A Blockchain-based Approach for Assessing Compliance with SLA-guaranteed IoT Services

Ali Alzubaidi*†, Karan Mitra‡, Pankesh Patel§ and Ellis Solaiman¶
*Newcastle University, School of Computing, UK, Email: A.A.K.Alzubaidi2@newcastle.ac.uk
†Umm Al-Qura University, Saudi Arabia, Email: aakzubaidi@IEEE.org
‡Luleå University of Technology, Sweden, Email: karan.mitra@ltu.se
§Pandit Deendayal Petroleum University, India, Email: pankesh.patel@sot.pdpu.ac.in
¶Newcastle University, School of Computing, UK, ellis.solaiman@newcastle.ac.uk

***Abstract*—Within cloud-based internet of things (IoT) applications, typically cloud providers employ Service Level Agreements (SLAs) to ensure the quality of their provisioned services. Similar to any other contractual method, an SLA is not immune to breaches. Ideally, an SLA stipulates consequences (e.g. penalties) imposed on cloud providers when they fail to conform to SLA terms. The current practice assumes trust in service providers to acknowledge SLA breach incidents and executing associated consequences. Recently, the Blockchain paradigm has introduced compelling capabilities that may enable us to address SLA enforcement more elegantly. This paper proposes and implements a blockchain-based approach for assessing SLA compliance and enforcing consequences. It employs a diagnostic accuracy method for validating the dependability of the proposed solution. The paper also benchmarks Hyperledger Fabric to investigate its feasibility as an underlying blockchain infrastructure concerning latency and transaction success/fail rates.**

***Index Terms*—Blockchain, Smart Contract, SLA, Cloud, Monitoring, IoT.**

I. Introduction

A Service Level Agreement (SLA) is a contracting method, commonly employed by cloud providers to ensure the quality of an X-as-a-Service (i.e. IaaS, PaaS, SaaS, etc.) [1]. It specifies rights and obligations of involved participants with relation to agreed Service Level Objectives (SLOs). An SLA declares to which level a Quality of Service (QoS) must be maintained, and what associated consequences (e.g. penalty, remedy, etc.) applied in case of SLA violation [2] [3].

In current practice of incident management, cloud providers promise to process incidents in good faith, assuring their consumers to impose SLA consequences on themselves. While intriguing, it is typically the consumer's responsibility to report a service level degradation, supported by evidence deemed irrefutable by the service provider. This is usually a tedious process and manually handled [4]. Most related enforcement studies, such as in [1] and [5], assume trust in either service providers or trusted third parties. However, it can be inviting for some consumers to manipulate evidence to support violation incidents. On the other hand, providers may not react well to poorly formed claims, regardless of their validity [6]. In some scenarios, unresolved disputes have to be escalated to mediators or other jurisdiction means [7].

Recently, the rise of Blockchain technology invites revisiting existing solutions, where trust is taken for granted; Incident management is of no exception. The concept of smart contracts has enabled blockchain-based decentralised applications to serve various problem domains. Thus, we see an opportunity in exploiting Blockchain features to enable non-repudiable enforcement of SLA consequences.

We can list a set of advantages of shifting SLA enforcement to blockchain environment as follows:

- Improving incident management's procedures. Reported incidents can be automatically raised and instantly handled in a non-repudiable manner.
- SLA terms are expressed in a logical format in the form of a smart contract.
- Records' immutability can help minimise dispute cases as well as the need for escalation.
- Service providers can reduce the workforce size allocated for handling such tedious tasks.

A. Contribution

The main contributions of this paper are:

1) The paper proposes a decentralised approach for enforcing consequences of SLA violations. It models and implements a smart contract logic that addresses incidents processing, and automates decision-making on the compliance level of obligated providers with their offered SLA.
2) It employs a validation method, known as Accuracy Diagnostic [8], for examining the dependability of the modelled smart contract in every development iteration.
3) The paper evaluates the feasibility of Hyperledger Fabric in terms of both the latency and transactions Success/fail rate, and provides a set of observations.

The source code of our implementation is publicly available under GNU GPL V3.0 License on GitHub[1]. We hope it forms a base that can help other interested researchers and industry alike to advance blockchain-based SLA management.

Outline. The rest of the paper is organised as follows: First, section II elaborates on the research context, by setting an IoT scenario as a motivating example, and making use of a real SLA. It also briefly overviews Hyperledger Fabric, that we

[1]https://github.com/aakzubaidi/MQTT-SLA-Blockchain-QoS-Enforcement

adopt as underlying blockchain infrastructure. In section III, we overview the proposed approach, and delve into modelling the proposed logic of enforcing SLA consequences. Section IV presents a simulation experiment on the feasibility of the approach. It also presents how we validate the smart contract for every development iteration. Finally, section V evaluates the performance of Hyperledger Fabric, aiming to assess its viability as an underlying blockchain infrastructure.

II. Research Context

The IoT World Forum sets an Internet of Things (IoT) reference model, where components are organised in a multi-layered architecture [9]. It suggests a point where communication, processing, and storage are being centrally managed. Cloud services have been widely employed to serve such centric tasks in most IoT conceptualised architectures [10]. In practice, cloud providers, such as Google IoT Platform[2], provide a set of pay-as-you-go services for registering devices, handling exchanged data through HTTP servers or MQTT brokers, processing data, providing analytical tools, and big-data storage means. Consumers (i.e. healthcare providers) may find it appealing to outsource such tasks to cloud providers since it alleviates several IT burdens, including scalability and maintenance. In return, cloud providers ensure a satisfactory service delivery through the concept of SLA.

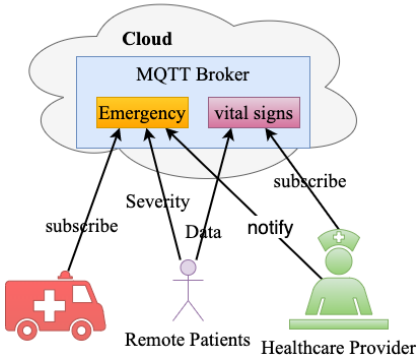

Fig. 1. Example IoT healthcare scenario employing MQTT for data exchange

A. Motivating Scenario

Message Queuing Telemetry Transport (MQTT)[3] is an asynchronous data exchange protocol based on publish/subscribe model. Since HTTP is document-oriented, it is less favourable when it comes to constraint resources. Therefore, MQTT, being data-oriented, is a popular data exchange protocol for IoT applications purposes. The mechanism of MQTT relies on the concept of topics, to which authorised entities can publish or subscribe. MQTT brokers orchestrate the data exchanged between publishers and subscribers. Because of the popularity of MQTT, several cloud providers offer MQTT brokers as a service.

[2]https://cloud.google.com/solutions/iot

[3]http://mqtt.org/

To elaborate, consider a Telemedicine example where a healthcare provider hires cloud services. These services depend on a MQTT broker, which is used for communication and data exchange. For example, Figure 1 depicts an IoT healthcare application that enables monitoring patients remotely and reacting in case of emergency. In this simplified scenario, three major entities communicate over the MQTT protocol, which are patients, the healthcare provider, and an ambulance department. These entities publish/subscribe to topics of their interest. For instance, health data can be collected by sensors/devices, and then published to a topic called *vital signs*. If a patient's condition becomes severe, both the healthcare provider and the ambulance department will be notified through the Emergency topic.

B. SLA Example

Cloud providers cope with IoT requirements by offering enabling services such as MQTT servers. For IoT scenarios such as the one presented in section II-A, cloud providers guarantee the quality of MQTT servers using the concept of SLA. For this paper, we use the GCP (Google Cloud Platform) SLA [4] for modelling our approach. The SLA covers a component for bridging between cloud services and IoT devices using MQTT. The GCP SLA ensures compensating consumers when a set of valid MQTT requests results in accidental device connections. The SLA holds the cloud provider accountable for consequences when the error rate exceeds 10%. For simplicity, we consider the error rate as the total accidental device disconnections divided by the total number of valid MQTT requests. For clarity, we formally define the error rate as in Equation 1, where F stands for MQTT Fail Requests and V stands for MQTT Valid Requests.

$$ErrorRate = \left(\frac{\sum_{i=1}^{n} F}{\sum_{i=1}^{n} V}\right) \times 100 \quad (1)$$

Within this paper we consider the GCP SLA as a representative example, and base our approach on it for assessing provider compliance, and for enforcing the violation consequences (when the error rate exceeds the 10% threshold).

C. Service Monitoring

Service monitoring is vital for enforcing satisfactory SLA compliance [7]. Examples of monitoring/reporting tools are those surveyed in [5] [11], which can be classified into proactive or reactive categories. The former predicts potential failures while the latter identifies incidents that are already in place. Ideally, cloud providers would apply proactive measures to prevent violation in the first place. While proactive violation methods present an exciting class of problems, we still need to discuss the aftermath of SLA violations [11]. This paper is limited to reactive monitoring methods, which we employ for incidents identification.

[4]https://cloud.google.com/iot/sla

D. Hyperledger Fabric as Enabling Technology

SLA as any other contracting method, is prone to trust and enforcement issues. Whenever trust is a burden, Blockchain present itself as a potential underlying infrastructure. Blockchain holds appealing principles including, but not limited to, immutability, decentralised computation, and a distributed shared ledger controlled by a consensus mechanism. The concept of a smart contract allows applications to benefit from these features, forming what is commonly referred to as Decentralised Applications (DAPPs) [12]. We see an opportunity in taking advantage of Blockchain to serve SLA monitoring and enforcement purposes. This does not only address trust issues [6] but also expedites and automates associated processes such as handling violations and decision making. It also helps reduce the need for manual processing and human intervention.

We select Hyperledger Fabric (hereafter HLF) [13] for realising the proposed blockchain-based SLA enforcement. HLF is classified as a consortium blockchain, composed of per-identified validating nodes. The permissioned nature of HLF enables adopting plugable consensus protocols lighter than Proof-of-work (PoW). The latest official version of HLF (v1.4.6 as of today) officially supports various Crash-Tolerant protocols as consensus mechanisms such as Raft [14] and Kafka/ZooKeeper. HLF supports general-purpose programming languages for representing the business logic as a smart contract. Supported languages include Golang, Javascript, and Java. Since all nodes are authenticated and committed for execution and validation tasks, there is no need for any execution fee, as it would be the case with public blockchain technologies. All transactions, whether successful or not, are immutably stored on the ledger for auditing purposes. Further details on the transaction flow can be found in [13].

III. Blockchain-based Consequences Enforcement

This section provides an overview on our approach, and then models a smart contract logic that enforces SLA consequences as per the GCP SLA example presented in section II-B.

A. Overview

As any contractual method, SLA is susceptible to breaches. In the current practice, SLA violations must be acknowledged by obligated providers in order to execute SLA consequences. Figure 2, suggests revisiting the current trust model, by shifting some such critical tasks from obligated providers to executable contracts operating in a non-repudiable fashion. In particular, We shift three main incident management tasks from obligated providers, which are Incidents processing, SLA compliance assessment, and SLA consequences enforcement. Blockchain-based Smart contracts can realise this aim beyond the influence of any single entity.

We highlight essential considerations for realising our approach as follows:

- The SLA document in place: We study the SLA content, and extract key elements which include but not limited to:
  - Participants: their rights and obligations.
  - QoS metrics: their definition, measurement, and violation consequences.
- Executable actions in the SLA must be represented in the smart contract, for example compliance assessment and consequences enforcement. The smart contract is then deployed to the blockchain network.
- SLA awareness within Blockchain: smart contracts need to have the necessary SLA awareness in order to execute relevant actions.
- Monitoring agent awareness: we have to consider the fact that smart contracts are supposed to be terminable and deterministic [15]. Smart contracts are not optimal for conducting endless activities such as monitoring. Thus, external monitoring/reporting means have to be in place to help smart contracts in forming a decision on the compliance level of obligated providers. For illustration, consider the GCP SLA definition of MQTT error rate. Monitoring agents should collect metrics related to the performance of the MQTT server such as up-time, the number of received/sent messages. These metrics help the smart contract to form a decision on the compliance status of the obligated provider. That is, monitoring agents must submit any identified incidents to the smart contract.

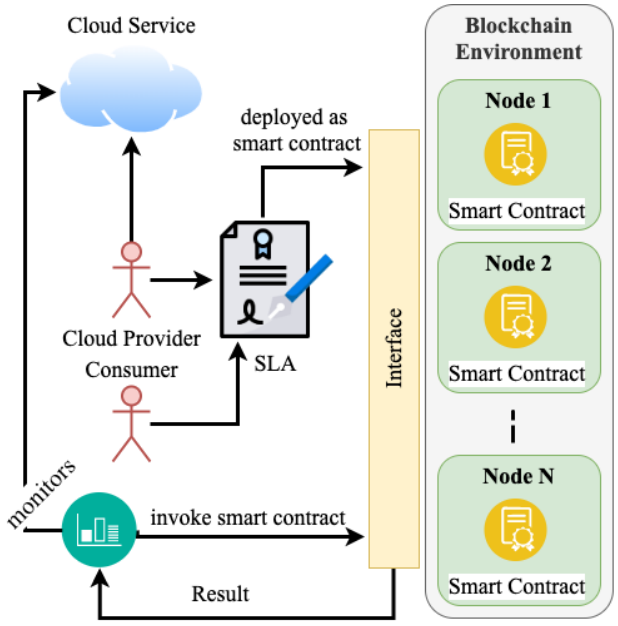

Fig. 2. Overview on the proposed Blockchain-based SLA enforcement

The rest of the paper delves into exploiting the smart contract for processing monitoring logs, and assessing the conformance of cloud providers to their offered SLAs.

B. Smart Contract Logic Modelling

We use the concept of smart contracts to automate decision-making on the compliance level of obligated providers (Cloud provider), beyond the control of centralised authorities. The decision making is formed with the aid of monitoring tools.

Figure 3 illustrates the proposed enforcement logic modelled as a smart contract. Two stages take place in every billing cycle (every month, for example). In the first stage, the smart contract waits for failure events, as per defined in the SLA. For that, there is an interface for authorised monitoring agents to invoke whenever there is an incident, as shown in Figure 2. For every transaction, the smart contract accommodates incidents by updating the total number of two elements, which are *Valid* and *Fail* MQTT requests. These two elements are essential for calculating the error rate. It is worth mentioning that update operations are executed at the state database provided by HLF, and not in the the immutable ledger. However, every update operation is backed-up with an immutably stored transaction for auditing purposes.

The second stage takes place at the end of the billing cycle. At this stage, the smart contract decides on the compliance status of the obligated provider. The decision is mainly based on *Error Rate*, calculated as per Equation 1. Then, it examines the calculated error rate against the threshold stipulated in the SLA in place; say 10% for example. If the smart contract concludes that the obligated provider has fulfilled its promise by not reaching the stipulated threshold, then the performance will be marked as *compliant*. Otherwise, it marks the performance as *Violation* and applies the agreed penalty. Either way, the result will be immutably recorded on the ledger for reference purposes. The count of both *Valid Requests* and *Failure Requests* are then reset for the next billing cycle.

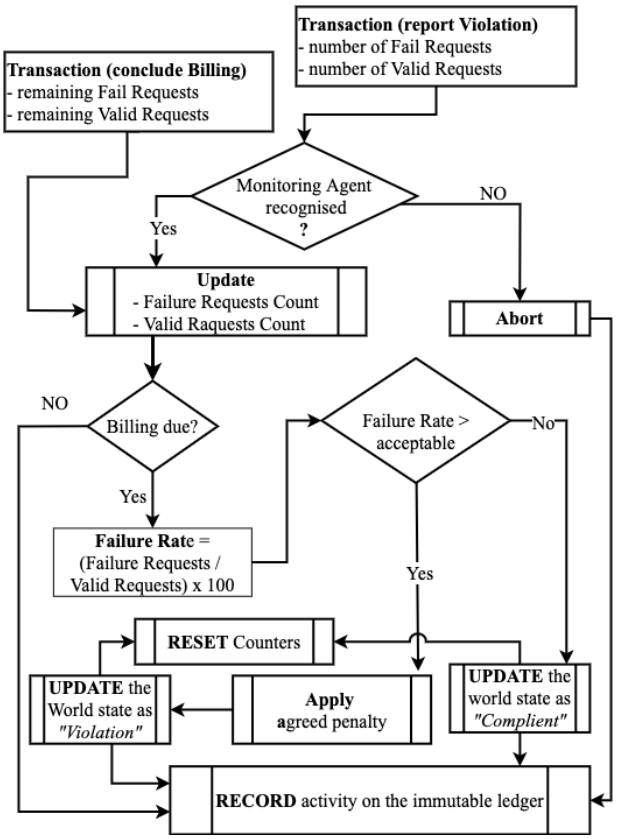

Fig. 3. Modelling the enforcement logic as a smart contract

IV. Simulation and Validation

This section attests the feasibility of implementing our approach as a smart contract. It also describes how we validate the dependability of the proposed approach. The smart contract, simulation, and the validation process are implemented in Java programming language and available as open source in GitHub[5].

A. Simulation and Implementation

The purpose of the simulation is to experiment with the feasibility of the proposed approach. Figure 4 illustrates the outlook of the implemented components. First, we employ an MQTT broker using *Eclipse Mosquitto*[6]. Second, we implement a simple behaviour of a monitoring agent using *Eclipse Paho*[7]. Third, the implemented smart contract, discussed in section III-B, is then deployed and instantiated using IBM Blockchain Platform [8]. We also developed a bridge using Fabric Java SDK[9], which represents the interface between the blockchain network and external entities (e.g. monitoring agents) to enable configuring all necessary elements (such as identity, access level, crypto materials, etc.).

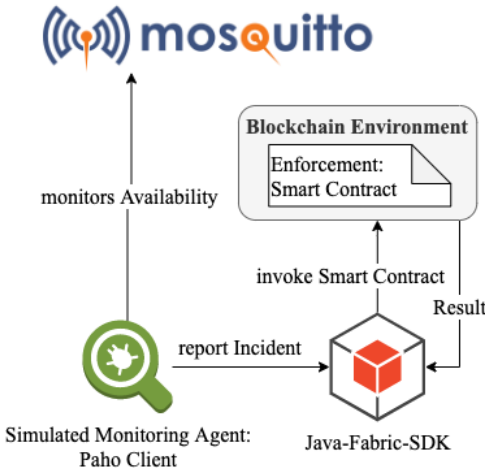

Fig. 4. Simulation of MQTT Broker and Monitoring agent

The logic of our simulation and testing is as follows. The monitoring agent continuously publishes testing samples to the MQTT Broker (Mosquitto). The goal of the monitoring agent is to observe Valid and Fail MQTT requests. The monitoring agent uses exception handlers and error codes provided by *Eclipse Paho* to identify errors and reason about them. We deliberately cause the MQTT broker downtime to simulate cloud failure by simply shutting down the MQTT broker (Mosquitto). Once the monitoring agent detects an availability issue (downtime), it submits a transaction to the smart contract, reporting latest *Valid* and *Fail* MQTT request. The smart

[5]https://github.com/aakzubaidi/MQTT-SLA-Blockchain-QoS-Enforcement
[6]https://mosquitto.org/
[7]https://www.eclipse.org/paho/
[8]https://marketplace.visualstudio.com/items?itemName=IBMBlockchain.ibm-blockchain-platform
[9]https://github.com/hyperledger/fabric-sdk-java

contract records incidents as well as valid MQTT requests, and will keep accumulating incidents until the error rate reaches 10%. Then we test the the smart contract's ability to mark the cloud provider's performance as *violation*. Overall, by conducting the above, our proposed approach proves to work as expected. Following, we describe our validation approach.

B. Decision Accuracy Validation

To ensure the reliability of our approach we consider two facts. First, the enforcement logic is an assessment tool in nature, which decides on the compliance level of obligated providers. Therefore, the dependability of the decision logic has to be validated [16]. Second, smart contracts operate beyond the control of a single authority. Therefore, rectifying a logical error in decentralised applications is not as straightforward as it would be in traditional applications [17]. Thus, this section presents how we validate the dependability of the enforcement logic for every development iteration on the smart contract. It examines the two stages of the enforcement life-cycle, which are the incident reporting as in Algorithm 1 and compliance assessment as in Algorithm 2. For every development iteration on the smart contract, we execute these validation algorithms using Junit testing framework.

*1) **Stage 1: Incidents Reporting**:* The decision logic is based on Equation 1, which is composed of two elements: *MQTT Fail Requests* and *MQTT Valid Requests*. Algorithm 1 investigates the smart contract ability to update the values of these elements correctly. It simulates a monitoring agent that repeatedly reports incidents to the smart contract. For each incident, the simulated monitoring agent submits MQTT Fail Requests F, and MQTT Valid Requests V. The smart contract will record this incident and update values. The algorithm queries the current count of both *Valid* and *Fail MQTT requests*, V_count and F_count, and assert that they are appropriately updated.

Algorithm 1 Validating correct values update

Input: F, V
Output: F_count and V_count are correctly updated
1: **for** $i \leftarrow 1$ to $Transactions\_count$ **do**
2: **Invoke** Report_Violation (F, V)
3: **Wait** for transaction resolution
4: **Query** (F_count , V_count)
5: **if** F_count OR V_count NOT correctly updated **then**
6: **Terminate** with error
7: **end if**
8: **end for**
9: **return** V_count , F_count

We always assume the worst-case scenario, where the MQTT broker exhibits malfunction behaviour lasting for the entire 30 days. We adopt the default reporting interval employed by Google Stackdriver monitoring tool [10], which is every 60 seconds. Thus, the monitoring agent will keep triggering the smart contract every minute. This means the smart contract should receive 43200 transactions by the end of the month. Therefore, we iterate for 43200 times, such that there are new Valid V and Fail F MQTT requests for every iteration. Thus, by the end of algorithm execution, V_count should equals $V \times 43200$ and F_count should equals $F \times 43200$.

*2) **Stage 2: Compliance Assessment**:* At the end of every billing cycle (assume 30 days), the smart contract assesses the compliance level of obligated providers based on the agreed *Error Rate Threshold*, For example 10%. It calculates the Error rate as per Equation 1, and compares it with the agreed *Error Rate Threshold* in the SLA. To examine decision accuracy we prepared a set of cases already known to us to be either *Violation* or *Compliant*. The Violation group has 50% of these cases, which exceed the 10% Error Rate Threshold. The Compliant group has the other half of the cases, which does not exceed the 10% Error Rate Threshold. Given the importance of this assessment task we employ rigorous testing that validates the decision made by the smart contact. We rely on a method called *Diagnostic Accuracy* [8], which employs a set of measures and calculations based on 2X2 table, as shown in Table I. The table is composed of 4 elements which are True Positive (TP), True Negative (TN) False Positive (FP), False Negative (FN). Let us assume an SLA that stipulates a service delivery with an Error Rate that does not exceeds 10%; Otherwise, it is considered a violation case. Then, we consider *TP* is the count of the cases that is correctly classified to be breaching this terms (the 10% threshold). Therefore, *TN* is the count of the cases that are correctly classified to be compliant and does not exceed the 10% threshold. *FP* and *FN* are the cases where the smart contract incorrectly identify cases as Violation or Compliant; respectively.

TABLE I
DIAGNOSTIC ACCURACY 2X2 TABLE

	Violation	Complaint
Positive	True Positive (TP)	False Positive (FP)
Negative	False Negative (FN)	True Negative (TN)

Based on the Diagnostic Accuracy table, we select and define the following measurements:

- *Sensitivity:* the proportion of violation cases correctly classified by the smart contract; calculated as follows:

$$Sensitivity = \frac{TP}{TP + FN} \tag{2}$$

- *Specificity:* the proportion of compliant cases correctly classified by the smart contract; calculated as follows:

$$Specificity = \frac{TN}{TN + FP} \tag{3}$$

- *Positive Predictive Value (PPV):* The probability of accurate decision on violation cases; calculated as follows:

[10]https://cloud.google.com/monitoring/api/metrics_gcp

$$PPV = (\frac{TP}{TP + FP}) \times 100 \quad (4)$$

- *Negative Predictive Value (NPV):* The probability of accurate decision on compliant cases; calculated as follows:

$$NPV = (\frac{TN}{TN + FN}) \times 100 \quad (5)$$

The goal is to examine the smart contract ability to correctly classify the prepared cases into their corresponding groups; either *Violation* or *Compliance*. Algorithm 2 illustrates the conduct of the Diagnostic Accuracy method. First, We feed all prepared sets of cases into the smart contract. We fill in table I according to the outcome of the smart contract. That is, for each case we classify the result of the smart contract decision on every case to be one of the listed categories (TP, TN, FP, or FN). We conduct the calculation of *Sensitivity*, *Specificity*, *PPV* and *NPV* to find out about accuracy of the compliance assessment. We implemented this validation method (Diagnostic Accuracy) using Junit testing framework, and executed it for every development iteration on the smart contract. Results are shown in table II.

TABLE II
OPTIMUM RESULTS OF THE DIAGNOSTIC ACCURACY METHOD

Sensitivity	Specificity	PPV	NPV
100%	100%	100%	100%

Algorithm 2 Conducting The Diagnostic Accuracy Method

Input: $TEST\_CASES$
Output: $Sensitivity$, $Specificity$, PPV , NPV

```
C = {c : c ≤ threshold} // Compliant cases
V = {v : v > threshold} // Violation cases
TEST_CASES = {C ∪ V}
for each tc ∈ TEST_CASES do
    assessCompliance (tc) //invoke the smart contract
    determine whether decision is TP, TN, FP or FN
    if TP then
        TP_count++
    else
        if TN then
            TN_count++
        end if
    else
        if FT then
            FT_count++
        end if
    else
        if FN then
            FN_count++
        end if
    end if
end for
Calculate Sensitivity , Specificity , PPV , NPV
```

V. PERFORMANCE EVALUATION AND RESULTS

HLF has proven to perform well for several scenarios. There exists a set of studies on HLF performance such as those in [13] [18] [19] [20] which are based on Kafka/Zookeeper protocol and [21] which is based on PBFT. To the best of our knowledge, there is no performance benchmarking coping with the latest development on HLF, where Raft protocol [14] has been officially recommended as a consensus protocol. In this paper, we adopt Raft, and thus we focus on the latest official HLF version 1.4.6.

The previous section looked into the smart contract logic dependability. However, we still need to confirm the feasibility of HLF as an underlying Blockchain technology. Therefore, this section investigates how HLF network copes with consecutive incidents (assuming a very poorly performing cloud provider). In particular, we look into HLF's latency and transaction success/fail rate.

A. Experimental Setup

We focus on the most demanding functionality in our modelled smart contract, which is processing the received incidents, as described in Algorithm 1. At the infrastructure level, Table III illustrates both, the testing environment and HLF configuration.

TABLE III
HYPERLEDGER FABRIC (HLF) NETWORK DEPLOYMENT CONFIGURATION

Factor	Settings
Host Machine	Local running MacOS Catalina OS, 2.9GHz Dual-core intel Core i5 CPU, 2GB LPDDDR3 Memory Ram.
Containerization	Docker version 2.2, Engine version 19.3.5. Allocated Resources: CPUs: 3, Memory: 7GB, Swap: 3GB
Network Topology	HLF Version 1.4.6, 2 Organisations, 4 committing peers, 5 orderers, each one in a separate container. One dedicated channel, LevelDB is used as a ledger database.
Consensus Protocol	Raft Protocol
Endorsement Policy	One peer of each organisation
Chaincode Settings	
Execution Timeout	60 seconds
Programming Language	Java
Logging	Enabled

We employ a modular blockchain benchmarking project under Hyperledger Umbrella, called *Hyperledger Caliper* [11] for deploying the smart contract, simulating the behaviour of monitoring agents, and benchmarking.

Table IV summarises the experimental settings of 4 different test cases. They are all similar in terms of benchmarking settings, where we denote similarity as $\sim$ to indicate no change

[11] https://hyperledger.github.io/caliper/v0.3/getting-started/

in the settings. However, these test cases are different in terms block batching configurations. For this experiment, we assume consecutive transactions. Therefore, we aim to figure out a suitable block batching configuration that avoids concurrency issues while maintaining a reasonable throughput and latency. We employed two factors for defining the readiness of a block for validation, which are: *timeout*, and maximum allowed number of *transactions per block*; whichever occurs first. These two factors vary in every testing, seeking the best performance possible.

TABLE IV
EXPERIMENTAL SETTINGS (HYPERLEDGER CALIPER)

Facet	Test 1	Test 2	Test 3	Test 4
Function under test	Report Violation (Asset update)	∼	∼	∼
total workers (client thread)	1 worker	∼	∼	∼
control rate: Fixed Rate:	300 transactions	∼	∼	∼
Sent Transactions (per second)	1 Tps	∼	∼	∼
Execution Timeout	30 seconds for: - Chaincode engine - Worker - Caliper runtime	∼	∼	∼
Fabric Block Batching Configuration				
Block Batching Timeout (milliseconds)	1000	500	1000	500
Transactions per block	10	10	1	1

Our testing plan is that, we set all parameters in Table IV to the minimal values possible, and then we scale gradually until we reach a bottleneck. For example, we started from 1 worker sending only one transaction. However, we encountered conflicting transactions for all different test cases as reported and discussed below. So we limited the number of workers to 1 for all four test cases and be more focused on Block batching configuration. We started with 1 second timeout and 10 transactions per seconds in test case 1. Then, we went to see how HLF would perform with lower values in the rest of test cases.

B. Results and Observations

1) ***Latency****:* All four test cases in Table II reveal that HLF latency, measured in seconds, is generally acceptable given the limited resources of the host machine. We observe no major difference in average Latency, which exhibits a relatively similar behaviour. We can see from case 4 that, these is a clear fluctuation between minimum and maximum latency. We can attribute this to the high rate of commits on the ledger across the network [22], resulted by generating a block every half second milliseconds. This leads many transactions to miss the chance of being included in the current block, and thus wait for the next one.

TABLE V
RESULTS OF LATENCY AND TRANSACTIONS SUCCESS RATE

	T1	T2	T3	T4
Success	298	297	299	295
Fail	2	3	1	5
Max Latency (s)	1.22	1.54	1.43	1.68
Avg Latency (s)	0.77	0.77	0.69	0.78
Min Latency (s)	0.68	0.68	0.77	0.34

2) ***Success/Fail Rates****:* We have to recognise the fact that HLF employs an optimistic locking mechanism, namely: Multi-Version Concurrency Control (MVCC) to prevent a known blockchain problem called double-spending problem [23]. This mechanism caused all test cases to experience transactions failures due to conflict in MVCC Read-Write sets. The MVCC conflict is triggered because HLF experienced consecutive transaction reporting incidents to the smart contract. The evaluation results reveal that, test case 3, (1 Tps and 1s timeout) experiences the least transactions failures. The worst performance is observed in test case 4, where block batching is set to the minimum possible (1 Tps and 500ms timeout).

C. Observation and Remarks

While MVCC has proven to work well for some scenarios such as money transfer, it is not the case for demanding applications. For example, it is abnormal for a money transfer application to receive frequent updates on the same account within a few seconds. However, In our case, we ideally expect consecutive transactions from monitoring agents. Nevertheless, the lock mechanism causes some monitoring transactions to experience MVCC conflicting Read-Write sets, represented as (key, value version). When a transaction tries to update a key, it acquires the latest version of that key. Since our scenario assumes consecutive monitoring transactions, failure can accrue when transactions carry out update operations based on an obsolete key version. This situation can happen when an unsettled update transaction (not committed yet to the ledger) finally manages to be committed, causing a change of the current value and version. Therefore, any transaction based on an obsolete version is to be invalidated regardless of how correct they are [24].

There have been several workaround solutions to address this matter. For example, Composition-key can bypass MVCC validation because there is a new key generated for every transaction. However, this contradicts with the purpose of MVCC and introduces cumbersome key management. A retry mechanism is another workaround, such that client applications are notified of such conflicts and then retry submitting again. While applicable for some scenarios, this may not be suitable for others. There are also other techniques such as queuing transactions before submitting them to the blockchain.

However, this does not benefit from the high throughput promised by HLF.

To sum up, in all test cases presented in table IV, the MVCC conflict was present, which is an unpleasant issue that has to be addressed. We do not expect such an issue to appear when a few incidents are occurring less frequently. However, in certain extreme scenarios, we expect HLF to exhibit malfunctions due to MVCC conflicts. For example, when a very poorly performing cloud provider causes monitoring agents to report incidents every 1 second. Even when we attempt to increase the frequency of block batching, there is value in addressing and mitigating this malfunction [22] [25].

VI. Conclusion and Future work

This paper presents a blockchain-based SLA enforcement approach that aims to automate SLA incident management. It focuses on processing SLA violations and decision-making on the compliance status of cloud providers obligated by an SLA. It validates the dependability of the proposed enforcement logic based on accuracy diagnostic method. This paper also investigates HLF as underlying technology using Hyperledger Caliper. The evaluation showed that HLF performs well in terms of latency and possibly throughput. HLF can potentially serve the purpose of our proposed enforcement logic. However, since our approach expects high throughput from monitoring agents, the issue of HLF MVCC conflict has to be addressed. Otherwise, we cannot safely assume dependability. In future work, we will investigate a viable enhancement on HLF, or at least adapting our proposed approach in a way that does not compromise the performance of HLF.

References

[1] F. Faniyi and R. Bahsoon, "A Systematic Review of Service Level Management in the Cloud," *ACM Comput. Surv.*, vol. 48, no. 3, pp. 1–27, dec 2015. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2856149.2843890

[2] A. Alqahtani, E. Solaiman, P. Patel, S. Dustdar, and R. Ranjan, "Service level agreement specification for end-to-end IoT application ecosystems," *Softw. Pract. Exp.*, vol. 49, no. 12, pp. 1689–1711, dec 2019. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2747

[3] A. Alqahtani, Y. Li, P. Patel, E. Solaiman, and R. Ranjan, "End-To-End Service Level Agreement Specification for IoT Applications," in *Proc. - 2018 Int. Conf. High Perform. Comput. Simulation, HPCS 2018*. IEEE, oct 2018, pp. 926–935.

[4] A. Alzubaidi, E. Solaiman, P. Patel, and K. Mitra, "Blockchain-based SLA Management in the Context of IoT," *IT Prof.*, 2019.

[5] S. Mubeen, S. A. Asadollah, A. V. Papadopoulos, M. Ashjaei, H. Pei-Breivold, and M. Behnam, "Management of Service Level Agreements for Cloud Services in IoT: A Systematic Mapping Study," *IEEE Access*, vol. 6, pp. 30 184–30 207, 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8016558/

[6] E. J. Scheid, B. B. Rodrigues, L. Z. Granville, and B. Stiller, "Enabling Dynamic SLA Compensation Using Blockchain-based Smart Contracts," in *2019 IFIP/IEEE Symp. Integr. Netw. Serv. Manag.*, 2019, pp. 53–61.

[7] O. F. Rana, M. Warnier, T. B. Quillinan, F. Brazier, and D. Cojocarasu, "Managing Violations in Service Level Agreements," in *Grid Middlew. Serv.* Boston, MA: Springer US, 2008, pp. 349–358. [Online]. Available: http://link.springer.com/10.1007/978-0-387-78446-5_23

[8] A.-M. Šimundić, "Measures of Diagnostic Accuracy: Basic Definitions." *EJIFCC*, vol. 19, no. 4, pp. 203–11, jan 2009. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/27683318 http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC4975285

[9] W. Stallings, *Foundations of modern networking: SDN, NFV, QoE, IoT, and Cloud*. Addison-Wesley Professional, 2016.

[10] G. White, V. Nallur, and S. Clarke, "Quality of service approaches in IoT: A systematic mapping," *J. Syst. Softw.*, vol. 132, pp. 186–203, oct 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016412121730105X

[11] W. Hussain, F. K. Hussain, and O. K. Hussain, "Maintaining Trust in Cloud Computing through SLA Monitoring." Springer, Cham, 2014, pp. 690–697. [Online]. Available: http://link.springer.com/10.1007/978-3-319-12643-2_83

[12] W. Cai, Z. Wang, J. B. Ernst, Z. Hong, C. Feng, and V. C. Leung, "Decentralized Applications: The Blockchain-Empowered Software System," *IEEE Access*, vol. 6, pp. 53 019–53 033, sep 2018.

[13] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains," jan 2018. [Online]. Available: http://arxiv.org/abs/1801.10228 http://dx.doi.org/10.1145/3190508.3190538

[14] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 {USENIX} Annu. Tech. Conf. ({USENIX}{ATC} 14)*, 2014, pp. 305–319.

[15] S. Wang, L. Ouyang, Y. Yuan, X. Ni, X. Han, and F.-Y. Wang, "Blockchain-Enabled Smart Contracts: Architecture, Applications, and Future Trends," *IEEE Trans. Syst. Man, Cybern. Syst.*, pp. 1–12, 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8643084/

[16] D. Magazzeni, P. Mcburney, and W. Nash, "Validation and verification of smart contracts: A research agenda," *Computer (Long. Beach. Calif).*, vol. 50, no. 9, pp. 50–57, 2017.

[17] K. Christidis and M. Devetsikiotis, "Blockchains and Smart Contracts for the Internet of Things," pp. 2292–2303, 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7467408/

[18] P. Thakkar, S. Nathan, and B. Vishwanathan, "Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform," *arXiv Prepr. arXiv1805.11390*, 2018.

[19] M. Kuzlu, M. Pipattanasomporn, L. Gurses, and S. Rahman, "Performance analysis of a hyperledger fabric blockchain framework: Throughput, latency and scalability," in *Proc. - 2019 2nd IEEE Int. Conf. Blockchain, Blockchain 2019*. Institute of Electrical and Electronics Engineers Inc., jul 2019, pp. 536–540.

[20] C. Gorenflo, S. Lee, L. Golab, and S. Keshav, "FastFabric: Scaling Hyperledger Fabric to 20,000 Transactions per Second," in *ICBC 2019 - IEEE Int. Conf. Blockchain Cryptocurrency*. Institute of Electrical and Electronics Engineers Inc., may 2019, pp. 455–463.

[21] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "BLOCKBENCH: A Framework for Analyzing Private Blockchains," in *Proc. 2017 ACM Int. Conf. Manag. Data - SIGMOD '17*. New York, New York, USA: ACM Press, 2017, pp. 1085–1100. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3035918.3064033

[22] H. Sukhwani, N. Wang, K. S. Trivedi, and A. Rindos, "Performance Modeling of Hyperledger Fabric (Permissioned Blockchain Network)," in *2018 IEEE 17th Int. Symp. Netw. Comput. Appl.* IEEE, nov 2018, pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/document/8548070/

[23] G. Chung, L. Desrosiers, M. Gupta, A. Sutton, K. Venkatadri, O. Wong, and G. Zugic, "Performance Tuning and Scaling Enterprise Blockchain Applications," dec 2019. [Online]. Available: http://arxiv.org/abs/1912.11456

[24] H. Meir, A. Barger, Y. Manevich, and Y. Tock, "Lockless Transaction Isolation in Hyperledger Fabric," in {*IEEE*} *Int. Conf. Blockchain, Blockchain 2019, Atlanta, GA, USA, July 14-17, 2019*. IEEE, 2019, pp. 59–66. [Online]. Available: https://doi.org/10.1109/Blockchain.2019.00017

[25] P. Yuan, K. Zheng, X. Xiong, K. Zhang, and L. Lei, "Performance modeling and analysis of a Hyperledger-based system using GSPN," *Comput. Commun.*, vol. 153, pp. 117–124, mar 2020.