UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Indrek Värva

Autonomy and Efficiency Trade-offs on an Ethereum-based Real Estate Application

Master's Thesis (30 ECTS)

Supervisor(s): Luciano García-Bañuelos, PhD

Tartu 2018

Autonomy and Efficiency Trade-offs on an Ethereum-based Real Estate Application

Abstract:

Marketplaces in sharing economy have traditionally been organized as web applications running on top of centralized databases. The advent of blockchain technology brings new opportunities, with the promise of transforming the landscape with tamper-resilient storage and the potential of reduction in intermediaries. In this context, in this thesis we look at exploring the use of blockchain technologies in the domain of real estate rental process. More specifically, we designed a solution on top of Ethereum and implemented three consecutive prototypes to analyze the impact of moving data and processing to the blockchain. The results show a trade-off between efficacy versus efficiency when moving toward decentralization.

Keywords:

Blockchain case study, smart contract design, disintermediation

CERCS: P170, Computer science, numerical analysis, systems, control

Autonoomsuse ja tõhususe kompromisside tuvastamine Ethereumi baasil arendatud kinnisvara rakenduses

Lühikokkuvõte:

Siiani on jagamismajanduse vahendusplatvorme arendatud tsentraliseeritud andmebaaside abil. Plokiahela esiletõus on aga ilmutanud uusi võimalusi, et muuta valdkonda võltsimiskindlaks ning vähendada vajadust vahendajate järele. Käesolevas töös uuritakse plokiahela kasutusvõimalusi kinnisvara rentimise protsessi näitel. Täpsemalt, töös disainitakse lahendus Ethereumi abil ning teostatakse kolm järjestikust prototüüpi, et analüüsida andmete ning arvutuste tõstmist plokiahelasse. Tulemused näitavad, et detsentraliseerimisel tuleb teha kompromisse teostatavuse ning tõhususe vahel.

Võtmesõnad:

Räsiahela juhtumiuuring, nutilepingu disain, vahendajatest vabanemine

CERCS: P170, Computer science, numerical analysis, systems, control

Table of Contents

1 Introduction

1.1 Context

In the world of business, a ledger is an essential tool for bookkeeping. A ledger is essentially a formatted medium to hold the business transactions such as debits and credits for summarization purposes. For centuries, the tool has had a physical form as a paper notebook in which the records were simply concatenated. However, the technological evolution has now provided measures for ledgers to move to a digital medium for obvious benefits such as faster insertions and analysis. For example, a centralized database system with insert, update and read access could be considered an implementation of such digital medium.

Lately, another implementation of a digital ledger - blockchain - has been trending[1]. This technology was first popularized as a backbone behind cryptocurrency Bitcoin in 2008 by Nakamoto [1]. It is a special type of ledger - a distributed one - which means that its data is shared, replicated and synchronized across multiple destinations using a specialized consensus system. As one of the biggest benefits of the blockchain, the consensus mechanism guarantees that its records are not tampered once appended to the ledger [1]. Furthermore, tamper resilience has been identified as a potential enabler for collaborative process execution between trusted partners [2].

In addition to providing tamper protection in a distributed setup, blockchain has also allowed ledgers to expand its functionality towards a whole new dimension - scripting. In the context of ledger systems, script is just another format of a record which is designed to perform a certain task when initiated by other scripts or external actors. For example, in the case of Ethereum blockchain [3] and through the concept of smart contracts, flexible scripting possibilities are available through a specialized programming language for instructs such as conditional blocks, loops and also automated triggers to enforce complex business rules and promote automation [4].

While the blockchain may prove itself useful for financial services [5], it also shows potential benefits in other fields as an effective mean to track asset life cycle [6]. In this context, an asset could be tangible, such as an aftermarket plane part moving through a supply chain, or intangible, such as a purchase order on an e-commerce platform. Furthermore, centralized marketplace/sharing economy applications have been a de facto standard for real estate rental. Moreover, the business model in this sector consists of collecting intermediary fees from the transactions [7]. But, naturally, the ultimate goal of sharing economy is to move towards a *peer-to-peer* (P2P) model in which one can expect the role of an intermediary to be kept at minimum to reduce transactional costs for end-users. So, in this aspect, smart contract scripting may be able to move associated business logic from a centralized server to the blockchain.

1.2 Research Goals and Methodology

This work considers the domain of real estate rental as a case study to explore the use of blockchain technologies. In the scope of the study, 3 proof of concept prototypes are designed and developed (Figure 1). The first prototype (variant 1) establishes a design baseline by including the blockchain into the platform providers system architecture as an internal service. Then, the research sets an objective of complete disintermediation by moving the internal blockchain service outside of the platform to be consumed directly by the external actors of the system. To move the service outside of the providers system, two iterations of

[1] https://trends.google.com/trends/explore?date=all&q=blockchain [Accessed: 24-Nov-2018]

redesign are made with the purpose to decouple the blockchain intermediated process from outside dependencies and achieve complete autonomy. For this purpose, a strategy is followed to first migrate the data (variant 2) and then port the operations (variant 3). In the accompanying service and design analysis of the prototypes, the paper focuses on the changes in the core components which are native and impactful to the applications of the domain such as geodetic real estate search functionality and other types of system data filtering operations. In addition to the architectural changes, empirical implications to tamper resilience and privacy are noted. Finally, the results are evaluated in terms of efficiency to find out how what are the trade-offs of between the autonomy-related design decisions and the efficiency of the system.

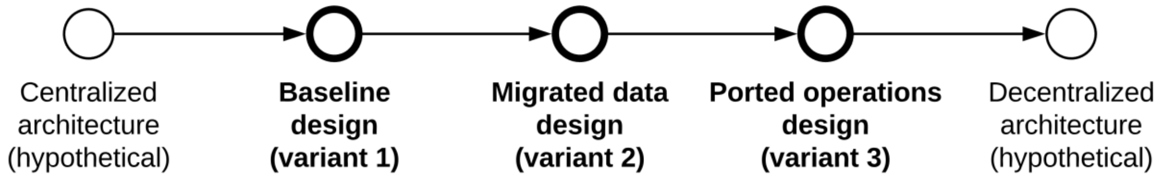

Centralized architecture (hypothetical)
Baseline design (variant 1)
Migrated data design (variant 2)
Ported operations design (variant 3)
Decentralized architecture (hypothetical)

Figure 1. Design and development roadmap.

In order to drive the representation and analysis of the prototypes to an effective contribution, the paper follows the design science (DS) research approach described by March *et al.* [8]. The DS research proposes to solve identified problems by using a framework of building, evaluating, theorizing and justifying (i.e. research activities) a set of effective artifacts (i.e. research outputs). Furthermore, the theory specifies four types of artifacts:

1) *Constructs* - conceptualize the domain by providing vocabulary on the problem space (e.g. entity on a data model).
2) *Models* - describe associations between constructs and act as a medium to capture requirements (e.g. business process model) or a define a system design (e.g. service map).
3) *Methods* - capture the steps necessary to perform a task on constructs and/or models (e.g. filtering algorithm on a data structure).
4) *Instantiations* - realize the constructs, models and methods to demonstrate their feasibility (e.g. prototype). [8]

The activities and outputs of the DS research form a 16-cell framework, each of which is could be considered a viable research effort [8]. To explore the use of blockchain in the domain of real estate rental, this study aims to build the constructs, models and methods of the three instantiations (prototypes). To assist with choosing the relevant artifacts of the variants, the toolset of data-driven service design [9] is used as a heuristic to capture a service map, state machine, domain and data models and the operation set. The created artifacts are thereafter empirically evaluated in terms of their differences to the previous iteration(s) on the roadmap (Figure 1) to derive insights for the next phase of development. Finally, the research instantiates the three prototypes to evaluate their performance in terms of efficiency to extract generalizable information about the implementations.

The approach by March *et al.* [8] only concerns with solution design and evaluation [10] and does not deal with exploring the problem space. To include problem identification to the framework, an extension by Peffers *et al.* [11] is used which adds relevance cycle of environment [12] to the process (Figure 2). Furthermore, the problem space is captured using the artifacts proposed by the process-driven methods [13].

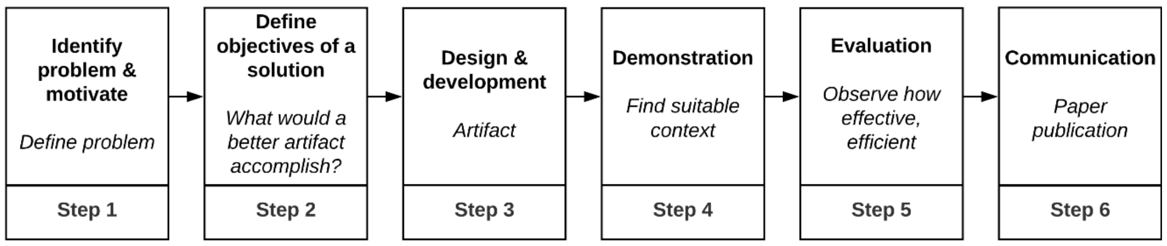


Figure 2. DS research methodology process model (adapted) [11].

1.3 Document Organization

The remaining of the paper is structured as follows. Firstly, chapter 2 introduces the foundations of the study. Then, chapter 3 aligns with the steps of the DS research approach (Figure 2) by identifying the problems and derives objectives of a solution (steps 1 and 2) in section 3.1. Next, in sections 3.2, 3.3 and 3.4, the design and development of the three prototypes is described (repeating step 3) with a brief analysis on their effects. Contribution is followed with chapter 4 which instantiates the prototypes for an evaluation (steps 4 and 5) and discussion. Finally, the paper finishes with concluding remarks and an outlook in chapter 5.

2 Background

This chapter introduces a more detailed background of the concepts, tools and previous literature related to the study.

2.1 Sharing Platforms

Sharing economy is a social paradigm in which on-demand access to the resource is valued over ownership. In this economy, suppliers seek to make profit from their personal underutilized assets or resources by loaning or renting them consumers. The phenomenon is nowadays largely driven by specialized online applications – sharing platforms – which are often domain-specific such as AirBnB[2] for accommodation, Uber[3] or BlaBlaCar[4] for ridesharing or TransferWise[5] for bank accounts for the sake of currency exchange. [14]

While the term *sharing* may also cover business-to-consumer exchanges (such as traditional car rental) [15], most of these platforms allow exchanges to take place between individuals in a P2P fashion [16]. In this model, the P2P sharing platforms act as multi-sided platforms which bring together multiple groups of users and enable their interaction to facilitate the exchange [7]. For example, on a two-sided platform (Figure 3) such as Monestro[6], the consumer is a person in need for financial resource (borrower) and the supplier a person with available money (lender).

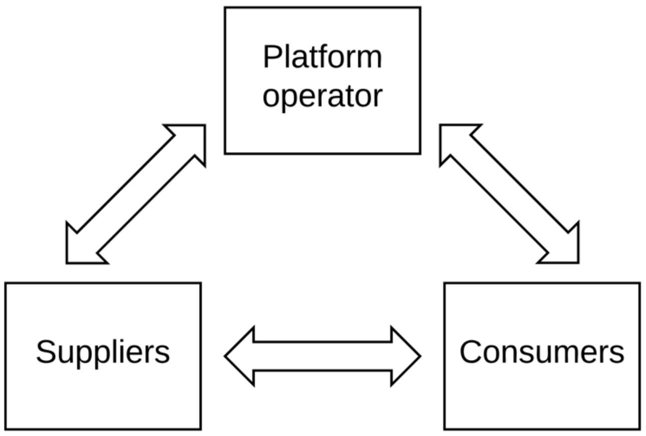


Figure 3. Two-sided market (adapted) [16].

To facilitate the exchange, the P2P sharing platform must conform to perform some general tasks. First, it must be able to collect potential offers from suppliers. Secondly, it enlists these offers to consumers. Now, if the consumer finds an interesting asset, the platform must provide the means for the stakeholders to establish a personal contact. Finally, and most importantly, the platform must enforce the rules of the game. Meaning that the transaction follows some set rules that all parties agree upon. [17]

In their tasks, these sharing platforms also need to be profitable. Business is said to have a sustainable model in case it creates, delivers and captures value that benefit themselves as well as its' stakeholders such as investors, customers, suppliers [18]. Now, Bocken *et al.* [19] have found that creating value from waste is an archetype of a sustainable business

[2] https://www.airbnb.com [Accessed: 19-May-2018]
[3] https://www.uber.com [Accessed: 19-May-2018]
[4] https://www.blablacar.com [Accessed: 19-May-2018]
[5] https://transferwise.com [Accessed: 19-May-2018]
[6] https://www.monestro.com [Accessed: 19-May-2018]

model. As P2P sharing platforms mediates what is waste to one party (in the form of under-utilised resource) to another one as a useful asset, it could be argued that P2P sharing platforms have sustainability embedded in their core [16].

2.2 Blockchain

In a nutshell, blockchain is a system which provides trust without a centralized authority so that interested parties can share data without trusting each other. Technically, it is a distributed transactional database/ledger which is enforcing a set of rules to make sure that the committed transactions (i.e. records inside a block) as well as the blocks including them are valid. In order to do so, blockchains employ a network of nodes which communicate with each other in a P2P manner to serve different tasks:

1) Validating and relaying transactions sent to the network.
2) Mine blocks consisting of transactions by solving cryptographic puzzles as *proof-of-work*. As an incentive to do so, a prize is included to the worker which is a combination of a calculated transaction cost and a static reward [3].
3) Propagating and validating mined blocks.

Most importantly, as a result of mining, propagation and validation mechanisms, a consensus is achieved when enough of the nodes have validated the block. Consensus, in turn, determines which transactions and blocks to persist in the chain as well as their order. Furthermore, the validation mechanism uses hashing based validation rules (e.g. a Merkle tree [20]) to make sure that the history (previous blocks) have not been changed and by that, providing tamper protection of the setup [3].

The transactions inside a blockchain are initiated by actors who are represented as addresses. These addresses are closely related to asymmetric cryptography. More specifically, a private key is used to cryptographically sign and public key to target and validate the transactions. As each transaction is signed by the initiator, it is trivial for the nodes to verify whether the associated address has necessary balance (e.g. Bitcoin or Ether) to pay for the costs as well as the transfer itself. [3]

Multiple implications of blockchain could be identified, for example:

1) Blockchain could be considered transparent in terms of the included transactions. For example, if the blockchain is public (e.g. Bitcoin), everyone is able to see and verify which transactions have taken place (and do this with no cost), further enhancing the trust-factor. However, if privacy of the data is an issue, there are also means to deploy a private blockchain to protect sensitive data [21].
2) It is transparent about its inner-workings as the associated software systems are managed in an open-source manner [3], [21], [22]. This further eliminates the possibility of fraudulence.
3) Blockchain typically improves read availability of a system. However, due to the technical overhead of consensus mechanisms, write availability is actually low. Furthermore, the transaction commit time is effected by chosen gas price and network delay causing out-of-order transactions. [23]

Undeniably, static-record blockchain has technical implications which could disrupt bookkeeping industries such as finance as the success of Bitcoin has already proven. However, their records remain to be static which means that it is not possible to include dynamic business rules in the system. To expand the abilities of a blockchain to include programmatic execution logic, a notion of smart contracts is used.

2.3 Smart Contracts

The idea of smart contracts is not new in the scientific field and was coined already in 1996 by Szabo [24] who then summarized them as a set of digital promises complemented with protocols to follow. Among requirements of such constructs, he enlists four objectives of common contract design:

1) *Observability* – stakeholders must be able access the performance of a contract.
2) *Verifiability* – an ability to prove that a contract has been performed or breached.
3) *Privity* – access and control of the contract should only be distributed as much as it needed to perform the contract.
4) *Enforceability* – power to execute the terms and protocol of the contract.

The gist behind smart contracts is to include business logic which must be followed to drive a state transition. For example, a conceptual smart contract of a collection fund could look like this: "[person] deposits an [amount] to be released to an [account] on a [date]". This smart contract could then be utilized by, say, Alice to automatically give Bob $10 000 if the date is 01.01.2020. This kind of invocation is called *contract execution*.

However, at the time of 1996 [24], smart contracts could not have been implemented due to technological limitations. Now, blockchain is starting to prove itself capable for the matter [3], [22]. More specifically, in the context of blockchain, smart contracts are introduced as special records written as a program code which could then be deployed and manipulated using transactions. They make it possible to write blockchain applications (called *distributed applications* or *dapps*). However, naturally, programming such contracts should follow strict rules in order for the blockchain to keep its trustlessness.

2.4 Ethereum

One of such platforms that implements programming of smart contracts in the trusted environment of blockchain is Ethereum [3]. Smart contracts could be written, deployed and used in either the public or a private Ethereum blockchain [25]. What makes Ethereum platform special, is its popularity which could, for example, be characterized by the number of active nodes (22319[7]). As the Ethereum platform was launched only in 2014, the growing number of nodes as well as its open-source nature[8] could be considered a sign of trust and reliability. Good reputation, built-in incentives, self-enforcing protocols and verifiability of transactions of Ethereum – qualities required for enforceability [24] – all contribute to make it a viable candidate at the time to target distributed application logic. Additionally, from the technical perspective, the high number of nodes also acts as a measure that helps to mitigate a potential security threat which would realize if a malicious party operates more than half of the nodes on the network which would help him manipulate with the consensus [1].

Workflow

The contracts in Ethereum are written using a specific programming language called Solidity. While the syntax and capability of the language is actively evolving, it provides necessary instructs to write smart contracts which could be used to capture more complex data structures as well as, for example, do looping and branching for advanced scripting. [4]

Solidity smart contracts are targeted to be executed on Ethereum Virtual Machine (EVM) which is a Turing-complete, isolated runtime environment with no access to network,

[7] https://ethernodes.org/network/1 [Accessed: 24-Nov-2017 01:03]
[8] https://github.com/ethereum [Accessed: 19-May-2018]

filesystem, other processes or any information which is outside of the blockchain or transaction [3]. An implementation of EVM is included in client node implementations (e.g. Geth [25]) to evaluate calls (read-only invocations) and transactions (invocations which change the state of the network). For user interactions, client nodes expose a JSON RPC API (JavaScript Object Notation Remote Procedure Call Application Programming Interface) [26]. A simplified workflow through RPC calls with a smart contract on Ethereum could be described as follows:

1) An EVM-compiled smart contract is deployed. This means that a transaction with a payload of contract bytecode and initialization parameters is sent to the pool of pending transactions managed by network nodes. As an output, a transaction hash is returned which could be used to check its status in regards with the next step. [26]
2) The miners take the transaction with the contract, validate it and run the initialization block to instantiate an initial state. If the transaction has been confirmed enough times in the network (i.e. it has been in enough blocks), a transaction receipt is issued and the contract will now have an artifact with an associated address and an application binary interface (ABI) [3], [26]. Similarly to external accounts managed by key pairs, the contract address has an associated balance and they are handled equally by the EVM [3].
3) The instantiated contract at a specific address could be invoked by calling the defined methods using the ABI. Again, if the message modifies the state of the network, the miners will run the code and upon it has been mined, the state of the contract is updated. If the ABI call does not modify the state (i.e. it is read-only), there is no need to propagate anything into a network and the call could be immediately served within a single node, free of charges.

In a development environment, the management of RPC calls and different invocation callbacks could be supported by frameworks such as web3js[9] and/or Truffle[10]. In addition to providing higher level language abstraction over the ABI invocations, Truffle also provides streamlined measures for contract compilation, linking and testing purposes.

Fee System

Deploying a smart contract or invoking a transactional write method requires resources from the Ethereum network to make changes to the ledger. To compensate the miners, a fee system is in place. A fee is measured in an internal currency tokens called Ether and it is reduced from the balance of the sending address when a transaction is mined. However, as Ether is also has a (non-fixed) value with fiat currencies[11], an internal notion of *gas* is introduced to disjoint the transaction execution cost/metering from fiat value. Therefore, the relation between transaction fee and an execution value could be represented with the following formula:

$$transaction\ fee\ (Ether) = \ execution\ cost\ (gas) * \ gas\ price\ \left(\frac{Ether}{gas}\right), \qquad (2.1)$$

where *execution cost* is the amount of gas that the miner uses to execute the transaction and *gas price* a parameter which sets a ratio to translate gas to Ether. [3]

When deploying a smart contract or sending a transaction, the invoker must consider multiple parameters, such as gas price [23]. For finding an optimal value to get a transaction

[9] https://github.com/ethereum/web3.js/ [Accessed: 19-May-2018]
[10] http://truffleframework.com/ [Accessed: 19-May-2018]
[11] https://www.coindesk.com/ethereum-price/ [Accessed: 19-May-2018]

chosen and mined within the pool of all pending transactions, the services of a client node could be used to reveal a median gas price the from last block[12]. However, using higher than median gas price helps to get the transaction mined faster as they are more profitable for miners [3], [23].

The sender must also set the amount of gas available for transaction execution. Moreover, to get a record successfully mined with optimal cost, this parameter should be between certain range. The lower bound of the range is the execution cost of the transaction. Similarly to gas price, an estimation of how much gas is required to execute a transaction could be retrieved from a client node[13] which is capable of doing a dry run of an execution without adding it to the blockchain [26]. If the realized cost of a transaction execution is lower than the set limit, the remaining of the gas is refunded [3]. However, if the realized gas consumption exceeds the amount set by transactor, a node throws an `Out of gas` exception at which point a transaction would be reverted (if not specifically coded otherwise) but a fee of the specified limit is not refunded [3]. On the other hand, there is also an upper boundary for transaction execution as EVM is specified to be *quasi*-Turing complete to limit the computations [3]. The upper boundary is in place so that the system could not be clogged with, for example, infinite loops. In practice, an upper boundary of block gas limit should be considered. To validate that the required gas for a transaction does not exceed the upper boundary, the client node could be queried to check the gas limit of the latest block[14].

The basis for metering of transaction execution gas is a pricing table which includes all possible executable atomic operation defined in EVM. For example, a base fee of 21000 gas must be paid for each transaction or 32000 when creating a contract. In addition to a flat fee of 32000, 200 gas per byte must also be paid for contract bytecode storage. Writing to contract storage could also be considered one of the most impactful operations: 20000 gas must be paid when setting 32-byte value in storage from zero to non-zero using opcode of `SSTORE` or 5000 for the same operation when modifying a previously set value. On the other hand, each execution of `AND`, `OR`, `MLOAD` and `MSTORE` operations have a flat cost of 3 gas. The cost for EVM memory usage (always expanded in 32-byte words) is linear until 724B after which it grows substantially faster. [3]

2.5 Application of the Blockchain Technology

The following is a review on the scientific literature which study applying blockchain on a broader scope of domain. The purpose of the review is to give an overview to the current state of the art.

Spielman [27] evaluates the applicability of blockchain for land registry as a recorder of deeds. The proposed system implements the concept of *Smart property* originally proposed by Szabo[15] through coins which represent real estate. To transfer ownership, the representing coin is transferred from one account to another. As one of the benefits, Spielman brings out that the smart property is able to collect verifiable signatures throughout the life cycle of the transfers and trace the ownership history. The author also proposes a hybrid model of the application to save data to a centralized database and maintain a hash of the data on-chain for verification purposes.

[12] https://github.com/ethereum/wiki/wiki/JSON-RPC#eth_gasprice [Accessed: 19-May-2018]
[13] https://github.com/ethereum/wiki/wiki/JavaScript-API#web3ethestimategas [Accessed: 19-May-2018]
[14] https://github.com/ethereum/wiki/wiki/JSON-RPC#eth_getblockbynumber [Accessed: 19-May-2018]
[15]http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/idea.html [Accessed: 19-May-2018]

Eberhardt and Tai [28] find that the expensive storage and computational power of the Ethereum platform may negatively impact performance and scalability of the implementation. Moreover, they also claim that often the technical limitations of the Ethereum platform do not allow an application to fully operate on the chain. For example, the very core of the blockchain operation such as transaction validation, consensus establishment and smart contract execution may create too big of an overhead and takes considerable amount of time. So, based on the identified efficiency and technical issues, they explore how to move operations off-chain without losing the benefits of the Ethereum blockchain.

Eberhardt and Tai make an example of an Ethereum blockchain chess game which persists the game state in a smart contract. While the blockchain application removes the intermediary to make the game logic require no explicit trust, it is found that the algorithm of checking the check mate condition is too complex for on-chain transactions. So, it is learned that highly complex computations such as chess end-game checks should be done on the client side and/or as rare as possible on the chain, using an initiation of an actor. [28]

Eberhardt and Tai also implement a marketplace to enable intermediation between providers and consumers of service APIs using cryptocurrency for payments. In the system, service providers are able to expose their API descriptions for discovery and consumers could buy access to them. In this case, it turns out that the necessary API descriptions are too big for efficient storage on the blockchain. On the subject, they also find that it is not possible to use a reference to off-chain data because this would defeat the purpose of decentralization. Secondly, the marketplace implementation introduces an issue of privacy when consumers need to prove their purchased access by sending a private token to the blockchain for validation. All in all, they learn that there is a necessity for optimization schemas to store data off-chain in a trustless and privacy-preserving manner. In addition, they call out to develop techniques to use off-chain computations on private data and then use smart contracts only for output validation to preserve privacy. [28]

To show how the study relates to the existing literature, the next chapter presents the contribution of the study.

3 Contribution

This chapter encapsulates the contribution of the research by describing the requirements for a real estate rental service (3.1) and the design of three prototypes (3.2, 3.3, 3.4). The structure of each section includes both results and short passage to the next section in the form of discussion on the built artifacts.

The results of problem identification and developing a baseline prototype is a collaborative effort with Kopylash [29].

3.1 Problem Identification

The problem is identified by studying the domain of the real estate rental. The case is studied through a 7 top-down manual interviews with a CEO of an industry partner who currently operates a business of real estate rental based on a centralized web application. The duration of the interviews varied from 60 to 120 minutes and the calls were being recorded for analysis. This section summarizes the information gathered through the interviews to present an intermediate-level description of the service under study. Based on the information gathered, the **objectives of a solution** are derived in the following subsection of analysis.

3.1.1 Service and Actors

From the domain perspective of the business case, a support for the process of real estate rental expected (Figure 4). There are three (external) actors interacting with the service: a landlord, a tenant and a moderator. Landlords represent the supply side of the market with a real estate to be rented through the platform. Tenants, on the other hand, act as the demand side: they interact with the ultimate goal to rent real estate. Lastly, the moderator is an actor who interacts with the system on behalf of the platform manager to add business value by accepting and rejecting property enlistment applications based on manual checks made off-system.

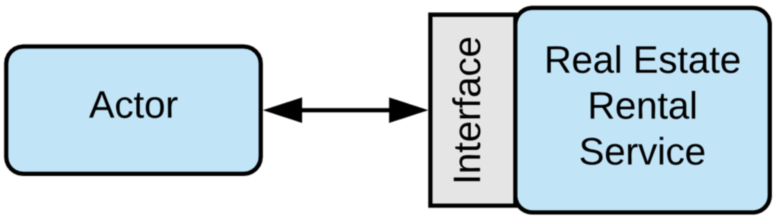


Figure 4. Service map of a real estate rental platform.

3.1.2 Business Process

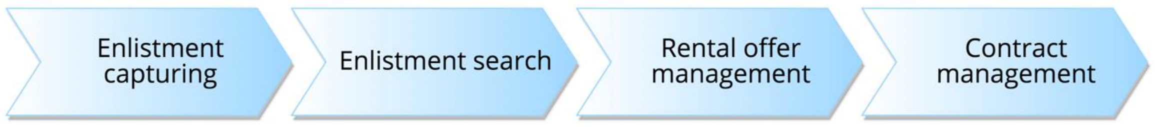


Figure 5. Real estate rental platform value chain.

The value chain of a real estate rental platform is based on four intermediation activities between landlords and tenants (Figure 5). The platform allows users to enlist a property which would then appear in the application. The intermediation service then continues by allowing tenants to make bids on an enlistment to implicate an interest and for a landlord to respond to the offers. Finally, the intermediator also allows the stakeholders to construct and exchange tenancy contracts.

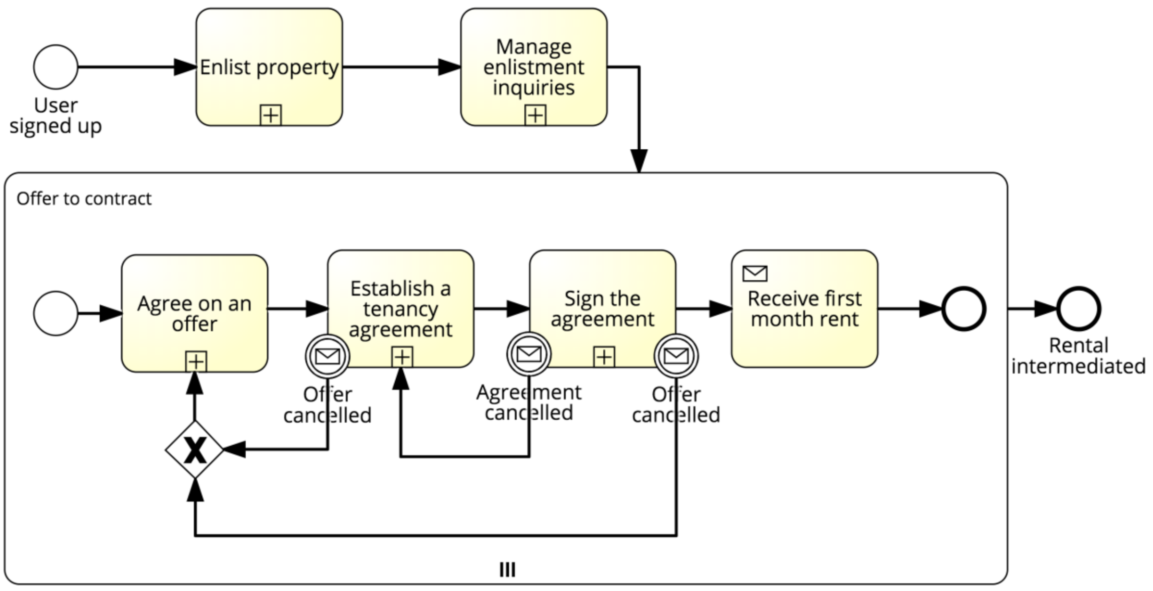


Figure 6. Main process of real estate rental.

From the perspective of the actors, the value chain expands to a main business process consisting of 6 steps starting with the event of user being signed up and finishing when the rental activites have been completed (Figure 6). Next, the details of each step and flows of the main process are elaborated on.

Enlist Property

Firstly, an enlistment must be posted to the marketplace by a landlord (Figure 7). This enlistment contains data about the landlord as well as the real estate property, including: landlord name; street name; house, floor, apartment, zip code numbers; geographical location of the property.

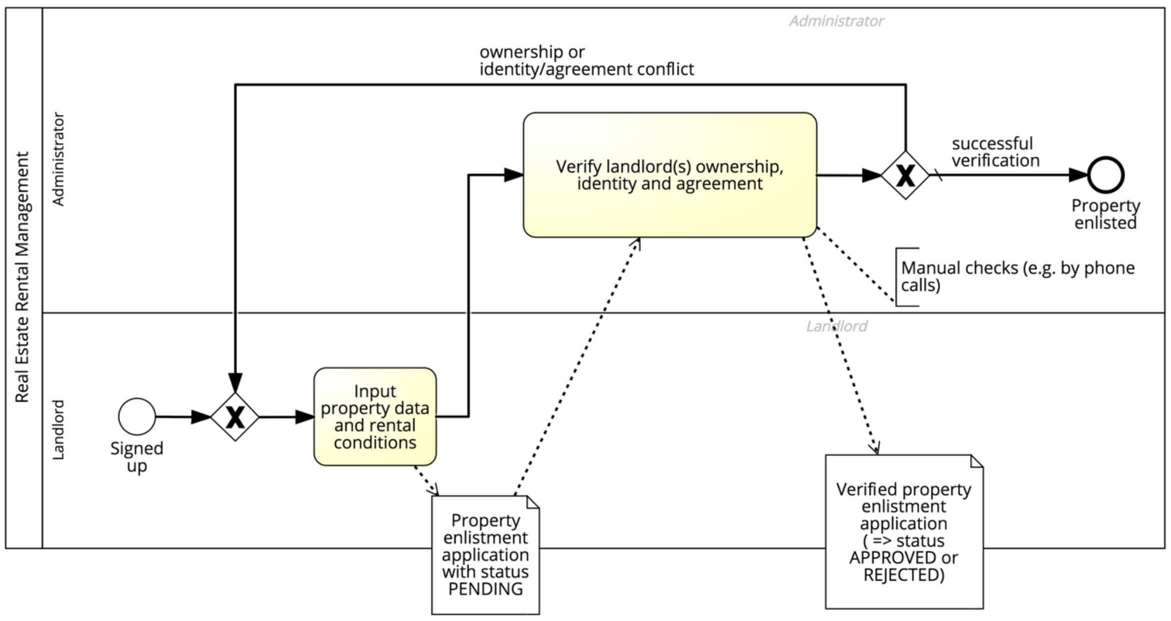

Real Estate Rental Management
Administrator
Landlord
Administrator
Landlord
ownership or identity/agreement conflict
Verify landlord(s) ownership, identity and agreement
successful verification
Property enlisted
Manual checks (e.g. by phone calls)
Signed up
Input property data and rental conditions
Property enlistment application with status PENDING
Verified property enlistment application (=> status APPROVED or REJECTED)

Figure 7. Enlisting a property.

After the enlistment has been posted, it is processed for eligibility by a service provider to make sure that the landlord is the rightful owner of the property and that it is actually him (or someone authorized) making the enlistment request. Additionally, the moderator may do more tasks such as ensure that the images attached to the enlistment application are of acceptable quality. If approved, the enlistment is made public in the application.

Manage Enlistment Inquiries

After the enlistment is approved to the platform, the tenant is able to find it through a listing or a search query (Figure 8). If the tenant finds an interesting property, he may chat with the landlord or visit the property to gather more information about the suitability. From the perspective of a landlord, he needs to be on the other end of these requests: answer to all of the questions as well as host visits to property.

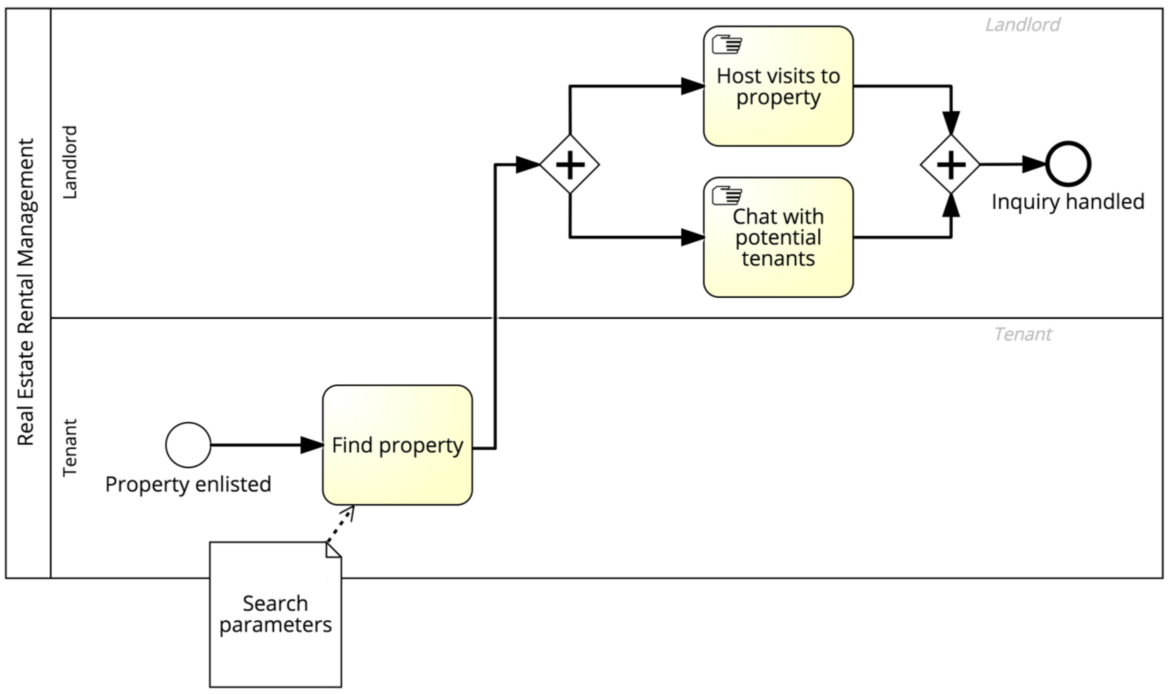


Figure 8. Managing enlistment inquiries.

Agree on an Offer

After the tenant finds a suitable property, he sends an offer for a rental payment to indicate interest towards the property (Figure 9). The offers are only visible to the owner of the enlistments and hidden from other tenants. To proceed, the landlord must accept the offer. It is also possible for a tenant to cancel an offer while it is pending. If the offer is rejected or cancelled and the tenant may submit a new one. From the perspective of a landlord, there may be multiple offers from different tenants to choose from at a given point of time. Also, there is no restriction for him to accept more than one offer.

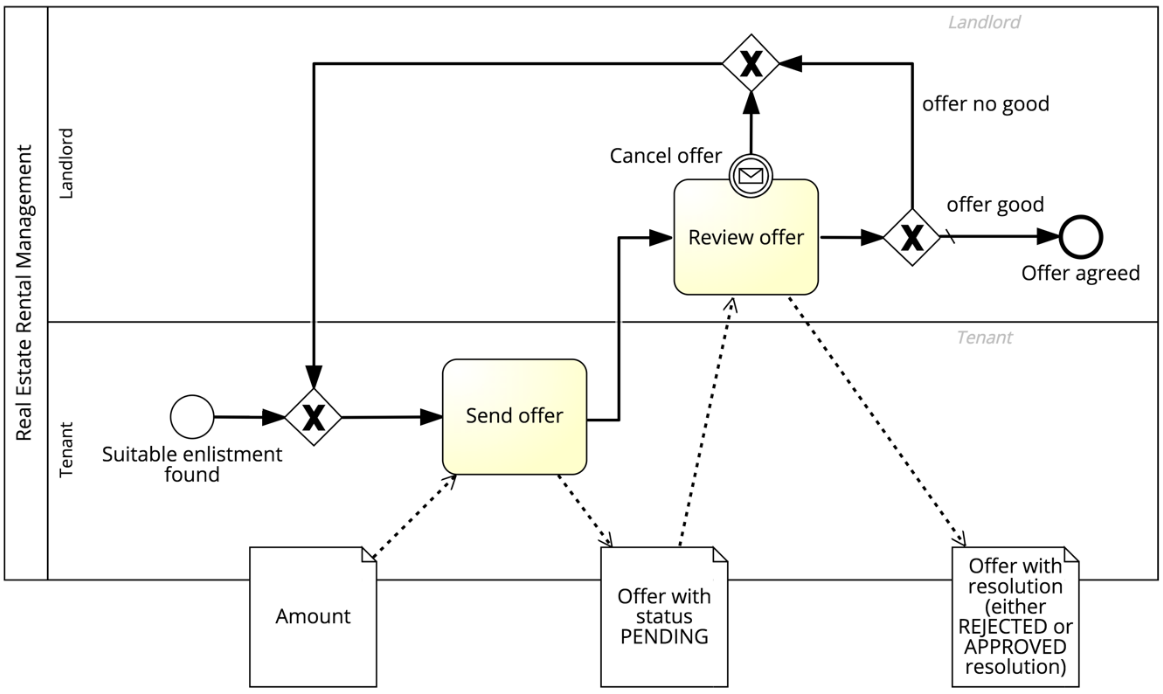
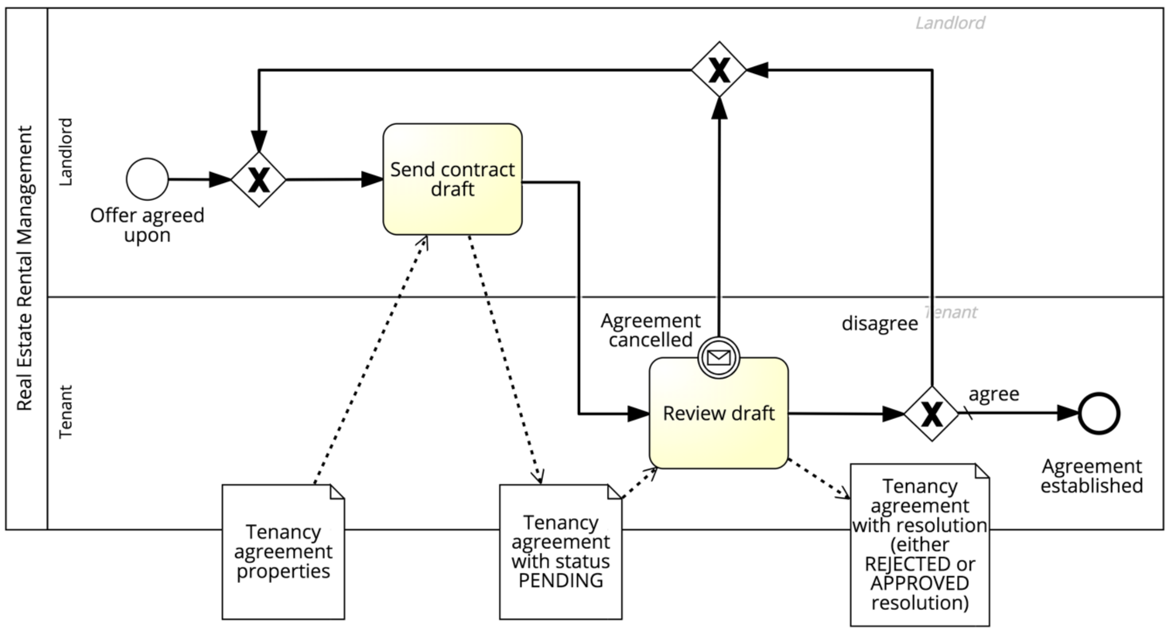

Real Estate Rental Management
Landlord
Tenant
Landlord
Tenant
Cancel offer
offer no good
offer good
Review offer
Offer agreed
Suitable enlistment found
Send offer
Amount
Offer with status PENDING
Offer with resolution (either REJECTED or APPROVED resolution)

Figure 9. Agree on an offer.

Establish a Tenancy Agreement

When an offer is accepted, the parties have to agree on the conditions of the rental by establishing a tenancy agreement (Figure 10). This is a feedback loop, similar to agreeing on an offer. But this time, the loop is initiated by the landlord by issuing a contract draft which includes data from the enlistment in a contract template document. As the official contract includes new (legal) details, the tenant may reject the contract draft to either propose adding, removing or editing the conditions. Additionally, the landlord is able to withdraw the contract draft while it is in review to make changes. The loop finishes when both parties agree on a contract which is indicated by an agreement from the tenant.

At this point, both parties may still back out of moving further with each other in the rental process by cancelling the offer. If a cancellation action is pursued, process flow moves back to agreeing on an offer (Figure 6).



Real Estate Rental Management
Landlord
Landlord
Offer agreed upon
Send contract draft
Tenant
Tenant
Agreement cancelled
disagree
Review draft
agree
Agreement established
Tenancy agreement properties
Tenancy agreement with status PENDING
Tenancy agreement with resolution (either REJECTED or APPROVED resolution)

Figure 10. Establish a tenancy agreement.

Sign the Agreement

Next, the parties sign the agreement to make the contract legitimate (Figure 11). As a practice, landlord as the contract issuer is the first to sign, followed by the tenant. Similarly to agreeing on a contract, both parties have the freedom to back out to one of the previous phases of the process by either cancelling on an established contract draft to start building a new one or reset the interaction altogether by pulling back the offer (this also cancels the agreement). Cancelling the offer or the tenancy agreement is possible up until the point where tenant has signed the agreement .

As an additional business rule, no more offers could be sent after the landlord has signed an agreement and it is not possible for him to sign more than one tenancy agreement for an enlistment at the time. This constraint is in place to avoid spam behaviour which can be originated by the landlord (double spending) or tenant. On the one hand, it makes the landlord to choose wisely as upon signing he will be committed to the contract and essentially locks the enlistment for potentially better offers. On the other hand, it suggests the tenant to act upon signing request quickly as the landlord may cancel at any time.

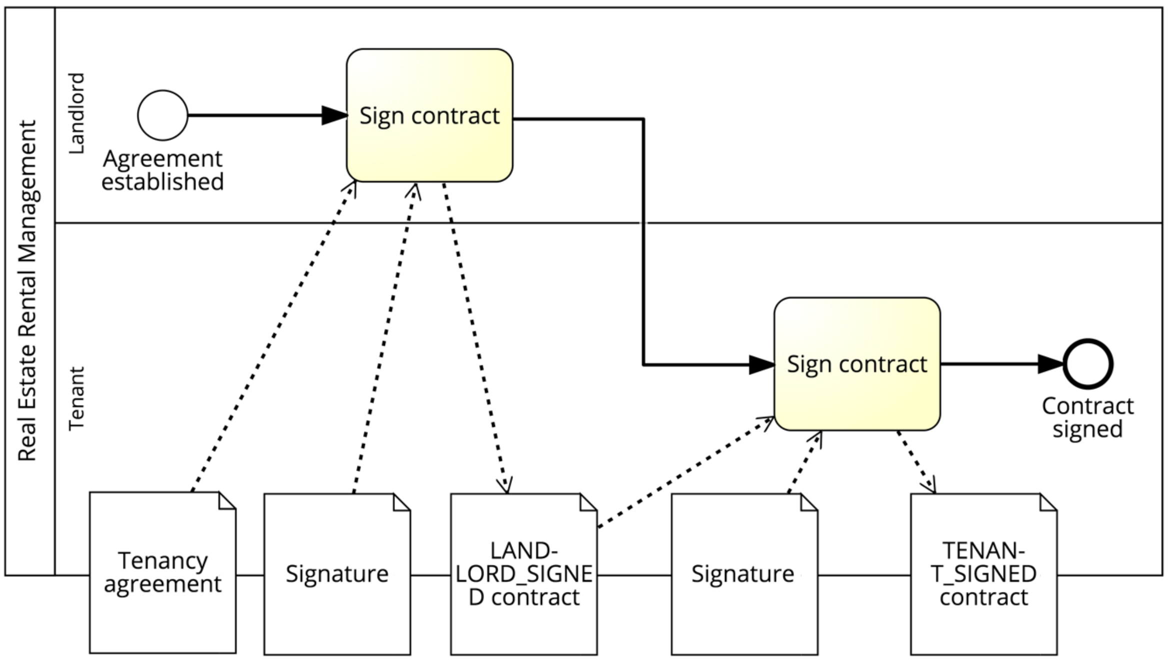


Figure 11. Sign the agreement.

Receive First Month Rent

To finish off the intermediation, the system also waits for a notification about the successful start of the tenancy agreement contract by requiring a confirmation about the receival of the first month rent. No additional rules apply here as this is only a business task to track that the intermediated contract is in action.

3.1.3 Exclusions

It must be noted that the captured process of real estate rental excludes some details to simplify the scope of the research:

1) There is only one tenant signing the agreement while in reality, there may be multiple.
2) It should be possible for a landlord to withdraw an enlistment at any time and modify its details (by possibly triggering the moderator review loop again).
3) It should be possible for a landlord to cancel the enlistment from the moderator review.
4) Based on the specific business model of an intermediator, additional steps are usually in place to collect service fees (e.g. by either paying per enlistment, by enlistment listing time or a cut from the first month rent payment).

3.1.4 Operations

From the captured business process of real estate rental, a list of provided atomic operations have been identified. An overview of such operations could be found from Table 1 where they are organized by their nature (write or read) and the underlying resource.

Table 1. Provided operations of real estate rental management service.

Resource	Write operations	Read operations
Enlistment	- Submit - Review (approve or reject)	- Retrieve one - Filter by reviewed - Filter by landlord - Filter by tenant (as the bidder) - Filter by geographical proximity for a given location and a search radius
Offer	- Send - Review (approve or reject) - Cancel	- Retrieve one for a given enlistment - Retrieve all for a given enlistment
Agreement	- Submit - Review (confirm or reject) - Cancel - Landlord sign - Tenant sign - Receive first month rent	- Retrieve one for an enlistment

3.1.5 Analysis

It is interpreted from the captured business process that the interactions between the landlords and the tenants and the role of the intermediator resembles of a two-sided market model [16]. What is more, the role of the real estate rental service provider aligns with the description of a platform which in the form of landlords and tenants has two distinct group of users who are both subject for revenue generation [7]. In the context of marketplaces, the discovered business model also captures the resource (i.e. real estate property) which is subject to sharing (i.e. renting).

Therefore, the **objectives of the solution** follow the principles of sharing economy and the main goal is business disintermediation to reduce overhead costs. As next, a system is described which provides the discovered process and uses a blockchain in its architecture.

3.2 Baseline Design and Development

This chapter describes a baseline design of a real estate rental system which introduces Ethereum platform to the architecture of the solution. The design of the prototype is described through a series of signposts moving from the high-level service artifacts to a lower-level interface design. In addition, the study focuses on the key components of the application which have high impact on the system or which procedures change with consecutive prototypes and therefore provide input for efficiency evaluation. Therefore, for key components, a step-by-step operation drill-throughs are provided.

3.2.1 Service Map

From actor involvement point of view, the real estate rental business process analysis reveals that the last and only value adding manual activity carried out by the platform operator (i.e. moderator) is auditing the property enlistments. Based on this observation, the services of a real estate rental platform could be abstracted in an alternative value chain, consisting of two segments: *Enlistment review* and *Enlistment to contract* (Figure 12). The first of the two segments encapsulates the subprocess involving a landlord and a moderator going through the motions to publish the enlistment. On the other hand, the rest of the business process in *Enlistment to contract completion* now only includes activities carried out by tenants and landlords.

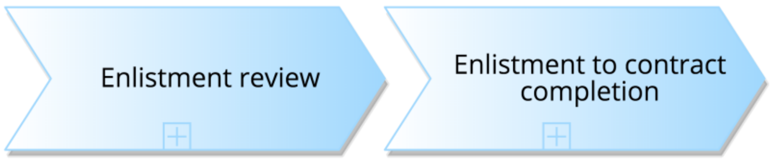


Figure 12. Abstract value chain of a real estate rental service.

Originating from the actor involvement driven process split between *Enlistment review* and *Enlistment to contract completion*, the *Real Estate Rental Management Service* (Figure 4) is divided into two corresponding subservices: an externally accessible *Enlistment Review Service* (*ERevS*) and a required internal *Rental Intermediation Service* (*RIS*) (Figure 13). Both, *ERevS* and *RIS* are responsible for the execution/progress of their respective subprocesses. Additionally, there is also a difference in the underlying implementational technologies of the services: *RIS* runs on an Ethereum distributed ledger node [25] while the *ERevS* is chosen to be implemented using a NodeJS[16] server application coupled with a relational PostgreSQL[17] database system[18].

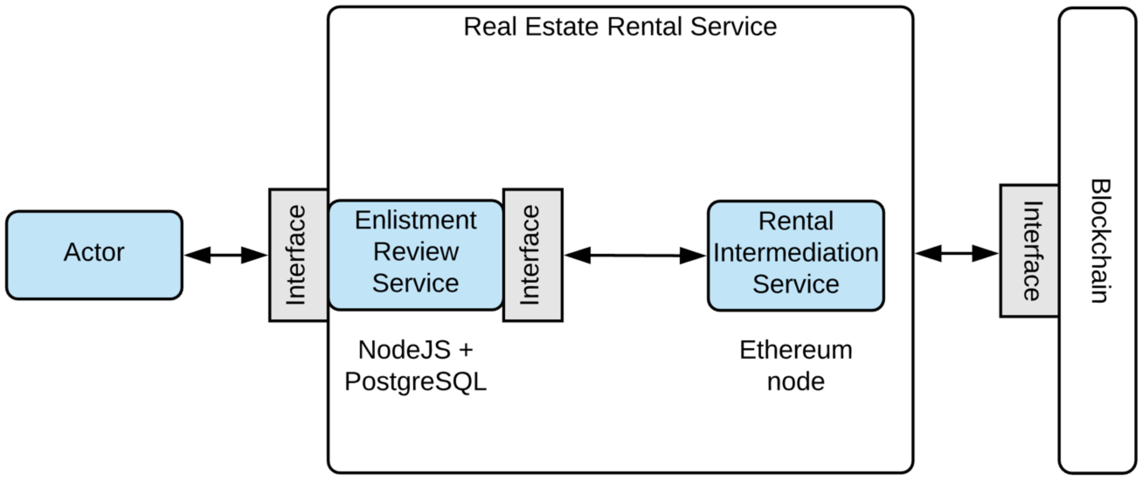

Real Estate Rental Service
Actor
Interface
Enlistment Review Service
Interface
Rental Intermediation Service
Interface
Blockchain
NodeJS + PostgreSQL
Ethereum node

Figure 13. Service map of the baseline architecture.

[16] https://nodejs.org/en [Accessed: 03-May-2018]
[17] https://www.postgresql.org [Accessed: 03-May-2018]
[18] Other implementational technologies may be used in place of NodeJS and PostgreSQL. NodeJS was preferred due to the JavaScript language compliance with widely accepted web3js library.

3.2.2 Resource State Transitions

Three core assets could be identified from the discovered business process: an enlistment, an offer and a tenancy agreement. When the process moves forward, the respective asset states change in a predictable manner through a set of possible events defined at each step. So, by considering business process execution as a series of state transitions on the underlying assets, one could determine the progress by examining their collective state.

Based on the states identified from the business process, a corresponding life cycle - presented as a state machine - and a coupled event interface have been designed[19] (see Figures 14-16). At its root, the state machine has two orthogonal regions to represent the parallel life cycles of the resources which are managed off-chain (enlistment) or in the smart contract (offer and agreement). When an enlistment is accepted, the state machine enters the nested smart contract state as this is the spot where the process execution control flow moves on-chain. Next, offer life cycle starts when it is submitted. Subsequently, agreement waits for offer to reach state “Accepted” and a first submission of the contract draft by a landlord. Only when the agreement is accepted, signed and started, the state machine finishes. The state machine is also complemented with business logic to enforce the flow of the real estate rental process. For example, it contains enlistment locking and offer cancelling procedure which propagates the event to also cancel the agreement (see 3.1.2).

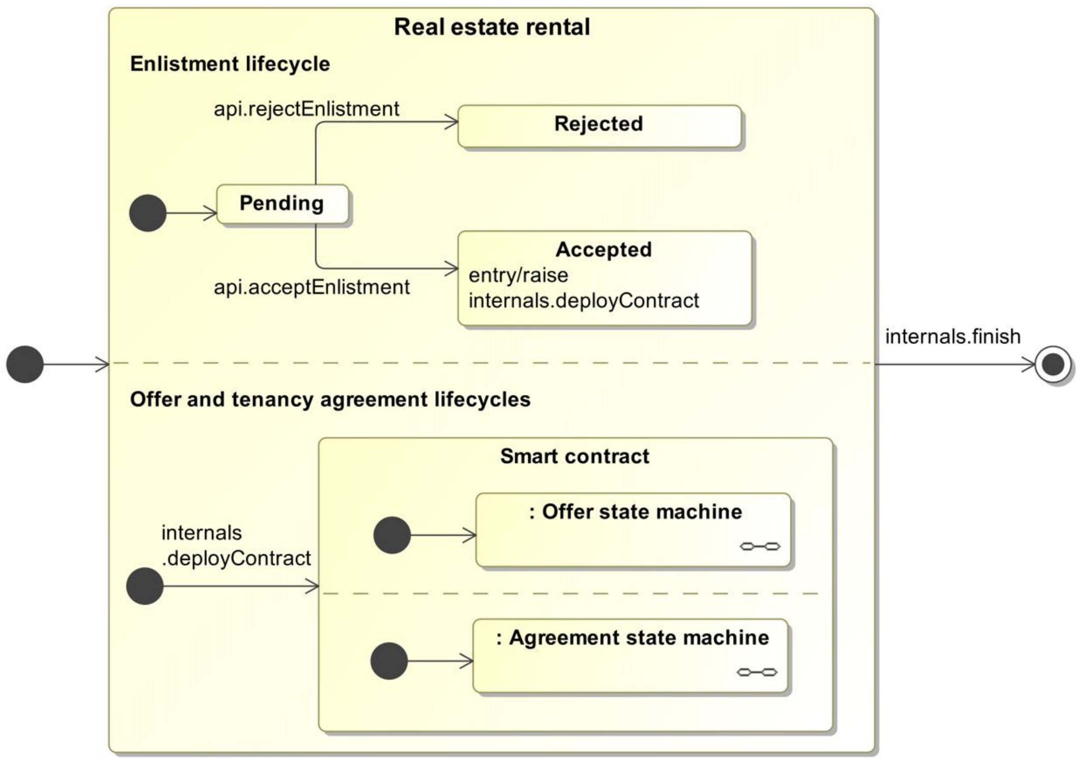


Figure 14. Real estate rental life cycle state machine.

[19] For presentational purposes, the state machine is only conceptual since the initial transitions must not have triggers.

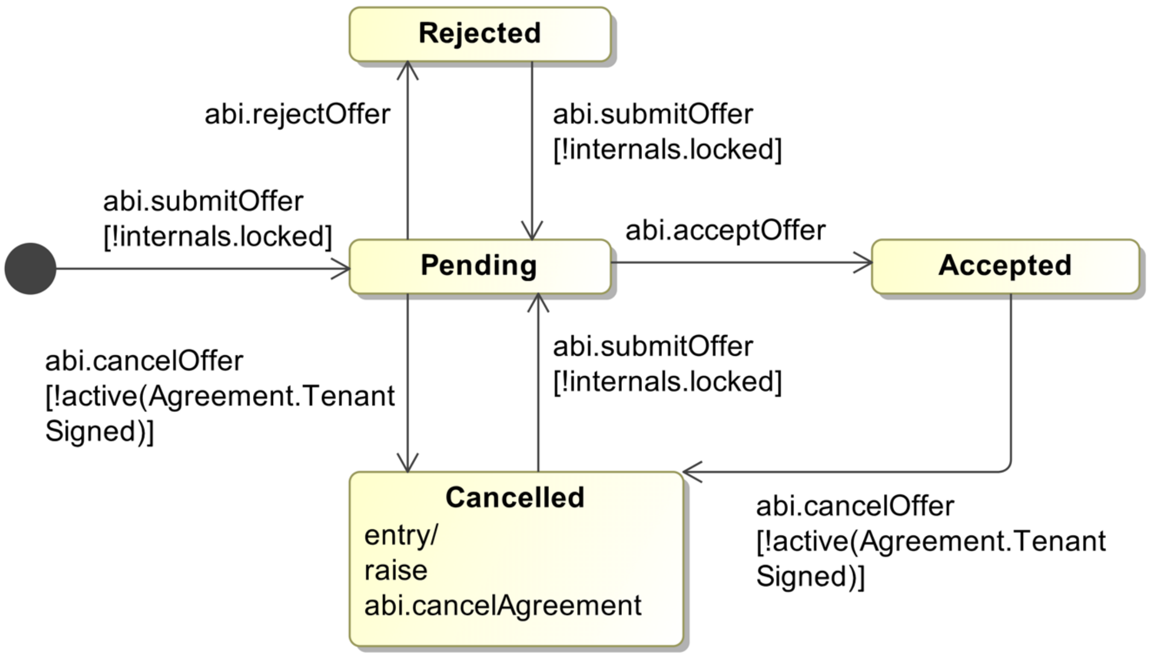

Rejected
abi.rejectOffer
abi.submitOffer [!internals.locked]
abi.submitOffer [!internals.locked]
Pending
abi.acceptOffer
Accepted
abi.cancelOffer [!active(Agreement.Tenant Signed)]
abi.submitOffer [!internals.locked]
Cancelled
entry/ raise abi.cancelAgreement
abi.cancelOffer [!active(Agreement.Tenant Signed)]

Figure 15. Offer life cycle state machine.

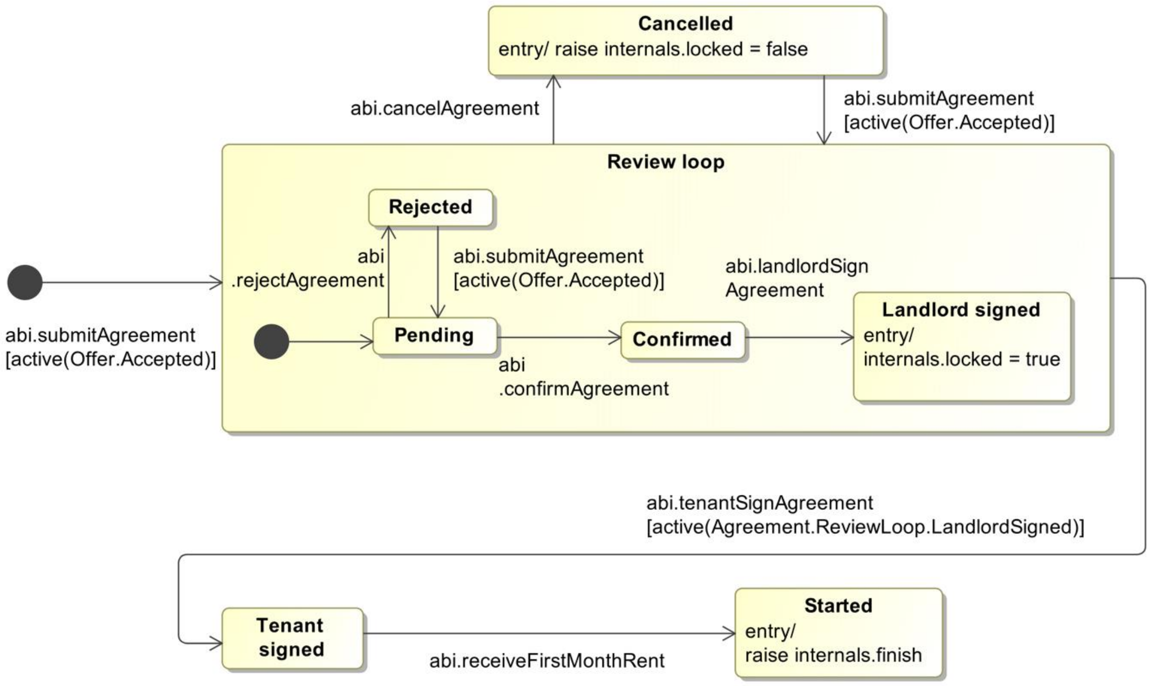

Cancelled
entry/ raise internals.locked = false
abi.cancelAgreement
abi.submitAgreement [active(Offer.Accepted)]
Review loop
Rejected
abi .rejectAgreement
abi.submitAgreement [active(Offer.Accepted)]
abi.landlordSign Agreement
Landlord signed
entry/ internals.locked = true
abi.submitAgreement [active(Offer.Accepted)]
Pending
Confirmed
abi .confirmAgreement
abi.tenantSignAgreement [active(Agreement.ReviewLoop.LandlordSigned)]
Tenant signed
abi.receiveFirstMonthRent
Started
entry/ raise internals.finish

Figure 16. Agreement life cycle state machine.

3.2.3 Data Model

Based on the asset state transitions identified and views required to assist the actors to fetch relevant information to move the business process forward, the service components implement the application based on the conceptual application class model of the system (Figure 17).

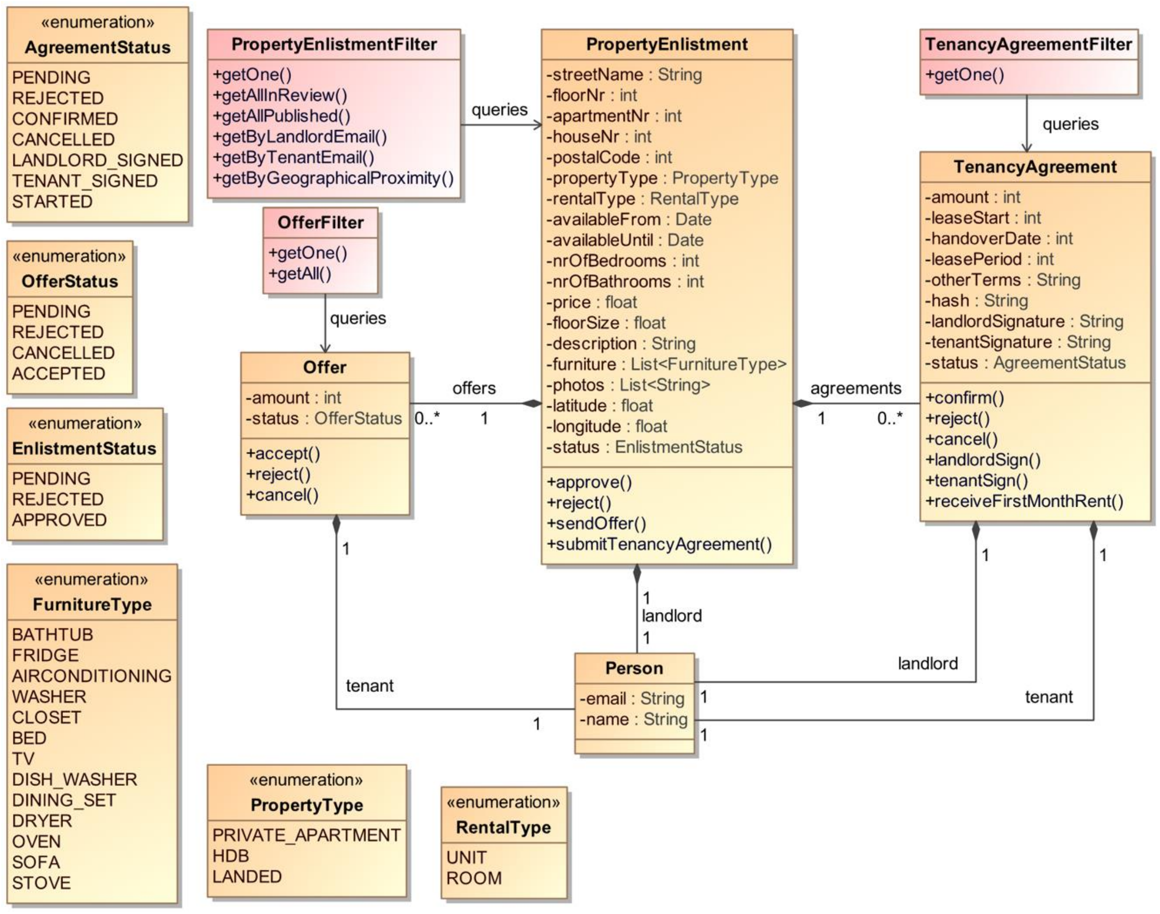


Figure 17. Application class model (conceptual).

However, from the moment the blockchain is included the application, a distributed system is created. From this point of view, it is required to understand which subsystem owns what information and how it can be modified and retrieved.

The process execution starts in *ERevS* (hereinafter referred to as **off-chain** component) as all the submitted details of the created enlistment[20] is persisted in a centralized database where it is assigned an unique ID. A moderator is able to query all enlistments from the database which are in review and may either approve to reject them (Table 1). If the enlistment is approved, one smart contract instance called *Enlistment* is deployed to Ethereum blockchain and its reference is attached to the object model in the (centralized) database. At this point, the control flow of the business process execution moves to *RIS* (hereinafter referred to as **on-chain** component).

[20] Uploading photos are managed off-system. The data model persists hyperlink references to the photo resource.

Until the enlistment is not approved, all the data is owned and could be queried and modified through the centralized database system in *ERevS*. However, after the smart contract instance gets deployed and it takes over the process execution control flow, the data is split into two systems (Figure 18). In order to be autonomous in its write operations, the smart contains and manages all the data it requires to move the process execution forward and apply business logic. This means that objects of offers and tenancy agreements reside on the blockchain. However, together with the data required for operational autonomity, the Enlistment smart contract also deploys some identifying decorational data (i.e. intersection of *EnlistmentOffChain* and *EnlistmentOnChain* models) which has been verified by the moderator and would benefit from tamper resilient nature of the blockchain. This creates a vertical partition of the attributes of enlistment with the ownership of the ported data being transferred to the blockchain. Proceedingly, after data migration, the off-chain copy of the data could be considered dead.

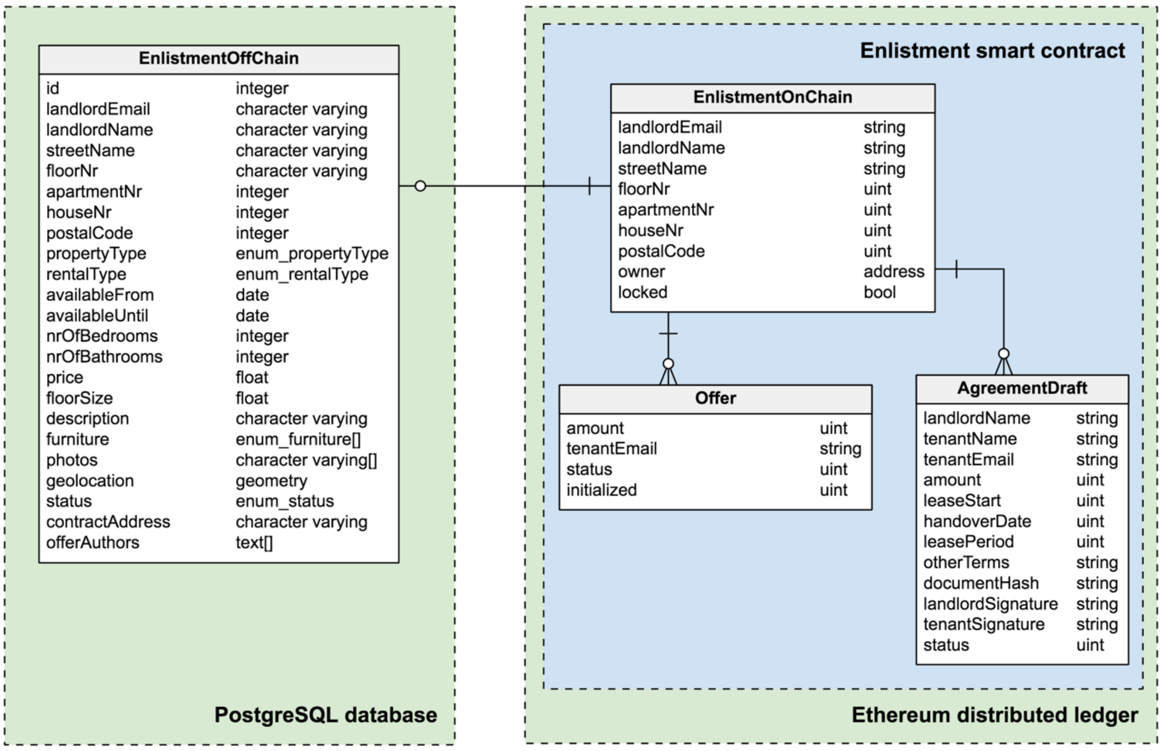


Figure 18. Data model (variant 1).

3.2.4 Interface Design and Key Operations

The ownership of the data shapes how resources and their collections are accessed and modified in the system. As the service is divided between two subservices, there is also a division of operations. Due to the fact that all external actor interactions are served through *ERevS*, the former also directly inherits the set of **provided** operations of a single service system (Table 1). However, in its operation, it relies on the **required** operations *RIS* (Table 2). In the next subsections, the interface and relevant operations of each service are dissected.

Table 2. Required operations from the *RIS* (variant 1).

Resource	**Write operations**	**Read operations**
Enlistment	- Deploy smart contract (implemented by the underlying Ethereum software)	- Retrieve one
Offer	- Send - Review (approve or reject) - Cancel	- Retrieve one for a given enlistment
Agreement	- Submit - Review (confirm or reject) - Cancel - Landlord sign - Tenant sign - Receive first month rent	- Retrieve one for a given enlistment

Interface and Operations of the Rental Intermediation Service

RIS is implemented as smart contract (Enlistment) which is designed to be operating on the Ethereum node. Enlistment smart contract implements the data model (Figure 18) through the use of Solidity structs[21] and mappings[22]. The constructed structs represent the entities in the data model[23]. The members of the structs are of elementary types: strings, unsigned integers, booleans and addresses. In addition to the data of the assets, the smart contract also holds stateful data which it needs to enforce the enclosed business logic. These values include statuses of the offers and agreement drafts which are implemented as enums in the source code. On the other hand, mappings are chosen to be utilized to store associations between entities. Using the email address of the tenant as the key of the mapping, the data structure gives a natural guarantee that there is only one offer or tenancy agreement in progress for a single tenant of the given enlistment.

The smart contract also implements business logic for its part of the process. For this reason, first and foremost, the smart contract enforces the sequencing of the tasks in the business process. This logic is implemented into a script utilizing function modifiers[24] on the methods which take the process into the next stage (e.g. all write functions of Table 2). Modifiers inject code to the beginning of the function and, so, offer a declarative way to include checks and throw exceptions before moving on to serve a user transaction. For example, the function for the action of cancelling an offer is annotated with 3 modifiers (Code block 1):

[21] http://solidity.readthedocs.io/en/v0.4.19/types.html#structs [Accessed: 19-May-2018]
[22] http://solidity.readthedocs.io/en/v0.4.19/types.html#mappings [Accessed: 19-May-2018]
[23] In the implementation, values of `owner` and `locked` are not actually part of the Enlistment struct but stored as separate contract variables
[24] http://solidity.readthedocs.io/en/v0.4.19/contracts.html#function-modifiers [Accessed: 19-May-2018]

1) `ownerOnly()` is used to implement access control to write transactions based on the owner property set to the address which deployed the smart contract. The value of owner is set in the constructor function when the contract is first initialized.
2) `offerExists(tenantEmail)` ensures that the tenant actually has made an offer to the enlistment. For this purpose, the modifier turns to the respective association mapping with the tenant email and checks whether the mapped offer is initialized.
3) finally, `offerCancellable(tenantEmail)` ensures that the tenant is allowed to cancel his/her offer at the given phase of the business process execution as depicted on Figure 15.

```
function cancelOffer(string tenantEmail) payable public
        ownerOnly()
        offerExists(tenantEmail)
        offerCancellable(tenantEmail)
    {
        tenantOfferMap[tenantEmail].status = OfferStatus.CANCELLED;
        if (tenantAgreementMap[tenantEmail].status != AgreementStatus.UNINITIAL-
IZED) {
            tenantAgreementMap[tenantEmail].status = AgreementStatus.CANCELLED;
        }
        locked = false;
    }
```

Code block 1. Smart contract function for cancelling an offer.

As one of the requirements of the business logic, the system must also ensure that landlord is able to have only one signed tenancy agreement at the time. For this reason, a simple boolean flag variable of `locked` is used which value is modified and validated when required by the process (Figure 17).

Interface and Operations of the Enlistment Review Service

Based on the business relevant operations identified in the service analysis (Table 1) and resource associations established by the application class model (Figure 17), a resource model is designed which reveals the structure of the underlying resources (Figure 19).

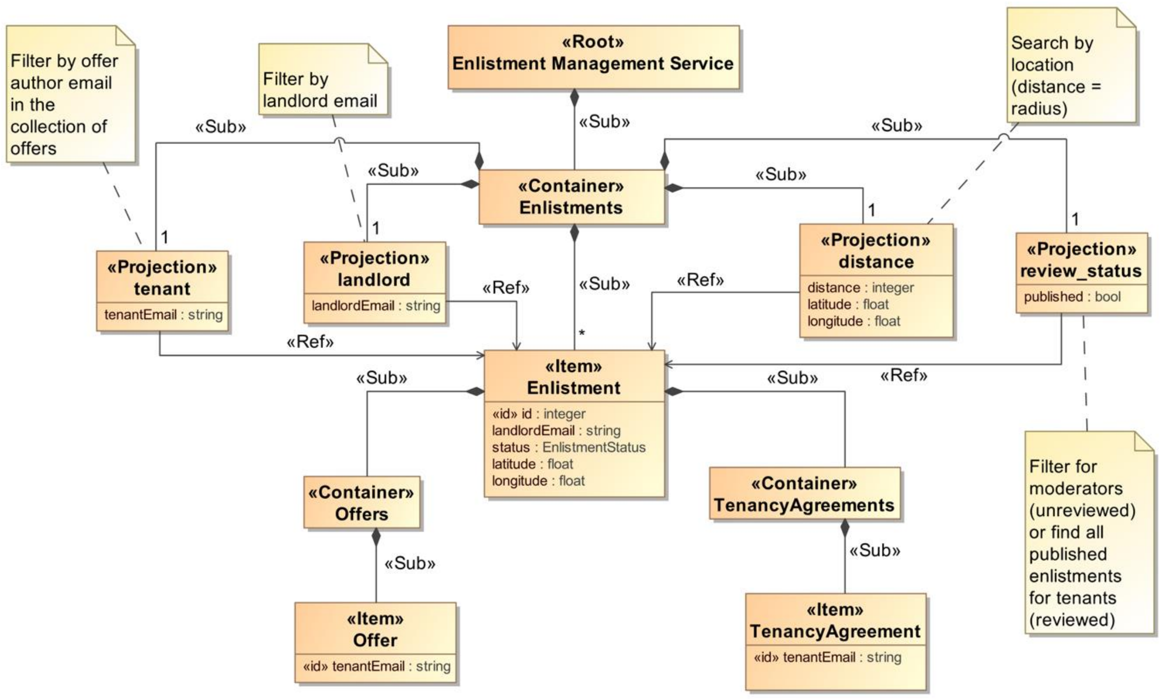


Figure 19. Enlistment Review Service resource model.

The exposed API of *ERevS* expects the primary key database reference of the enlistment entity (ID) which it then maps to on-chain reference (i.e. Ethereum address) if necessary for the request. For example, to retrieve an approved enlistment, an HTTP GET request of `/enlistments/:id` would need to be called which, in turn:

1) retrieves the referenced object from the relational database;
2) calls Enlistment smart contract function to retrieve the model from the blockchain using the associated address from the previous step;
3) merges the two objects and returns the result.

On the other hand, *ERevS* is also an access point to resources of *RIS* which owns and manages child entities of an enlistment: offers and tenancy agreements. For this reason, it requires the on-chain identifier for its subresource requests. That is, to review a tenancy agreement, an HTTP POST request to `/enlistments/:id/agreements/:tenantEmail/review` would be called. As a response, the procedure mapped to the API endpoint will:

1) retrieve the object referenced with an ID from the relational database;
2) calls Enlistment smart contract function to review the agreement with the parameters of tenant and review resolution;
3) await for the transaction to be mined and then send response.

Analogous procedure as previously described is used for all application actions which require a write transaction to be transmitted to the blockchain, i.e. operations of offers and tenancy agreements in Table 1.

In the proposed design, *ERevS* is also responsible for managing the collection of the enlistments. This makes the service a natural choice for all filtering operations. As a result, the service provides for the type of filtering operations:

1. Finding either all published or unpublished enlistments. For this action, a simple filter query to the centralized database is sufficient which includes an enlistment item when its status is approved.
2. Retrieving enlistments by landlord email. A simple `SELECT` query is sent to the relational database which retrieves all enlistments based on the input parameter of landlord email.
3. Retrieving enlistments by offer tenant email. The query expects iterating through 2 levels of entities: enlistments and offers. To simplify the procedure, the data of the enlistment offer authors is mirrored off-chain. So, to retrieve enlistments by tenant email, the filtering is done off-chain by running a `SELECT` query over all enlistments which include the input tenant email in the helper array of offer authors (Figure 18).
4. Filtering enlistments based on geographical proximity. The filter allows to run search queries in *ERevS* based on the input WGS 84 [30] latitude, longitude and search radius parameters. The search is implemented on a database service level using a PostGIS[25] spatial database extender for PostgreSQL. The underlying procedure sends a `SELECT` query to the database, utilizing `ST_Distance_Sphere`[26] function of the PostGIS library to filter on enlistments for which the calculated distance is smaller than the one specified in the input parameters.

In addition to the procedures dissected above, due to the vertical split of the enlistment attributes between multiple services, a round trip to *ERevS* is required in order to merge the data of all the filtered elements.

3.2.5 Analysis

The established system provides all the necessary operations for the real estate rental process. The results also show how Ethereum is introduced to the system and what is its role. Moreover, the basis of the blockchain integration proves to be the artifacts generated in the problem identification phase. In more detail, a subprocess is identified which only includes the intermediation between a landlord and a tenant. The identified subprocess is thereupon extracted from the main business process and abstracted to a dedicated service for an execution using a smart contract.

In the proposed architecture, blockchain is added as an internal service which is inaccessible to external actors. From the functional aspect, this means that as opposed to features of standard decentralized applications, end users do not need to own a cryptocurrency wallet to use the service. The associated transaction costs must be paid by the platform operator. Moreover, the only noticable changes to external actors should be non-functional. Next, the perceived changes on tamper resilience, availability and privacy are noted.

Tamper Resilience, Availability and Privacy

In a fully centralized architecture, there is a risk of data tampering as there may be people in the production line or with malicious intent who have credentials and/or direct access to the database. Inherently, this kind of access opens the application to a potential security threat as the person may be able to change the data for fraudulent purposes. However, by introducing blockchain to the architecture of the system under research, the risk of data tampering is mitigated with the underlying properties of Ethereum block validation mechanisms [3].

[25] https://postgis.net/ [Accessed: 19-May-2018]
[26] https://postgis.net/docs/manual-1.4/ST_Distance_Sphere.html [Accessed: 19-May-2018]

However, with the proposed service design, there is still a single point of entry to the system (*ERevS*) which may be subject to issues such as down-time and data loss. As a result, it could be concluded that the exposed service of the variant does not benefit from higher read availability usually associated with decentralized applications [23]. However, an increase of availability is perceived for the internal service running on smart contract.

What is more, the data model of the smart contract exposes a threat to the privacy of the users. Namely, the current design exposes private information of the actors such as their email addresses to a (public) blockchain where it could be easily queried or decoded. However, the purpose of an email address in the established design is crucial: it is the unique identifier which identifies actors on-chain (based on the hypothetical user model data gathered off-chain). The email addresses are also used as association keys between on-chain resources. So, trade-offs in privacy should be noted as a limitation of the design.

As an alternative to email addresses, an internal off-chain ID could be considered to reference identities on-chain. This, however, would again lose the autonomy of the on-chain process by losing data completeness as also identified by Eberhardt and Tai [28]. Meanwhile, a viable solution may emerge when the on-chain service is made externally available in the next phases of the research. If the on-chain service is exposed externally, identities of Ethereum accounts could be considered for the place of email addresses. However, as also found by Spielman [27], using on-chain identities may have legal limitations which need evaluation in future studies. In contrast, an implementation of the subprocess on a fully centralized architecture with a user management system could be trivially designed such that it does not expose the email address or any other person identifying information. In such system, the data could live in a database with no public access and the users may be assigned with internal IDs which make it possible to uniquely identify them in association with their data. Therefore, in summary, using references to off-chain resources is found to be imcompatible to improve privacy.

Key Components

In terms of the key components of the system, four operations of filtering are identified based to their perceived complexity level. The operations are: retrieving all unpublished or published enlistments; filtering by enlistments by landlord; filtering enlistments by tenant; filtering enlistments based on the geodetic distance. However, in the context of the baseline prototype, the key components are all implemented off-chain where their impact on the efficiency is considered negligible for the study. Nevertheless, their neglibility in the current design is notable because complexity of these components is expected to change in the next phases of the study.

Autonomy

In summary, we have identified core data and process perspective for a simplified system which provides all the necessary operations for the real estate rental process. The analysis has also allowed us to identify where blockchain technology can be used.

However, it could be observed that the baseline variant does not make the sharing economy part of rental intermediation more decentralized. Namely, the service which allows P2P interactions through a smart contract cannot operate and be accessed without the assist *ERevS* which plays the role of an intermediator. Therefore, the baseline design iteration is a starting point to make the decentralized service more independent and is subject to be moved outside of the platform providers system, as described in the next sections.

3.3 Migrating Data to the Blockchain

In the baseline design, *ERevS* is managing the collection of resources. For this reason, it is also fit for filtering enlistments which are managed by *RIS*. What is more, in addition to collection management, the details of the property enlistments are split between two services.

This section describes the design of a system which moves all required data for managing the real estate rental intermediation into the blockchain. The system builds on top of the baseline variant and in terms of the established design analysis framework, only the dimensions of significant change are described.

3.3.1 Service Map

To move the management of resource collection of reviewed enlistments out of *ERevS*, another required service is added next to *RIS* with the name of *Enlistment Registry Service* (*ERegS*). *RIS* and *ERegS* are included in an abstract *Enlistment Rental Service* which is served in an Ethereum node (Figure 20).

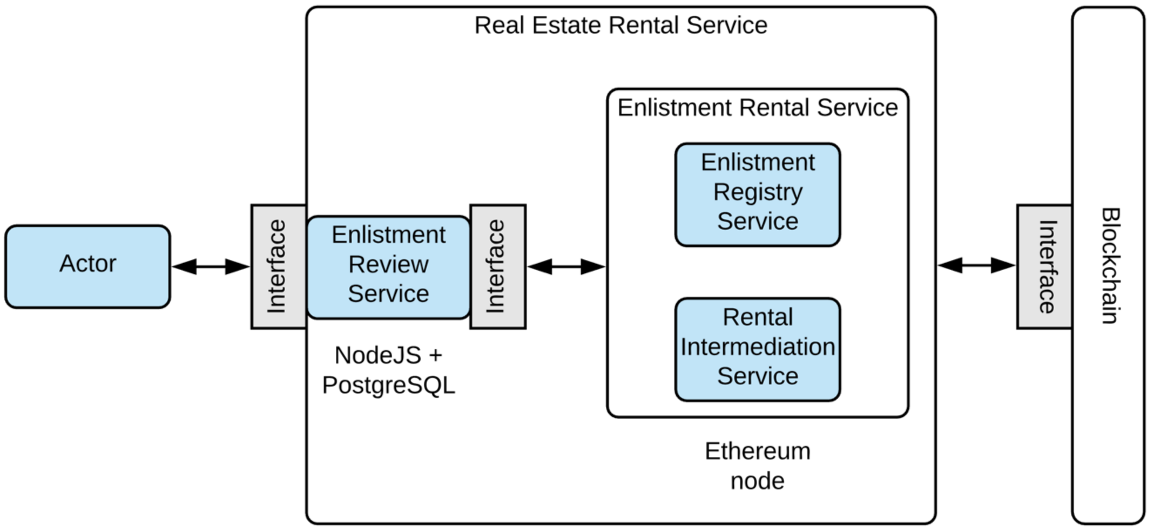

Real Estate Rental Service
Enlistment Rental Service
Enlistment Registry Service
Actor
Interface
Enlistment Review Service
Interface
Rental Intermediation Service
Interface
Blockchain
NodeJS + PostgreSQL
Ethereum node

Figure 20. Service map (variant 2).

3.3.2 Data Model

With the inclusion of another service, the data is now managed between 3 stores (Figure 21). Equivalently to the baseline variant, an instance of Enlistment smart contract is deployed to Ethereum distributed ledger once moderator approves it. However, now, in addition to the baseline procedure, the blockchain address of the deployed Enlistment instance is added to Enlistment Registry smart contract.

To migrate all enlistment data on-chain, the constructor of the Enlistment smart contract accepts details which were previously managed in *ERevS*. To persist enlistment location on the blockchain, a representation of geohash [31] is used. In its inner workings, geohashing divides the sphere of the Earth into a hierarchical grid and allows encoding of spatial latitude and longitude values into a arbitrary-length string of Base32 characters: the longer the string, more precision for the location. For the nature of the resources represented by the location, a 9-character length is used which at the worst case provides a rectangular cell with the precision of $\approx 4.77\ m \times 4.77\ m$ on the Equator according to Veness [32]. On the other hand, as other migrated details of the enlistment only serve a decorative purpose and never need on-chain processing, they are persisted as a JSON-encoded string. As the final change, an internal helper array of offer authors is migrated which allow for iteration over Solidity mappings.

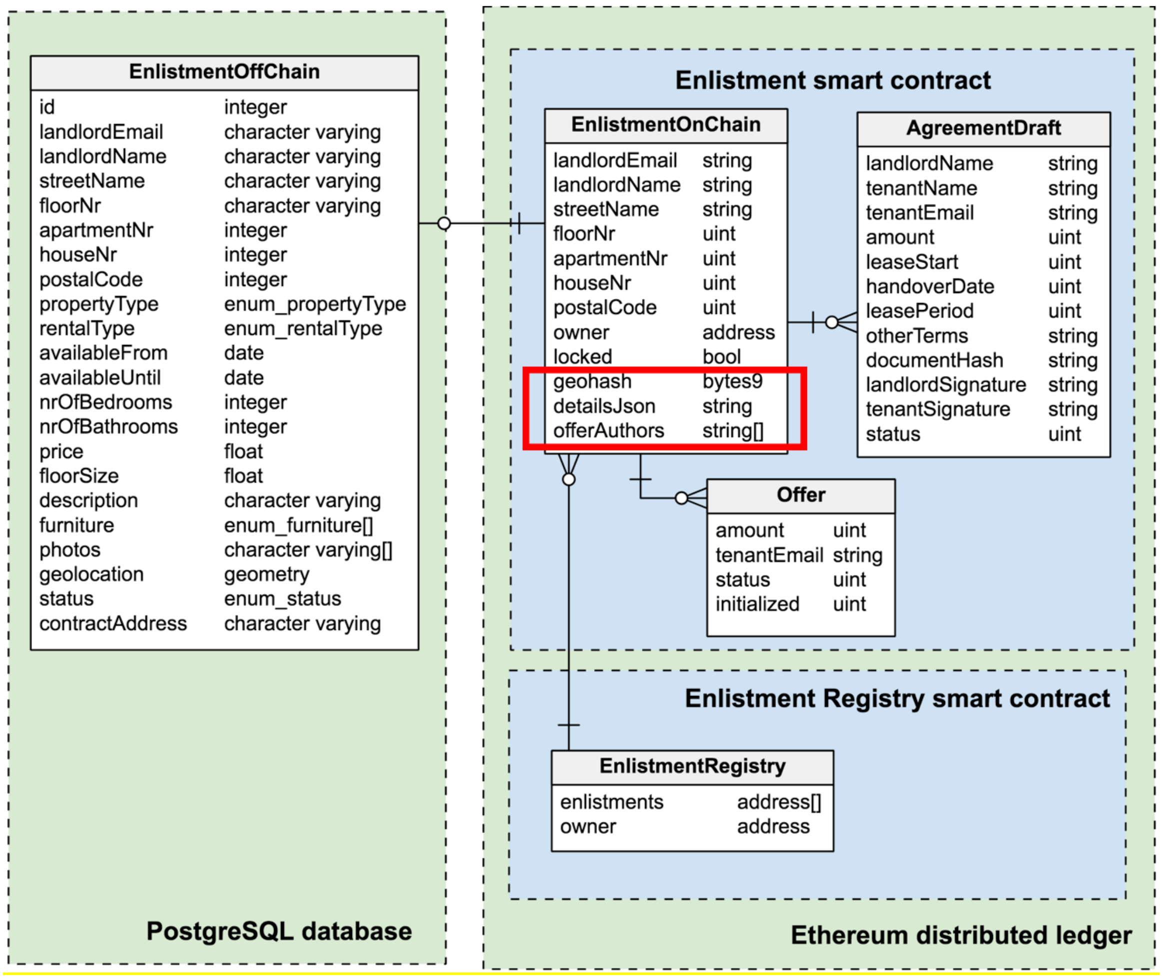


Figure 21. Data model (variant 2).

3.3.3 Interface Design and Key Operations

The root service is divided between three subservices (Figure 20) and because of this, another split of operations is implied. As the external actors are still served through *ERevS*, its operation set remains unchanged (Table 1). However, in its operation, it now relies on the required operations of *ERegS* (Table 3) and *RIS* (Table 4). Proceedingly, in the following subsections, the anatomy of the interfaces and impactful functions of the services is presented.

Table 3. Required operations from Enlistment Registry Service (variant 2).

Resource	**Write operations**	**Read operations**
Enlistment	- Add	- Retrieve all - Retrieve with mapped data for ge-osearch - Retrieve with mapped data for offer author filtering

Table 4. Required operations from Rental Intermediation Service (variant 2).

Resource	**Write operations**	**Read operations**
Enlistment	Same as baseline	Same as baseline
Offer		- Retrieve one for a given enlistment - Retrieve the number of offers - Retrieve offer by its index in the helper array
Agreement		Same as baseline

Interface and Operations of the Enlistment Registry Service

EregS is implemented as a smart contract called Enlistment Registry operating in Ethereum client node. The registry smart contract is a singleton instance, meaning that it is deployed once and the resulting reference is hard-coded in the consuming services.

The Enlistment Registry implements its corresponding data model from Figure 21 through two top-level storage variables: `owner` and `enlistments`. Similarly to the Enlistment smart contract, `owner` is utilized as to enforce access to its deployer (i.e. the platform provider). Variable of `enlistments`, however, references an append-only array of Enlistment instance addresses.

In addition to the trivial implementations of adding an enlistment to the registry and retrieving the underlying array, the contract also exposes 2 functions for operations which were an implicit burden of the off-chain service in the baseline design:

1) Retrieving enlistments with mapped data for geosearch. The invocation of this method (Code block 2) loops over all enlistment addresses in the registry and for each of them, makes an inter-contract call to retrieve a geohash. As a result, it returns two arrays of fixed length of the size of the registry: one of which contains all addresses and the other one the corresponding geohashes. The output of this function is input to run filtering computations off-chain.
2) Retrieving enlistments for offer author filtering. The anatomy of this function is similar to the previous one with the difference that instead of retrieving a geohash, an intra-contract returns the number of offers for an enlistment.

```
function getEnlistmentsForGeosearch() view public returns (address[], bytes9[])
{
  bytes9[] memory geohashes = new bytes9[](enlistments.length);
  for (uint i = 0; i < enlistments.length; i++) {
    Enlistment enlistmentContractInstance = Enlistment(enlistments[i]);
    geohashes[i] = enlistmentContractInstance.getGeohash();
  }
  return (enlistments, geohashes);
}
```

Code block 2. Data retrieval operation from an on-chain registry.

Interface and Operations of the Rental Intermediation Service

When compared to the ABI of *RIS* of baseline variant, the operational changes are minor: the smart contract is updated to return the migrated data and exposes a new function to return the length of the helper array of `offerAuthors` as required by *ERegS*.

Interface and Operations of the Enlistment Review Service

Migrating data does not change the state transitions or resource hierarchy of the baseline variant. However, as *ERevS* no longer owns the collection of approved enlistments, the API explicitly expects an on-chain reference of an enlistment for all write and read queries once it is approved by a moderator. For example, to cancel an offer, HTTP POST request to end-point `/enlistments/:enlistmentContractAddress/offers/:tenantEmail/cancel` needs to be called.

Changes to the data governance also modify the collection-related procedures of *ERevS*:

1) <u>Finding either all published or unpublished enlistments.</u> While unpublished enlistments are still retrieved from the relational database, the published ones are collected by requesting all addresses from *ERegS*.
2) <u>Retrieving enlistments by offer tenant email</u>. The underlying procedure is now as follows:
    a. A function of Enlistment Registry is called to retrieve the collection of on-chain enlistment identifiers and their corresponding number of offers.
    b. For all enlistments, all offers are retrieved one-by-one by accessing them through the index in the tenant email helper array of the Enlistment smart contract.
    c. The filtering is done by checking if the provided tenant email is in the array of enlistment offers.
3) <u>Retrieving enlistments by landlord email.</u> To retrieve enlistments by landlord email:
    a. The array of all enlistment addresses is retrieved from the Enlistment Registry smart contract.
    b. Using enlistment references, each of their landlord email is retrieved from *RIS*.
    c. Filtering is done by checking if the input landlord email matches the ones retrieved from the previous step.
4) <u>Filtering enlistments based on geographical proximity.</u>
    a. Similarly to the first step of the two previous operations, Enlistment blockchain addresses and their geohashes are retrieved from *EregS*.
    b. Filtering is done by:

i. decoding the geohash to latitude/longitude representation of float datatype;
ii. calculating the distance between the input location and the enlistment using a JavaScript implementation of Haversine formula [33] and comparing it to the search radius input.

Similarly to the baseline variant, for each matching enlistment in the filter, an additional call must be made to retrieve the actual data to be presented in the API request output view.

3.3.4 Analysis

The results show that the data which need migration are atomic resource attributes of enlistments and collection data structures. To adapt the migrated atomic resource attributes, the data model of the enlistment smart contract is appropriately modified to accept the new details as a geohash and JSON-encoded collection of attributes.

While the migrated location data of the enlistment is perceived necessary for the operation of future variants (i.e. for location-based filtering), the inclusion of purely decorative information could be considered debatable. In more detail, the migrated attributes of enlistment (except location) serve no other purpose nor add more value other than to reach the goal of complete decoupling. The concern of high data storage costs aligns with the findings of Eberhardt and Tai [28] who also describe a setup in which the amount of data grew so big that it was not reasonable to store the values on-chain. However, Eberhardt and Tai [28] do not provide any proof or examples about how big is the overhead. As such, the effect on the added data storage overhead could be considered a limitation on the design which needs to be evaluated in terms of efficiency and alternatives.

In terms of collection data structure migration, a new smart contract is added to the system to serve the purpose of a lightweight registry. What is found significant, is that to ensure that the enlistment was really reviewed, the append operation is initiated from the *ERevS*. Alternatively, a chained operation could be considered which automatically adds the enlistment to the registry in the constructor of Enlistment smart contract. Although such alternative would add automation to the system by replacing two consecutive transactions with one, it opens up the system for spam in public blockchain. The potential attack on chained registry addition approach trivial: the attacker would be able access the contract bytecode by inspecting the transactions and then use it to instantiate a new enlistment without going through a review process. So, to give the provider of *ERevS* ensurance that only they are able to manage the registry, two separate transactions are preferred over a chained one.

Key Components

The data migration, especially moving to an on-chain registry approach considerably changes the anatomy and perceived efficiency of the key operations. Whereas in the baseline system only one request to *RIS* per match in the filter was necessary, the current variant adds another layer(s) of requests just to retrieve the migrated data from the on-chain service for the operations. Furthermore, this per-resource data retrieval strategy may have negative impact on the performance of the operations and is subject for closer inspection during evaluation.

Autonomy

The changes to baseline system design migrated all rental intermediation related data to the service actually managing the underlying process. This results in a gain in autonomy but the empirical analysis also finds potential loss in efficiency. However, migrating data alone is not enough to achieve autonomy for the on-chain services because the filtering computations are still done off-chain. Because of this, the objective for the next prototype is to port operations.

3.4 Porting Operations to the Blockchain

The previous results show how to integrate blockchain and, thus, gain in autonomy. In more detail, it is shown how to extract a peer intermediation subprocess from a business process model of a real estate sharing platform and design it as a separate internal service. Then, it is found that in order to decouple from the platform manager to enable autonomous P2P process execution on-chain, the internal service needs to move outside of the providers system. Moreover, for the autonomy to happen, the required data has been migrated from off-chain datastores to on-chain storage.

This chapter describes the service design to move the processing involved with the intermediation process. The design builds on top of the previous results.

3.4.1 Service Map

To port the operations of filtering enlistments by geographical proximity, an internal dependency to a service of *Geodistance* is added to *ERegS*. Although the operations of the added service could also be implemented as embedded, the explicit decoupling separates its standard mathematical operation (see 3.4.3) from the business domain of the application.

3.4.2 Data Model

There is only one change to the data model to adapt to the underlying operation of geosearch: geohash format [31] for persisting enlistment location is replaced by WGS 84 [30] coordinates encoded as 32-bit signed integers (Figure 22). The signed integer stores 6 decimal places of a latitude/longitude degree which, based on widest sphere of the Earth ($R = 6378137$ m)[27], guarantees more than enough precision to locate real estate:

$$\frac{R \times 2 \times \pi}{360° \times 10^6} \approx 0.111\ (m). \tag{3.1}$$

[27] http://epsg.io/7030-ellipsoid [Accessed: 19-May-2018]

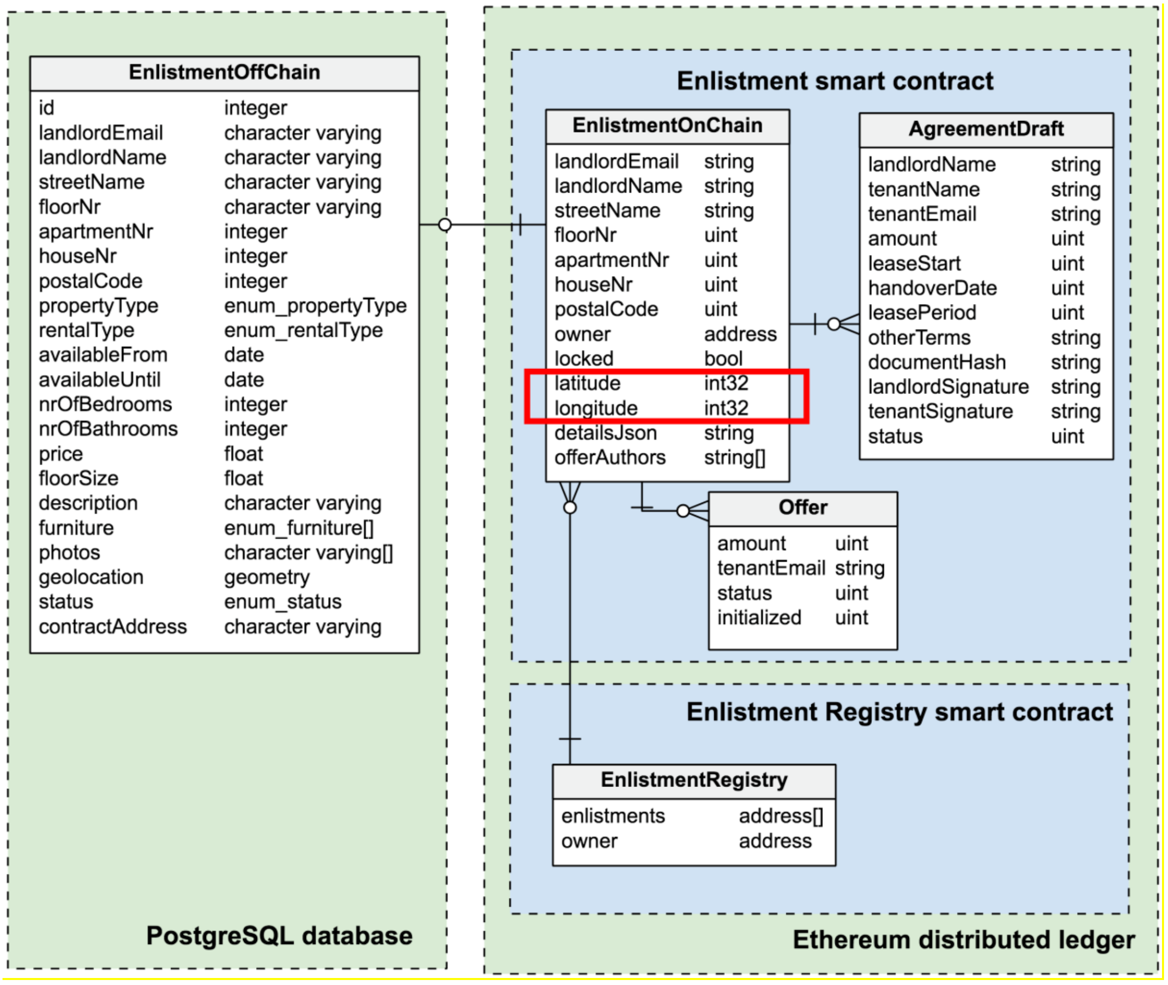


Figure 22. Data model (variant 3)

3.4.3 Interface Design and Key Operations Drill-through

Regarding changes to the interface of the services, there are 3 external read operations to be migrated from *ERevS* (Table 1) to *ERegS*: filtering by landlord, by offer tenant and by geographical proximity (Table 5). Proceedingly, the operation set of *RIS* needs to cater the needs of *ERegS* for it to do the filtering. For this purpose, *RIS* now provides the attributes of landlord and location of the enlistment as well as a function to check whether a given tenant has made an offer (Table 6). On the other hand, the sole operation of Geodistance Service is to calculate distance between two locations. Next, each of the interfaces and significant operations are elaborated on.

Table 5. Required operations from Enlistment Registry Service (variant 3).

Resource	**Write operations**	**Read operations**
Enlistment	Same as variant 2	- Retrieve all - Filter by landlord - Filter by tenant (as the bidder) - Filter by geographical proximity for given location and search radius

Table 6. Required operations from Rental Intermediation Service (variant 3).

Resource	**Write operations**	**Read operations**
Enlistment	Same as variant 2	- Retrieve one **- Retrieve landlord email** **- Retrieve location** **- Check whether a given tenant has made an offer**
Offer		- Retrieve one for a given enlistment - Retrieve offer by its index in the helper array
Agreement		Same as variant 2

Interface and Operations of the Geodistance Service

The Geodistance Service is implemented as Solidity library[28] and it exposes one operation: calculating geographic distance between two sets of coordinates. As performance is important for on-chain computations and the distances set by the domain are expected to be relatively small on the global scale, it is chosen to implement the operation using Euclidean distance on an equirectangular projection of the Earth (also known as *Equirectangular approximation*).

To calculate the trigonometric function of cosine, GeoDistance library is dependent on another library: *Trigonometry*. In its operation, the library divides a circle to 16384 angle units and uses a lookup table on the first quadrant of sine together with linear interpolation.[29]

Interface and Operations of Enlistment Registry Service

Implementation of filtering functions are now migrated to the interface of *ERegS*:

[28]https://github.com/vindrek/blockchain-real-estate/blob/enlistment-filtering-on-chain/ethereum/contracts/GeoDistance.sol [Accessed: 19-May-2018]
[29] https://github.com/Sikorkaio/sikorka/blob/master/contracts/trigonometry.sol [Accessed: 19-May-2018]

1) Retrieving enlistments by landlord email. The filtering procedure iterates over the registry and by inter-contract call to *RIS* retrieves a Keccak-256 hash of the landlord email[30] to be compared with the hashed value of the input.
2) Retrieving enlistments by offer tenant email. Again, the registry is iterated and filter is applied by inter-contract call to Enlistment which checks whether a given tenant has made an offer.
3) Filtering enlistments based on geographical proximity.
    a. By inter-contract call to Enlistment, the coordinates of an enlistment is retrieved.
    b. A deployed Geodistance library instance calculates the distance between the location of the input and enlistments using equirectangular approximation method.
    c. Filter is applied by comparing the calculated distance and the search radius provided.

All the filtering operations of *ERegS* output a 256-bit integer which bitset maps the indices of the matching enlistment in the registry array. Meaning, for each matching enlistment, another call is required to retrieve the blockchain address of the Enlistment smart contract instance which would then be used to make another call to get the actual data of the enlistment.

3.4.4 Analysis

The results reveal that the operations which need to be ported to the smart contracts for more autonomy are related to collection management and do the procedure of filtering resources. The study identifies such enlistment filtering operations and describes a functionally equivalent implementation using smart contracts.

Key Components

Results show that the key component of location-based filtering implementation poses limitations to the service. Namely, the design uses equirectangular projection as the basis for geodetic distance calculation. The reason behind such decision is that equirectangular approximation works well with the processing-limited nature of EVM because it is trivial to compute. For example, equirectangular projection only requires one square root and trigonometric function call[31]:

$$d = r \times \sqrt{\left((\lambda_2 - \lambda_1) \times \cos\left(\frac{\varphi_1 + \varphi_2}{2}\right)\right)^2 + (\varphi_2 - \varphi_1)^2}, \qquad (3.2)$$

where r is the radius of the sphere, φ is the latitude and λ the longitude. On the other hand, an implementation of Haversine formula requires 6 trigonometric function calls and a 1 square root:

[30] Due to the limitations of EVM and/or Solidity v0.4.19, returning strings of arbitrary-length in external functions is not possible. For this reason, a 32-byte hash of landlord email must be used, instead.
[31] Implementation is modified to use the length of 1 degree on the Equator instead of spherical radius to operate on degrees throughout the algorithm instead of radians which would need conversion.

$$d = 2r \times \sin^{-1}\left(\sqrt{\sin^2\frac{\varphi_2 - \varphi_1}{2} + \cos\varphi_1 \cos\varphi_2 \sin^2\frac{\lambda_2 - \lambda_1}{2}}\right). \qquad (3.3)$$

Moreover, the implications of using Spherical Law of Cosines require 6 trigonometric function calls:

$$d = r \times \cos^{-1}(\sin\varphi_1 \times \sin\varphi_2 + \cos\varphi_1 \times \cos\varphi_2 \times \cos(\lambda_2 - \lambda_1)). \qquad (3.4)$$

However, the equirectangular projection introduces significant distortions with a reflecting accuracy error depending on the bearing, latitude and distance [34]–[36]. Similarly, to the current study, Esenbuğa *et al.* also compromise accuracy for efficiency when choosing equirectangular approximation [36]. What is more, while evaluating different geodetic distance algorithms for province assignment of cities in Turkey, Esenbuğa et al. show that the maximum relative percentage error of the equirectangular approximation on their data is only 0.2323% ($s = 0.0650\%,\ \bar{x} = 0.1396\%$) [36]. So, the low error and compliance for inter-city measurements implies that the method should be potent for intra-city purposes. The analysis on determining the exact latitude value where the real estate search radius distance is longer acceptable is left open for future studies to specify.

Another design limitation is that the maximum capacity of enlistments in the current variant is 256 enlistments. This limitation is set by the filtering operations which uses bitset technique and returns an unsigned integer to represent the matches. However, the maximum unsigned integer size currently supported by Solidity is 256 bits. To scale up, the procedure has to be adjusted to return an array of such integers.

Autonomy

As a result of the migrating intermediation related data and operations on-chain, the former is now capable of managing both read and write operations of rental intermediation between landlords and tenants. From the domain segregation point of view, this design reduces the responsibility of the off-chain service (which is plays the role of an intermediatior) to serve tasks that it is designed to do: review enlistments. Once an enlistment is approved, the review service simply forwards requests to smart contracts without enforcing any business-logic itself.

4 Evaluation and Discussion

In the scope of the study, 3 progressive variants of a real estate rental system are described. While the supported process and the externally available operations of the different variants remain the same, they differ from internal aspects. This chapter assesses and analyses the qualities of the variants.

4.1 Quantitative Analysis

The quantitative analysis focuses on the performance observed during the execution of smart contracts by the EVM. On the other hand, the efficiency of smart contract operations are measured by observing gas consumption as the execution cost of the message calls to a contract address. While the measurement value of gas does not express the computational complexity directly, it does so in a non-direct way by producing a fair cost for the transactor to compensate for the code execution. The fairness of gas calculation is guaranteed by operation costs and polynomial cost functions (such as memory extension) as defined by the specification of EVM [3].

4.2 Scenario

As a test scenario, a happy path of *Enlistment to contract completion* subprocess is used, complemented with read operations to hypothetically assist the actors with the execution. The execution is observed in the prototypes: baseline (variant 1), data migration (variant 2) and operation migration (variant 3). As described in the subsections of interface design (3.2.4, 3.3.3, 3.4.3), the implementation includes a total of 4 smart contracts: `Enlistment`, `EnlistmentRegistry` and libraries of `GeoDistance` and `Trigonometry`. In the analysis, `EnlistmentRegistry`, `GeoDistance` and `Trigonometry` contracts are also referred to as *singleton* contracts because they are only required to be deployed once for the operation of the system (as opposed to multi-instance nature of `Enlistment` which is deployed each time an enlistment is approved into the system).

The scenario is divided into the following steps:

1) An enlistment is deployed and added to the public registry (*write*).
2) Tenant retrieves all published enlistments (*read*).
3) Tenant runs a geographic approximity search (*read*).
4) Tenant requests the enlistment data (*read*).
5) Tenant places an offer *(write*).
6) Landlord queries for his enlistments (*read*).
7) Landlord queries all offers for an enlistment (*read*).
8) Landlord retrieves one offer (*read*).
9) Landlord accepts the offer (*write*).
10) Landlord issues a tenancy agreement (*write*).
11) Tenant queries for the enlistments that he has bid on (*read*).
12) Tenant retrieves a tenancy agreement (*read*).
13) Tenant accepts the tenancy agreement (*write*).
14) Landlord signs the agreement (*write*).
15) Tenant signs the agreement (*write*).
16) Tenant sends the first month rent (*write*).

The experiment measures:

1) Gas usage of the steps and singleton contract deployment with the following strategy:

   a. Deployment cost is calculated by subtracting the balance of the account before and after the transaction.
   b. Ordinary transactions (i.e. write calls such as sending an offer) are measured by checking gas usage from the receipt.
   c. Read-only calls (e.g. geofiltering) are measured using the `estimateGas` method of web3js[32].

2) count of JSON RPC calls for each step;
3) size of the deployed bytecode of the contracts retrieved from the truffle compilation asset file[33].

Moreover, the following should be considered when interpreting the results:

- If a step requires multiple atomic subrequests (e.g. when retrieving all published enlistments), then the measured gas consumption and JSON RPC call count is the sum of the requests.
- Each enlistment in the test run has an additional 3 offers from other tenants (including the one being operated on).
- The landlord in the scenario has no previous enlistments.
- The tenant in the scenario has not placed any previous offers to any enlistment.
- All the added enlistments except the one in the scenario are outside of the geosearch area.
- A single step includes either only write or read operations but not both.

4.3 Results

In terms of singleton smart contract deployment gas consumption, variant 3 is the most demanding (Figure 23). Variant 3 deploys a total of 3 contracts with a total cost of 1187972 gas. Meanwhile, variant 2 only requires 1 singleton contract (677559 gas) and variant 1 operates without any singletons. The most significant impact to the cost of the singleton deployment is made by `EnlistmentRegistry` which proves to be ~14.3% higher for variant 3 than it is for variant 2.

[32] https://web3js.readthedocs.io/en/1.0/web3-eth-contract.html#methods-mymethod-estimategas [Accessed: 19-May-2018]
[33] https://www.npmjs.com/package/truffle-contract-schema [Accessed: 19-May-2018]

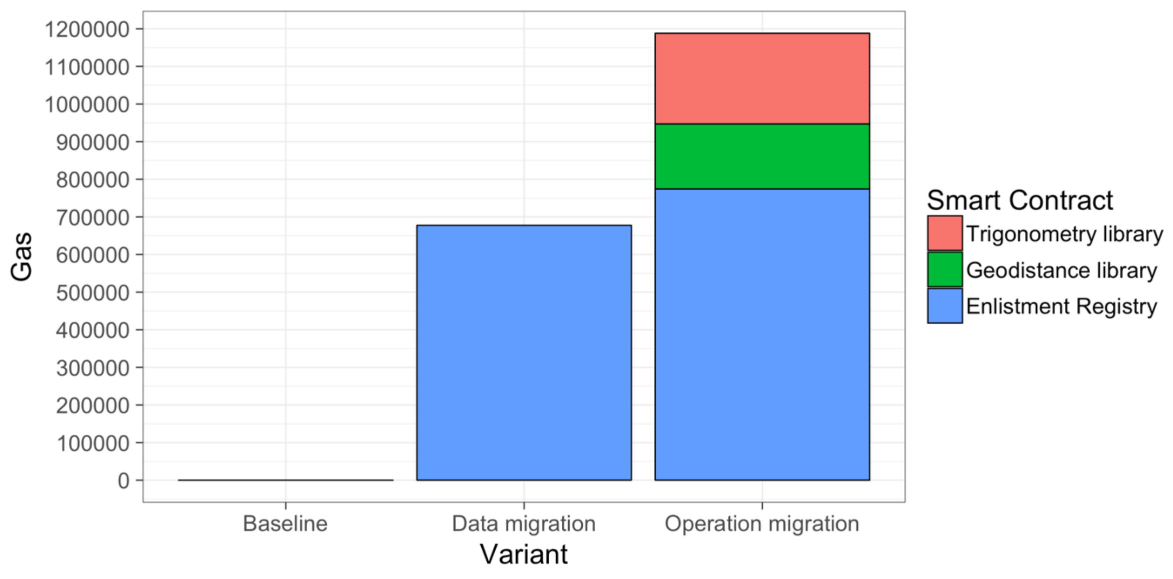


Figure 23. Bar plot analysis for singleton smart contract deployment gas usage.

However, when aggregating and comparing all the contract deployment costs in the system, then it is evident that `Enlistment` deployment cost is the most impactful (Figure 24). The gas usage for baseline `Enlistment` deployment is 4574231 and it rises ~24.6% for the second variant. Noticable is also the difference between Enlistment deployment cost of the last 2 variants (98501).

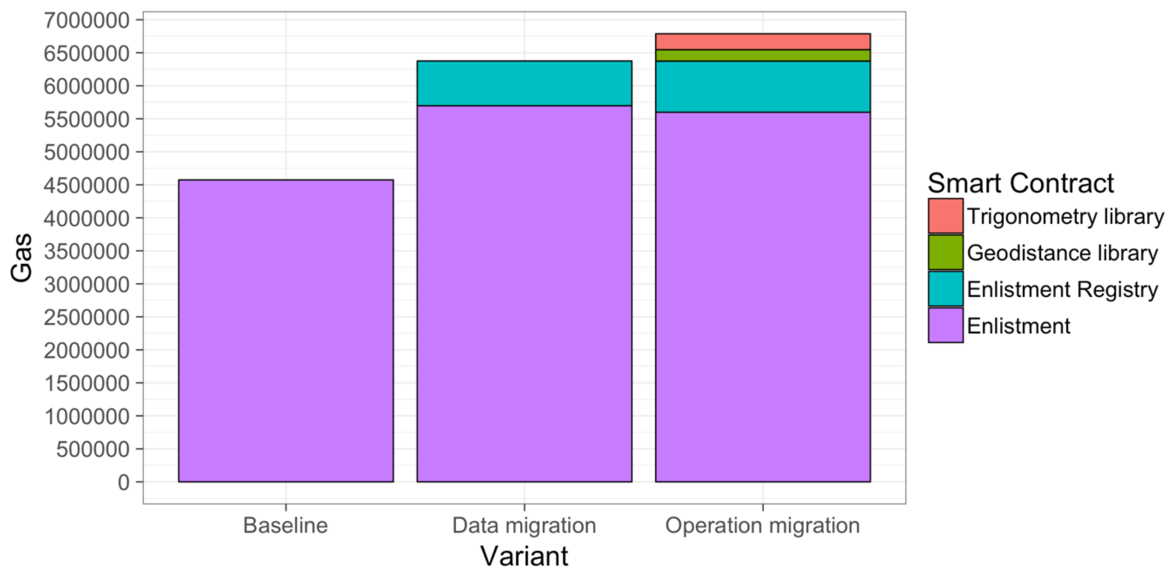


Figure 24. Bar plot analysis for smart contract deployment gas usage.

The bytecode size of the contracts are given in Table 7 and it reveals that the largest bytecode size is for `Enlistment` contract of variant 2, followed by variants 3 and 1.

Table 7. Size of the deployed smart contract bytecode.

	`Enlistment` (bytes)	`EnlistmentRegistry` (bytes)	`Geodistance` (bytes)	`Trigonometry` (bytes)
Baseline	16177	-	-	-
Data migration	18217	2270	-	-
Operation migration	17808	2648	458	716

An observation to the scenario execution with $N = 10$ previous enlistments in the registry (Figure 25) reveals that the transactional gas usage of the write operations is by far the most expensive for step 1. The next most gas consuming step is step 10. In relation to step 1 and step 10, the cost of other write steps (5, 9, 13, 14, 16) could be considered negligible.

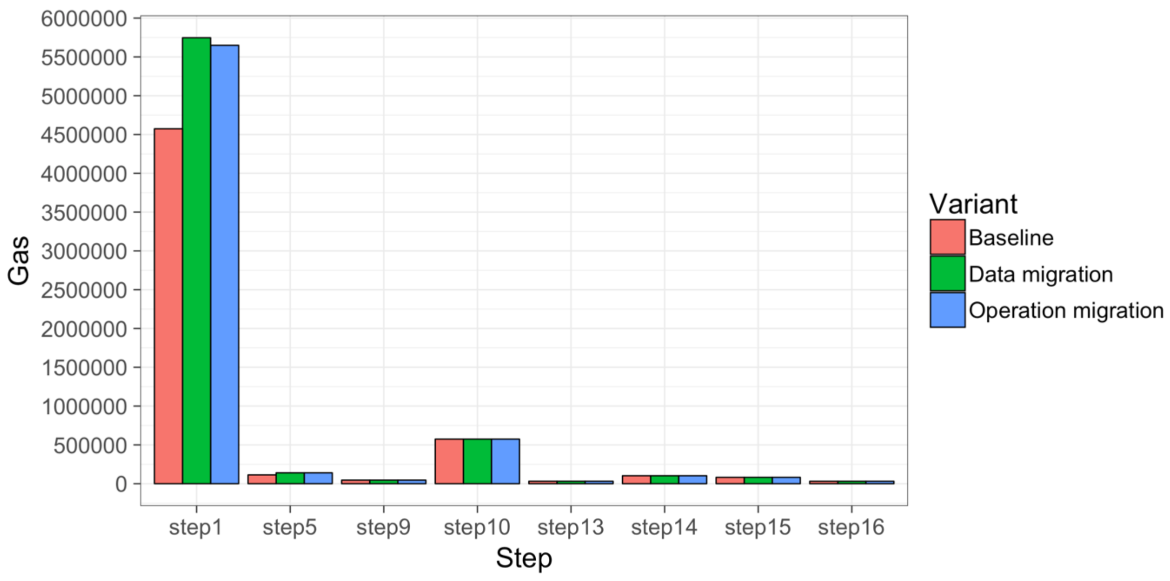


Figure 25. Bar plot analysis for write steps gas usage for 10 previous enlistments in the registry.

By eliminating the enlistment deployment cost from step 1, an inter-variant perspective to progressive operation gas cost (Figure 26) reveals the similarities of the variants. There are two exceptions to otherwise similar behaviour: firstly, for step 5, the usage for baseline variant (113569) is slightly lower than for other variants (139877 and 139899 respectively) and secondly, baseline variant does not use any gas for step 1.

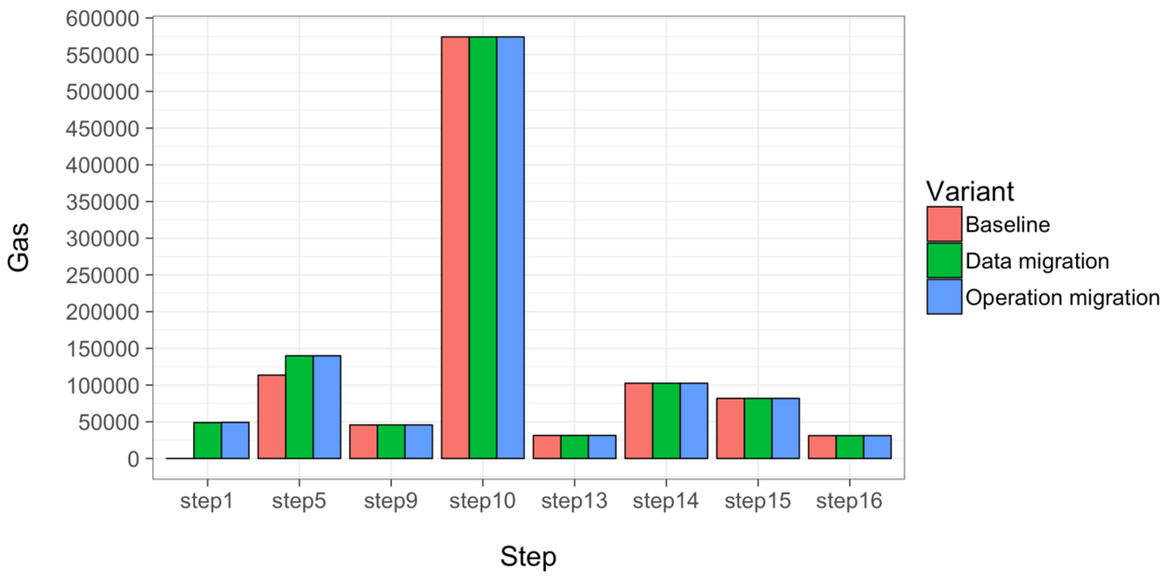


Figure 26. Bar plot analysis for write steps gas usage for 10 previous enlistments in the registry (excluding `Enlistment` contract deployment).

Figure 27 shows that when the amount of enlistments grows, the transaction gas usage for the write steps remains the same. However, a closer look reveals one exception to this: for variants 2 and 3, step 1 is slightly more expensive when there are no previous enlistments in the registry (Table 8).

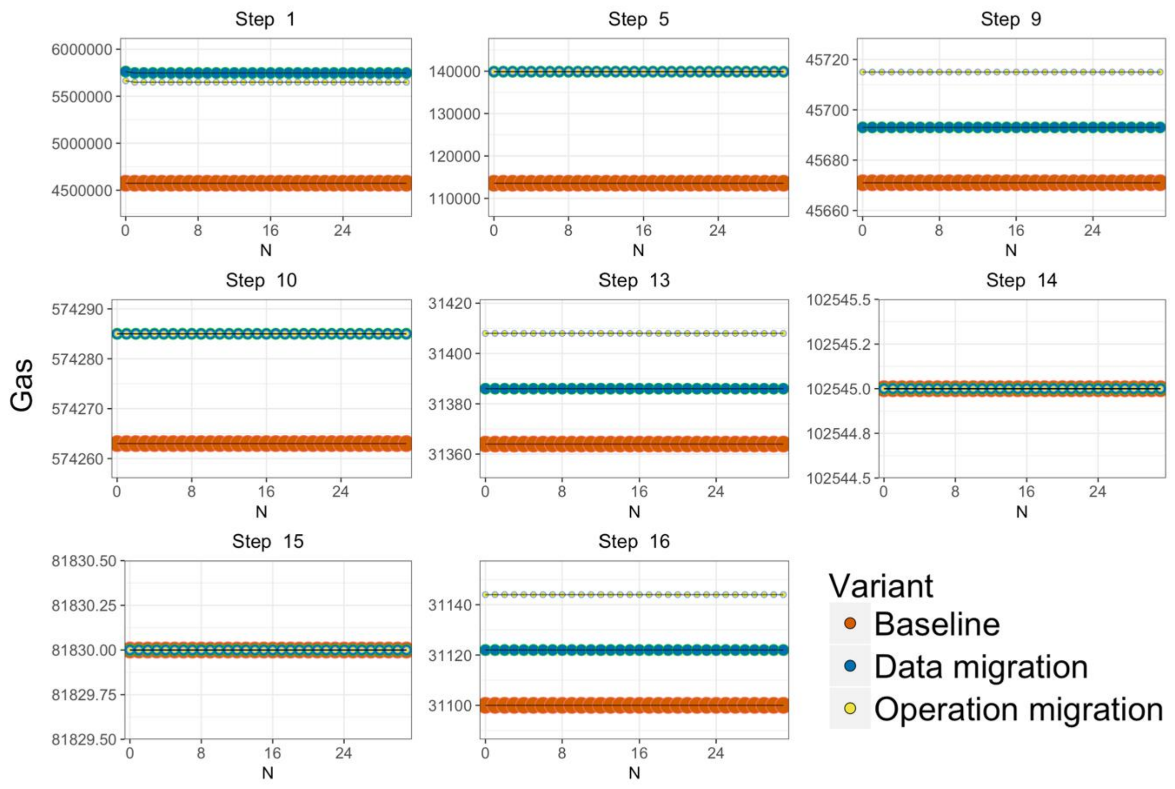


Figure 27. Gas usage of write transactions for arbitrary number of enlistments (N) in the registry.

Table 8. Gas usage for step 1 of different variants for arbitrary number of enlistments in the registry.

	0 previous enlistments (gas)	**1 previous enlistment** (gas)	**2 previous enlistments** (gas)
Baseline	4574231	4574231	4574231
Data migration	5762189	5747189	5747189
Operation migration	5664030	5648966	5648966

The total mean cost for scenario write steps execution is the biggest for data migration variant (6754396), followed by operation migration (6656319: 1.5% lower) and baseline (5554573: 21.6% lower).

Request count for write operations is similarly constant as is their gas consumption (Figure 28). However, it could be noticed that the first write step requires two requests to the smart contract while the others do one request.

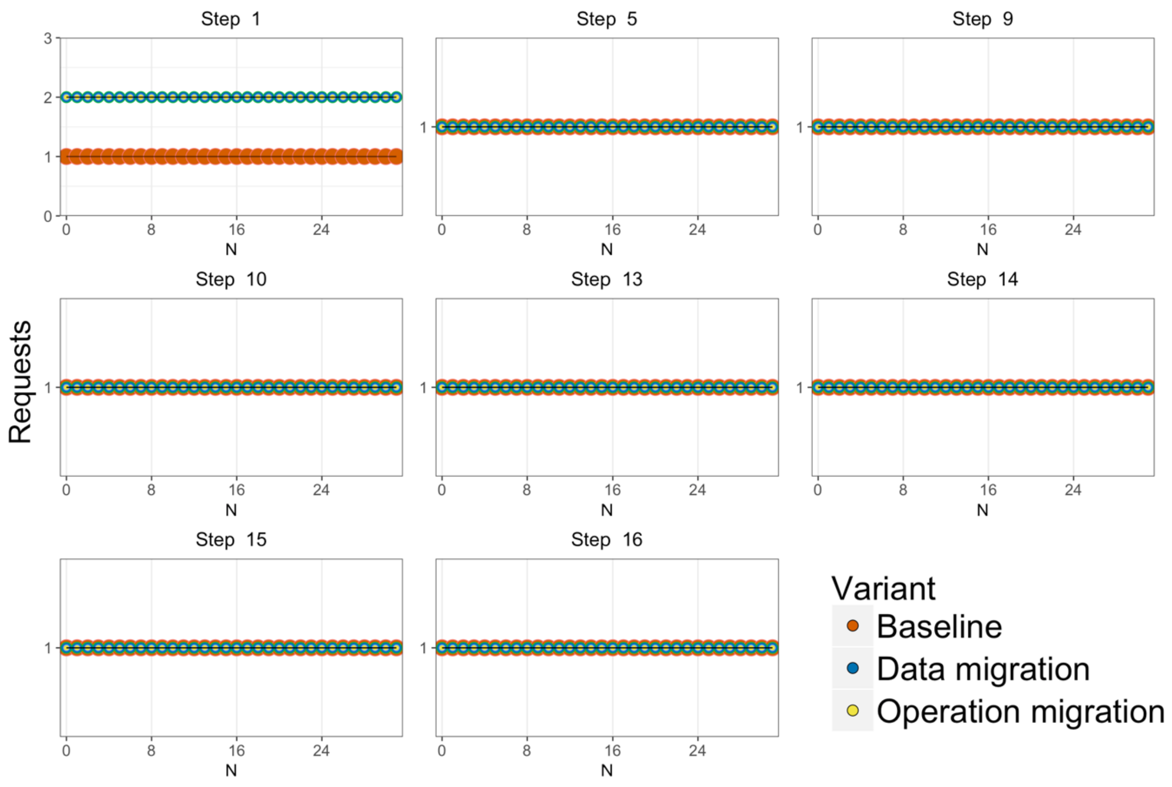


Figure 28. Request count for write operations for arbitrary number of enlistments (N) in the registry.

Figure 29 shows that read steps gas usage for $N = 10$ previous enlistments in the registry vary. The data shows that the cost difference of steps 8 and 12 is negligible while steps 4 and 7 are getting slightly more expensive with each variant. On the other hand, for steps 2 and 3, the gas usage rises significantly with successive variants. Most significantly, the results reveal an anomaly for the behavior of data migration variant on steps 6 and 11 which are multiple times more expensive than their respective results on baseline and operation porting implementations.

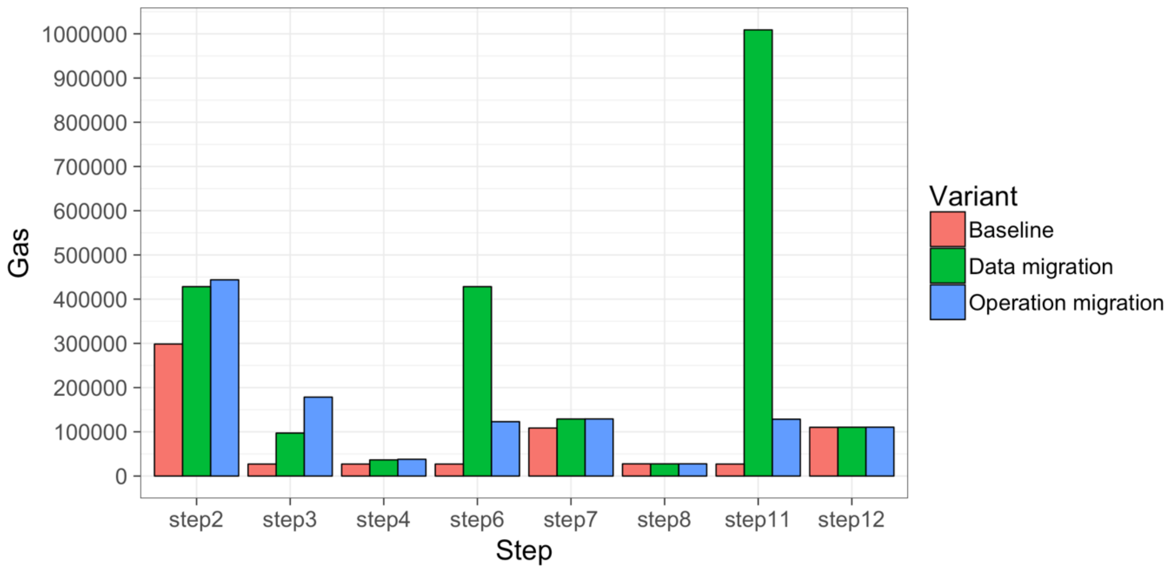


Figure 29. Bar plot analysis for write operation gas usage for 10 previous enlistments in the registry.

With growing number of enlistments (N) before the execution of the scenario, gas usage for steps 4, 7, 8 and 12 remain constant for all variants, Figure 30 shows. The gas consumption also stays constant for the baseline variant in steps 3, 6 and 11. For steps 2, 3, 6 and 11, a linear growth could be recognized. For steps 6 and 11, data migration prototype has a significantly steeper slope than the baseline and operation migration versions. On the other hand, the operation migration variant cost grows the fastest for steps 2 and 3. Table 9 gives an overview of the growth (slope of the line) and reveals that the gas usage grows the fastest for the data migration variant of step 11: 83860 per added enlistment in the registry.

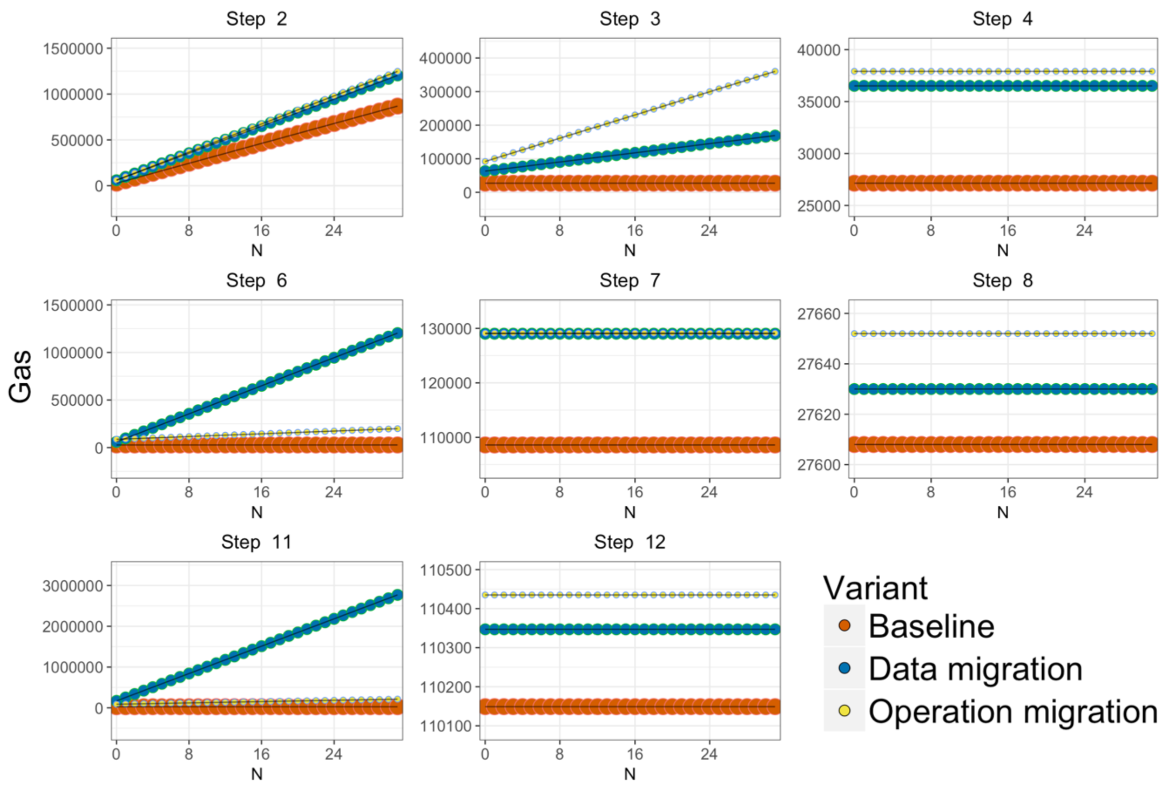


Figure 30. Gas usage of read operations for arbitrary number of enlistments (N) in the registry.

Table 9. Gas rise for an added enlistment for read steps.

	Step 2 (gas)	**Step 3** (gas)	**Step 6** (gas)	**Step 11** (gas)
Baseline	27144	0	0	0
Data migration	36886	3417	36886	83860
Operation migration	38280	8654	3590	4063

The total mean cost for scenario read steps execution is found to be the least for baseline variant (802811), followed by operation migration (1478994: 84.2% higher) and data migration variant (3151951: 292.6% higher).

Figure 31 shows that the characteristics of read operation request counts resemble the affine or constant nature of their gas consumption counterparts (Figure 30) with an exception of step 3. In more detail, the gas usage for step 3 is linear but the request count remains constant. Also, for steps 4, 8 and 12 it could be noticed that the number of requests are the same for different variants but gas consumption differs: for all of them, the operation migration variant is the most expensive, followed by data migration and then baseline.

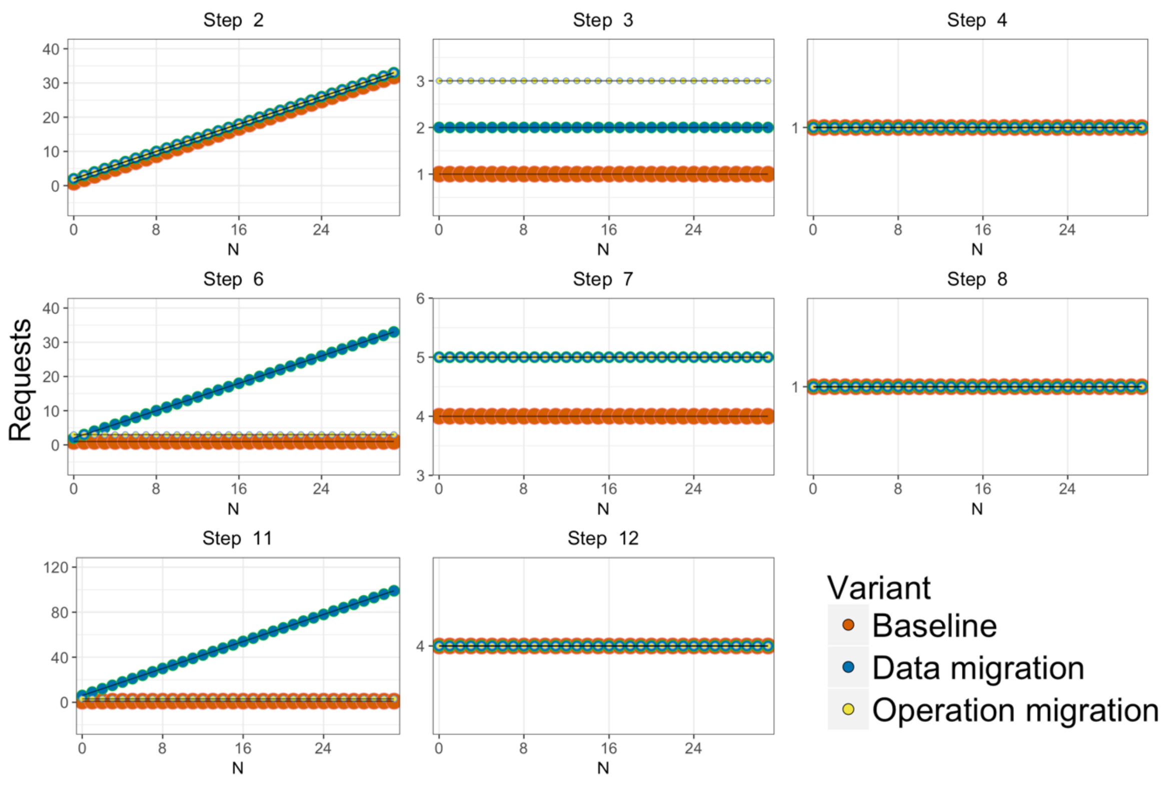


Figure 31. Request count for read transactions for arbitrary number of enlistments (N) in the registry.

By aggregating the mean gas usage for scenario read and write steps execution[34], the results reveal that the most gas consuming variant is the data migration variant with a value of 9906347 which is ~21.8% higher than the operation counterpart and ~55.8% higher than baseline.

4.4 Discussion

Given that the implementations build on top of each other by design and that the role of the on-chain service increases with successive variants, the evaluation expectedly proves the baseline version to be the most efficient. However, it is surprising that the least efficient variant is the second and not the third variant. Moreover, the second variant not only proves to be the most expensive for read but also write steps.

Results show that the second variant is less efficient for write steps than the third one because of the first step in the scenario: deploying an enlistment and adding it to the registry. Because the operation of adding an enlistment to the registry is the same for both variants (3.3.2), the underlying reason should lie in the differences in the `Enlistment` smart contract. However, in terms of their data models, the third variant should even be less efficient than the second one. Namely, concerning data model, the only difference between the variants is that the second variant uses a single storage mapping to persist the location (geohash) while the third one uses two values of latitude and longitude (3.4.2). In summary, this means that the culprit of the remaining difference comes from the added bytecode storage fees (200 gas per byte [3]) as shown in Table 7.

[34] Excluding singleton deployment gas because they are one-time expenditure.

Meanwhile, variant 2 is shown to be cumulatively the most expensive for scenario read steps because of its execution cost of two operations: filtering enlistments by landlord and by a tenant. For the two operations, variant 2 proposes a similar anatomy: with the first call, the procedure retrieves an initial reference data from the on-chain registry and then access each resource one-by-one (or multiple levels of them in child resources are required) in order to do the filtering (3.3.3). However, such filtering strategy is creating request overhead as well as redundancy. The overhead appears because the strategy accesses resources one by one, creating a severe N + 1 query problem. As a sign of severity, the results show that the number of JSON RPC calls grows near 100 for 31 previous enlistments in the registry (Figure 31). A solution to the N + 1 issue would be to retrieve all the data at once but such possibility is found to be currently limited by native incapability of EVM to return arrays of dynamic length and depth as well as complex data structures. Redundancy, however, is introduced by the interim nature of the data migration phase design: data is not being processed in its origin and in order to compensate for that, more information needs to be exchanged to achieve the desired result. To quantify the effect of high request count to quality of service, further studies should measure changes in response time.

On the other hand, in comparison with variant 2, the filtering strategy of variant 3 fixes redundancy by doing the processing at the origin of the data (3.4.3). However, on-chain filtering only alleviates the request overhead by eliminating one level of N + 1 problem (or multiple if filtering child resources). In another words, if the test scenario were such that the landlord or tenant had other enlistments or offers or there were more than one match for geosearch, the slope of the gas usage and request count would increase for both data and operation migration variants. So, in conclusion, the paper finds that on-chain filtering strategy is superior due to its ability to remove redundancy and the associated request overhead: filtering by landlord is nearly 7 times more efficient on-chain whereas by tenant it is more than 20 times[35] (Table 9).

While for other filtering operations the on-chain method outperforms off-chain, it is not the case for geosearch. Instead, the evaluation results shows that the gas consumption of geosearch grows ~2.5 times slower off-chain in case of variant 2 (Table 9), making it more efficient. This is found to be a direct result of a computation-heavy nature of the operation to calculate distances between coordinates (3.4.3). Even though a lot of optimization techniques were put into the implementation (e.g. sacrificing accuracy for performance by choosing a fast algorithm, using look-up tables for trigonometric functions, fixing the accuracy of the latitude and longitude angles), the operation would be better off done off-chain.

Moreover, the evaluation brings forward the high impact of step 1 in relation to the other write transactions which contribute to an effective cost of the scenario. It turns out that the associated cost of the multi-instance `Enlistment` deployment and adding it to the registry is more than 7 times more expensive than any other write transaction for all variants. This kind of inefficiency hints that the proposed design of the on-chain service should be reconsidered. In more detail, to promote separation of concerns, the on-chain services currently uses an approach of two smart contracts: a multi-instance intermediation subprocess smart contract working together with a separated singleton managing registry (3.3.2). So, to increase efficiency, the operations of the two smart contract could be merged. The feasibility and implications of the proposed single contract instance approach solution is left for future studies to discover.

[35] Due to the limitations of EVM and the chosen implementation, the efficiency gain would decline depending on the number of matches. This is due to the fact that filtering result (bitset indices) must be mapped through another layer of requests to retrieve a reference to an enlistment.

In addition to the high deployment cost of the `Enlistment` smart contract, the experiment also reveals the significant impact of its on-chain data storage strategy. Namely, it could be seen that while the bytecode size of `Enlistment` in the first two variants only differs 12.6% (Table 7), the gas usage of its deployment spikes nearly twice as much. The difference between the bytecode size and deployment cost could be explained by the variants data model transformation (3.3.2, 3.4.2). In more detail, the `Enlistment` smart contract of the data migration variant adds three storage variables compared with the baseline. Out of the three added variables, two are initialized with the data passed through the smart contract in the constructor during the deployment phase: `geohash` and `detailsJson` of the model of enlistment (the third, `offerAuthors`, is not valuated). Now, while the 9-byte `geohash` only initiates one `SSTORE` opcode of EVM [3], the `detailsJson` is a string of arbitrary length and a storage cost must be paid for each 32-byte chunk (or 32 characters of ASCII). Evidently, this kind of limitless storage strategy is not sustainable, especially with no maximum length guards in place. In the identified issue of high data storage cost, the results align with the work of Eberhardt and Tai [28].

What is more, building on the significance of the identified data storage cost, it would be beneficial to overlook the whole data model of the smart contracts. In a redesign, a leaner approach should be used to have the smart contract only include the data it needs for business logic processing. For example, in addition to the decorative data of an enlistment (e.g. number of bedrooms), there should be no place for duplicate data (e.g. currently, the information about landlord is both, on the contract and on the tenancy agreement). With gas usage effiency in mind, the data which is not necessary for the smart contract should seek other storage solutions. For example, the opportunities of content-addressed storage techniques may be used [28], [37].

The results also identify a peculiarity of EVM which may have an impact for any Ethereum application. In more detail, the experiment reveals a spike in gas consumption when there are no previous enlistments in the on-chain registry: 15000 more for variant 2 and 15064 for variant 3 (Table 8). Such cost spike could be explained with the pricing strategy of EVM which states that `SSTORE` operation costs 20000 gas only when the variable value had not been initialized before but 5000 when it is being changed from a previous value [3]. In the context of case study and the to-be externally available intermediation service, this means that the landlord who is the first to get his/her enlistment added to the registry, must pay extra.

Another identified low-level implication of EVM is that due to its stack size limitations, trivial function implementations may become non-trivial and require additional established data exchange protocols. For example, the results show that the design uses 4 JSON RPC calls to retrieve a single tenancy agreement (Figure 31). The reason behind such design is that due to a memory stack size of 16 [3], there could only be a limited number of local variables. So, as the accessible resource of tenancy agreement has a high number of members to be served in the output, a `Stack too deep, try removing local variables` compilation error appears[36]. To overcome the issue, a number of possible solutions may be considered. For example, the tenancy agreement accessor function is split into 4 smaller functions. However, this approach creates request overhead. An another approach to tackle the stack limitation for high-count variables, is to group together variables or members of the same type to arrays, as it was designed for the constructor of the `Enlistment` smart contract. However, this approach does not work with strings of arbitrary length because this would

[36] While the issue is partly brought forward by a debatable design decision to persist that much decorational data in the smart contract, the problem stands.

imply a dynamic two-dimensional array which is not supported[37]. So, to gain more efficiency as well as flexibility for a high number of variables with arbitrary length, a solution which involves custom encoding for information exchange between off- and on-chain components may be explored in the future studies.

In summary, we evaluate the least efficient prototype to be the data migration variant and the most efficient to be baseline. To increase the sustainability by raising efficiency for write as well as read operations, the most significant optimization design decisions are found to concern the strategies of smart contract deployment and data storage.

[37] http://solidity-doc-test.readthedocs.io/en/latest/frequently-asked-questions.html#can-a-contract-function-accept-a-two-dimensional-array [Accessed: 19-May-2018]

5 Conclusion

By building and analysing design artifacts, the paper shows how Ethereum blockchain can be used in implementing a process of real estate rental. To do so, it gradually moves an internal decentralized service of extracted peer intermediation subprocess to an autonomous external distributed application for disintermediation. In the scope of the study, 2 of the 3 proposed service migration phases are successfully carried out. By utilizing smart contracts, the implemented system allows actors to submit, review and establish assets of real estate property enlistments, offers and tenancy agreements.

In particular, the study shows the efficacy of Ethereum-enabled real estate rental but limited efficiency. By instantiating the internally-changing prototypes for evaluation, the paper provides a detailed quantitative analysis on the effect of smart contract design decisions. The results show that an increase in autonomy comes with a trade-off on efficiency. Moreover, the study warns application developers about the dramatic effect of multi-instance smart contract design and over-loading the on-chain storage with decorative data. Futhermore, the found evidence hints that service design in combination with the data querying limitations of EVM may negatively impact the quality of service by creating request overhead. Therefore, while blockchain read operations are free of fees, it is suggested that no less consideration should be put into their design decisions to find a balance between off-chain and on-chain operations and keep the overhead at minimum.

In future studies, we plan to design and develop the last phase of the service migration. The implementation raises some interesting areas for research such as migrating off-chain computations in to the browser application of external actor, providing role-based access in smart contracts and collecting signatures on tenancy agreements using an Ethereum account.

The source code for the prototypes and efficiency evaluation scripts could be found in the following Git repositories:

1) https://github.com/vindrek/blockchain-real-estate/releases/tag/Thesis-V1
2) https://github.com/vindrek/blockchain-real-estate/releases/tag/Thesis-V2
3) https://github.com/vindrek/blockchain-real-estate/releases/tag/Thesis-V3

The author of this thesis has received grant from IT Academy.

6 References

[1] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System.” p. 9, 2008.

[2] J. Mendling *et al.*, “Blockchains for Business Process Management - Challenges and Opportunities,” *ACM Trans. Manag. Inf. Syst.*, vol. 9, no. 1, p. 4:1--4:16, 2018.

[3] G. Wood, “Ethereum: a secure decentralised generalised transaction ledger,” *Ethereum Proj. Yellow Pap.*, pp. 1–32, 2014.

[4] Ethereum Foundation, “Solidity — Solidity 0.4.19 documentation.” [Online]. Available: http://solidity.readthedocs.io/en/v0.4.19/. [Accessed: 10-Apr-2018].

[5] R. Beck, J. Stenum Czepluch, N. Lollike, and S. Malone, “Blockchain - The Gateway to Trust-Free Cryptographic Transactions,” in *Twenty-Fourth European Conference on Information Systems (ECIS), Istanbul,Turkey, 2016*, 2016, pp. 1–14.

[6] J. Mattila, “The Blockchain Phenomenon - The Disruptive Potential of Distributed Consensus Architectures,” 2016.

[7] T. Eisenmann, G. Parker, and M. W. Van Alstyne, “Strategies for Two-Sided Markets,” *Harv. Bus. Rev.*, vol. 84, no. 10, p. 12, 2006.

[8] S. T. March and G. F. Smith, “Design and natural science research on information technology,” *Decis. Support Syst.*, vol. 15, pp. 251–266, 1995.

[9] P. Selonen, “From Requirements to a RESTful Web Service: Engineering Content Oriented Web Services with REST,” in *REST: From Research to Practice*, New York, NY: Springer New York, 2011, pp. 259–278.

[10] P. Offermann, O. Levina, M. Schönherr, and U. Bub, “Outline of a Design Science Research Process,” in *Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology*, 2009, p. 7:1--7:11.

[11] K. Peffers, T. Tuunanen, M. Rothenberger, and S. Chatterjee, “A Design Science Research Methodology for Information Systems Research,” *J. Manag. Inf. Syst.*, vol. 24, no. 3, pp. 45–77, 2007.

[12] A. Hevner and S. Chatterjee, “Design Science Research in Information Systems,” in *Design Research in Information Systems: Theory and Practice*, Boston, MA: Springer US, 2010, pp. 9–22.

[13] T. Erl, *Service-Oriented Architecture, Concepts, Technology, and Design*. 2005.

[14] R. Botsman, “The Sharing Economy Lacks A Shared Definition | Fast Company,” *Fast Company*. [Online]. Available: https://www.fastcompany.com/3022028/the-sharing-economy-lacks-a-shared-definition. [Accessed: 23-Oct-2017].

[15] N. A. John, “Sharing and Web 2.0: The emergence of a keyword,” *New Media Soc.*, vol. 15, no. 2, pp. 167–182, Mar. 2013.

[16] L. Piscicelli, G. D. Simone Ludden, and T. Cooper, “What makes a sustainable business model successful? An empirical comparison of two peer-to-peer goods-sharing platforms,” *J. Clean. Prod.*, Aug. 2017.

[17] M. P. Wilhelms, K. Merfeld, and S. Henkel, “Yours, mine, and ours: A user-centric analysis of opportunities and challenges in peer-to-peer asset sharing,” *Business Horizons*, vol. 60, no. 6, Elsevier, pp. 771–781, 01-Nov-2017.

[18] F. Boons, C. Montalvo, J. Quist, and M. Wagner, “Sustainable innovation, business

models and economic performance: An overview," *Journal of Cleaner Production*, vol. 45. Elsevier, pp. 1–8, 01-Apr-2013.

[19] N. M. P. Bocken, S. W. Short, P. Rana, and S. Evans, "A literature and practice review to develop sustainable business model archetypes," *Journal of Cleaner Production*, vol. 65. Elsevier, pp. 42–56, 15-Feb-2014.

[20] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," in *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, 1988, pp. 369–378.

[21] G. Greenspan, "MultiChain Private Blockchain — White Paper," 2015.

[22] V. Bharathan *et al.*, "Hyperledger Architecture," 2.

[23] I. Weber *et al.*, "On Availability for Blockchain-Based Systems," in *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, 2017, pp. 64–73.

[24] N. Szabo, "Smart Contracts: Building Blocks for Digital Markets," 1996. [Online]. Available: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html. [Accessed: 26-Nov-2017].

[25] Ethereum Foundation, "Geth · ethereum/go-ethereum Wiki." [Online]. Available: https://github.com/ethereum/go-ethereum/wiki/geth. [Accessed: 10-Apr-2018].

[26] Ethereum Foundation, "JSON RPC · ethereum/wiki Wiki." [Online]. Available: https://github.com/ethereum/wiki/wiki/JSON-RPC. [Accessed: 10-Apr-2018].

[27] A. Spielman, "Blockchain: digitally rebuilding the real estate industry," Massachusetts Institute of Technology, 2016.

[28] J. Eberhardt and S. Tai, "On or Off the Blockchain? Insights on Off-Chaining Computation and Data," pp. 3–15, Sep. 2017.

[29] V. Kopylash, "An Ethereum-based Real Estate Application with Tampering-resilient Document Storage," University of Tartu, 2018.

[30] NIMA, "Department of Defense World Geodetic System 1984," St. Louis, 2000.

[31] G. Niemeyer, "Geohash." 2008.

[32] C. Veness, "Geohash encoding/decoding." [Online]. Available: https://www.movable-type.co.uk/scripts/geohash.html. [Accessed: 07-May-2018].

[33] M. Kisanrao Nichat, N. RChopde, and M. K. Nichat, "Landmark Based Shortest Path Detection By Using A* Algorithm and Haversine Formula," *Int. J. Innov. Res. Comput. Commun. Eng.*, vol. 1, no. 2, p. 299, 2013.

[34] E. Schubert, A. Zimek, and H.-P. Kriegel, "Geodetic Distance Queries on R-Trees for Indexing Geographic Data," in *Advances in Spatial and Temporal Databases*, 2013, pp. 146–164.

[35] B. Jenny, B. Šavrič, N. D. Arnold, B. E. Marston, and C. A. Preppernau, "A Guide to Selecting Map Projections for World and Hemisphere Maps." pp. 213–228, 2017.

[36] Ö. G. Esenbuğa, A. Akoğuz, E. Çolak, B. Varol, and B. Erol, "Comparison of Principal Geodetic Distance Calculation Methods for Automated Province Assignment in Turkey." Istanbul, 2016.

[37] J. Benet, “IPFS - Content Addressed, Versioned, P2P File System (DRAFT 3).”

I. License

Non-exclusive licence to reproduce thesis and make thesis public