Proposing a Tool to Monitor Smart Contract Execution in Integration Processes[*]

Mailson Teles-Borges[1], **Rafael Z. Frantz**[1], **José Bocanegra**[2], **Sandro Sawicki**[1], **Fabricia Roos-Frantz**[1]

[1]Unijuí University – Ijuí, RS – Brazil

mailson.borges@sou.unijui.edu.br

{rzfrantz, sawicki, frfrantz}@unijui.edu.br

[2]Universidad Distrital Francisco José de Caldas – Bogotá – Colombia

jjbocanegrag@udistrital.edu.co

Abstract. *Smart cities take advantage of digital services to enhance citizens' experiences. Integration processes facilitate interactions between these services, providing or improving functionalities. The integration can operate under specific restrictions, which can be represented through smart contracts deployed in a blockchain. Monitoring systems track interactions between integration processes and digital services by recording events from communication ports. In this paper, we argue that current monitoring tools lack the ability to observe these ports or invoke smart contracts. We propose a monitoring system to track integration processes, capture port-reported events, and invoke smart contracts on a blockchain platform.*

1. Introduction

According to recent research, it is estimated that by the year 2050, urban areas will be home to over two-thirds of the global population [Fabolude et al. 2025]. Urban migration, population growth, and the emergence of megacities (cities with over 10 million inhabitants) have boosted investment, research, and development in smart cities. Although there is no agreed definition [Serrano 2018], the concept of smart cities, which integrates physical infrastructure with social, environmental, and economic factors, is becoming widely accepted [Zou et al. 2019, Addas 2023]. Information and Communication Technology (ICT) plays a pivotal role in smart cities by providing software systems that deliver digital services to their citizens; however, these systems require data and functionality spread across the city's software ecosystem during their operation. The smart city ecosystem includes a diverse range of software solutions operating in both private and public sectors, developed with different technologies and data models.

Integration processes can be developed to enhance existing digital services by combining data and functionality from different systems. For instance, a shop could automatically pay a customer's parking ticket if they spend over a certain amount on their products. Alternatively, a municipality could book a taxi for an elderly person to attend an appointment at the hospital for medical examinations. Integration processes

[*]This research is partially funded by the Co-ordination for the Brazilian Improvement of Higher Education Personnel (CAPES) and the Brazilian National Council for Scientific and Technological Development (CNPq) under the following project grants 311011/2022-5, 309425/2023-9, 402915/2023-2, and by the Universidad Distrital Francisco José de Caldas under the grant PR2-2025.

are pieces of software that enable data and functionality sharing amongst digital services. These processes are developed and executed using integration platforms, which are specialised software tools designed to create, implement, test, and run integration processes [Rosa-Sequeira et al. 2018]. In general terms, an integration process consists of communication ports and an internal workflow of tasks. Ports are implementations of communication protocols (e.g., HTTP, FTP, SSH, JDBC, etc.) that abstract the interaction with digital services. Conversely, tasks execute atomic operations (e.g., filter, merge, split, transform, copy, aggregate, route, etc.) on the data input encapsulated in messages that flow through the integration process.

The interaction of communication ports with the integrated digital services must comply with certain restrictions, such as the maximum number of requests allowed per unit of time, the maximum permissible response time, or the maximum amount of data transmitted per request. These restrictions can be expressed through contract clauses and may be regulated by smart contracts [Dornelles et al. 2022]. Smart contracts are self-executing software programs that enforce contractual terms without human intervention and contain programmable logic that defines the rules and conditions of the agreement between the parties involved [Zou et al. 2019]. They are typically written in cross-domain languages, such as Solidity, or even in general-purpose languages, such as Go, and executed on a blockchain platform, where they are published and made immutable.

The execution of a smart contract regulating interactions between an integration process and a digital service requires data from each event reported by the port. Therefore, a monitoring system must not only observe and capture these reported events from an integration process but also invoke the corresponding smart contract to check for contract violations. Each event triggers the contract's execution. Unfortunately, existing monitoring tools found in the literature focus on monitoring the integration process internal tasks execution or monitoring hardware resources (e.g., memory consumption, disk usage, network consumption, etc.). The lack of tools capable of invoking smart contracts using event data from ports represents a research and technological challenge since it requires configuring many tools and manual coding.

This paper discusses our proposal for a monitoring system. This system observes integration processes, captures events reported by communication ports, and invokes the corresponding smart contract deployed on a blockchain. Currently, we are evaluating smart contracts written for the Ethereum and Hyperledger Fabric platforms, but it is possible to extend support to others. The system is event-based, and its architecture consists of internal components (i.e. queues and event handlers) that asynchronously process every event received. Application Programming Interfaces (API) enable the monitoring system to communicate and interact with the integrated digital services. This paper is organised as follows: Section 2 presents tools for monitoring and discusses relevant works in the field; Section 3 outlines the design and function of the monitoring system's components; Section 4 presents the current stage of the Monitoring System's development; and Section 5 concludes and outlines ongoing and future work related to our proposal.

2. Related Work

There are tools available for the integration context, but they lack support for smart contract execution or require complex configuration. Open-source solutions, such as those provided by Grafana Labs [Grafana 2025] or Elastic [Elastic 2025], either require instrumentation of the integrated digital services or the configuration of multiple specialised tools to work together. For instance, if Grafana technologies are used, it is necessary to configure Promtail, Loki, and Grafana Dashboard. Promtail is an agent for collecting logs in software applications, Loki is a storage system, and Grafana Dashboard is a user interface for visualising metrics. In contrast, cloud-based solutions (e.g., Datadog [Datadog 2025] or Dynatrace [Dynatrace 2025]) may significantly reduce the configuration process. However, they incur high costs, as their business model is based on a pay-as-you-go approach.

The works available in the current literature do not address monitoring in the smart contract context. Usually, they propose solutions either for cloud-based systems [Otero et al. 2024, Tundo et al. 2019], where monitoring involves tracking resource utilization, network traffic, application performance, or hardware monitoring [Bautista et al. 2022, Sukhija and Bautista 2019]. Another approach is to implement machine learning algorithms to enhance monitoring capabilities, such as predicting or adding adaptive identification of critical failures [Klymash et al. 2024], or even filtering alerts that pottencially may generate unecessary notifications [Taheri et al. 2024]. In conclusion, although there are some works addressing monitoring in the software development field, none address monitoring in the smart contract context domain.

3. Envisioned Solution

Traditionally, smart contracts enforce business rules within a business process. In our context, enforcement would turn the integration process port into a type of filter, evaluating each received event immediately and preventing the operation from being executed if any clause is breached. Conversely, in the monitoring approach, the execution of the smart contract and the integration process are independent. As a result, if any clauses are breached, the integration process has already been executed, necessitating a compensatory action. Although, at first glance, the enforcement approach may seem the most suitable option, it introduces challenges such as processing overhead and increased latency, as the operation must await its completion—an issue that does not arise in the monitoring approach. Our approach is based on monitoring, as it aligns more naturally with event-driven and asynchronous workflows, avoiding the strict synchronous dependencies imposed by enforcement mechanisms.

The monitoring system is accessed through APIs that facilitate the configuration of the system and enable external applications to access information regarding ongoing monitoring activities. The design of the proposed system is illustrated in Figure 1. Internally, the system consists of six primary components: the Inbound Events Queue, Event Handler, Smart Contract Execution Queue, Contract Invoker, Smart Contract Outbound Queue, and Event Updater. These components store, validate, and manipulate events received from the integration processes. The adopted design is event-based, meaning that component processing is triggered whenever events are present in the queues. This approach ensures that the events stored in the queues are processed according to the availability of computing resources. In our proposal, the system also includes four repositories: three for internal configurations and another for storing and querying received events.

3.1. API

The blockchain platforms (e.g., Ethereum, Hyperledger Fabric) can be configured using `API 1`, which the monitor will use to execute smart contracts registered on these platforms. Smart contracts associated with each integration platform and their metadata can be registered via `API 2`, enabling their subsequent activation on the corresponding blockchain platform. Activating a contract requires metadata, such as the contract's name, clauses, clauses arguments, and the blockchain on which the contract will be executed. The interface defined by `API 3` enables the recording of events reported by the integration processes. These events must contain: a header with data that allows for the authentication and authorisation of their processing; a body with a payload containing a dictionary that informs the `id` of the contract, the clauses registered in `APIs 1` and `2`, and the arguments for these clauses, as shown in Listing 1. The interface defined by `API 4` allows for the configuration of parameters necessary for the correct functioning of the monitor. Parameters include both base registration data (e.g., users, user groups) and operating parameters (e.g., permission and access levels, event retry policy). The interface defined by `API 5` will serve as the monitoring system's output port, allowing queries regarding the status and result of smart contract executions.

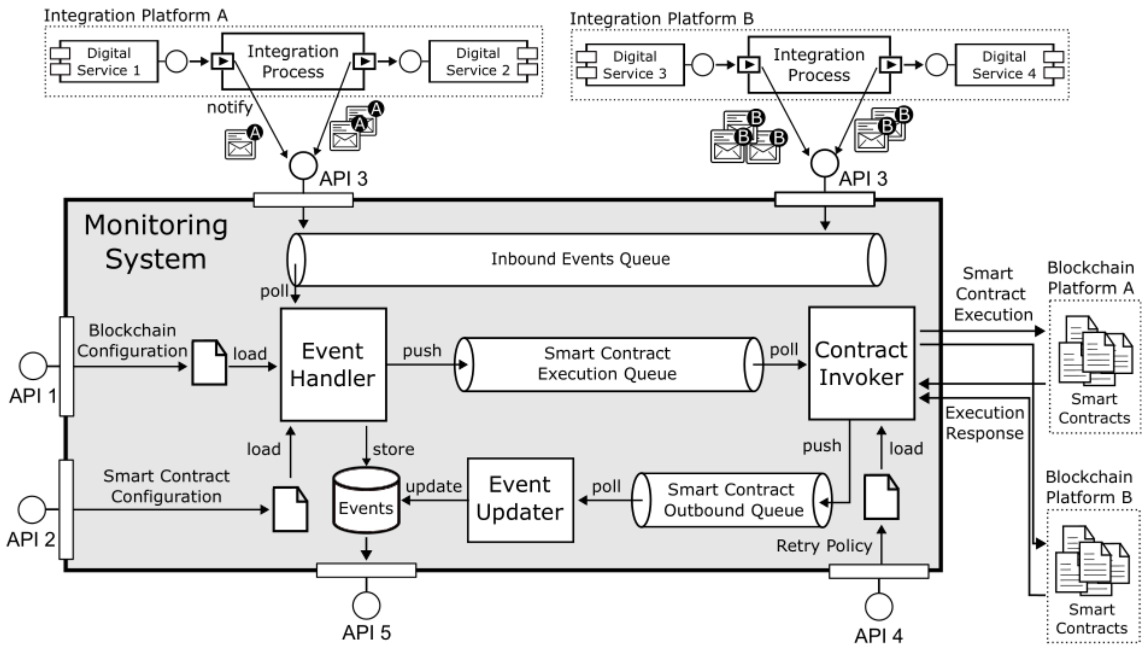

Figure 1. Monitoring System Architecture.

```
{ "contract id": { "clause id [1]": { "arg1": "value 1", "arg2": "value 2" }}
```

Listing 1. Inbound Event Model.

3.2. Event Handler

The Event Handler executes polling operations on the Inbound Events Queue, asynchronously reading each received event. Upon initiating event processing, this component consults the information configured through `APIs 1` and `2`. Initially, validation is performed to check whether the data reported in the event payload aligns with the configurations of the corresponding smart contract. If an irregularity is detected, an exception is generated and associated with the event `ID`. Exceptions are categorised into two types: the Structure Integrity Category and the Execution Category (see Table 1). The Event Handler only handles Structure Integrity exceptions, which relate to differences between the event payload and the data structure expected by the Monitoring System. The Contract Invoker handles Execution-type exceptions, which relate to issues encountered during smart contract execution. Following this, the event can proceed through two different channels. If an exception occurs, its processing is terminated, and it is directed to the event repositories. If no inconsistency is found, the event, along with the smart contract and blockchain configuration data, is redirected to the Smart Contract Execution Queue.

Category	**Exception**	**Description**
Structure Integrity	ContractNotFoundException	Contract ID not found.
	ClauseNotFoundException	Clause ID not found.
	InvalidClauseException	Clause belongs to another contract.
	InvalidArgumentClauseTypeException	Invalid argument type.
	MissingArgumentException	Required argument missing.
Execution	BlockchainConnectionException	Blockchain connection issue.
	ContractClauseViolatedException	Clause breach detected.

Table 1. Exception Categories

3.3. Contract Invoker

The Contract Invoker retrieves stored events from the Smart Contract Execution Queue and executes the corresponding smart contract on the corresponding blockchain. The communication between the Contract Invoker and the blockchain platform is synchronous. If the blockchain connection fails, the event will be returned to the Smart Contract Execution Queue for reprocessing. This reprocessing will occur *n* times, where *n* is a general system parameter registered through `API 4`. If the error persists after *n* attempts, an exception of type `BlockchainConnectionException` will be generated. If the execution proceeds as usual but there is a violation of one or more contract clauses, an exception of type *ContractClauseViolatedException* will be generated. Finally, the Contract Invoker will release an event to the Smart Contract Outbound Queue.

3.4. Event Updater

The Event Updater is the final component of the monitor's internal workflow. It updates the state of event execution by associating the result, generated by the Contract Invoker and made available in the Smart Contract Outbound Queue, with the initial event registered by the Event Handler. This information is utilised to construct the monitor output event for `API 5`. Listing 2 illustrates such an event. Exceptions are categorised into a list, where the exceptions in line 3 belong to the Structure Integrity Category, and those present within the clauses (lines 4 and 7) belong to the Execution Category. The `isValid` attribute represents the state of the object (whether it is valid or not). A contract will be deemed invalid if any exception is raised during event processing.

```
1  {"event id": { "contract id": {
2     "isValid": false,
3     "exceptions": [{"type": "type", "description": "description"}],
4     "clauses": { "clause id [1]": { "isValid": true, "exceptions": [] },
5       "clause id [n]": {
6         "isValid": false,
7         "exceptions": [{"type": "type", "description": "description"}] }}}}}
```

Listing 2. Outbound Event Model.

4. Ongoing work and preliminary results

The source code of the Monitoring System is available on GitHub[1]. It is already possible to register the smart contract and its clauses with their respective arguments.

[1]https://github.com/gca-research-group/smart-contract-execution-monitoring-system

However, this task may seem redundant, as the user must also register the smart contract code. The process of registering clauses could be improved if the Monitoring System were able to analyse the smart contract and infer the clause metadata automatically. The management of blockchain connections requires connection keys. If the blockchain is accessed through a cloud-based service, an access key is required, for instance. For security reasons, storing keys in the Monitoring System database or file system may be risky. Services such as Bitwarden, Azure Key Vault, and AWS Key Management Service offer secure storage solutions and have been evaluated as potential options to address this issue.

Although we have outlined the model presented for the events reported by the processes, we cannot assume that integration platforms will adopt this model. In such a scenario, three solutions are being evaluated: updating the implementation of communication ports on integration platforms, updating the software application's source code, or adding a new layer within the monitor. Modifying ports requires significant effort, as it involves accessing, understanding, and updating the integration platform's source code implementation. This approach is also limited to open-source platforms. Our recommended solution is to update the software application's source code by adding a layer between it and the communication port. We have conducted a proof of concept using the Decorator design pattern, which allows us to dynamically add behaviour to objects without modifying the original code. This approach enables communication between the software application and the monitoring system without interfering with the business process. The initial results are promising. Finally, the last option would involve adding a layer in front of the Inbound Events Queue, capable of converting the received event into the required format adopted by the monitoring system. The weakness of this option is that the layer would need to be specifically implemented for a particular need, whereas the monitor should remain software-agnostic.

5. Conclusions and future work

This paper proposes and details an event-based architecture for a system that monitors integration processes and invokes smart contracts on blockchain platforms. The system comprises a set of APIs, which facilitate both system configuration and the retrieval of information regarding monitoring activities. It also consists of internal components: Inbound Events Queue, Event Handler, Smart Contract Execution Queue, Contract Invoker, Smart Contract Outbound Queue, and Event Updater. The Inbound Events Queue receives events for processing. The Event Handler polls these events and prepares them as required for the Contract Invoker. The Smart Contract Execution Queue receives events from the Event Handler. The Contract Invoker polls events from the Smart Contract Execution Queue and executes the smart contract on the blockchain platform. The Smart Contract Outbound Queue receives the execution results. Finally, the Event Updater updates the necessary information and makes the event available for queries. Even though we are addressing the enterprise integration domain, the monitoring system is blockchain and domain agnostic, as its operation depends primarily on configuring the blockchain connection, registering the smart contract, and sending standardised events containing the data required by the smart contract. We plan to evaluate the system overhead of using the monitoring system in an integration process; additionally, we need to evaluate the system under high-demand usage, pushing it to its limits, as even though the event-based architecture can support the demand, external agents such as the blockchain and the integrated application may not. Furthermore, Artificial Intelligence can pottencially enhance some functionalities. For instance, it could be employed to identify parameter misconfigurations or predict potential errors.

References

Addas, A. (2023). The concept of smart cities: a sustainability aspect for future urban development based on different cities. *Frontiers in Environmental Science*, pages 1–2.

Bautista, E., Sukhija, N., and Deng, S. (2022). Shasta log aggregation, monitoring and alerting in hpc environments with grafana loki and servicenow. In *Int. Conf. on Cluster Computing*, pages 602–610.

Datadog (2025). https://www.datadoghq.com/. Accessed: January 2025.

Dornelles, E., Parahyba, F., Frantz, R. Z., Roos-Frantz, F., Reina-Quintero, A., Molina-Jiménez, C., Bocanegra, J., and Sawicki, S. (2022). Advances in a DSL to specify smart contracts for application integration processes. In *Ibero-American Conference on Software Engineering*, pages 46–60.

Dynatrace (2025). https://www.dynatrace.com/. Accessed: January 2025.

Elastic (2025). https://www.elastic.co/. Accessed: January 2025.

Fabolude, G., Knoble, C., Vu, A., and Yu, D. (2025). Smart cities, smart systems: A comprehensive review of system dynamics model applications in urban studies in the big data era. *Geography and Sustainability*, pages 1–12.

Grafana (2025). https://grafana.com/. Accessed: January 2025.

Klymash, M., Zablotskyi, S., and Pohranychnyi, V. (2024). Improving alerting in the monitoring system using machine learning algorithms. In *Int. Conf. on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering*, pages 150–153.

Otero, M., Garcia, J. M., and Fernandez, P. (2024). Towards a lightweight distributed telemetry for microservices. In *Int. Conf. on Distributed Computing Systems Workshops*, pages 75–82.

Rosa-Sequeira, F., Basto-Fernandes, V., and Frantz, R. Z. (2018). Enterprise application integration: Approaches and platforms to design and implement solutions in the cloud. *Advances in Engineering Research*, pages 277–303.

Serrano, W. (2018). Digital systems in smart city and infrastructure: Digital as a service. *Smart cities*, pages 134–154.

Sukhija, N. and Bautista, E. (2019). Towards a framework for monitoring and analyzing high performance computing environments using kubernetes and prometheus. In *SmartWord*, pages 257–262.

Taheri, J., Gördén, A., and Al-Dulaimy, A. (2024). Using machine learning to predict the exact resource usage of microservice chains. In *Int. Conf. on Utility and Cloud Computing*, pages 25–34.

Tundo, A., Mobilio, M., Orrù, M., Riganelli, O., Guzmàn, M., and Mariani, L. (2019). Varys: An agnostic model-driven monitoring-as-a-service framework for the cloud. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1085–1089.

Zou, W., Lo, D., Kochhar, P. S., Le, X.-B. D., Xia, X., Feng, Y., Chen, Z., and Xu, B. (2019). Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering*, pages 2084–2106.