Implementation and evaluation of smart contracts using a hybrid on- and off-blockchain architecture

Ellis Solaiman | Todd Wike | Ioannis Sfyrrakis

School of Computing, Newcastle University, Newcastle upon Tyne, UK

Correspondence
Ellis Solaiman, School of Computing, Newcastle University, Newcastle upon Tyne, UK. Email: ellis.solaiman@ncl.ac.uk

Funding information
School of Computing, Newcastle University

Summary
Smart contracts can be implemented using either centralized or decentralized (blockchain) platforms. However, for a large class of applications, neither of these two alternatives on its own can provide adequate scalability, performance, quality of service, security, and trust requirements. We argue that for many applications, hybrid solutions that combine both on and off-blockchain components are more adequate. In this article, we introduce the design and implementation of a novel hybrid smart contract architecture built using the Ethereum blockchain connected to a centralized smart contract management system developed by us. We then compare and evaluate the implementation of an asset tracking service using three different architectures: on-chain, off-chain, and hybrid. We demonstrate that using a hybrid architecture, we can substantially improve performance of applications while retaining security and trust for critical tasks.

KEYWORDS
blockchain, hybrid, off-chain, on-chain, performance evaluation, smart contracts

1 | INTRODUCTION

We have recently seen a growing interest in the exploration of smart contracts built on blockchain technologies for the development of a wide range of innovative applications in healthcare, smart cities, cloud computing, the Internet of Things, and so on. A smart contract is an executable program (written in a programming language such as Java, C++, Solidity, Go, etc.) that is deployed to monitor interactions between two or more parties. Its main aim is to prevent or at least detect deviations from agreed upon behavior. To achieve this, the smart contract (i) intercepts each initiated operation, (ii) analyses it to determine if it is contract compliant, (iii) produces a verdict (contract compliant or none contract compliant), and (iv) records the outcome in an indelible log that is available for inspection, for example, to sort out disputes. In order to convert a legal contract into a smart contract, it is important to have clear rights, obligations, and prohibitions for each of the parties within the clauses defined within the contract. Penalties may also be defined within the smart contract for when a prohibited action is performed or when certain obligations are not met.

1.1 | Research problem

Despite the undoubted potential for the implementation of smart contracts on blockchain technologies such as, Bitcoin, Ethereum, and Hyperledger, blockchain based solutions are plagued with questions about their scalability, performance, transaction costs, and other questions...
that emerge from their decentralized nature. For example, Bitcoin can only process seven transactions per second—a poor throughput compared to visa’s 2000 transactions per second. Also, it takes Bitcoin about 10 minutes to publish a transaction to a block.

Long before Satoshi’s seminal paper that launched Bitcoin, alternative solutions to the execution of smart contracts based on centralized architectures were being proposed. However, a noticeable problem with implementing smart contracts using these platforms is that because of their centralized nature, they suffered from limitations such as trust being placed on trusted third parties (TTP), and the centralized server being a single point of failure. The central argument of this article is that in several applications, hybrid platforms composed from the integration of on and off blockchain platforms are more adequate. Unfortunately, the use of hybrid architectures in smart contract implementations is largely unexplored.

1.2 Research contributions

This article aims to help with covering this research gap by describing a hybrid architecture consisting of a centralized smart contract monitor known as the contract compliance checker (CCC), connected to the Ethereum blockchain. We then present and discuss a performance comparison for an asset tracking application implemented into three different ways: on-blockchain, off-blockchain (centralized), and hybrid. Therefore, the main contributions of this article can be summarized as follows:

1. It introduces the design and implementation of a novel hybrid smart contract architecture that combines an Ethereum based blockchain platform with a centralized smart contract management platform developed at Newcastle University (the CCC).
2. It validates the proposed architecture by evaluating its performance with the aid of an asset delivery scenario. Specifically, we compare the performance of the hybrid architecture with a blockchain only implementation and also with a centralized only solution using the CCC. Experimental analysis verifies the efficacy of our hybrid solution.

The remainder of this article is organized as follows: In Section 2, we present the example of an asset tracking service as a motivating scenario. In Section 3, important background concepts are discussed, and we provide an overview of work that is related to this research. The architecture of the system is presented in Section 4, and implementation details are discussed in Section 5. The performance analysis is presented in Section 6 and evaluated in Section 7. Finally, we finish with conclusions and future work directions in Section 8.

2 MOTIVATING SCENARIO

Figure 1 presents a scenario where parties interact with each other over the Internet in order to record correct delivery of an asset while it is being transported from a delivery service provider to a customer. Alice (the “delivery service provider”) is a delivery agency that has the resources to deliver an asset which Bob (the “recipient”) has purchased. For example, an asset such as a container that travels several ports from source A to destination D, the asset leaves Node A, is transported by sea to a neighboring Node B, then gets shipped to Node C, until it finally reaches its destination at Node D. An example of a contract that governs their relationship could include the following clauses:

1. The recipient (Bob) has the right to place with the sender (Alice) a delivery request for a specific asset, indicating a specified period which the asset must be delivered.
2. The sender has the obligation to respond with either a confirmation or rejection within 2 days of receiving the request. No response from the sender within 2 days will be treated as a rejection.
3. The recipient has the obligation to either pay or cancel the request within 2 days of the confirmation. No response from the recipient within 2 days will be treated as a cancellation.
4. The sender has the obligation to initiate the delivery process upon receiving confirmation from the sender and deliver the asset within the period. If the asset is not delivered within the specified period, the recipient has the right to a refund or accept the late delivery.
5. The recipient has the right to query the location of the asset at any time before the asset has delivered.
6. Upon delivering the asset, the recipient has the obligation to confirm that the asset has been delivered.

The clauses include contractual operations (for example, delivery request, rejection, and confirmation) that the parties have the right or obligation to execute under strict time constraints to honor the contract. We have highlighted the operations in bold. Although the clauses are relatively simple, they are realistic enough to illustrate our arguments. The clauses of the contract example describe the set of legal execution paths that the interaction between the two parties can follow. The aim is for the contract written in English to be converted into a smart contract that can be used to monitor adherence of the contracted parties to the stipulated clauses. As a first step to achieving this, it helps to convert the English text contract into a systematic notation. Figure 2 shows a graphical view of the contract example.
The operations in the English text contract have been mapped to messages sent by a party to its counterpart. For example, the execution of the operation delivery request corresponds to the DelReq message sent by the recipient to the sender. Similarly, the execution of the operation reject corresponds to the Reject message sent by the sender to the recipient. The diamonds represent exclusive splits in the execution path and have been labeled with O (Obligation) and R (Right). TO stands for Time Out and represents the time constraints stipulated in the contract. Failure to execute and obligatory operations results in abnormal contract end with disputes to be sorted off line.

Figure 2 demonstrates that our contract example can be modeled and implemented as a finite state machine (FSM) in a Turing-complete language. The challenge for the developer is to select a suitable architecture and technology for implementation. There are several approaches to smart contract implementations:

- **Centralized**: The smart contract is deployed on a Trusted Third Party. This approach is also known as off-blockchain implementation since there is no blockchain involved.
- **Decentralized**: The smart contract is deployed on a blockchain platform, for example, on hyperledger or Ethereum. This approach is also known as on-blockchain.
- **Hybrid**: The contract is split and deployed partially off-blockchain and partially on-blockchain. Some of the clauses are monitored/enforced off-blockchain, whereas others are enforced on-blockchain. The separation of operations into on and off-blockchain is based on several criteria including cost, performance, smart contract languages capabilities, and privacy; see References 14, 16, and 17.

Smart contracts that are entirely built on blockchains have notable limitations, and these are mainly scalability and costs. This is due to the algorithms which blockchains use to verify that the information being stored on the blockchain is correct. It is also potentially expensive to store data...
on a blockchain as there is a cost associated with writing data to a blockchain such as Ethereum. Also, smart contracts built entirely on a blockchain have an extra time related cost incurred when performing the required calculations to check whether the actions being performed by the contracted parties are contract compliant or not, in addition to time costs for any required data manipulation. This means on an industrial scale, there are problems for using public blockchains for any applications, which have a large user base. To compare the proposed application to a real-world example, consider Amazon. Amazon moved 5 billion assets in 2017, equating to over 13 million assets per day, while the public Ethereum blockchain peaked at just over 1 million transactions per day in December 2017, meaning smart contracts would currently not be able to keep up with the throughput required for commercial usage. In addition to this, there would be a significant cost to the clients of Amazon if they were to move their asset tracking system to the Ethereum blockchain due to the volume of data which would need to constantly be manipulated and overwritten on a daily basis.

3 | BACKGROUND AND RELATED WORK

3.1 | Smart contracts: Background

A smart contract is an event-condition-action stateful computer program, executed between two or more parties that are reluctant to trust each other unguardedly. It can be regarded as an FSM that keeps a state that models the development (from initiation to completion) of a shared activity. For instance, in References 15, and 18-20, the state is used for modeling changes in rights, obligations, and prohibitions as they are fulfilled or violated by the parties.

Research on executable contracts can be traced back to the mid 80s and early 90s. In 1997, Szabo used the term smart contract to refer to contracts that can be converted into computer code and executed. However, commercial interest in smart contracts emerged only in 2008 motivated by the publication of Satoshi’s Bitcoin paper that inspired the development of cryptocurrencies, smart contracts, and other distributed applications. Satoshi departed from the centralized approach taken in previous research and demonstrated how smart contracts can be decentralized.

Depending on the number of instances (copies) of the smart contract deployed to monitor and enforce the contract, we distinguish between centralized and decentralized (distributed) approaches (Figure 3). In the figure, A and B are business partners, for example, Alice and Bob of our contract example of Section 2. SC is the corresponding smart contract. op stands for operation executed against SC, and rp is the corresponding response. TTP node is a node under the control of a TTP. N1, …, N4 are untrusted nodes. CP stands for Consensus Protocol. As shown in Figure 3A), a contract can be implemented as a centralized application that uses a single instance of the smart contract (SC) running in the TTP node. Besides the disadvantages that a TTP introduces (single point of failure, trust placed on the TTP etc.), this approach is comparatively simpler that the decentralized approach. The decentralized approach relies on a set of untrusted nodes instead of a single TTP that are used for running several identical instances (shown as SC1, …, SC4) of the smart contract. In this approach, A and B are free to place their operation against any of the instances. The price that the decentralized approach pays for getting rid of the TTP is that the untrusted nodes must run a consensus protocol to verify that a given operation has been executed correctly, and to keep the states of SC1, …, SC4 identical. Depending on the protocol used, its computational, communication, and performance degradation cost might be unbearable or its consistency guarantees inadequate to the extent of rendering the decentralized approach unsuitable.

3.2 | Related work

Research work on the monitoring and enforcement of interactions between parties using smart contracts was pioneered by Minsky in the mid-1980s with work on law governed interaction (LGI). LGI is a law enforcer that regulates the interaction between autonomous and distributed agents linked by a communication network. This was followed by Marshall in the 1990s. Although some of the tools developed exhibited some
decentralized features, those systems took mainly a centralized approach. Within this category falls References 11 and 25. To the same category belongs the contract enforcer suggested by IBM, and the Heimdhal engine aimed at monitoring state obligations (for example, the service provider is obliged to deliver an asset within x business days). Directly related to our work is the contract compliant checker reported in References 15 and 20, which also took a centralized approach to gain in simplicity at the expense of suffering from all the drawbacks that TTPs inevitably introduce.

The publication of the Bitcoin paper in 2008 motivated the development of several platforms for supporting the implementation of decentralized smart contracts. Platforms in References 1-3 are some of the most representative. A good summary of the features offered by these and other platforms can be found in Reference 27. Although they differ on language expression power, fees, and other features, they are convenient for implementing decentralized smart contracts.

An early example of a permissioned distributed ledger that is similar in functionality to the hyperledger blockchain is the "Business to Business Objects" platform (B2Bobjects). B2Bobjects is a component based middleware implemented at Newcastle University in the early 2000s and used for the enforcement of decentralized contracts. B2BObjects offers consensus services (based on voting initiated by a proposer of a state change) and storage for recording non-repudiable and indelible records of the operations executed by the contracting parties. B2Bobjects is permissioned (as opposed to public) in the sense that only authenticated parties are granted access to the object. Work at Newcastle University also looked at techniques for verification (model checking) of executable contracts deployed using B2BObjects.

The hybrid approach that we suggest and evaluate in this article addresses problems that neither the centralized or decentralized approaches can address separately. The original idea for hybrid contract implementation emerged with the off blockchain payment channel discussed in References 1 and 30. The concept of logic-based smart contracts discussed in Reference 31 has some similarities with our hybrid approach. They suggest the use of logic-based languages in the implementation of smart contracts capable of performing on blockchain and off-blockchain inference. The difficulty with this approach is that current blockchain technologies lack support for logic-based languages. In our work, we rely on the native languages offered by the blockchain platforms, for example, Ethereum's Solidity. On- and off-blockchain enforcement of contractual operations is also discussed in Reference 32; though an architecture is presented, no technical details about its implementation or functionality are discussed. Another conceptual design directly related to our work is private contracts executed in the Enigma architecture. Like in our work, a private contract is a conventional business contract with contractual operations separated into on- and off-blockchain categories. Similar to our hybrid design, they use a blockchain platform (Ethereum) to execute blockchain operations. However, unlike in our work, instead of using a TTP to execute off-blockchain operations, they use a set of distrusting Enigma nodes running a Secure Multi-party Computation (SPC) protocol that guarantees privacy. In this collaborative architecture, the blockchain is in charge. It is responsible for guaranteeing that the contractual operations are honored and for delegating tasks to the Enigma nodes as needed. The integration of the SPC protocol ensures that the smart contract running in the Ethereum blockchain never accesses raw data that might compromise privacy. Unlike our TTP, the Enigma nodes charge computation and storage fees, just like Ethereum and Bitcoin do. The cost that the Enigma architecture pays for privacy protection is complexity. Similar to our work also is Etkiden, a system for combining blockchains with Trusted Execution Environments (TEEs). The authors report significant performance improvements; however, they do not discuss the challenges of testing and verification hybrid smart contracts as we do in, where we also discuss the advantages and disadvantages of each of the on-blockchain, off-blockchain, and hybrid approaches. Another technique suggested for addressing the scalability problems of blockchains is sharding. This approach involves alteration of the blockchain architecture. The central idea of sharding is parallelization with subsequent cross-shard consensus: several transactions are executed in parallel by different independent groups of nodes (called shards) selected from the group of conventional miners (see for example). To avoid the risk of consuming inconsistent data, one can replace consensus algorithms that offer only eventual consistency like the proof of work (PoW) algorithms with algorithms that offer stronger consistency. An example of such algorithms is classical Byzantine fault-tolerant (BFT).

In Reference 39, the authors present an improvement to the architecture of the Hyperledger Fabric designed to improve the throughput of the blockchain. The method by which the number of transactions per second was increased involved altering some of the operations on the blockchain by manipulating data and memory and implementing concurrent operations. The proposed changes to the blockchain would require altering the entire Hyperledger platform and would only be applicable to applications based upon the Hyperledger blockchain. However, this work shows that there is potential for significant increases in performance, and particularly increasing the throughput of Hyperledger by nearly 600% with indications that the throughput could be further increased to 50,000 transactions per second which would rival VISA. In addition to the potential benefits to Hyperledger, hopefully lessons learnt from this work can be applicable to other blockchains solutions and hybrid solutions such as ours. The advantage of our work is that it aims to provide a solution to the scalability issues regardless of what blockchain is being used.

The idea of applying blockchain technologies to asset tracking is discussed in Reference 40, whereby an asset has a starting point and a destination, with multiple intermediary locations. The design of the system is also aimed toward following the conventional delivery model of signing for the delivery of an asset, meaning that the sender and recipient both agree when the asset has been delivered and leaves the possibility of completely automating the system.

One of the current solutions to this problem is designed by IBM. The system uses RFID tags and barcodes to keep track of the location of a given asset and each person in the supply chain publishes their information to a shared ledger. The system is designed using Hyperledger and is designed more to keep track of business assets and to be used in an industrial scenario rather than a retail setting. One of the limitations with the system is that it only handles the part of the contract concerned with the delivery of the asset. This means the system does handle
other events such as placing of payment, and therefore cannot ensure the entire series of events involved in the delivery process are contract compliant.

4 \hspace{1cm} \textbf{SYSTEM ARCHITECTURE}

As discussed earlier, our aim is to build smart contracts for an asset tracking service in three different ways (on-blockchain, off-blockchain, and hybrid) and to compare their performance. The on-blockchain solution consists of a web front end which is used on the client’s system and connects to the blockchain (Ethereum). The off-blockchain solution connects the web front end to a centralized Smart Contract Engine known as the CCC, built at Newcastle University. Details of the architecture of the CCC will be provided later.

For the hybrid solution, one of the challenges is the complexity that inevitably emerges from the interaction between the off-blockchain and on-blockchain components. Several interaction models can be devised. For example, we can deploy the off-blockchain and on-blockchain smart contracts in a master-slave or peer-to-peer relationship. Alternatively, we can place them in a pipe-relationship where an off-blockchain smart contract is deployed by one of the contractual parties to mirror the work of the on-blockchain smart contract, say to double check its outputs. Other deployment alternatives are discussed in References 13 and 14. We believe that our chosen solution is sufficient for our purpose, which is to provide a performance comparison between the hybrid, on-blockchain, and off-blockchain solutions. The central idea of the hybrid approach is to split the contractual operations into two sets: off-blockchain operations and on-blockchain operations. Operations from the first set are evaluated for contract compliance by a centralized smart contract deployed on a TTP. By contrast, operations from the on-blockchain set are evaluated by a decentralized smart contract deployed on a blockchain. The hybrid architecture we have chosen to implement can be seen in Figure 4. It can be implemented on the basis of several alternative technologies. In this work, we use the Ethereum blockchain to realize the decentralized smart contract component. To realize the centralized smart contract component, we use the latest version of the centralized CCC. The integration follows a master-slave relationship between the centralized and decentralized smart contract components where the former is in charge. A description of the CCC architecture is provided in the next sections.
4.1 The contract compliance checker

The CCC offers several features that ease integration with a blockchain platform such as Ethereum. It is an open-source tool designed for the monitoring and enforcement of smart contracts. The CCC is the platform we use for our centralized solution, and as part of the hybrid solution. The CCC processes each event to determine if it is contract compliant (CC) or none contract compliant (NCC). The execution of a business operation is CC if it satisfies the following three conditions and is NCC if it does not:

1. \( b_o \in BO \); the business operation matches an operation within the set of business operations expected by the CCC,
2. \( b_o \vdash ROP_{rp} \); the business operation matches the ROP set of its role player (meaning, the role player that performed the operation has a right/obligation/prohibition to perform that particular operation). By “match,” we mean that for a valid business operation \( b_o \), and a particular role player’s ROP set; \( ROP_{rp} \) where: \( R_p = \{r_1, \ldots, r_m \} \), \( O_p = \{o_1, \ldots, o_m \} \), \( P_p = \{p_1, \ldots, p_m \} \), and \( m \geq 1 \), their relationship should be that: \( b_o \in r_j \) or \( b_o \in o_j \) or \( b_o \in p_j \), where \( 1 \leq j \leq m \).
3. the business operation must also satisfy the constraints stipulated in the contractual clauses. An example of a constraint is the obligation to make a delivery within “the agreed upon period” in clause 4 of our example contract.

We also consider that the execution of a given sequence of operations is NCC if it includes one or more operations that are flagged by the CCC as NCC. A sequence of operations is also known as an execution sequence or execution trace and drives the choreography of interaction from its initial state to a final state.

4.1.1 Architecture of the CCC

A high level view of the CCC is shown in Figure 5. It consists of two layers: The CCC Engine (The Logical Layer), and the CCC Service (The Presentation Layer). The CCC Engine is responsible for processing business events and for determining whether they are contract compliant or not. The CCC Service is an interface to the CCC Engine, it is used for delivering business events to the CCC, and for collecting the corresponding responses. In addition, the CCC Service can be used for loading and editing the clauses of the smart contract. The functionality of the architecture is as follows: A business
event is received through the monitoring channel as an XML document that includes the names of the participants, the business operation, and its outcome from the set: \{(Success, BizFail, TecFail)\}:

```xml
<event>
  <originator>Recipient</originator>
  <responder>Sender</responder>
  <type>DelReq</type>
  <status>success</status>
</event>
```

The XML document representing the business event is passed to the BEvent queue. Business events are retrieved and converted using the xml2Obj/Obj2xml Converter from their XML format into business event objects. Events are then passed to the Logical Layer. The filter discards mismatched business events that are not among the permitted events defined within the ROP set. Business events that pass this filter are inserted into the Event queue. All deadlines are set and reset by the relevance engine and enforced by the timer. Timeout events are added to the filter as required by the contract and are examined by the filter to decide if events are mismatched. For example, receiving a Confirm event from the Sender after the deadline has elapsed will be treated as mismatched. The relevance engine removes a business event from the head of the event queue and compares it to the rules stored in the Smart Contract. Rules that match the event under examination are triggered to determine if their conditions are satisfied. The actions of the rules whose conditions are satisfied are executed, and this may alter (add/del) the current state of the ROP set. For our example in Figure 2, a rule triggered by a Confirm business event and finds its conditions satisfied will delete (disable) the Sender’s right to execute another Confirm business operation, and delete the Sender’s right to execute a Reject operation. The rule also will add (impose) an obligation on the Recipient to either initiate the execution of Pay, or initiate the execution of Cancel. The event is then stored in the Event logger as a record for any future dispute resolution. The relevance engine eventually declares the business event either CC or NCC and produces a response as a business object, which is sent out to the Presentation Layer. The business object passes through the xml2Obj/Obj2xml Converter, where it is serialized into an XML message of the following format:

```xml
<result>
  <contractcompliant>true | false
</contractcompliant>
</result>
```

The xml2Obj/Obj2xml Converter inserts the response into the outcome queue, which can be accessed by the contracted parties. The Presentation Layer allows a “rule manager” to update the contract rules at run time. For this purpose, rules can be edited using the Smart Contract editor (in a browser) and sent to the upload smart contract service as a conventional RESTful POST operation. The upload smart contract service is responsible for producing a drl (Drools) file (for example new-rules.drl) from the payload of the POST operation, and for uploading it to the CCC Logical Layer to replace the Smart Contract.

The CCC Logical Layer is implemented using JBoss’s Drools rules Engine. Drools is a declarative Turing complete language designed for writing business rules. The Drools rules engine powers the decision making capabilities of the relevance engine. The relevance engine acts as a wrapper for the Drools rule engine and its responsibilities include the initialization of the contract, as well as the addition and processing of events received from the Presentation Layer. The Presentation layer exposes the CCC as a RESTful web service. Its aim is to enable the exchange of XML event messages between the CCC and the contracted clients, and to ease the editing and update of the contract rules. The Presentation Layer is implemented using the JBoss Enterprise Application Platform (EAP). The Event queue and the outcome queue are implemented using JBoss’s HornetQ (a message oriented middleware layer), and using the Java Message Service (JMS) API. A Message-Driven Bean (MDB) receives business events from HornetQ and passes them to the XML2Obj/Obj2XML converter, which is implemented using Java.

### 4.2 Ethereum

We chose the Ethereum platform for implementing the decentralized smart contract service for the following reasons: It is currently one of the most mature blockchains. It supports solidity—a turing–complete language that designers can use for encoding stateful smart contracts of arbitrary complexity. Equally importantly, Ethereum provides, in addition to the main Ethereum network (Mainnet), four experimental networks (Ropsten, Kovan, Sokol, and Rinkeby) that developers can use for experimenting with their ideas using Ethereum tokens instead of real ether money. A summary of technologies used is as follows. Solidity is the main programming language used to create Ethereum smart contracts. Remix IDE is an online IDE for development of Solidity smart contracts. It can identify any semantic errors in code as well as offering warnings when functions can potentially...
run over the gas limit of the contract. Node Package Manager (NPM) is an open-source collection of JavaScript packages. It is used connect the website to the Solidity code on the blockchain. MetaMask is a browser extension for Google Chrome, which connects the browser to the Ethereum network. Truffle is a development framework used to interact with the Ethereum blockchain. Ganache is a personal blockchain. It is used for testing contracts in combination with Truffle and MetaMask. JBoss: middleware designed to aid in hosting applications.

4.3 Deployment of the hybrid smart contract

The technology used in the integration of the CCC with the Ethereum Blockchain is shown in Figure 6. The contract is split into two parts: (i) the smart contract \( SC_c \) deployed on the a TTP (CCC), encoded in drools, and deployed on a Mac computer (regarded as a TTP node). On the Mac, we also deploy an Ethereum client connected to the rinkeby Ethereum network. (ii) The blockchain based smart contract \( SC_d \), encoded in the Solidity language. \( SC_d \) receives string messages from the \( SC_c \) and replies with another string message.

The client acts as a web client to the CCC. We use it to test the implementation of the contract example implemented by the combination of \( SC_c \) and \( SC_d \). We provide the client with all the execution sequences required to measure the performance of the hybrid architecture. We generate all the execution sequences in advance and store them in a folder with \( N \) sub-folders: each sub-folder corresponds to one execution sequence and contains as many files as messages are included in each execution sequence. As shown in Figure 6, the CCC relies on the web3j library to communicate with the Ethereum client. Among other services, the web3j library includes a command line application that mechanically generates wrapper code from a smart contract specified using solidity and compiled using the solc compiler. The CCC (a Java application) can use the generated wrapper code to communicate with the \( SC_d \) contract, through the json-rpc API provided by Ethereum. In addition, the web3j library provides an API for the CCC to unlock an Ethereum client account by providing the path to the key-store file and the password.

5 IMPLEMENTATION OF THE ASSET TRACKING SERVICE

The set of functional requirements for the systems we aim to build is based upon the clauses in the contract described prior, which are detailed in Figure 2. In order to determine the cost and time differences when implementing the solution using the three different approaches (on-blockchain, off-blockchain, and hybrid), the requirements of the system need to be established first so that the on-blockchain, off-blockchain, and hybrid systems consistently enable the same functionality. The functional requirements can be seen in Figure 7.

5.1 On-chain implementation

The initial step of implementing the on-chain section of the asset tracking project was to provide the data structures necessary for representing an asset and a location. A location requires a unique identifier. To make the system easier to understand and use, a string representation was applied to represent a name which is the unique identifier. A location also contains a unique integer identifier used to check if the identity of two locations is the same. This is because Solidity does not have a string comparison method, meaning to compare if two strings are the same, the strings need to be
hashed and then have the hashed values compared, which is computationally expensive and increases the gas cost of some of the methods, whereas comparing two integers is a cheaper operation with regards to gas cost. While it is more expensive to store the data initially, if the proposed solution was implemented on a widespread use, the more important factor would be the average cost of executing methods, meaning integer comparison would be preferred to string comparison. A location also required an address field to represent the owner of a specified location to allow the contract to compare the owner of a location with the owner of an asset or the address initiating any of the methods in the contract. A Boolean was also used to check that the name of the location is unique when it is created and used when adding assets to the map of all assets in the system.

An asset required a unique identifier which was also represented as a string for ease of use, and similarly to the location data structure also had a unique integer identifier for comparison within the contract for the same reasons described above. An asset also required an address variable which is the address of the user who currently is in possession of the asset. To represent what stage in the delivery process each unique asset is currently in, an integer was used with the numbers 0 to 6 representing the different stages as indicated in Figure 8. The rights, obligations, and prohibitions described are generated from the contract design in Figure 2.

An integer variable is stored in the asset class which is used to keep count of the length of the route between the starting and ending locations when adding new locations to the route. There is also a Boolean check variable in the Asset class to ensure that the string identifier is unique. In order to track the delivery time of each stage of delivery, a mapping of integers to integers is used. The integer keys in the mapping are equal to the integer keys in the mapping for previous locations. A location is stored in the asset structure as the destination of the delivery. This is so that the contract can check when the asset has reached the end of its route. Each of the assets and locations must be stored on the blockchain. This was achieved by having two mappings: one of the location names to the location struct and one of the asset names to the asset struct. Mappings were chosen over arrays to store the locations and assets as there is no need to iterate over every asset or location, so it is computationally less expensive to use a mapping to store and access individual assets or locations, which reduces the cost of interacting with the methods on the blockchain. Each of the functions in the code also has a list of require() statements at the beginning. This is a way of validating the data, which is passed into the methods and also controls who can execute each method at a particular time. A require() statement is the same as an if-else statement but if the conditional statement is false, the function is not executed, and the gas cost of the function is returned to the sender of the message. Therefore, require() statements are used as a method of saving as much gas as possible by refunding gas when functions are not called and is an important method of reducing the gas cost of executing functions on the blockchain.

### 5.2 Off-chain implementation

The CCC is a piece of software which evaluates contractual compliance given a set of rules written in Drools and a series of events in the form of XML files. The XML files have four fields: originator, responder, type, and status. In the contract used in this project, the sender and recipient are the originators and responders of the messages, which are sent between each other. Using the Drools language, each of the role players in the scenario is assigned an initial set of rights, obligations, and prohibitions. The CCC is then able to check when each event arrives, and whether the event complies with the ROP set. If an event is compliant with the ROP set, then the appropriate alteration of the ROP set involved is made; for example, upon initialization the recipient has the right to request delivery of an asset. Once the event comes in whereby the recipient requests the delivery of

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asset creation</td>
<td>A user must be able to add an asset to the system to be tracked</td>
</tr>
<tr>
<td>Location creation</td>
<td>A user must be able to add a location to the system to store where an asset is located.</td>
</tr>
<tr>
<td>Asset locating</td>
<td>A user must be able to search for the location of a unique asset.</td>
</tr>
<tr>
<td>Asset identification</td>
<td>A user must be able to uniquely identify an asset</td>
</tr>
<tr>
<td>Location identification</td>
<td>A user must be able to uniquely identify a location</td>
</tr>
<tr>
<td>Asset history</td>
<td>A user must be able to retrieve a list of all the locations an asset has been located previously</td>
</tr>
<tr>
<td>Asset movement</td>
<td>A user must be able to move an asset from its current location to a specified location</td>
</tr>
<tr>
<td>Delivery request</td>
<td>A user must be able to request the delivery of a specified asset to a specified location</td>
</tr>
<tr>
<td>Delivery response</td>
<td>Once a delivery request has been sent, a user must be able to accept or deny the request</td>
</tr>
<tr>
<td>Delivery initiation</td>
<td>Upon accepting a delivery, a user must be able to indicate that movement of the asset has begun</td>
</tr>
<tr>
<td>Delivery of asset</td>
<td>A user must be able to indicate they have delivered an asset to a specified location</td>
</tr>
<tr>
<td>Confirmation of delivery</td>
<td>A user must be able to confirm that an asset has been delivered</td>
</tr>
<tr>
<td>Asset history</td>
<td>A user must be able to see a list of all the previous locations where an asset has been stored</td>
</tr>
</tbody>
</table>
FIGURE 8  The rights, obligations, and prohibitions of the parties involved in the contract at each stage of the delivery process

<table>
<thead>
<tr>
<th>Integer</th>
<th>Description</th>
<th>Rights</th>
<th>Obligations</th>
<th>Prohibitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Newly created asset</td>
<td>Recipient can request a delivery.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Asset has been delivered to the destination location</td>
<td>Recipient can request a delivery.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Delivery requested</td>
<td>-</td>
<td>Sender must accept or reject the delivery request</td>
<td>Another recipient cannot request delivery for the same asset.</td>
</tr>
<tr>
<td>3</td>
<td>Pay for or cancel the delivery</td>
<td>-</td>
<td>The recipient must respond to the confirmation of delivery by either paying for the delivery of cancelling the request</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Creating delivery route</td>
<td>Recipient can request the location of the asset.</td>
<td>Sender must add locations to the route until the route is completed</td>
<td>Sender cannot add a location to the route more than once</td>
</tr>
<tr>
<td>5</td>
<td>Route finished, out for delivery</td>
<td>Recipient can request the location of the asset.</td>
<td>Sender must deliver the asset to the recipient within the agreed upon time.</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Asset delivered to location, awaiting confirmation from recipient</td>
<td>Recipient can request the location of the asset.</td>
<td>Recipient must accept the delivery of the asset.</td>
<td></td>
</tr>
</tbody>
</table>

an asset, the right to request delivery for an asset is removed from the recipient and the sender is assigned the obligation to respond to the request with either a confirmation that the recipient can have the asset delivered or a rejection of the delivery.

5.3  Hybrid implementation

When implementing the hybrid solution, the first step was to decide upon which parts of the system would be placed on-blockchain and which parts would remain off-blockchain. Due to the blockchain being used as a decentralized database, this meant that any parts of the contract where a significant change in data takes place must be added to the blockchain. Any methods that are used for payment must also be on the blockchain. The areas where the biggest change in data occurs are when a recipient has paid for delivery of an asset, as it means that the asset must then begin the process of delivery. If the payment method is placed on the blockchain, the cancellation method must also be on-blockchain. This is so that whether the recipient pays for or cancels the delivery, the status is on the blockchain and the result of this can be retrieved at any time. Due to the process of adding to the list of locations where the asset is being delivered being a data change, this method also needs to be on the blockchain; however, it may be possible to adjust the CCC to take in extra fields in the Event class to pass locations into the CCC to add to the route and have one final method when the route is finished which pushes the result to the blockchain. The final indication of when an asset has been delivered needs to be on the blockchain, as well as the result of whether the recipient has claimed a refund or accepted a late delivery as this is related to payment. On the hybrid architecture, the stage variable in the asset data structure was no longer required, as all requests go through the CCC first, which checks that the correct clause of the contract is being executed. So this is removed from the asset data structure.
6 | PERFORMANCE ANALYSIS

6.1 | Gas cost

To calculate the cost of each implementation, the gas cost of each method within the Ethereum solidity code was measured using the truffle console. The console logs the gas cost when a function is executed. Some functions are shared between the on-blockchain and hybrid solution. The gas price of each function is not constant due to the use of strings in each method, which are of variable length. So each function has a base cost and a cost per character. Figure 9 shows a table of values for the gas cost of these functions. The gas cost of the findAsset() function is a constant value of 0 as it is a function which does not manipulate any data and only returns a value. There are also functions that are only used in the fully on-blockchain solution. Figure 10 shows a table of these functions. There are functions that are shared between the hybrid and on-blockchain implementations but have different gas costs. This is due to a reduced amount of information required to be manipulated and posted to the blockchain in the hybrid architecture of the implementation. Figure 11 details the functions that are shared between the two implementations but have different gas costs. Finally, Figure 12 shows a table of the deployment and migration cost of the on-blockchain and hybrid implementation of the asset tracker. The last step for testing the average gas cost of each of the implementations was to calculate the gas cost of each of the series of contract compliant events which can happen in the system. Figure 13 shows the results. The minimum total gas cost of each function was calculated by taking the minimum gas cost of adding two locations and an asset to the system along with invoking all the methods which are required to iterate through each clause in the contract. The rejection of deliver test requires 23.6% less gas using the hybrid implementation, the cancellation test requires 23.3% less gas using the hybrid implementation, and the full contract requires 32.1% less gas using the hybrid implementation.

6.2 | Average execution time

To test the average execution time of the contract, Ganache was set up to mirror the rate at which Ethereum blocks are mined. In practice, this solution would be implemented on a private blockchain with multiple nodes connected to the network. Etherscan shows that the average block time for Ethereum has ranges between 10 and 30 seconds per block, being mostly restricted between 10 and 20 seconds. To test the average execution time of the solutions, we ran the tests at 10 seconds per block to get an indication of the best-case time for the solution. To make sure that the tests were fair for each system, a predetermined series of events needed to be derived for each solution to see how each solution handles the requests. The series of events which were chosen were a combination of full sequences of events which can be executed on the blockchain. This meant that
no events which were not contract compliant were chosen, as when these events are attempted to be executed in the on-chain solution, they are caught in the require statements and not executed at all. The tests were conducted via the web browser created using MetaMask to confirm every transaction, meaning there is latency added for the accepting of transactions. All the series of tests were run for 10 assets and executed 5 times each. Figure 14 shows the results of each test. The first test was a recipient placing an order for delivery, and that order being rejected. The second test was the order being placed and accepted, then canceled by the recipient. The next test was to execute the entire delivery process, with the asset being sent from one location to the next location with no extra stops on the route (a direct delivery). The chart shows that for the test where the delivery request was rejected, the response time was exactly the same for the hybrid implementation and the off-blockchain implementation. This is because no information is actually sent to the blockchain at all for the rejection of a delivery request. The on-blockchain implementation had an execution time, which was nearly 35 times longer than the off-blockchain architecture for this test. The test case where the recipient cancels the delivery after the order has been placed is the first case where the hybrid implementation takes longer than the off-blockchain implementation. However, it is clear to see that there is a significant improvement in the time taken for this architecture to process the requests, reducing the time taken by over 80%. The final test also showed a significant reduction in the time taken for a full delivery to be completed, taking one third of the time to execute the same number of orders.

7 | EVALUATION

7.1 | Gas cost

As can be seen from the testing of the gas costs for each of the implementations discussed, the hybrid implementation is a lot cheaper than the entirely on-chain implementation. The most interesting difference in the gas costs is between the functions which are shared between the hybrid and on-blockchain implementations. The addToRoute() function is particularly interesting as the difference in gas cost between the implementations is very low. This is due to the code between the two implementations being quite similar. On the other hand, the payOrCancel() method requires 17.8% less gas on the hybrid implementation. This is because the hybrid smart contract does not contact the blockchain to confirm which actions
have occurred, meaning there is no writing to the blockchain in the hybrid solution, whereas in the on-blockchain solution, the smart contract has to change the stage variable. It is clear to see that in order to decrease the amount of gas required for each of the functions is to remove computation from the Ethereum smart contract where possible. This links to research in Reference 16, which suggested off-chaining as much computation as possible. A particularly relevant idea from the paper was the Delegated Computation Pattern, where a third party does computation off the blockchain and rather than having to replicate this on the blockchain, generate a proof of correct execution. While this solution does not have a proof of correct execution on the blockchain, using the CCC to check contracts compliant off the blockchain is a similar idea.

### 7.2 Average time

The big difference between the three solutions is the average time to execute each of the contracts. While this was a known issue with using a public blockchain such as Ethereum, our results clearly demonstrate the sizable difference in performance between the three solutions. Although this could be inflated by having to accept each transaction through MetaMask, there is still a limit on how fast the blockchain can process the data, which is dependent on the gas limit on each block. One of the problems with the testing method used for the on-chain solution was that each time a clause in the contract was called and accepted with MetaMask, the next clause in the contract for that asset could not be checked for contract compliance until the blockchain had mined the block. This is because the data are not actually put onto the blockchain until the block is mined. This means that the require() statements cannot be passed until the block is mined. This then creates a backlog of transactions that are waiting to be accepted when it takes too long to mine a block before the next clause in the contract is executed. This problem would be further exacerbated with slower block mining speeds. Our tests were conducted for a best case scenario taking the peak of the average block speed for Ethereum from the past year. If the speed was to slow down to 15 or 20 seconds per block, more of a backlog would be created, leading to an even slower throughput. Despite this, we believe this would not be a problem for the proposed asset tracking system and for a host of similar applications. In a real-world application based on a blockchain asset tracker, the users of the software would not be going through the clauses in the contract within 10 to 30 seconds of each clause being executed. This would mean that the issue of blocks being mined too slowly for the data to be updated would only be a problem for applications where contract clauses need to be executed at a faster rate. Also, the above discussion assumes implementation of a solution using the public Ethereum blockchain. Mining speeds maybe significantly reduced if implementing a solution using a private permissioned chain.

### 7.3 Transactions per block

Taking into consideration the impact of mining speeds on the average execution time discussed in the previous section, another area can also be measured to discover the maximum potential throughput of the system. Each of the functions in the contracts has a gas cost that is only variable by a small percentage per character passed into the function parameters. This means that the base cost of the function could be used to calculate the maximum number of each function that could be called within a particular block. This would give us an idea of how many clients could be interacting with the smart contract at a particular time. The default block gas limit on Ganache is 6,721,975 gas per block. Using the values generated for the minimum gas cost per function, we can generate the maximum number of function calls per block, as shown in Figure 15. The methods where there is no orange bar are methods that do not have an on-blockchain execution on the hybrid architecture. Figure 16 shows the increased throughput of each method using the hybrid architecture. As the table shows, the throughput of the system is increased significantly in most cases. However, the maximum throughput is only as large as the smallest throughput in the system. Even though adding an asset to the system has the smallest number of possible function calls per block, it is not what will be used to calculate the throughput, as there is only a finite number of assets which can be added to the system. But we assume that there are an infinite number of deliveries, which can occur across time. This means the lowest throughput of the system is in the addToRoute() method. This is problematic for the hybrid solution as the increased throughput using this solution is only one function call per block, equating to a 1.78% increase in maximum throughput. This suggests that when using a public blockchain such as Ethereum, the hybrid solution is actually not very impactful at combating time related scalability and high throughput requirements of using a blockchain. To add to this problem, the average block gas limit has changed drastically over time, ranging from 500,000 to 10,000,000. This means that the throughput of the system is still open to changes depending on how much the block gas limit changes over time. Therefore, in addition to giving careful consideration to which contract clauses are to be placed on-blockchain and which are to be placed off-blockchain within a hybrid solution, it is also important, as discussed in Section 3, to identify additional methods that can improve the overall performance of blockchain technologies.

### 8 CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

We have presented and evaluated the performance of a novel hybrid architecture for the implementation of a smart contract. The hybrid architecture consists of a centralized smart contract monitor known as the CCC, connected to the Ethereum blockchain. The performance of an asset
delivery application built in three different ways (on-blockchain, off-blockchain, and hybrid) is also measured and evaluated. Our results demonstrate that hybrid on-blockchain/off-blockchain based solutions may be very effective for implementing applications such as asset tracking. Using a blockchain allows users to trust that the information provided is accurate and removes the need for the user to trust the sender of the asset. The time related issues with using a blockchain can also be significantly improved as we have demonstrated by implementing hybrid smart contracts. However, when using public blockchains such as Ethereum as part of a solution, there is a problem with the maximum throughput of the system. Therefore, in addition to implementing hybrid architectures, it is important, as discussed in Section 3, to identify methods that can improve the performance of blockchain technologies such as Ethereum and Hyperledger. Another downside to using a public blockchain based system for asset tracking is that the user would have to pay a small fee to interact with the blockchain in order to use the service, which would be undesirable as the solutions which are currently used generally do not require the user to pay for them. However, as we have seen, a hybrid solution reduces the costs involved when compared to a blockchain based solution on its own. The costs could also be eliminated entirely if applications are implemented as part of hybrid solutions that involve blockchains with zero or no fees.

Future research directions include the integration of the hybrid architecture with other blockchains such as Hyperledger. This will enable the performance evaluation of hybrid architectures across different blockchains. In addition, we aim to develop a framework to investigate different centralized, hybrid, and blockchain architectures and execute performance benchmarks according to certain criteria in an automated and reproducible method. This way we can generalize the performance evaluation of smart contracts targeting a variety of use cases and architectures. In another direction, we are currently investigating the application of off-chain, on-chain, and hybrid smart contract solutions to complex problems such as cyber-physical systems and IoT. For example, we are in the process of developing capabilities that can automatically convert Service Level Agreements (SLAs) for IoT applications where performance monitoring is important, into Smart Contracts that can monitor adherence of interacting parties to the agreed SLAs. Toward this goal, in Reference 46, we discuss the development of a machine readable end-to-end IoT application.

ACKNOWLEDGEMENTS
This is an extended version of the paper: “On and Off-Blockchain Enforcement Of Smart Contracts.” We would like to thank our colleagues from the Computer Laboratory at the University of Cambridge, and from the Hat Community Foundation; Carlos Molina-Jimenez, Jon Crowcroft, and Irene Ng for their valuable contributions to this work.

ORCID
Ellis Solaiman https://orcid.org/0000-0002-8346-7962

REFERENCES
How to cite this article: Solaiman E, Wike T, Sfyrakis I. Implementation and evaluation of smart contracts using a hybrid on- and off-blockchain architecture. Concurrency Computat Pract Exp. 2020;e5811. https://doi.org/10.1002/cpe.5811