junior', 'level_of_education': 'Bachelor', 'years_of_experience': 1, 'salary': '0'}}
```

Listing 17: Resulting event log after execution

After that we can upload the event log into the IPFS using the same method shown previously, and finally push the new content identifier into the smart contract in order to update the state of the process in blockchain using the following code:

```
def test_update_process_state():
    my_client = CoreClient(
        process_dir_name="process1", hostname="localhost", port=26500
    )

    event_log_hash = "QmapJbBtMjJA85CfhqHC3yQPnowHVC9oSusvu4UGCgB2He"
    process_id = "process1"
    private_key = "0xcbd80d93112fe4671dc7a8e2d41fc1e456a37e60c1c1f02f4836b12e6b4c13e1"
    caller_address = "0x5efE526378B2Ccb19ACD447a8ACd5302278f8122"

    my_client.instance_oracle.set_process_event_log_hash(
        process_id=process_id,
        event_log_hash=event_log_hash,
```



Process Name	Process Instance Key	Version	Start Date	End Date	Parent Process Instance Key	Called Process Instances
Onboarding Employee Workflow	2251799813686336	3	2023-08-31 22:40:08	2023-08-31 22:40:31	None	None

Instance History

- Onboarding Employee Workflow
  - Start onboarding
  - Create a company email
  - Assign authorization level
  - Gateway_008srfv
  - Provide courses
  - Gateway_0v0elf7
  - Provide software license
  - Calculate salary
  - Complete onboarding

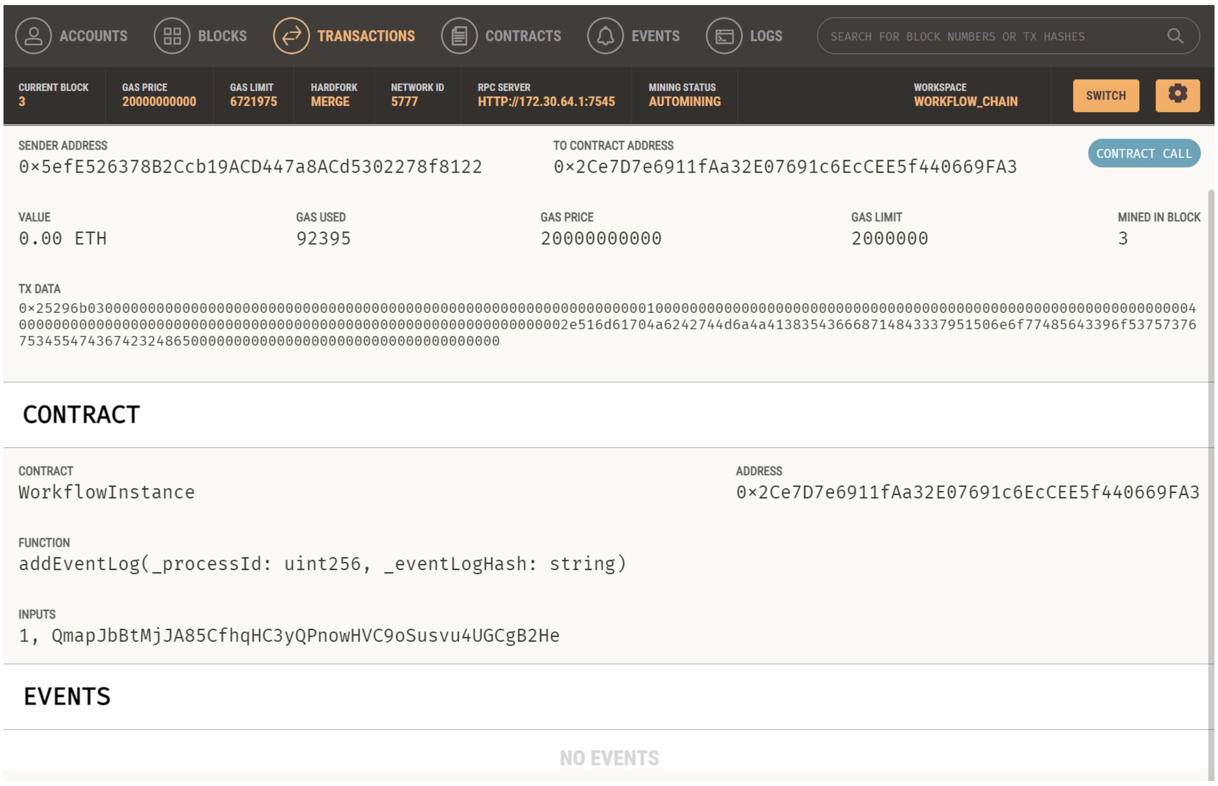
Variables

Name	Value
auth_level	1
course	"Introduction to C++"
email	"gasan_rzaev@organization.org"
first_name	"Gasan"
last_name	"Rzaev"
level_of_education	"Bachelor"
position_level	"Junior"
profession	"Software Engineer"
salary	"0"
soft_license	"Visual Studio"
years_of_experience	1

Figure 30: Completed process on Camunda Operate

```
        private_key=private_key,
        caller_address=caller_address,
    )
```

Finally, to finalize the evaluation we can see that the transaction has arrived using the instance oracle in the Figure 31.



CURRENT BLOCK 3 | GAS PRICE 20000000000 | GAS LIMIT 6721975 | HARDFORK MERGE | NETWORK ID 5777 | RPC SERVER HTTP://172.30.64.1:7545 | MINING STATUS AUTOMINING | WORKSPACE WORKFLOW_CHAIN

SENDER ADDRESS
0×5efE526378B2Ccb19ACD447a8ACd5302278f8122

TO CONTRACT ADDRESS
0×2Ce7D7e6911fAa32E07691c6EcCEE5f440669FA3

VALUE
0.00 ETH

GAS USED
92395

GAS PRICE
20000000000

GAS LIMIT
2000000

MINED IN BLOCK
3

TX DATA
```
0×25296b03000000000000000000000000000000000000000000000000000000000000000100000000000000000000000000000000000000000000000000000000000000004
00000000000000000000000000000000000000000000000000000000000000002e516d61704a6242744d6a4a413835436668714843337951506e6f77485643396f53757376
7534554743674232486500000000000000000000000000000000000
```

CONTRACT

CONTRACT
WorkflowInstance

ADDRESS
0×2Ce7D7e6911fAa32E07691c6EcCEE5f440669FA3

FUNCTION
addEventLog(_processId: uint256, _eventLogHash: string)

INPUTS
1, QmapJbBtMjJA85CfhqHC3yQPnowHVC9oSusvu4UGCgB2He

EVENTS

NO EVENTS

Figure 31: Ganache showing updated contract state

6 Conclusions

In conclusion, this study embarked on a comprehensive exploration of the integration of blockchain smart contracts with the Zeebe process engine to establish a middleware solution. Through the thorough examination and implementation of this middleware, significant insights and implications have emerged, illuminating the potential and challenges inherent in such an innovative amalgamation.

The project's trajectory has been marked by meticulous analysis and execution. A pivotal aspect of this endeavor was the seamless interaction between blockchain smart contracts and the Zeebe process engine, thereby fostering a novel approach to task implementation and workflow automation. The middleware successfully orchestrated the execution of smart contracts as integral components of Zeebe workflows, illustrating the fusion of trustless execution and orchestrated processes.

Throughout the exploration, it became evident that the amalgamation of these technologies not only introduces numerous benefits but also unveils certain considerations and opportunities for future enhancements. The introduced middleware showcases the immutability of workflow data, bolstered by blockchain's inherent characteristics. It further empowers the concept of trustless task execution, imbuing workflows with heightened transparency and accountability.

However, the journey has also unveiled areas that warrant further investigation and refinement. Notably, concerns regarding data security emerged, accentuated by the storage of sensitive information in the IPFS and the subsequent persistence of hash identifiers within smart contracts. This vulnerability calls for the exploration of encryption methods or the introduction of authorization layers to safeguard data integrity.

Looking ahead, the project's future trajectory presents intriguing avenues for expansion. Implementing on-chain approval and escalation mechanisms offers the promise of enhanced security and a more decentralized workflow framework. The integration of tokenization standards, such as ERC-721, emerges as a compelling option to fortify access control, immutability, and task validation, potentially introducing an incentivized ecosystem.

In evaluating the middleware's merits and limitations, a dichotomy emerges. While it champions data immutability and trustless task execution, potential scalability issues loom, particularly when applied to intricate workflows in resource-intensive industrial contexts. The study has also highlighted the importance of real-world input verifiability, an aspect that necessitates ongoing consideration.

As the exploration draws to a close, it becomes evident that the synthesis of blockchain smart contracts and Zeebe process engines holds substantial promise, while continually beckoning us to delve deeper into the uncharted realms of decentralized automation.

References

[1] IBM Cloud Education. “What is workflow automation?” (2022), [Online]. Available: `https://www.ibm.com/cloud/blog/workflow-automation`.

[2] AgilePointBPMS. “What is a process engine?” (2022), [Online]. Available: `https://documentation.agilepoint.com/supportportal/docs/productdocumentation/05.00.0200/documentationlibrary/maps/overviewWorkflowEngine.html`.

[3] Vitaly Buterin. “Ethereum: A next-generation smart contract and decentralized application platform. by vitalik buterin (2014).” (2014), [Online]. Available: `https://ethereum.org/669c9e2e2027310b6b3cdce6e`
`Ethereum_Whitepaper_-_Buterin_2014.pdf`.

[4] Ethereum.org. “Ethereum consensus mechanism.” (2023), [Online]. Available: `https://ethereum.org/en/developers/docs/consensus-mechanisms/`.

[5] “Solidity documentation.” (2023), [Online]. Available: `https://buildmedia.readthedocs.org/media/pdf/solidity/develop/solidity.pdf`.

[6] G. Wood. “Ethereum yellow paper.” (2014), [Online]. Available: `https://ethereum.github.io/yellowpaper/paper.pdf`.

[7] A. M. V. C. Sguanci R. Spatafora, “Layer 2 blockchain scaling: A survey,” 2021.

[8] Ethereum.org. “Ethereum scaling.” (2023), [Online]. Available: `https://ethereum.org/en/developers/docs/scaling/#:~:text=Layer%202%20is%20a%20collective,decentralized%20security%20model%20of%20Mainnet.`.

[9] Ethereum.org. “Sidechains.” (2023), [Online]. Available: `https://ethereum.org/en/developers/docs/scaling/sidechains/`.

[10] “Vyper documentation.” (2020), [Online]. Available: `https://docs.vyperlang.org/en/stable/`.

[11] F. Zhang, E. Cecchetti, K. Croman, A. Juels, E. Shi, “Town crier: An authenticated data feed for smart contracts,” 2016.

[12] K. Mammadzada, M. Iqbal , F. Milani, L. Garcia-Banuelos, R. Matulevicius, “Blockchain oracles: A framework for blockchain-based applications,” 2020.

[13] “Vyper Reentrancy Vulnerability: An Explanatory Overview.” (2023), [Online]. Available: `https://medium.com/@justanotherdev/vyper-reentrancy-vulnerability-an-explanatory-overview-3a3411889f06`.

[14] T. Xiong, S. Feng, M. Pan, Y. Yu, “Smart contract generation for inter-organizational process collaboration,” 2023.

[15] A. B. Tran, Q. Lu, I. Weber, “Lorikeet: A Model-Driven Engineering Tool for Blockchain-Based Business Process Execution and Asset Management,”

[16] O. Lopez-Pintado, L. Garcia-Banuelos, M. Dumas, I. Weber, “Caterpillar: A Blockchain-Based Business Process Management System,”

[17] E. Ozaras, “Integration of blockchain smart contracts with zeebe process engine as task implementations,” 2022.

[18] E. Ozaras, “Integration of blockchain smart contracts with process engines,” 2022.

[19] Camunda. “Zeebe’s Architecture.” (2023), [Online]. Available: `%7Bhttps://docs.camunda.io/docs/components/zeebe/technical-concepts/architecture/%7D`.

[20] M. Yadav. “What is interplanetary file system (ipfs).” (2022), [Online]. Available: `%7Bhttps://devinfinity.hashnode.dev/what-is-interplanetary-file-system-ipfs%7D`.