University of Camerino

School of Advanced Studies

Doctor of Philosophy in Blockchain and Distributed Ledger Technology

XXXVIII Cycle

Flexible and Trustworthy Execution of Business Processes

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy (Ph.D.) in Blockchain and Distributed Ledger Technology.

Supervisor

Prof. Francesco Tiezzi

PhD Candidate

Nawaz Abdullah Malla

Co-Supervisor

Prof. Rumyana Neykova

Coordinator

Prof. Flavio Corradini

Academic Year 2024-2025

ACKNOWLEDGEMENTS

I would like to begin by expressing my deepest gratitude to Almighty Allah (Subhanahu wa Ta'ala), whose endless blessings, mercy, and guidance have enabled me to undertake and complete this doctoral journey. Without His will this work would not have been possible. I extend my heartfelt appreciation to my beloved parents for their unwavering love, constant encouragement, and countless sacrifices. Their prayers, patience, and belief in me have been a continuous source of strength and motivation throughout my life and this academic pursuit. I would also like to express my sincere gratitude to all the professors, colleagues, and friends who have guided and supported me throughout this journey. Their invaluable help, encouragement, and companionship have made these years both intellectually stimulating and personally enriching.

I would like to express my sincere gratitude to **Professor Francesco Tiezzi** from the University of Florence for his continuous support, insightful guidance, and for the opportunities he provided to collaborate on exciting research topics. His expertise, patience, and dedication have had a profound impact on my academic and personal growth.

I am also very grateful to **Professor Rumyana Neykova** from Brunel University London for her valuable mentorship, constructive feedback, and for inspiring me with her passion for rigorous research. Her collaboration and insightful discussions have greatly enriched the outcomes of this thesis.

I would like to express my sincere and profound gratitude to **Professor Francesco Santini** and **Professor Andrea Polini** for serving as reviewers of this doctoral thesis. I am deeply appreciative of the time, expertise, and careful consideration they devoted to evaluating this work. Their insightful comments, constructive criticisms, and valuable suggestions have greatly

contributed to enhancing the quality, rigor, and clarity of this thesis. Their thoughtful feedback has been instrumental in refining both the conceptual framework and the overall presentation of the research. I am truly honored to have benefited from their scholarly guidance and extend my heartfelt thanks for their support and contribution to this work.

My heartfelt thanks go to **Professor Andrea Morichetta** and **Professor Alessandro Marcelletti** from the University of Camerino for their continuous encouragement, technical guidance, and kind support during the most critical stages of my Ph.D. Their advice and availability have been crucial to overcoming challenges and achieving my goals.

I would like to express my sincere gratitude to **Dr. Khaleel Ahmad** from Maulana Azad National Urdu University (MANUU) for his guidance and mentorship during my master's studies. His support and advice inspired me to pursue higher education abroad and have had a lasting impact on my academic and personal growth.

My sincere thanks also go to my friends and companions **Dr. Waseem Mir**, **Dr. Ishfaq Malla**, **Dr. Mudasir Bhat**, **Dr. Rayees Parry**, and **Dr. Mujtaba Farooq**. The countless memories, discussions, laughter, and moments we shared together have been a great source of happiness and strength throughout this journey. Their friendship and support have made this phase of my life truly special and unforgettable.

I would also like to thank my friends and colleagues from the University of Florence, especially **Dr. Mohammad Atif**, **Dr. Saqib Hayat**, **Javed Jamshed** and **Fahad Ahmed Khokhar**, for their friendship, cooperation, and the wonderful memories we have shared, both inside and outside the research environment.

I am equally thankful to my friends from the University of Camerino especially **Zeeshan, Sameer and Rouf** for their companionship, encouragement, and for creating an environment full of collaboration and joy that made the research journey truly fulfilling.

My sincere appreciation also goes to the **Doctoral School of Advanced Studies** for believing in my potential and granting me the opportunity to serve as the Doctoral Representative. This experience has been deeply rewarding and has allowed me to grow both academically and personally.

To my friends from Brunel University London — **Ahmed, Silvia, Simone, Abdelhai, and Roberto** — thank you for your friendship, the many discussions, laughter, and support during my research visits. Each of you has

contributed to making this journey a memorable one.

I wish to express my heartfelt love and gratitude to my beloved nephew, ***Mikayeel Rayees***, who brought a new and profound happiness into our family as first child. His presence has filled our lives with joy, hope, and renewed meaning. Mikayeel has given me a reason to cherish every moment and milestone with warmth and excitement. He represents a beautiful beginning and a source of endless love for our entire family.

Finally, my heartfelt and deepest gratitude to **Dr. Sheema jan**, Doctoral Fellow at NIT Srinagar, whose unwavering support has been a constant throughout this journey. Her love, patience, and the countless beautiful moments we have shared together have been my greatest source of strength and joy. From being a dear friend to becoming my life partner, her presence has been my biggest motivation, and I am truly blessed to have her by my side.

Dedication

Dedicated to my family whose unwavering love and support have been my greatest strength.

To my mentors,
who inspired me to pursue knowledge with curiosity and integrity.

And to all those who believe in the power of perseverance.

"Success is not final, failure is not fatal: It is the courage to continue that counts." — Winston Churchill

ABSTRACT OF THE DISSERTATION

Blockchain technology has emerged as a transformative solution for executing inter-organizational business processes in distributed environments where trust between participants cannot be assumed. By encoding business logic into smart contracts, blockchain ensures tamper-proof and transparent execution according to predefined rules. The Business Process Model and Notation (BPMN) standard, particularly its choreography diagrams, provides a high-level, intuitive language for modeling these multi-party interactions. However, a significant challenge arises from the inherent immutability of blockchain, which conflicts with the need for business processes to adapt to unforeseen circumstances at runtime. Traditional platforms like Ethereum require complex, error-prone, and costly patterns (e.g., proxy contracts) to enable updates, undermining both flexibility and trust.

This thesis addresses this challenge by leveraging the Algorand blockchain, which natively supports the secure and efficient updating of stateful smart contracts. We propose a novel, end-to-end framework for the flexible and trustworthy execution of BPMN choreographies. The core contributions are fivefold. First, we present a formal, modular, and semantics-driven method for translating BPMN choreography models into smart contract code, ensuring a one-to-one correspondence between model elements and contract functions. Second, we detail a mechanism for runtime adaptation, where changes to the choreography model are seamlessly reflected in the underlying Algorand smart contract using its native update capabilities, eliminating the need for complex patterns. Third, we explore the use of Large Language Models to assist in migrating existing Solidity contracts to Algorand, reducing the barrier to entry. Fourth, we introduce a BPMN to Algorand smart contract generation tool that automates the translation of BPMN choreography models into executable smart contracts, simplifying

the development process and eliminating the need to write smart contracts from scratch. Finally, by exploiting our BPMN-to-Algorand translation tool we validate our approach through experiments and case studies, demonstrating its feasibility, cost-effectiveness, and superior performance compared to Ethereum-based solutions. The results confirm that the proposed Algorand-based approach provides a robust foundation for achieving both trustworthiness through decentralization and immutability, and flexibility through secure and straightforward contract updates, thereby effectively supporting dynamic business processes.

LIST OF PUBLICATIONS

- Malla, Nawaz Abdullah, Alessandro Marcelletti, Andrea Morichetta, and Francesco Tiezzi. "Unveiling Algorand Storage Peculiarities." In DLT. 2024.
- Malla, Nawaz Abdullah, Rumyana Neykova, Giuseppe Destefanis, and Francesco Tiezzi. "LLM-Based Translation of Ethereum Solidity Contracts to Algorand Python." In 2025 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS), pp. 7-12. IEEE, 2025.
- Malla, Nawaz, Alessandro Marcelletti, Andrea Morichetta, and Francesco Tiezzi. "Algorand's Path to Flexibility in Decentralized Applications." In International Conference on Data Analytics & Management, pp. 177-188. Singapore: Springer Nature Singapore, 2024.
- Malla, Nawaz Abdullah, Khaleel Ahmad, Jameel Ahamed, and Halimjon Khujamatov. "Implementing Flexible Business Processes with Algorand Blockchain." In Fostering Machine Learning and IoT for Blockchain Technology: Smart Cities Applications, Volume 1, pp. 215-227. Singapore: Springer Nature Singapore, 2025.
- Malla, Nawaz Abdullah, Alessandro Marcelletti, Andrea Morichetta, and Francesco Tiezzi. "A Blockchain-Based, Semantics-Driven, Modular Implementation of BPMN Choreographies." In International Conference on Society 5.0, pp. 181-193. Cham: Springer Nature Switzerland, 2025.
- Malla, Nawaz Abdullah, Alessandro Marcelletti, Andrea Morichetta, and Francesco Tiezzi. "An Algorand-based Approach for Flexible Execution of BPMN Choreographies." (2025).

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

PART I

INTRODUCTION & BACKGROUND

CHAPTER 1

INTRODUCTION

Inter-organisational business processes deal with distributed organisations, controlling different components, e.g., software services, business units, and departments, to perform certain interactions to reach a common final objective. This kind of business involving many distributed parties is becoming crucial since it involves complex and distributed systems difficult to coordinate and execute. This thesis aims at devising a blockchain-based solution to support the flexible and trustworthy execution of such interorganisation business processes.

This chapter introduces the motivation of the thesis (Section 1.1), and its research objectives (Section 1.2) and contributions (Section 1.3). Finally, the chapter concludes with a description of the structure of the thesis (Section 1.4).

1.1 Motivation

In today's globalized economy, business processes increasingly span multiple independent organizations, creating complex networks of collaboration that transcend traditional corporate boundaries. These inter-organizational business processes—encompassing, e.g., supply chain management, cross-border payments, healthcare data sharing, and multi-party regulatory compliance—require sophisticated coordination mechanisms to achieve common goals across organizational divides. However, these distributed environments are fundamentally characterized by a lack of mutual trust; participants often demonstrate reluctance to grant control to central authorities while simultaneously harboring concerns about other participants potentially manipulating processes for personal gain.

The inherent complexity of managing these distributed processes is amplified by several interconnected factors: geographical dispersion of participants, heterogeneity of technical infrastructures, varying regulatory requirements across jurisdictions, and the dynamic nature of business relationships. This led to a more and more wide adoption of business process management solutions to support design, modeling, execution, and maintenance of business processes.

The Business Process Model and Notation (BPMN) standard [92] has emerged as the predominant language for modeling business processes in both industrial practice and academic research. Developed by the Object Management Group (OMG) [22, 121], BPMN provides a rich graphical notation that serves as a common vocabulary for process specification, enabling clear communication among stakeholders across organizational boundaries. Within the BPMN ecosystem, different diagram types serve distinct purposes, with **choreography diagrams** specifically addressing the unique requirements of inter-organizational scenarios by focusing exclusively on public message exchanges between participants while deliberately concealing their internal behaviors. This modeling approach allows organizations to maintain privacy over proprietary operations while still specifying the essential interactions required for effective collaboration.

Traditional Business Process Management (BPM) systems have primarily focused on intra-organizational processes, where a single entity maintains complete control over process execution, data storage, and governance mechanisms. However, this centralized paradigm proves fundamentally inadequate for inter-organizational scenarios, where multiple autonomous participants with potentially conflicting interests must collaborate toward common objectives [13]. The limitations of conventional approaches become particularly evident in environments characterized by information asymmetry, divergent operational priorities, and incomplete trust relationships between participants [45].

The execution of business processes across organizational boundaries inherently involves complex trust relationships that manifest in several challenging forms. Most conventional inter-organizational systems rely on a central coordinating entity that hosts the process engine, manages process state, and orchestrates interactions between participants. This architecture requires all participants to trust this central authority to behave correctly, maintain data confidentiality, and ensure fair treatment—a problematic assumption given that the central authority may have conflicting interests, suffer security breaches, or become a single point of failure that disrupts the entire business network. Even in peer-to-peer architectures without central coordination, participants must trust each other to faithfully execute their roles in the process, report accurate information, and refrain from oppor-

tunistic behavior. This mutual trust is particularly challenging to establish in dynamic business networks where participants may have limited prior interaction or compete in other domains. Additionally, participants must trust that the technical infrastructure accurately records process states, prevents unauthorized modifications, and preserves the integrity of process history—a requirement that traditional distributed systems struggle to meet without compromising performance, scalability, or security.

These trust deficits generate tangible business consequences, including increased transaction costs due to verification requirements, delayed process execution caused by reconciliation activities, limited participation in business networks due to trust barriers, and heightened vulnerability to fraud or manipulation. The challenges extend to transparency and auditability, where traditional systems demonstrate significant limitations. Process data typically resides in siloed systems across different organizations, making consolidated auditing challenging and requiring complex data integration efforts. Furthermore, centralized databases allow authorized parties to modify historical records, either legitimately for error correction or maliciously for fraudulent purposes, fundamentally undermining the reliability of audit trails.

Blockchain technology has emerged as a pivotal innovation to address this trust deficit [85]. As a decentralized and distributed ledger, blockchain enables the creation of tamper-proof, transparent, and auditable records of transactions. Smart contracts—programs deployed and executed on a blockchain—automate the enforcement of business rules, ensuring that processes unfold exactly as designed without requiring intermediaries. This makes blockchain an ideal platform for the **trustworthy execution of business processes**, particularly in environments where traditional trust mechanisms prove insufficient.

However, the adoption of blockchain technology introduces a critical conflict: the inherent tension between the need for **flexibility** and the property of immutability. Real-world business processes are dynamic entities that must continuously evolve in response to unforeseen events, changing regulations, market conditions, and internal optimizations. Organizations constantly identify opportunities to optimize processes for efficiency, cost reduction, or quality improvement, requiring modifications to existing process models and implementations. The business ecosystem itself evolves continuously, with new participants joining, existing participants leaving, roles changing, and capabilities expanding—all dynamics that necessitate corresponding adaptations in process implementations. Furthermore, legal and regulatory frameworks change over time, requiring business processes to be updated to maintain compliance across different jurisdictions.

In traditional BPM systems, these changes can be achieved through process model updates and system reconfiguration. However, in blockchain-

based systems, the immutability of deployed smart contracts makes such adaptations challenging, costly, and sometimes prohibitively complex. Traditional blockchains like Ethereum are inherently immutable; a smart contract's code cannot be altered once deployed. While solutions like proxy patterns [44] or off-chain engines [54] exist, they introduce significant complexity, security vulnerabilities, and high transactional costs (gas fees), ultimately diminishing the very trust and efficiency the blockchain was meant to provide [116].

Beyond the core challenges of trust and flexibility, inter-organizational business processes introduce additional complexities related to participant multiplicity and sophisticated access control requirements. Business processes require predictable operational costs for budgeting and economic planning, yet the gas fees characteristic of platforms like Ethereum create significant uncertainty that hinders adoption. Real-world business processes often involve high volumes of transactions and interactions, and platform limitations on transaction processing capacity can create bottlenecks that render blockchain-based solutions impractical for many business scenarios.

The comprehensive analysis of challenges in inter-organizational business process execution reveals a significant research gap: while blockchain technology provides a powerful foundation for addressing trust deficits, its current implementations introduce new challenges related to flexibility, cost, scalability, and access control that limit practical adoption. Existing approaches to blockchain-based business process execution have primarily focused on either trust enforcement through immutability or flexibility through complex workarounds, but not both simultaneously. Solutions that prioritize trust typically sacrifice flexibility, while those that emphasize flexibility often compromise trust guarantees. This dichotomy represents a fundamental limitation in the current state of the art.

The central **research question** guiding this work is:

> "*How can we achieve flexible execution of BPMN choreographies on blockchain without compromising the trustworthiness provided by its decentralized and immutable nature?*"

This research question encompasses the fundamental challenge of creating business process execution systems that can adapt to changing requirements while maintaining the core benefits of blockchain technology—a capability essential for the practical adoption of blockchain in real-world business scenarios across diverse domains, including supply chain management, healthcare coordination, financial services, and cross-border commerce.

This thesis addresses this gap by proposing a novel framework that leverages the Algorand blockchain's unique capabilities to achieve both trustworthiness and flexibility in business process execution. Algorand's native support for smart contract updates, combined with its scalable architecture

and predictable cost structure, provides a technical foundation that previous platforms lacked.

1.2 Research Objectives

The primary goal of this thesis is to design, implement, and validate a framework for the flexible and trustworthy execution of BPMN choreographies on blockchain. This overarching goal is divided into the following specific research objectives:

RO1 To develop a framework for translating BPMN choreography models into blockchain smart contracts that is modular and driven by BPMN choreography formalization.

RO2 To design and implement controlled runtime adaptation of blockchain-based business processes using Algorand's update capabilities.

RO3 To design and develop an automated BPMN-to-Algorand smart contract generation tool that translates BPMN choreography models into executable Algorand smart contracts.

RO4 To provide an AI-assisted methodology for migrating existing business process implementations to the Algorand blockchain platform.

RO5 To validate the proposed approach, and in particular its economic viability and performance characteristics, through empirical evaluation across diverse business use cases.

1.3 Thesis Contributions

The work presented in this thesis is based on an encoding of choreographies into smart contracts executable on a blockchain infrastructure; works along this line are already available in the literature (see Chapter 3), but with different focus and purpose. This thesis makes the following novel key contributions.

1. **A Formal Semantics and Modular Translation for BPMN Choreographies.** We provide a rigorous formalization of BPMN choreographies using a Backus-Naur Form (BNF) grammar and operational semantics. This foundation enables a modular translation to smart contract pseudocode, which can be implemented in various target languages like Solidity and Algorand Python. This is presented in Chapter 4 and permits achieving RO1.
2. **A Novel Flexibility Mechanism using Algorand.** We present a framework for updating BPMN choreography smart contracts at runtime by directly leveraging Algorand's *UpdateApplication* transaction.

This approach [79] eliminates the need for complex proxy patterns, reducing complexity and cost while maintaining security. To this aim, we analyze the Algorand's storage options (i.e., Local, Global, and Box) highlighting their behavior during update and deletion operations, which is critical for designing flexible decentralized applications [78]. This is presented in Chapter 5 and permits achieving RO2.

3. **Automated BPMN-to-Algorand Smart Contract Generation Tool.** We design and implement a tool that automatically translates BPMN choreography models into executable Algorand Python smart contracts. The tool enables rapid deployment of choreography-based business process models. This contribution bridges the conceptual gap between high-level business process design and blockchain implementation, reducing manual implementation effort and minimizing errors. This is presented in Chapter 6 and permits achieving RO3.
4. **An Exploration of LLM-Assisted Cross-Platform Translation.** We conduct an empirical study on the use of LLMs (e.g., Claude, ChatGPT) for translating Solidity contracts to Algorand Python, providing a mapping of language constructs and a hybrid human-AI workflow for developers. This is presented in Chapter 7 and permits achieving RO4.
5. **A Comprehensive Cost-Benefit Analysis.** We demonstrate through practical case studies (e.g., healthcare, e-commerce) that the proposed Algorand-based approach offers significant cost savings and performance advantages over traditional Ethereum-based platforms. This is presented in Chapter 8 and permits achieving RO5.

1.4 Thesis Structure

This thesis is organized into five parts and eight chapters and they are described as follows.

Part I: Introduction & Background. Chapter 1 provides the motivation, research objectives, and a brief description of the main contributions. Chapter 2 provides the necessary background on blockchain, with a special focus on Ethereum and Algorand, and BPMN.

Part II: State Of The Art: Related Work. Chapter 3 reviews related work on blockchain-based business process execution, flexibility solutions, and model-driven engineering.

Part III: Flexible Execution of Business Processes. This part constitutes the core technical contribution of the thesis, detailing the modular

translation, the Algorand-based flexibility mechanism, and the LLM-assisted migration approach. The part first provides an insights into the storage mechanism of Algorand blockchain and later concludes with the flexibility approach to support the runtime adaption and AI translation of smart contracts for different blockchain platforms. More specifically:

- Chapter 4 provides a rigorous formalization of BPMN choreographies using a BNF grammar and operational semantics. This foundation enables a modular translation to smart contract pseudocode, which can be implemented in various target languages like Solidity and Algorand Python.
- Chapter 5 presents our framework for updating BPMN choreography smart contracts at runtime by directly leveraging Algorand's *UpdateApplication* transaction. This approach eliminates the need for complex proxy patterns, reducing complexity and cost while maintaining security.
- Chapter 6 introduces the BPMN-to-Algorand Smart Contract Generation Tool, focusing on the automation of smart contract creation from a BPMN choreography model. It details the process of translating BPMN diagrams into Algorand-compatible smart contracts, highlighting the tool's capabilities, and providing an in-depth explanation of its functionalities.
- Finally, Chapter 7 presents the empirical study on the use of LLMs for translating Solidity contracts to Algorand Python, providing a mapping of language constructs and a hybrid human-AI workflow for developers.

Part IV: Validation of the Approach. Chapter 8 presents the experimental evaluation, including flexibility and cost analysis of three different use cases of business processes.

Part V: Conclusions & Future Works. Finally, Chapter 9 summarizes the thesis contributions, starting from flexible execution of business processes and challenges related to flexibility, and limitations, and finally outlines directions for future research.

CHAPTER 2
BACKGROUND

This part introduces all the concepts used in the next part of the thesis. In particular, a first description of blockchain technology and its characteristics is provided with a special concern on their application to the BPM domain. Here both the Ethereum and the Algorand blockchains are analysed, highlighting their peculiarities and usage scenarios. Then a section introduces the Business Process Management (BPM) discipline and the Business Process Modeling and Notation (BPMN) specification with a focus on choreography diagrams. The concluding part of a chapter reports a healthcare process scenario which will be used as a running example in the remainder of the thesis. In particular, the case study is analysed from different perspectives, emphasising the challenges in using blockchain for a trusted execution. For each challenge, a different context of a healthcare process is presented, with a focus on those characteristics requiring a novel approach.

2.1 Blockchain in BPM

A blockchain is a distributed, immutable, and transparent digital ledger maintained by a peer-to-peer network. In the context of Business Process Management (BPM), particularly for inter-organizational processes, these fundamental properties provide unique solutions to longstanding challenges in distributed system coordination and trust establishment. Decentralization represents a paradigm shift from traditional centralized BPM systems where a single organization controls the process engine and data storage.

In blockchain-based BPM, unlike conventional BPM systems where one organization hosts the process engine, blockchain distributes control among all participants. This eliminates the need for organizations give control to a

central authority or trust a third-party service provider.

However, as blockchain Process state transitions and message exchanges require validation by multiple nodes in the network. For BPM, this means that no single participant can unilaterally advance the process state or manipulate process data without consensus from other participants. The distributed nature ensures that the business process continues to operate even if some participants experience technical failures or attempt malicious behavior. This is particularly crucial for critical business processes spanning multiple organizations.

In practice, this enables truly collaborative business processes where no single entity has disproportionate power, making blockchain ideal for, e,g., supply chain management, cross-organizational healthcare processes, and multi-party financial transactions.

Immutability ensures that once recorded, process data cannot be altered or deleted. For BPM applications, this property provides, every process instance, message exchange, and state transition is permanently recorded and cryptographically secured. This creates an incontrovertible audit trail that cannot be modified by any participant, ensuring the integrity of business process execution. Since participants cannot deny their actions in the process, immutability provides strong evidence of compliance with agreed-upon business protocols and auditability of the data exchanged by participants during the process execution. This is especially valuable in regulated industries and for dispute resolution. The immutable record prevents participants from rolling back process states or reordering activities to gain unfair advantages, ensuring that all parties adhere to the same process version throughout execution.

However, this strength also presents the core challenge addressed by this thesis: while immutability ensures trustworthiness, it conflicts with the natural evolution of business processes that require adaptation to changing business environments.

Transparency in blockchain enables unprecedented visibility into business process execution. All authorized participants can view the same process state and history, eliminating information asymmetry and reducing coordination overhead. This is particularly valuable in complex supply chains where multiple organizations need synchronized visibility. Participants can independently verify process progress without relying on centralized status reports, enabling more responsive business operations and quicker exception handling. The transparent nature facilitates auditing and compliance verification, as regulators can directly access the immutable process history without requiring documentation from individual participants.

In BPMN choreography execution, transparency ensures that all message exchanges and process states are equally visible to all participants, creating

a level playing field for collaborative business processes.

The security properties of blockchain provide crucial foundations for trustworthy process execution. Every process action is cryptographically signed, ensuring that only authorized participants can execute tasks and send messages in the business process. Cryptographic hashing ensures that process data cannot be modified without detection, protecting against unauthorized changes to process states, message content, or execution paths. The requirement for multiple validations of each process state transition prevents single points of failure and collusion attacks, ensuring that the business process executes exactly as modeled.

These security properties make blockchain particularly suitable for business processes involving sensitive transactions, valuable assets, or critical operations where trust between participants cannot be assumed. The convergence of blockchain properties with BPM creates a new paradigm for inter-organizational process execution. While traditional BPM systems struggle with trust establishment in multi-party scenarios, blockchain-native BPM provides:

Trustless Collaboration: Organizations can collaborate on business processes without establishing prior trust relationships, relying instead on cryptographic guarantees and decentralized consensus. Smart contracts enforce process rules automatically, ensuring that all participants adhere to the agreed-upon choreography without manual monitoring or intervention. The combination of immutability and transparency creates unprecedented levels of accountability in business process execution.

However, as this thesis addresses, the very properties that enable trustworthiness—particularly immutability—also create challenges for process flexibility and evolution. The subsequent sections explore how modern blockchain platforms like Algorand provide pathways to reconcile these seemingly contradictory requirements.

2.1.1 Smart Contracts

A smart contract is a program stored on a blockchain that automatically executes when predetermined conditions are met. They enable the automation of contractual clauses, removing the need for intermediaries and reducing the risk of manipulation. In the context of BPM, a smart contract enforces the flow of a choreography, ensuring that participants can only perform actions that are permitted by the current state of the process.

The advent of blockchain technologies provided the means to execute smart contracts in a decentralized fashion without the need of trusting a central authority: the correct execution of the smart contract is ensured by the consensus protocol of the blockchain on which it is deployed. Blockchain smart contracts' property of trustless execution make it an interesting tech-

nology for the business processes. However, business processes execution often requires processing large amounts of data which are often difficult to manage in a blockchain environment. In fact, traditional blockchains can process only few transactions per second, making it unsuitable for the needs of many business processes execution. Furthermore, traditional public blockchains require users to pay fees for every piece of data committed to the chain, this makes the use of blockchains too expensive in many business processes execution scenarios, since they often involve inter-organizational transactions exchanging large amounts of messages. In this thesis, it presents a solution for executing smart contracts that improves scalability on blockchains in terms of throughput and costs. The solution is particularly suited for the business processes execution and its generality makes it possible to be used in a variety of scenarios not necessarily related to the business processes execution provide a description of the key smart contract languages: solidity, teal and algorand python.

2.1.2 Ethereum

Ethereum is the most prominent blockchain platform for smart contracts. Its ecosystem is vast, and its native language, Solidity, is widely adopted. Early approaches to blockchain-based BPM, such as Caterpillar [11], targeted Ethereum. However, Ethereum faces two major challenges for business process execution:

1. Scalability and Cost: The Proof-of-Work (now transitioning to Proof-of-Stake) consensus mechanism and network congestion can lead to low transaction throughput and high, unpredictable gas fees. This is prohibitive for processes requiring numerous interactions.

2. Smart Contract Immutability: Once deployed, a Solidity smart contract is immutable. This is a significant barrier to flexibility. To update a contract's logic, developers must deploy a new contract and migrate the state, or use proxy patterns (e.g., UUPS, Transparent Proxies). These patterns are complex, have led to severe security vulnerabilities [11, 31], and increase gas costs. This complexity undermines the trustworthiness of the system.

2.1.3 Algorand

Algorand [19] founded in 2017 first carbon-negative blockchain protocol, provides institutional-grade blockchain infrastructure, offering high performance, security, and scalability. Designed for decentralized applications, digital assets, and financial solutions, it leverages a unique Pure Proof-of-Stake (PPoS) consensus mechanism to ensure instant finality, low transaction costs, and robust security.

Consensus Protocol

The problem with many blockchains is they sacrifice at least one of the key properties of security, scalability, and decentralization, known as the blockchain trilemma. Algorand addresses the blockchain trilemma with its unique PPoS mechanism, achieving a strong balance between security, scalability, and decentralization.

Algorand's consensus protocol [107] works by selecting a block proposer and a set of voting committees at each block round, to propose a block and validate the proposal, respectively. The proposer and committees are randomly chosen from the pool of all Algo holders, and the likelihood of being chosen is proportional to the account's stake in the network. The technical specifics of Algorand's consensus include:

Verifiable Random Function (VRF): Algorand uses cryptographic sortition based on VRFs to randomly select users to participate in the consensus protocol. The VRF acts as a random number generator that provides a proof that the selection was truly random.

Byzantine Agreement Protocol: Once users are selected, they participate in a Byzantine agreement protocol that ensures consensus even if some participants are malicious.

Two-Phase Block Production: The Two-Phase Block Production process consists of distinct yet coordinated steps to ensure secure and efficient block finalization. In the Propose Phase, selected proposers submit candidate blocks for consideration. During the Soft Vote stage, a committee of validators reviews these proposals and votes to indicate preliminary agreement on the most suitable block. Finally, in the Certify Vote phase, a separate committee formally certifies the chosen block, confirming its validity and inclusion in the blockchain.

Cryptographic Self-Selection: Users privately check if they're selected for committees using their private participation keys, without revealing themselves until necessary, preventing targeted attacks.

Committee Rotation: New committees are selected for each step of the consensus process, enhancing security.

Algorand consensus provides the following technical guarantees: The system ensures fork resistance, maintaining with over 99.9% probability that no forks occur. It guarantees strong consistency, so all users share an identical view of confirmed transactions. Additionally, it upholds liveness, continuing to make progress and confirm new transactions even under severe or unstable network conditions.

2.1.4 Proxy Patterns

In blockchain systems, proxy patterns represent a foundational design approach to enable smart contract upgradability without changing the contract address or user-facing interface. Since most blockchain platforms (e.g., Ethereum) deploy immutable bytecode, the proxy pattern introduces an intermediate proxy contract that delegates calls to a separate logic (or implementation) contract, thereby decoupling state storage from executable logic. This allows the contract's logic to be replaced or extended over time while preserving the same state variables and contract address, ensuring continuity for users and integrated services. The basic principle relies on the delegatecall instruction in Ethereum Virtual Machine (EVM) [9, 48], which allows the proxy to execute functions defined in another contract within its own storage context.

- **Simple Proxy Pattern** The simplest form of proxy, where a fixed intermediary contract forwards all calls to an implementation contract. While easy to deploy, it lacks flexibility and cannot be upgraded once deployed [93].

- **Transparent Proxy Pattern** Introduced by OpenZeppelin, this model routes user calls to an implementation contract through a proxy while reserving administrative calls for the proxy itself, ensuring state consistency and preventing selector collisions [95].

- **Universal Upgradeable Proxy Standard (UUPS)** A refinement of the transparent proxy, UUPS embeds the upgrade logic within the implementation contract itself rather than the proxy, reducing gas costs and simplifying contract management [101, 95]

- **Beacon Proxy Pattern** In this approach, the proxy delegates logic lookups to a separate beacon contract that points to the active implementation. This enables multiple proxies to share the same upgrade source, enhancing scalability [94].

- **Diamond Proxy (EIP-2535)** Designed for large-scale modular applications, the diamond proxy splits logic across multiple "facets" (contracts) that handle specific functionalities, supporting fine-grained upgrades and reducing monolithic contract risks [68].

- **Minimal Proxy (EIP-1167 or Clone Factory)** This lightweight proxy replicates existing contract logic using a minimal bytecode template, optimizing for deployment cost and efficiency when deploying many identical contracts [86].

Comparison with Algorand's Upgradability Model

While proxy patterns such as the Transparent Proxy, UUPS, Beacon, and Diamond Proxy have become standard solutions for achieving smart contract upgradability on platforms like Ethereum, they inherently introduce architectural complexity and security overhead due to their reliance on delegatecall and multi-contract state management. These designs require strict control of storage layouts, administrative privileges, and function selectors to ensure consistency across contract versions, as highlighted in studies by Feist et al. [48] and Zhou et al [126]. In contrast, Algorand provides a native mechanism for stateful smart contract updates, thereby eliminating the need for proxy-based indirection.

In Algorand's TEAL and PyTeal framework, smart contracts can be upgraded in place by authorized accounts using the platform's update transaction type, provided that the contract logic explicitly allows such updates through an update_approval_program. This model inherently maintains state continuity, improves security predictability, and avoids storage misalignment vulnerabilities that often occur in proxy implementations. Moreover, the absence of a delegatecall-like mechanism ensures that contract logic is executed deterministically and atomically, aligning with Algorand's Layer-1 design philosophy of on-chain simplicity and verifiable correctness.

Empirical analyses, such as those by Weber et al. [109], further demonstrate that Algorand's model not only reduces runtime costs but also improves process stability for blockchain-based business process management applications. Consequently, while proxy-based architectures remain essential in Ethereum-like environments for achieving upgradability, Algorand's native update mechanism represents a simpler, more secure, and maintainable paradigm for evolving smart contracts without external orchestration.

2.1.5 Blockchain Comparison

The current trend towards introducing different blockchain technologies, with different characteristics, has led to a proliferation of technologies to be acquired and learned. For our purpose, it is possible to distinguish among the following characterisations.

- **Permissionless vs Permissioned**: a permissionless blockchain is an open network where participants can join, and leave the network without the need for any authorisation. A permissioned blockchain runs a ledger among a set of previously identified and authorised peers.
- **Auditability vs Confidentiality**: an auditable blockchain has an immutable and transparent nature, and it naively allows independent auditing over the stored data. On the contrary, a permissioned blockchain

introduces confidentiality so that stored data are not visible to anyone. Moreover, it restricts the distribution of information only to authorised nodes.

- **High decentralisation vs Performance and scalability**: the usage of strong consensus algorithms allows to trust nodes previously unknown or not trusted, in a decentralised context. On the contrary, the introduction of access control mechanisms leads to a trusted network with higher scalability and transaction throughput
- **Anonymity vs Identity**: a blockchain technology could permit anyone to join the network without putting in place any access control mechanism. Trust over the stored data will be in any case guaranteed by the consensus algorithm. On the other hand, access to a blockchain can be restricted to authorised users introducing specific mechanisms. Consequently, it will be possible to associate identities with participants, and cryptographic credentials can be issued to new members. All communications can also be made use of authentication mechanisms.

2.2 Business Process Management

Business Process Management is the discipline overseeing the work conducted by organisations to ensure consistent outcomes and to take advantage of improvement opportunities [43]. In particular, BPM aims to manage the entire chain of events, activities, and decisions connected to an organisation. These chains are called *business processes* and BPM includes concepts, methods, and techniques to support their design, administration, configuration, enactment, and analysis [120]. Those processes can be used to represent many kinds of interactions as in the case of collaborative ones. Those are called in general multi-party business processes and usually take place when many parties are involved in collaborations where there is a global awareness of the relevant interactions [17].

Business process activities can be enacted automatically by information systems, without any human involvement. Hence, inter-organisational business processes are an important concept in facilitating collaboration between companies and information systems. More and more business processes also play an important role in the design and realisation of flexible information systems. These information systems are essential for providing the technical basis useful for a quick implementation of new functionalities that realise new products or services.

BPM is characterised by a set of steps that occur cyclically in order to adapt and improve the model. Hence, BPM involves a continuous cycle, comprising the following phases: *modelling*, *analysis*, *execution*, and *monitoring*.

2.2.1 BPMN Choreography Diagram

Nowadays, a prominent modelling language to describe collaborative distributed systems is the BPMN standard [92]. BPMN is a graphical notation to represent the graphical layout of business processes [22, 121], it has been standardised by the Object Management Group (OMG) and it is actually widely accepted both in industry and academia. BPMN permits the representation of business processes with different levels of detail through the use of diagrams. Each diagram consists of a set of modelling elements expressing different behaviours.

Although it has been initially introduced to define and document business processes, the use of BPMN models has successively gathered momentum as a starting point for model-driven engineering of distributed systems [98, 5]. For such a purpose, BPMN provides a set of flowchart-based notations, permitting the representation of different distributed systems perspectives, starting from the internal behaviour of the composing sub-systems, till their interaction. In inter-organisational cooperations, in addition to the need for coordination in a distributed setting, the involved parties also ask to keep private their internal behaviour [112]. For this reason, organisations have found in BPMN *choreography* diagrams a valid solution to represent such kind of coordination in a proper way [27, 6]. These diagrams allow to describe system interactions in terms of the exchange of messages from a global perspective, without exposing internal behaviour.

In a distributed environment, organisations wishing to collaborate can refer to specific choreographies that describe in detail how the different parties should interact to achieve common objectives. The integration of processes in this way leads to more peer-to-peer collaboration, shifting responsibility for each execution step of the collaborative process to the individual nodes. Consequently, in a choreography approach, each participant is responsible for partial orchestration, based on its individual rules without a central coordinator, and the final behaviour is specified as a family of permitted message exchange sequences.

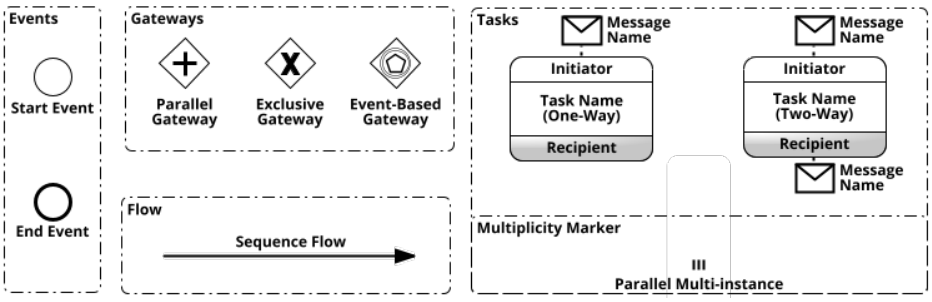


Figure 2.1: BPMN choreography core elements.

Figure 2.1 depicts the most used modelling elements that can be included in a BPMN choreography diagram. On the left, elements are used to define the control flow, while on the right, the elements are used to represent communication tasks.

- **Events** are used to represent something that can happen. An event can be a *Start Event* representing the point from which the choreography starts, or an *End Event* representing the choreography termination. Events are drawn as circles.
- **Sequence Flows** are edges used to connect events, gateways and tasks, permitting to specify the choreography execution flow.
- **Gateways** are used to drive the flow of a choreography. They can act either as join nodes (merging incoming sequence edges) or split nodes (forking the flow into multiple outgoing edges). Different types of gateways are available. An *exclusive gateway (XOR)* permits to represent choices. In particular, a XOR-split gateway is used after a decision to fork the flow into branches. When executed, it activates exactly one outgoing edge. A XOR-join gateway acts as a pass-through, meaning that it is activated each time the gateway is reached. A XOR gateway is drawn with a diamond marked with the symbol "×" A *parallel gateway (AND)* enables parallel execution flows. An AND-split gateway is used to model the parallel execution of two or more branches, as all outgoing sequence edges are activated simultaneously. An AND-join gateway synchronises the execution of two or more parallel branches, as it waits for all incoming sequence edges to complete before triggering the outgoing flow. An AND gateway is drawn with a diamond marked with the symbol "+". An *event-based gateway* is similar to the XOR-split gateway, but its outgoing branches activation depends on taking place of catching messages. Basically, such messages are in a race condition, where the first event that is triggered wins and disables the other ones. An event-based gateway is drawn with a diamond marked with the symbol of a double-rounded pentagon.
- **Tasks** are used to specify the message exchange between two participants. They are drawn as rectangles divided into three bands: the central one includes the name of the task, while the other two refer to the involved participants (the one in white is the initiator, while the grey one is the recipient). Messages can be sent either by one participant (One-Way tasks) or by both participants (Two-Way tasks). When creating tasks and participants it is also possible to specify the multiplicity aspect by including parallel Multi-instance markers. The resulting behaviour indicates that a participant can send/receive many messages or that a single message can be sent/received from/to many participants.

2.2.2 Running Example

To clarify the motivations and research challenges addressed in this thesis, this section presents a running example involving a multi-party healthcare process centered on X-ray examination management. The proposed example refers patient care workflows while ensuring trustworthinesses and flexibility among the involved parties. In this scenario, a healthcare provider initiates the X-ray examination process upon receiving a patient's diagnostic request. The process may also involve a specific radiology center or specialist explicitly indicated by the healthcare provider or patient. This example Fig. 2.2 illustrates an interaction framework where the requirements for trust, confidentiality, and data privacy dynamically evolve depending on the healthcare context and the stakeholders engaged in the process. The policy under consideration aims to streamline the workflow while safeguarding sensitive patient information in compliance with healthcare regulations.

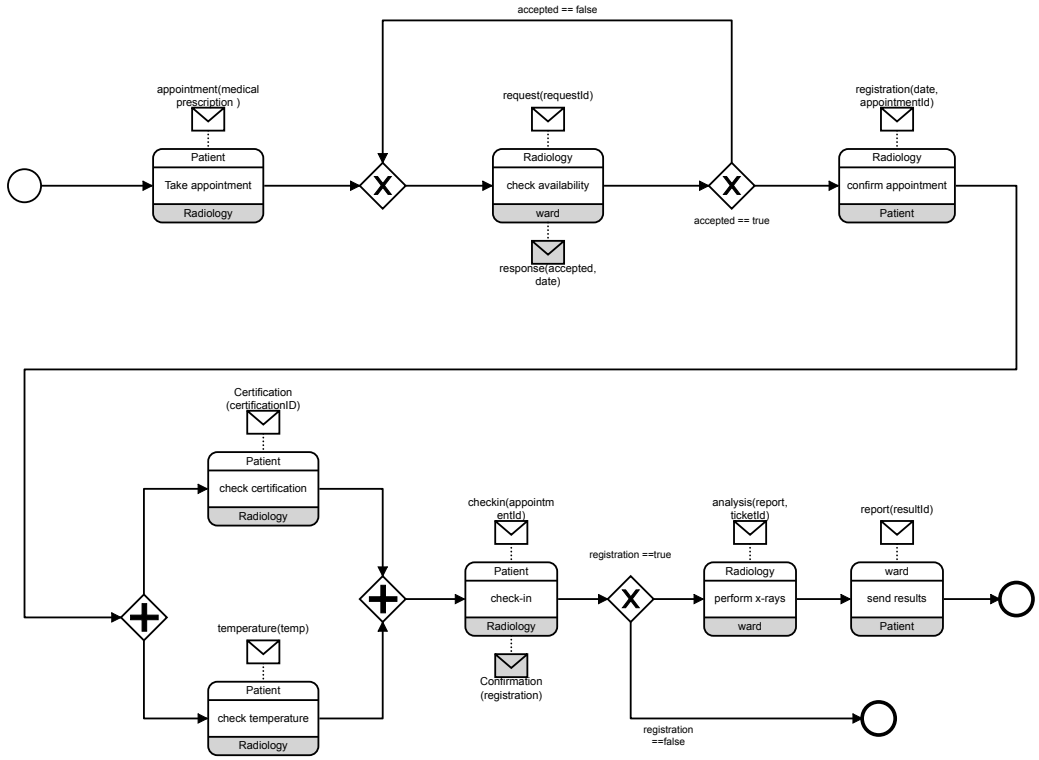


Figure 2.2: X-ray exam choreography diagram.

Running example description The choreography reported in Fig. 2.2 represents the communications that should take place among the participants for the scenario described above. The process begins with the patient requesting a quotation for an X-ray examination. If the examination slot

is available, the patient proceeds with the registration, and the healthcare provider confirms the appointment. Otherwise, if the preferred radiology center or specialist indicated by the patient is unavailable, the provider contacts an alternative radiology center, which then provides a quotation. Upon acceptance and registration, the radiology center schedules the examination and communicates the appointment details back to the healthcare provider, who finalizes the patient's request and coordinates the examination.

Trust requirement In the first healthcare context considered, the described interactions follow a decentralized model where patients and healthcare providers may not be known to each other beforehand, and each instance of the process can involve different patients, providers, and diagnostic centers. Several critical needs arise in this model. First, patients require assurance that their medical data and requests are handled transparently and securely, fostering trust in the healthcare ecosystem. Second, it is essential to verify the authenticity and certification of diagnostic results, ensuring that reports originate from accredited radiology centers or specialists.

To address these needs, blockchain technology can provide a secure and transparent way to share certified medical information across the care chain, enabling patients and providers to verify the provenance and integrity of diagnostic data. Moreover, the use of blockchain eliminates the need for additional central authorities or intermediaries to validate the process, thereby reducing administrative costs and enhancing the overall efficiency and fairness of healthcare delivery.

Flexibility requirement Considering the flexibility aspect in the proposed healthcare scenario, a possible case is illustrated in Figure 2.3. Here, the healthcare provider proposes an internal update to the process aimed at optimizing overall costs and reducing administrative steps. The new agreement involves a change in the workflow where the radiology center directly schedules and manages the patient's X-ray examination, bypassing the healthcare provider as an intermediary. This adjustment impacts the entire process: after the patient's initial request and the provider's response, the patient completes the registration and shares the necessary information for the appointment. If the preferred radiology center has availability, it schedules the examination and the process concludes. Otherwise, the healthcare provider coordinates with an alternative radiology center, which then directly manages the patient's appointment, thereby closing the process.

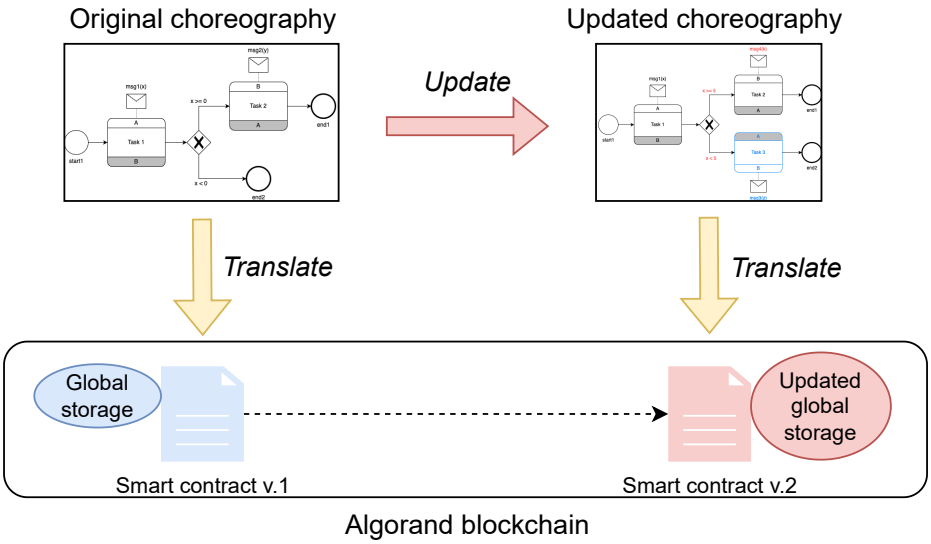


Figure 2.3: Update approach for runtime adaption for evolving Business Processes.

PART II

STATE OF THE ART: RELATED WORK

CHAPTER 3

RELATED WORK

The integration of blockchain technology with business process management represents a paradigm shift in how inter-organizational processes can be executed in trustless environments. This chapter situates our research within the broader academic and technical landscape, tracing the evolution from initial blockchain-based process execution systems to contemporary solutions addressing flexibility and interoperability challenges. We review existing work on blockchain-based business process execution, approaches for enhancing flexibility, and model-driven engineering approaches to propose the solution to the cental **research question** of this thesis.

> "*How can we achieve flexible execution of BPMN choreographies on blockchain without compromising the trustworthiness provided by its decentralized and immutable nature?*"

The goal is to highlight the advancements made by prior research and, crucially, to identify the specific gap that this thesis aims to fill. Through this comprehensive analysis, we identify the specific research challenge that this thesis addresses—the need for a native, secure, and efficient mechanism for flexible execution of business processes on blockchain without compromising trustworthiness. We systematically review related work across four key dimensions:

- Blockchain-based business process execution.
- Choreography-oriented blockchain implementations.
- Flexible Choreography Execution support in Blockchain Systems.
- Model-Driven Engineering and Code Generation.
- AI-Based Code Transpilation.

3.1 Blockchain-Based Business Process Execution

The seminal work in blockchain-based business process execution began with recognizing the technology's potential beyond cryptocurrencies. Mendling et al. [85] offered a foundational analysis of the opportunities and challenges of integrating blockchain with Business Process Management (BPM). Their survey underscores blockchain's capacity to enhance trust through transparent, immutable execution environments and explains how smart contracts can enforce process compliance in decentralized settings. They also examine the tension between transparency and privacy, noting that public process execution data demands careful design of monitoring logic. This work remains a cornerstone in understanding blockchain's implications for BPM, particularly regarding trust, transparency, and compliance.

Blockchain as a trust in business process execution

Research in [115, 41, 100] explores blockchain integration within multi-stakeholder business process environments requiring mutual trust but without centralized control. Their studies focus on how blockchain infrastructures enable trusted collaboration by tamper-proofing records and ensuring synchronized, reliable process states across independent parties. They discuss governance mechanisms and trust management over blockchains, which are crucial for successful business collaborations in decentralized settings. This work contributes to framing blockchain not only as a technological enabler but also as a basis for organizational and process trust in collaborative BPM.

Application-Oriented business process execution

Tripathi [111] and Çodur [25] examine blockchain's strategic role in supply chain management and broader business ecosystems. Tripathi's review provides insights into how blockchain facilitates transparency, traceability, and stakeholder synchronization in complex supply chains, reducing fraud and improving operational efficiency. Çodur's survey complements this by analyzing implementation guidelines, technical architectures, and practical challenges in deploying blockchain in diverse business ecosystems. These reviews position blockchain as a critical infrastructure component for modern digital supply chains and interconnected business processes, emphasizing practical aspects such as standardization, scalability, and interoperability.

Model-Driven Approaches to Business Process Execution

The Caterpillar framework [70] marked a significant milestone as the first comprehensive solution for executing BPMN collaboration models on Ethereum. Unlike earlier approaches that used blockchain merely as an immutable audit log, Caterpillar translated BPMN models directly into Solidity smart contracts that actively enforced process flow. This "compliance-by-design" approach ensured that participants could not deviate from the predefined process model. However, Caterpillar's focus on collaboration diagrams—which mix public message exchanges with internal participant tasks—resulted in unnecessary exposure of private logic on the blockchain, violating the principle of information hiding in inter-organizational settings.

Lorikeet [110] extended this paradigm by addressing asset-intensive business processes. The framework automatically generated Solidity smart contracts from business process models while also creating a registry data schema for managing non-fungible assets. This was particularly relevant for supply chain and property management applications where process execution and asset ownership are intertwined. Despite these advancements, Lorikeet inherited Ethereum's fundamental limitations regarding scalability and flexibility.

Practical Aspects of Adopting Blockchain for BPM

Practical industry assessments [99, 89] provide actionable insights for real-world blockchain BPM adoption. These reports discuss best practices for blockchain architecture design, security governance, and integration strategies in enterprise settings. They highlight the importance of clear governance policies, risk management frameworks, and compliance mechanisms to ensure blockchain deployments deliver business value while adhering to regulatory and security requirements. Such assessments bridge the gap between academic research and industrial practice, guiding organizations on how to leverage blockchain effectively for business process innovation and automation.

These early systems demonstrated the viability of blockchain for process execution but revealed critical challenges: high transaction costs, limited throughput, and most importantly, the inherent conflict between process flexibility and blockchain immutability. As Weber et al. [35] later demonstrated, these limitations significantly impacted the practical adoption of blockchain-based BPM solutions in enterprise settings.

Comparision with our work While prior blockchain-based BPM approaches primarily relied on Ethereum or similar general-purpose platforms, our work leverages Algorand to overcome several limitations identified in earlier systems such as Caterpillar and Lorikeet. Unlike Ethereum-based

solutions that face high transaction fees, limited throughput, and variable finality times, our approach benefits from Algorand's low-cost transactions, high performance, and instant finality, which enable more efficient and predictable process execution. Moreover, Algorand's Layer-1 smart contract update mechanisms support secure, stateful process enforcement without exposing unnecessary internal logic on-chain, enabling stronger privacy and clearer separation of concerns in inter-organizational settings. By exploiting these features, our work offers a more privacy-preserving, cost-effective, and scalable foundation for blockchain-based business process execution, addressing long-standing challenges that have hindered broader enterprise adoption.

3.2 Choreography-Oriented Blockchain Implementations

Choreography-based specifications have been widely investigated for developing multi-party distributed systems. EU-funded projects such as CHOReOS[1][11] and CHOREVOLUTION[2][11] highlighted the relevance of choreographies for inter-organizational busiess processes and the need for lifecycle management infrastructures. Recognizing the limitations of collaboration diagrams for inter-organizational processes, subsequent research shifted toward BPMN choreography diagrams, which exclusively model the public message exchanges between participants. This approach aligns perfectly with blockchain's transparency principle while preserving participant privacy.

Model-Driven Frameworks for Choreography Execution

ChorChain [30, 35] emerged as a comprehensive model-driven framework specifically designed for choreography-based systems. The framework supported the entire lifecycle—from modeling choreographies to generating Solidity contracts, executing processes, and enabling auditing. ChorChain generated a separate smart contract for each choreography instance, with functions representing message exchanges that could only be executed when permitted by the process state. This ensured that all interactions adhered to the choreography model.

Model- and Rule-Based Frameworks for Choreography Execution

The framework [52] introduces a model-driven approach for choreographic programming, enabling developers to specify choreographies with high-level

[1] https://cordis.europa.eu/project/rcn/96288/factsheet/en
[2] http://www.chorevolution.eu/bin/view/Main/

syntax and automatically generate correct-by-construction local endpoints through EndPoint Projection (EPP) techniques. This approach ensures faithful execution of choreographic protocols and supports modular programming via choreographic types.

ChorChain [36] is a model-driven framework that translates BPMN choreography models into blockchain-based smart contracts, enforcing choreography logic and ensuring correct sequence of actions across multiple parties. It leverages blockchain for trustable and auditable execution.

AIOCJ [40] presents a choreographic framework for safe adaptive distributed applications, focusing on rule-based adaptation and runtime modification of choreographies.

Recent approaches also explore combining rule-based techniques with blockchain, allowing for run-time updates of choreography rules without being limited by blockchain immutability, thus supporting flexible and certified execution of choreographic processes

Multi-Blockchain Frameworks for Choreography Execution

An approach to multi-blockchain frameworks for choreography execution is presented in [28]. The work introduces a model-driven methodology that enables the modeling, refinement, deployment, and execution of choreographies across multiple blockchains, supporting the entire lifecycle of inter-organizational business processes. The framework, often exemplified by systems like ChorChain, translates BPMN choreography models into blockchain-specific smart contracts, ensuring the correct sequence of actions and trustable execution across distributed participants. Such frameworks leverage blockchain's auditability and enforce choreography logic through smart contracts, allowing for secure, auditable, and flexible multi-party process execution on multiple blockchain infrastructures.

Other approaches like TABS [16] employed different modeling formalisms, specifically Discrete Event-Hierarchical State Machines (DE-HSM), and supported multiple blockchain platforms including Ethereum and Hyperledger Fabric. While innovative, TABS and similar frameworks continued to face the fundamental challenge of updating deployed contract logic to accommodate process changes.

A common limitation across these choreography-focused approaches was their reliance on Ethereum and similar platforms that lack native support for smart contract updates. This forced researchers to either accept rigidity or develop complex workarounds that often compromised security or decentralization.

Comparision with our work In contrast, our work leverages Algorand's native smart contract update capabilities, which allow process logic to evolve

without sacrificing trust assumptions or system integrity. This enables a more flexible and sustainable choreography execution environment where corrections, adaptations, and versioning can be handled securely and transparently. Furthermore, the platform's high throughput, low fees, and instant finality mitigate practical deployment barriers faced by earlier Ethereum-based approaches. Consequently, our solution provides a more adaptable, performant, and operationally viable foundation for blockchain-based choreography execution.

3.3 Flexible Choreography Execution support in Blockchain Systems.

The challenge of supporting flexibility during the execution of process-aware information systems has been widely studied, resulting in numerous systems providing different forms of run-time and design-time adaptation [104, 84]. Existing approaches vary considerably depending on the execution environment and the type of flexibility required. Corradini et al. [39, 31, 32, 33] further advanced this direction by exploring various aspects of multi-party business process execution on blockchain. Their work on FlexChain [33] introduced a rule-based approach where choreography rules were stored on-chain but evaluated off-chain, attempting to balance trust and flexibility. However, this introduced a trusted off-chain component, partially undermining blockchain's trust assumptions.

Context aware & behavioural flexibility

A real-time flexibility mechanism for cloud-service workflows is presented in [51]. The authors combine BPMN and UML to capture both functional and behavioural aspects and monitor predefined Quality of Service (QoS) attributes. Detected anomalies trigger corrective actions during execution, enabling adaptive workflow behaviour.

Context-aware run-time adaptation is explored in [113]. Here, business processes rely on rules that are dynamically selected and executed based on contextual information, enabling behavioural adjustments when execution conditions evolve.

Controlled and Role-Based Flexibility

Controlled flexibility is addressed in [71], where BPMN is extended through the CF4BPMN notation. This extension allows designers to specify which process elements may change at run-time and under which formally defined

constraints. The approach gives precise control over variability but requires designers to anticipate possible changes beforehand.

A related contribution is found in [72], which introduces dynamic role binding for run-time sub-process selection. The mechanism allows actors to be bound or unbound from roles based on consistency checks and agreements between participants. While effective for switching among predefined alternatives, it focuses on role management rather than enabling structural run-time adaptation, which is the core aim of the work proposed in this thesis.

Together, these approaches demonstrate that flexibility can be introduced through anomaly-handling mechanisms, controlled change specifications, version management, or context-aware logic. However, they differ substantially from our work, whose primary objective is to ensure flexibility during the execution of blockchain-based processes while maintaining trust—an area that remains comparatively underexplored.

Versioning & Upgradeability Flexibility

Process versioning is examined in [63], where dynamic updates to business processes are managed through BPEL extensions and structured change patterns. This method offers traceable support for evolving process definitions but primarily targets version management rather than flexible execution. Upgradeability in blockchain systems is addressed in [64], where each participant in a BPMN collaboration diagram is transformed into a separate smart contract. Flexibility is achieved through versioning structures and proxy contracts that forward calls to updated logic upon approval. However, this architecture introduces high deployment complexity and focuses on updating inactive instances. In contrast, our approach targets run-time modifications for active process executions.

The broader tension between blockchain immutability and business process flexibility has generated substantial research. On Ethereum, the predominant solution involves proxy patterns [18], such as UUPS and Transparent Proxy models, which delegate execution to updateable logic contracts while maintaining persistent state. Despite their popularity, empirical studies reveal significant limitations. Zhang et al. [123] identify vulnerabilities arising from storage clashes, function-selector collisions, and initialization errors, while Li et al. [69] observe that many upgradeable contracts are never updated in practice, indicating that the complexity often deters real use. Tools like PROXYEX [123] and USCDetector [69] were developed to detect such vulnerabilities, highlighting the inherent risks of proxy-based upgradeability.

More recently, researchers have begun investigating next-generation blockchain platforms to overcome the limitations of proxy-based upgradeability. Malik et al. [74] analyse upgradeability patterns and their implications

for business processes, whereas Klinger et al. [65] propose a collaborative upgradeability concept still dependent on complex contract patterns.

Technological/Off-Chain Flexibility

In [2], flexibility is pursued at the technological layer of process execution. By integrating on-chain and off-chain components within a federated blockchain environment, the authors address interoperability and heterogeneous platform interaction. This differs from our proposed approach of enabling flexibility within the business process execution itself, rather than at the infrastructure level.

Off-chain approaches provide an alternative. Corradini et al. [34] propose a hybrid architecture in which a flexible off-chain engine executes processes while relying on blockchain only for validation and auditing. Adams et al. [1] similarly combine on-chain and off-chain components within federated settings. Although these strategies enhance flexibility, they reintroduce trusted intermediaries, thereby undermining the decentralised trust model central to blockchain systems.

Within BPM specifically, Lopez-Pintado et al. [71] explore controlled flexibility through dynamic role binding, enabling runtime selection of participants for specific tasks. While suitable for certain scenarios, this approach supports only limited forms of flexibility and does not permit structural changes to the process model.

Source	BPMN	Blockchain	Flexibility
[51]	✓	✗	Scaling deployed services during the workflow
[83]	✓	✗	Modelling and executing controlled flexibility
[63]	✗	✗	Dynamic process version selection
[113]	✓	✗	Rule- and context-based modelling and simulation
[72]	✓	✓	Run-time selection of predesigned elements
[2]	✓	✓	Dynamic blockchain selection
[64]	✓	✓	Upgradeability and versioning approach
[71]	✓	✓	Policy-based dynamic control-flow
[31]	✓	✓	Rule-driven runtime modification
[60]	✓	✓	Verified BPMN-driven contract generation
[106]	✓	✓	Flexible decision logic integration

Table 3.1: Table of identified related works and their characteristics.

Table 3.1 summarises the analysed works concerning flexibility and considering three main characteristics: (i) if the work explicitly uses BPMN diagrams for modelling, (ii) if the work relies on blockchain, and (iii) which flexibility aspects the work supports, and how it implements them. The table shows that in most cases flexibility is achieved in a setting that does not rely on blockchain technology, hence without facing all issues raised by

the immutability of the blockchain. In the approaches that, instead, adopt blockchain for a trusted environment, the concept of flexibility they consider differs from work to work. In [2], the authors consider flexibility as the possibility to select different blockchains, while in [72], the authors provide the possibility to bind actors and control-flow decisions at run-time. In [64], the work focuses on the smart contract upgrading mechanism, thus proposing a versioning approach. Differently, our work focused on a richer flexibility mechanism for business processes, whose business logic and execution state is stored in the blockchain, supporting a voting mechanism that permits arbitrary changes in the model at run-time.

Comparision with our work Across the literature, flexibility in blockchain-based business processes is typically achieved either by delegating adaptation to off-chain components, by restricting flexibility to predefined alternatives, or by employing complex upgradeability patterns such as proxies. These strategies, while effective in specific contexts, introduce significant drawbacks: off-chain engines weaken decentralisation and the trust guarantees of blockchains; controlled or role-based approaches limit flexibility to designer-specified options; and proxy-based upgradeability on Ethereum incurs substantial complexity and security risks, as shown by empirical analyses identifying vulnerabilities and low real-world adoption of contract updates.

In contrast, our work provides a richer, fully on-chain flexibility mechanism that maintains the trust model of blockchain without relying on intermediaries or fragile upgrade patterns. By leveraging Algorand's native smart contract update capabilities, we enable arbitrary, model-level modifications at run-time—even for active process instances—while preserving correctness, traceability, and decentralised governance. Our policy-based adaptation protocol further ensures that process evolution remains collaborative and tamper-resistant, avoiding the rigidity of static smart contracts and the operational complexity seen in many Ethereum-based solutions. Consequently, our approach offers a more robust, decentralised, and practically deployable foundation for flexible choreography execution on blockchain.

3.4 Model-Driven Engineering and Code Generation

The translation of high-level business process models to executable smart contract code represents another active research area. Model-Driven Engineering (MDE) approaches have demonstrated significant potential for reducing development effort and ensuring consistency between models and implementations.

The Pupa approach [87] extended Caterpillar by supporting additional BPMN elements, including timer events and inclusive gateways. Similarly, Curty et al. [39] conducted a comprehensive review of MDE and low-code/no-code platforms for blockchain application development, identifying patterns and best practices across numerous frameworks.

Domain-Specific Modeling Languages

To bridge the gap between business models and smart contract code, the BPFM framework [42] provides a feature-modeling approach to map business variability to blockchain configurations, offering systematic derivation of tailored smart contract implementations. This modularity provides several advantages: it enhances code understandability, facilitates verification, and crucially, supports targeted updates when processes evolve. When a choreography element changes, only the corresponding function needs modification, unlike monolithic translation approaches that require complete redeployment.

Executable Process Models

Executable business process formalisms have also been integrated directly with blockchain back-ends. Weber et al. [119] pioneered an approach to execute business collaborations on blockchain by transforming BPMN models into verifiable smart contracts that manage process state transitions. This line of work influenced subsequent research such as TRANSAX [67], which studies how blockchain-based execution can enhance inter-organizational workflow trust and auditability.

Model-Driven Smart Contract Synthesis

Research has also explored systematic generation of smart contracts from higher-level specifications. The UML-to-smart-contract pipeline proposed by Marchesi et al. [82] uses UML statecharts to capture contract logic and automatically derive Solidity code. Additionally, the B-MERODE framework [10] leverages model-driven object-oriented analysis to generate secure smart contracts with explicit behavioral constraints.

Comparision with our work Model-driven approaches for generating executable blockchain-based process logic—such as Pupa, BPFM, Weber et al.'s BPMN-to-contract transformation, and UML- or MERODE-based synthesis—have demonstrated the value of systematic translation from high-level models to smart contract code. However, these works predominantly generate static, non-updatable smart contracts on platforms like Ethereum,

inheriting rigidity whenever process logic evolves or requires correction. Even frameworks that support modularity, such as BPFM, are limited by underlying infrastructures that do not natively support secure contract updating, forcing full redeployment or complex indirection mechanisms.

In contrast, our work combines a platform-agnostic model-driven translation pipeline with the advantages of Algorand's native smart contract update features, enabling safe and efficient evolution of deployed process logic. The generation of pseudocode independent of any specific smart contract language further enhances portability, allowing the same methodology to target multiple platforms—including Solidity and Algorand Python—without reengineering the core translation approach. This provides a more adaptable and maintainable foundation for blockchain-based process execution than prior framework-specific MDE solutions.

3.5 AI-Based Code Transpilation

This section examines existing research on approaches to code translation, with particular emphasis on evaluations in the blockchain domain. We focus on how researchers have assessed LLM performance for code translation tasks and the methodologies employed. The reviewed works are particularly relevant because they are related to our LLM-based solution for migrating business processes implemented as solidity smart contracts to the algorand platform in order to enable their flexibility.

Rule-Based Transpilers

Early approaches to code translation primarily relied on rule-based transpilers, which apply explicitly defined transformation rules to convert code from one language to another. These systems provide predictable and deterministic behavior by enforcing strict syntactic mappings between source and target languages. Examples include traditional Python-to-Java and Java-to-Python transpilers such as SolMover [62]. While rule-based systems can achieve high accuracy for well-structured languages, they require extensive programming expertise to craft and maintain rule sets. As languages evolve, the rule base becomes increasingly brittle, often failing to handle complex constructs, semantic subtleties, or idiomatic patterns. Consequently, such systems frequently require manual intervention, limiting their scalability and applicability to modern, dynamic languages—especially in contexts such as blockchain development, where subtle semantic differences can significantly affect execution behavior.

Statistical Machine Learning–Based Transpilers

Before deep learning became prominent, statistical ML techniques represented the dominant paradigm for automated code translation. These methods typically learned translation patterns from parallel corpora using phrase-based or grammar-enhanced models. Nguyen et al. [90] introduced one of the earliest phrase-based source code migration techniques, which was later refined by Karaivanov et al. [61] through the incorporation of grammatical contexts and handwritten rewriting rules. Aggarwal et al. [3] demonstrated that similar statistical approaches can support language migrations such as Python 2 to Python 3. Additional contributions include bidirectional transpilers [105] and systems like CRustS, which focus on reducing unsafe expressions when translating between memory-unsafe and memory-safe languages. Although statistical models improved flexibility compared to rigid rule-based systems, they remained highly dependent on large parallel datasets and often struggled to capture deep semantic dependencies or complex control-flow relationships.

Transformer-Based Code Translation

The introduction of transformer architectures brought a significant leap forward in code translation. The encoder–decoder attention mechanism introduced by Vaswani et al. [114] enabled models to capture long-range relationships and structural patterns more effectively than previous techniques. This innovation led to a wave of specialized transformer models for code. CodeBERT [49] and CodeGPT [73] enhanced bilingual and autoregressive code understanding, while CodeT5 [118] incorporated semantic-aware objectives for improved translation fidelity. PLBART [4] extended denoising autoencoding methods to handle cross-domain translation. Beyond these, structural models such as tree-to-tree transformers [21] enriched translation quality by explicitly incorporating abstract syntax tree (AST) information. These approaches achieved more semantic consistency than pure sequence-based models, marking a strong step toward automated, high-quality translation.

LLM-Based Approaches with Testing and Verification

Large Language Models (LLMs) further evolved the translation landscape by combining syntactic pattern learning with powerful contextual reasoning. They outperform earlier approaches by handling long-range dependencies, inferring semantics, and adapting to novel languages through few-shot prompting. To improve reliability, several researchers integrated LLMs with program testing or formal verification. Roziere et al. [103] used unit tests to generate synthetic parallel datasets, while Wang et al. [117] applied reinforcement

learning using compiler feedback. Other studies examined the translation capability and reliability of LLMs. Pan et al. [55] evaluated their effectiveness across languages, whereas Orlanski et al. [96] studied their performance for low-resource languages. Pan et al. [97] highlighted that LLM-generated translations frequently introduce subtle bugs, underscoring the importance of verification mechanisms. More recent works integrate theorem proving to ensure correctness. Karanjain et al. [62] used the Move Prover to validate Solidity-to-Move translations, demonstrating that structured prompting and fine-tuning enable accurate results even for underrepresented languages. Bhatia et al. [14] proposed converting both translated code and its proofs into a theorem prover's syntax to verify functional equivalence. Yang et al. [122] developed Vert, a tool that generates readable Rust translations while ensuring formal correctness. Lachaux et al. [66] additionally showed that unsupervised models can achieve competitive translation quality across C++, Java, and Python—with most errors requiring minimal manual correction. Surveys such as Huynh et al. [56] identify persistent issues like semantic drift, syntactic errors, and vulnerability introduction. Wang et al. [59] reinforce this by showing that LLM-generated code often contains vulnerabilities that are difficult for models to repair.

LLMs for Code Translation

LLMs have transformed code translation by enabling prompt-based and few-shot translation capabilities that frequently outperform both rule-based and statistical approaches. Rooted in the advances demonstrated by GPT-3 [46], LLMs can generalize from limited examples and produce translations that closely follow both syntax and intent. Evaluation metrics have also evolved, with modern approaches assessing code through compilation success, unit tests [103], formal verification [14], and expert review. Despite these strengths, LLMs continue to struggle with inconsistent semantics, incorrect handling of idiomatic constructs, and vulnerability-prone patterns. Huynh et al. [56] and Zhang et al. [47] demonstrate that simply increasing the number of examples in a prompt does not always improve translation quality—highlighting the need for carefully designed prompt engineering and verification pipelines.

LLMs in Blockchain Development

Smart contract translation represents one of the most challenging domains for LLM-based translation due to the strict execution, determinism, and security requirements in blockchain environments. Karanjain et al. [62] investigated LLM-driven translations of Solidity into Move, demonstrating that even languages underrepresented in training corpora can be translated effectively

when combined with structured prompting and the Move Prover. Their results show that LLMs can assist in building safer smart contracts, provided that formal verification accompanies the translation process. Similarly, Yang et al. [122] introduced Vert, which leverages LLMs to generate Rust smart contracts that adhere to formal correctness guarantees—exceeding the typical reliability standards of purely generative models. However, blockchain also amplifies the risks associated with unsafe translations. Wang et al. [59] reveal that LLMs frequently output vulnerable smart contract code and often fail to adequately repair security flaws, even when explicitly informed about vulnerabilities. These findings emphasize that while LLMs offer promising capabilities for cross-blockchain translation, reliable deployment requires hybrid approaches combining AI with symbolic reasoning and domain-specific verification tools.

Comparision with our work Our investigation of LLM-assisted Solidity-to-Algorand translation [81] represents the first systematic study focused on this specific language pair. The architectural differences between Ethereum and Algorand—particularly in state management, transaction handling, and type systems—present unique challenges that general-purpose LLMs struggle to overcome. Our contribution of a structured feature mapping and hybrid human-AI workflow addresses these challenges practically.

LLM-based translation has surpassed rule-based and statistical methods in flexibility and automation. Transformer models introduced semantic awareness, while modern LLMs incorporate reasoning, prompting, and verification techniques to further enhance translation reliability. However, challenges persist, including maintaining semantic equivalence, avoiding vulnerabilities, and ensuring domain-specific correctness—issues that are especially acute in blockchain development. Ongoing research combining LLMs with formal verification (e.g., Move Prover, theorem provers, Rust verification tools) indicates a promising path toward safer, more reliable code translation across heterogeneous programming ecosystems.

PART III

FLEXIBLE EXECUTION OF BUSINESS PROCESSES

CHAPTER 4

BPMN TO SMART CONTRACT TRANSLATION

This chapter presents the translation from BPMN choreography to smart contract pseudocode, achieving the first objective of this research.

> "*To develop a framework for translating BPMN choreography models into blockchain smart contracts that is modular and driven by BPMN choreography formalization.*"

The aim is to bridge high-level business workflow models and their trustworthy execution by transforming BPMN choreography models into modular smart-contract structures. More specifically, the target of the translation is not the smart contract language of a specific blockchain platform, but a pseudocode that can be easily mapped to different smart contract languages. This serves as an intermediate platform agnostic layer before generating smart-contract logic for a specific blockchain platform, enabling transparency, flexibility, and traceability in decentralized process execution.

Business Process Management (BPM) increasingly deals with distributed systems where independent parties collaborate to achieve shared goals. The BPM community standardized BPMN [91]. Among BPMN perspectives, choreography diagrams are essential, as they capture message-based interactions among participants without revealing internal process logic. This property makes them suitable for decentralized and cross-organizational settings.

Blockchain technology has emerged as a trusted execution layer for multi-party processes, particularly where inherent trust cannot be assumed [57]. Its decentralized infrastructure ensures tamper-proof state storage, transparent audit trails and trust-free coordination between parties. Choreography

models can be compiled into smart contracts [30, 70], enforcing process rules and enabling verifiable history of interactions. In this context, BPMN choreographies can be implemented as blockchain smart contracts, enabling verifiable process history and automated enforcement of interaction protocols [30, 70]. Blockchain-based execution ensures all participants adhere to the agreed choreography, making malicious deviations or unauthorized modifications detectable. Consequently, blockchain acts as both the execution engine and the audit layer for multi-party workflows, improving accountability and trustworthiness in distributed settings.

Overall, this chapter introduces the transformation method that bridges high-level BPMN choreography models to smart contract logic, laying the foundation for trustworthy, adaptable, and formally grounded blockchain-based process automation. To ensure broad applicability, our translation approach is blockchain-agnostic. Instead of directly producing code in a specific smart-contract language, we generate pseudocode that can be consistently rendered into mainstream languages such as Solidity and Algorand python. This allows us to evaluate the translation on two distinct blockchain platforms—Polygon (EVM-based) and Algorand. Experimental results demonstrate that the proposed approach achieves modularity and semantic fidelity while keeping execution costs affordable on modern blockchain infrastructures.

4.1 BPMN Choreographies

In BPMN, both collaboration and choreography diagrams serve to model interactions among multiple participants, but they do so from different perspectives. A collaboration diagram represents several interacting processes, each modeled within its own pool. Message flows between pools illustrate how these independent processes exchange information. This representation is particularly suitable when internal process logic must be captured alongside external communication, for example, in enterprise-to-enterprise integrations or partner network scenarios.

In contrast, a choreography diagram abstracts the internal details of each participant and focuses purely on the exchange of messages that define the observable interaction protocol. Every task in a choreography represents a message exchange between two participants, explicitly showing who sends and who receives. This interaction-centric view provides a concise and global perspective of the entire collaboration, ensuring that all participants share a consistent understanding of expected communication behavior. Such abstraction is especially valuable in decentralized or trustless environments—like blockchain-based workflows—where only the interaction logic is shared and executed collaboratively, while internal processes remain private. Table 4.1

presents the comparison between the two models.

Aspect	**Collaboration**	**Choreography**
Definition	Interactions among processes with internal workflows.	Sequence of message exchanges without internal logic.
Perspective	Process-centric—focus on process communication.	Interaction-centric—focus on message order.
Control	Each participant manages its own logic.	Coordination via agreed message flows.
Representation	Multiple pools showing activities and messages.	Single diagram of public message exchanges.
Complexity	Can be complex with many message flows.	Simpler, emphasizes interaction semantics.
Use Cases	Detailed inter organizational workflows.	Decentralized, trustless systems.

Table 4.1: Comparison of Collaboration and Choreography in BPMN Models

In this work, BPMN choreography diagrams are adopted as the primary modeling notation to represent multiparty business workflows. In a choreography, each task models a communication step where one participant initiates an interaction and another responds. Sequence flows define the execution order among tasks, while gateways express branching, looping, or parallel execution behavior. By relying on choreographies, our approach ensures that the translation to pseudocode—and later to smart contract logic—captures only inter-participant behavior, preserving privacy of internal procedures while guaranteeing collaborative correctness.

4.1.1 Choreography Modeling Example

To illustrate core BPMN choreography constructs, we refer to the model in Fig. 4.1, drawn from [33]. The example represents a healthcare process where a patient schedules and completes an X-ray examination. Blockchain is particularly relevant in this scenario as it increases transparency and auditability of the healthcare workflow, ensuring the authenticity of interactions and preventing disputes among stakeholders.

The choreography begins with a *start* event, represented as a circle with an outgoing edge. Sequence flows connect choreography elements and define execution progress. The first task models the interaction where the patient, holding a medical prescription, requests an appointment from the radiology

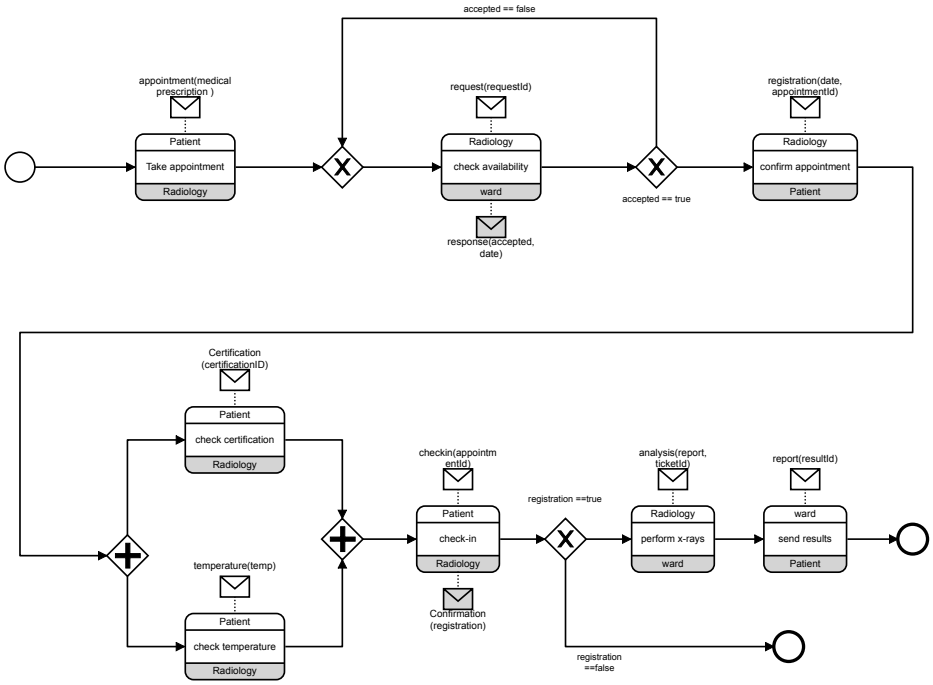


Figure 4.1: X-ray exam choreography.

department. A *one-way choreography task* captures this communication, represented as a rectangle divided into three bands: the middle band contains the task name, while the top and bottom bands denote the initiator and receiver, respectively. For clarity, Table 4.2 summarizes the main BPMN element categories referenced throughout this example.

Category	**BPMN Element Examples**
Events	Start, End, Intermediate
Activities	Tasks, Service Tasks
Gateways	XOR, AND, Event-based
Artifacts	Messages, Data objects
Connecting Objects	Sequence Flows, Message Flows, Associations
Data Elements	Data Store, Data Input, Data Output
Structure / Swimlanes	Pools, Lanes

Table 4.2: BPMN Elements by Category

Choreography execution is governed by *gateways*. An *exclusive (XOR) gateway*, represented by a diamond with a × symbol, models conditional branching. A XOR-split gateway activates exactly one outgoing path based

on conditions, while a XOR-join gateway merges flows without synchronization. In our running example, two XOR gateways specify a loop: the radiology department repeatedly checks ward availability until a valid appointment date is found.

Two-way choreography tasks represent message exchanges where both participants send information. Once availability is confirmed, the radiology department notifies the patient. Prior to physical check-in, pandemic-related safety checks require the patient to verify both temperature and vaccination status. These independent checks are modeled using *parallel (AND) gateways*, drawn with + symbols. The AND-split activates both branches, and the AND-join synchronizes them by waiting until both checks complete.

The patient then proceeds to physical check-in by submitting an appointment identifier, validated by the radiology department. If validation fails, an *end* event terminates the process. Otherwise, the patient undergoes the X-ray exam. The results are sent to the ward, which compiles a final report and forwards it to the patient, completing the choreography.

Notably, two XOR gateways are used to specify a sort of do-while loop: the radiology department checks the availability of the ward for the X-ray exam; if there is a free date, the appointment is confirmed; otherwise, the availability is checked again.

The request-response interaction between the radiology department and the ward is modeled by a *two-way choreography task*, in which both participants send a message. In case the X-ray analysis request is accepted, the response is sent to the patient. Before performing the physical check-in, due to pandemic-related entry restrictions, the patient must check his/her temperature and vaccine certification. These checks are performed in parallel, which is specified by using two *parallel (AND) gateways* (drawn with diamonds marked with the + symbol): the AND-split gateway activates all the outgoing edges simultaneously, while the join one has to wait to be reached by all its incoming edges to be activated. The In the *event-based gateway*, the outgoing branch activation depends on the reception of a message; the message events connected to the gateway are in race condition: the first one that is triggered activates the corresponding branch and disables the other ones.

The patient then performs the physical check-in, by providing the appointment identifier that is controlled by the radiology department, which returns the confirmation response. If the appointment is not confirmed, the choreography terminates. In this case the execution flow is directed by an XOR-split gateway to an *end* event, which is drawn as a circle with an incoming edge and denotes an ending point of the choreography. If instead the appointment is confirmed, the X-ray exam is done and the results are provided to the ward that compiles a final report, then sent to the patient.

4.2 BPMN Choreography Formalization

This section presents the formalization of BPMN choreographies we rely on. This formalization results from the extension of the one presented in [37] with additional features inspired by the formalization in [38].

Translating BPMN choreographies into smart contracts introduces several technical challenges due to the gap between graphical modeling abstractions and precise execution semantics. First, BPMN is intentionally expressive and informal, which can lead to semantic ambiguity when converting visual constructs into deterministic control logic. The distributed nature of choreographies adds complexity, as each participant executes independently and interacts through messages, requiring careful synchronization and ordering of communication events. Gateway constructs such as XOR and AND forks may lead to state-space explosion, especially in models with nested branching or concurrent paths. These factors make automated translation non-trivial and necessitate a semantics-driven, modular approach to preserve correctness, traceability, and readability in the resulting executable representation. Table 4.3 summarizes the BPMN-to-smart contract translation challenges.

Challenge	**Description**
Semantic Ambiguity	Graphical symbols lack execution logic
Distributed Nature	Multiple independent actors
State Explosion	Gateway combinations make the state space grow exponentially
Message Synchronization	Ensuring ordering & delivery
Traceability and Readability	Ensuring a one-to-one correspondence between BPMN elements and pseudocode

Table 4.3: Challenges in BPMN-to-smart contract translation

The graphical notation of BPMN choreographies makes unhandy the definition of the formal semantics. Therefore, the formalization is based on an alternative textual representation of choreographies, whose syntax is defined by the BNF grammar.

The BNF syntax of the choreography models structure is given in Fig. 4.2.

$$
\begin{aligned}
C ::= {} & \mathsf{start}(\mathsf{id}, \mathsf{e}_0, \mathsf{e}_1) \mid \mathsf{end}(\mathsf{id}, \mathsf{e}) \mid \mathsf{andSplit}(\mathsf{id}, \mathsf{e}, E) \\
& \mid \mathsf{andJoin}(\mathsf{id}, E, \mathsf{e}) \mid \mathsf{xorSplit}(\mathsf{id}, \mathsf{e}, X) \mid \mathsf{xorJoin}(\mathsf{id}, E, \mathsf{e}) \\
& \mid \mathsf{task}(\mathsf{id}, \mathsf{e}_1, \mathsf{p}_1, \mathsf{m}, \mathsf{p}_2, \mathsf{e}_2) \mid \mathsf{eventBased}(\mathsf{id}, \mathsf{e}, T) \mid C_1 | C_2 \\
X ::= {} & (\mathsf{e}, \mathsf{exp}) \mid X_1, X_2 \qquad\qquad T ::= (\mathsf{p}_1, \mathsf{m}, \mathsf{p}_2, \mathsf{e}) \mid T_1, T_2
\end{aligned}
$$

Figure 4.2: Syntax of BPMN Choreography Structures.

In the grammar, the non-terminal symbol C represents *Choreography Structures*, while the terminal symbols, denoted by the sans serif font, are the considered elements of a BPMN model, i.e. events, gateways and tasks. Non-terminal symbols X and T represent expression lists and event-based task lists, respectively; when convenient, we shall regard these lists simply as sets. As a matter of notation, let $\mathbb{E}$ be the set of *sequence edge* names, in the following $\mathsf{e} \in \mathbb{E}$ denotes an edge, while $E \in 2^{\mathbb{E}}$ a set of edges. Notice that the set $\mathbb{E}$ also includes spurious edges for denoting the enabled/disabled status of start events.

Moreover, the set $\mathbb{M}$ of *messages* is ranged over by m, while the set $\mathbb{V}$ of *values* is ranged over by v. Notation p ranges over names uniquely identifying *participants*, while id ranges over *element identifiers* (these unique identifiers are not visualized in the graphical representation of the diagram, but are included in the underlying XML file). We also use a set $\mathbb{EXP}$ of *expressions*, ranged over by exp, whose precise syntax is deliberately not specified; we just assume that expressions contain message parameters and values. Such design choice has been influenced by the fact that the expression language operating on data is left unspecified even by the BPMN standard.

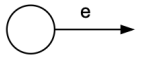
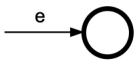
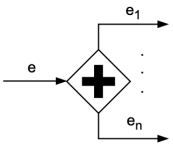
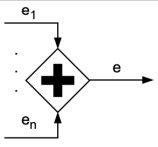
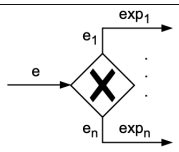
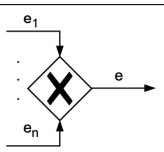
Graphical Notation	**Textual Representation**	**Graphical Notation**	**Textual Representation**
	$\mathsf{start}(\mathsf{id}, \mathsf{e}_0, \mathsf{e})$		$\mathsf{end}(\mathsf{id}, \mathsf{e})$
	$\mathsf{andSplit}(\mathsf{id}, \mathsf{e}, \{\mathsf{e}_1, \ldots, \mathsf{e}_n\})$		$\mathsf{andJoin}(\mathsf{id}, \{\mathsf{e}_1, \ldots, \mathsf{e}_n\}, \mathsf{e})$
	$\mathsf{xorSplit}(\mathsf{id}, \mathsf{e}, \{(\mathsf{e}_1, \mathsf{exp}_1), \ldots$		$\mathsf{xorJoin}(\mathsf{id}, \{\mathsf{e}_1, \ldots, \mathsf{e}_n\}, \mathsf{e})$

Figure 4.3: BPMN graphical and textual notation of Control Flow Elements.

Notably, we are not proposing a new modeling formalism, but we are only using a textual notation for the BPMN elements. With respect to the graphical notation, the textual one is more manageable for supporting the formal definition of the semantics and its implementation. The one-to-one correspondence between the graphical notation of BPMN and the syntax used for contol-flow is presented in Fig 4.3, tasks is shown in Fig 4.4, and events is shown in Fig 4.5, We will only consider terms of the textual syntax that are derived from BPMN models.

Notice that two-way tasks are rendered in our formal framework as pairs of one-way tasks, hence they are not explicitly included in the syntax.

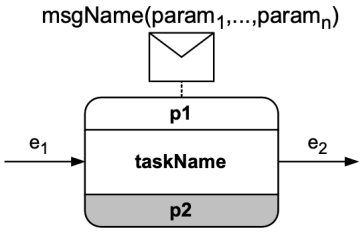
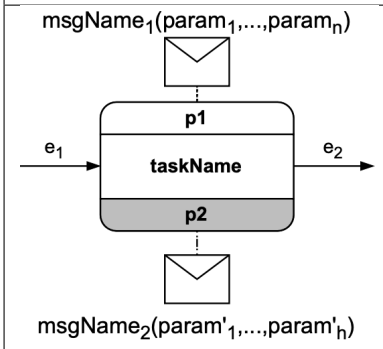
Graphical Notation	Textual Representation
	$\mathsf{task}(\mathsf{taskName}, \mathsf{e}_1, \mathsf{p}_1, \mathsf{msgName}(\mathsf{param}_1, \ldots, \mathsf{param}_n), \mathsf{p}_2, \mathsf{e}_2)$
	$\mathsf{task}(\mathsf{taskName}, \mathsf{e}_1, \mathsf{p}_1, \mathsf{msgName}_1(\mathsf{param}_1, \ldots, \mathsf{param}_n), \mathsf{p}_2, \mathsf{e})$ | $\mathsf{task}(\mathsf{taskNameResp}, \mathsf{e}, \mathsf{p}_2, \mathsf{msgName}_2(\mathsf{param}'_1, \ldots, \mathsf{param}'_h), \mathsf{p}_1, \mathsf{e}_2)$

Figure 4.4: BPMN graphical and textual notation of Task Elements.

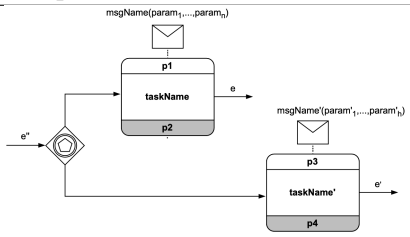
Graphical Notation	Textual Representation
	$\mathsf{eventBased}(\mathsf{id}, \mathsf{e}'', \{ (\mathsf{p}_1, \mathsf{msgName}_1(\mathsf{param}_1, \ldots, \mathsf{param}_n), \mathsf{p}_2, \mathsf{e}_1), (\mathsf{p}_3, \mathsf{msgName}_2(\mathsf{param}'_1, \ldots, \mathsf{param}'_h), \mathsf{p}_4, \mathsf{e}_2), \ldots (\mathsf{p}_k, \mathsf{msgName}_k(\mathsf{param}_1^{(k)}, \ldots, \mathsf{param}_m^{(k)}), \mathsf{p}_l, \mathsf{e}_k) \})$

Figure 4.5: BPMN graphical and textual notation of Event-Based Gateways.

To achieve a *compositional definition*, each sequence edge of the BPMN model is split in two parts: the part is outgoing from the source element, and the part is incoming into the target element. The two parts are correlated by means of unique sequence edge names in the BPMN model.

Example 1. *Let us consider the BPMN choreography model in Fig. 4.1. The textual representation of its structure is as follows (for reader's convenience, we use* e_i*, with* i *a natural number, to denote sequence edges,* $\mathsf{m}_{\mathsf{msgName}}$ *to*

denote messages, and $\mathsf{p_p}, \mathsf{p_r}, \mathsf{p_w}$ *to denote the three participants):*

$$
\begin{aligned}
&\mathsf{start}(\mathsf{start1}, \mathsf{e}_0, \mathsf{e}_1)\\
&\mid \mathsf{task}(\mathsf{takeAppointment}, \mathsf{e}_1, \mathsf{p_p}, \mathsf{m_{appointment}}, \mathsf{p_r}, \mathsf{e}_2)\\
&\mid \mathsf{xorJoin}(\mathsf{xor1}, \{\mathsf{e}_2, \mathsf{e}_6\}, \mathsf{e}_3)\\
&\mid \mathsf{task}(\mathsf{checkAvailability}, \mathsf{e}_3, \mathsf{p_r}, \mathsf{m_{request}}, \mathsf{p_w}, \mathsf{e}_4)\\
&\mid \mathsf{task}(\mathsf{checkAvailabilityResp}, \mathsf{e}_4, \mathsf{p_w}, \mathsf{m_{response}}, \mathsf{p_r}, \mathsf{e}_5)\\
&\mid \mathsf{xorSplit}(\mathsf{xor2}, \mathsf{e}_5, \{(\mathsf{e}_6, \mathsf{response.accepted} == \mathsf{false}),\\
&\qquad\qquad (\mathsf{e}_7, \mathsf{response.accepted} == \mathsf{true})\})\\
&\mid \mathsf{task}(\mathsf{confirmAppointment}, \mathsf{e}_7, \mathsf{p_r}, \mathsf{m_{registration}}, \mathsf{p_p}, \mathsf{e}_8)\\
&\mid \mathsf{andSplit}(\mathsf{and1}, \mathsf{e}_8, \{\mathsf{e}_9, \mathsf{e}_{10}\})\\
&\mid \mathsf{task}(\mathsf{checkCertification}, \mathsf{e}_9, \mathsf{p_p}, \mathsf{m_{certification}}, \mathsf{p_r}, \mathsf{e}_{11})\\
&\mid \mathsf{task}(\mathsf{checkTemperature}, \mathsf{e}_{10}, \mathsf{p_p}, \mathsf{m_{temperature}}, \mathsf{p_r}, \mathsf{e}_{12})\\
&\mid \mathsf{andJoin}(\mathsf{and2}, \{\mathsf{e}_{11}, \mathsf{e}_{12}\}, \mathsf{e}_{13})\\
&\mid \mathsf{task}(\mathsf{checkIn}, \mathsf{e}_{13}, \mathsf{p_p}, \mathsf{m_{checkin}}, \mathsf{p_r}, \mathsf{e}_{14})\\
&\mid \mathsf{task}(\mathsf{checkInResp}, \mathsf{e}_{14}, \mathsf{p_r}, \mathsf{m_{confirmation}}, \mathsf{p_p}, \mathsf{e}_{15})\\
&\mid \mathsf{xorSplit}(\mathsf{xor3}, \mathsf{e}_{15}, \{(\mathsf{e}_{16}, \mathsf{confirmation.registration} == \mathsf{true}),\\
&\qquad\qquad (\mathsf{e}_{17}, \mathsf{confirmation.registration} == \mathsf{false})\})\\
&\mid \mathsf{end}(\mathsf{end1}, \mathsf{e}_{17})\\
&\mid \mathsf{task}(\mathsf{performXRays}, \mathsf{e}_{16}, \mathsf{p_r}, \mathsf{m_{analysis}}, \mathsf{p_w}, \mathsf{e}_{18})\\
&\mid \mathsf{task}(\mathsf{sendResult}, \mathsf{e}_{18}, \mathsf{p_w}, \mathsf{m_{report}}, \mathsf{p_p}, \mathsf{e}_{19})\\
&\mid \mathsf{end}(\mathsf{end2}, \mathsf{e}_{19})
\end{aligned}
$$

The operational semantics we propose is given in terms of configurations of the form $\langle C, \sigma, \gamma\rangle$, where C is a choreography structure, $\sigma : \mathbb{E} \to \mathbb{N}$ is an *execution state function* mapping sequence edges to numbers of tokens ($\mathbb{N}$ is the set of natural numbers), and $\gamma : \mathbb{M} \to \mathbb{V}$ is a *message state function* mapping messages to values. The execution state obtained by updating in the state σ the number of tokens of the edge e to n, written as $\sigma \cdot \{\mathsf{e} \mapsto \mathsf{n}\}$, is defined as follows: $(\sigma \cdot \{\mathsf{e} \mapsto \mathsf{n}\})(\mathsf{e}')$ returns n if $\mathsf{e}' = \mathsf{e}$, otherwise it returns $\sigma(\mathsf{e}')$. The message state obtained by updating in the state γ the value of a message m to a value v, written as $\gamma \cdot \{\mathsf{m} \mapsto \mathsf{v}\}$, is defined similarly. The *initial* states of a choreography, where all sequence edges are unmarked (but for the spurious ones that are marked by one token) and all message values are undefined, are denoted respectively by σ_0 and γ_0.

The operational semantics is defined by means of a labelled transition system (LTS) on choreography configurations, formalising the execution of a choreography in terms of marking evolution and message exchanges. The LTS is a triple $\langle \mathcal{C}, \mathcal{L}, \to\rangle$ where: $\mathcal{C}$ is the set of choreography configurations; $\mathcal{L}$, ranged over by l, is the set of *labels* (of transitions that choreography configurations can perform); and $\to \subseteq \mathcal{C} \times \mathcal{L} \times \mathcal{C}$ is the *transition relation.* As usual, we will write $\langle C, \sigma, \gamma\rangle \xrightarrow{l} \langle C, \sigma', \gamma'\rangle$ to indicate that $(\langle C, \sigma, \gamma\rangle, l, \langle C, \sigma', \gamma'\rangle) \in \to$ and say that the choreography configuration $\langle C, \sigma, \gamma\rangle$ performs a transition

$\langle \mathsf{start}(\mathsf{id}, \mathsf{e}_0, \mathsf{e}_1), \sigma_0, \gamma_0 \rangle \xrightarrow{\tau} \langle inc(dec(\sigma_0, \mathsf{e}_0), \mathsf{e}_1), \gamma_0 \rangle$	$\sigma(\mathsf{e}_0) > 0$	$(Start)$
$\langle \mathsf{end}(\mathsf{id}, \mathsf{e}), \sigma, \gamma \rangle \xrightarrow{\tau} \langle dec(\sigma, \mathsf{e}), \gamma \rangle$	$\sigma(\mathsf{e}) > 0$	(End)
$\langle \mathsf{andSplit}(\mathsf{id}, \mathsf{e}, E), \sigma, \gamma \rangle \xrightarrow{\tau} \langle inc(dec(\sigma, \mathsf{e}), E), \gamma \rangle$	$\sigma(\mathsf{e}) > 0$	$(AndSplit)$
$\langle \mathsf{andJoin}(\mathsf{id}, E, \mathsf{e}), \sigma, \gamma \rangle \xrightarrow{\tau} \langle inc(dec(\sigma, E), \mathsf{e}), \gamma \rangle$	$\forall \mathsf{e}' \in E \;.\; \sigma(\mathsf{e}') > 0$	$(AndJoin)$
$\langle \mathsf{xorSplit}(\mathsf{id}, \mathsf{e}', \{(\mathsf{e}, \mathsf{exp})\} \cup X), \sigma, \gamma \rangle \xrightarrow{\tau}$ $\langle inc(dec(\sigma, \mathsf{e}'), \mathsf{e}), \gamma \rangle$	$\sigma(\mathsf{e}') > 0 \wedge$ $eval(\mathsf{exp}, \gamma)$	$(XorSplit)$
$\langle \mathsf{xorJoin}(\mathsf{id}, \{\mathsf{e}\} \cup E, \mathsf{e}'), \sigma, \gamma \rangle \xrightarrow{\tau} \langle inc(dec(\sigma, \mathsf{e}), \mathsf{e}'), \gamma \rangle$	$\sigma(\mathsf{e}) > 0$	$(XorJoin)$
$\langle \mathsf{task}(\mathsf{id}, \mathsf{e}_1, \mathsf{p}_1, \mathsf{m}, \mathsf{p}_2, \mathsf{e}_2), \sigma, \gamma \rangle \xrightarrow{\mathsf{p}_1 \to \mathsf{p}_2 : \mathsf{m}}$ $\langle inc(dec(\sigma, \mathsf{e}_1), \mathsf{e}_2), upd(\gamma, \mathsf{m}) \rangle$	$\sigma(\mathsf{e}_1) > 0$	$(Task)$
$\langle \mathsf{eventBased}(\mathsf{id}, \mathsf{e}', \{(\mathsf{p}_1, \mathsf{m}, \mathsf{p}_2, \mathsf{e})\} \cup T), \sigma, \gamma \rangle$ $\xrightarrow{\mathsf{p}_1 \to \mathsf{p}_2 : \mathsf{m}}$ $\langle inc(dec(\sigma, \mathsf{e}'), \mathsf{e}), upd(\gamma, \mathsf{m}) \rangle$	$\sigma(\mathsf{e}') > 0$	$(EvtBased)$
$\dfrac{\langle C_1, \sigma, \gamma \rangle \xrightarrow{l} \langle \sigma', \gamma' \rangle}{\langle C_1 \vert C_2, \sigma, \gamma \rangle \xrightarrow{l} \langle \sigma', \gamma' \rangle}\ (Int_1)$	$\dfrac{\langle C_2, \sigma, \gamma \rangle \xrightarrow{l} \langle \sigma', \gamma' \rangle}{\langle C_1 \vert C_2, \sigma, \gamma \rangle \xrightarrow{l} \langle \sigma', \gamma' \rangle}\ (Int_2)$	

Figure 4.6: Semantics of BPMN Choreographies.

labelled by l and becomes the configuration $\langle C, \sigma', \gamma' \rangle$. Since choreography execution only affects the current states, and not the choreography structure, for the sake of presentation, we omit the structure from the target configuration of the transition. Thus, a transition $\langle C, \sigma, \gamma \rangle \xrightarrow{l} \langle C, \sigma', \gamma' \rangle$ is written as $\langle C, \sigma, \gamma \rangle \xrightarrow{l} \langle \sigma', \gamma' \rangle$. Labels l represent computational steps and are defined as: τ, denoting movements of tokens; and $\mathsf{p}_1 \to \mathsf{p}_2 : \mathsf{m}$, denoting an exchange of message m from participant p_1 to p_2. Notably, despite the presence of labels, this has to be thought of as a reduction semantics, because labels are not used for synchronisation (as instead it usually happens in labeled semantics), but only for keeping track of the exchanged messages.

The transition relation over choreography configurations is defined by the rules in Fig. 4.6. Before commenting on the rules, we introduce the auxiliary functions they exploit. Specifically, function $inc : \mathbb{S}_e \times \mathbb{E} \to \mathbb{S}_e$ (resp. $dec : \mathbb{S}_e \times \mathbb{E} \to \mathbb{S}_e$), where $\mathbb{S}_e$ is the set of execution states, allows updating a state by incrementing (resp. decrementing) by one the number of tokens marking an edge in the state. Formally, they are defined as follows: $inc(\sigma, \mathsf{e}) = \sigma \cdot \{\mathsf{e} \mapsto \sigma(\mathsf{e}) + 1\}$ and $dec(\sigma, \mathsf{e}) = \sigma \cdot \{\mathsf{e} \mapsto \sigma(\mathsf{e}) - 1\}$. These functions extend in a natural way to sets of edges as follows: $inc(\sigma, \varnothing) = \sigma$

and $inc(\sigma, \{\mathsf{e}\} \cup E)) = inc(inc(\sigma, \mathsf{e}), E)$; the cases for *dec* are similar. The boolean predicate $eval : \mathbb{EXP} \times \mathbb{S}_m \rightarrow \{true, false\}$, where $\mathbb{S}_m$ is the set of message states, evaluates an expression with respect to a given message state; since the expression language is left unspecified, the definition of *eval* is unspecified as well. We also use the function $upd : \mathbb{S}_m \times \mathbb{M} \rightarrow \mathbb{S}_m$ that updates a message state by assigning a value to a given message; formally, the function is defined as follows: $upd(\gamma, \mathsf{m}) = \gamma \cdot \{\mathsf{m} \mapsto \mathsf{v}\}$ for $\mathsf{v} \in \mathbb{V}$.

We now briefly comment on the operational rules. Rule *Start* starts the execution of a choreography when it is in its initial state, i.e., only the spurious edge is marked, and no message has been exchanged yet. The effect of the rule is to consume the token in the spurious edge and increment the number of tokens in the edge outgoing from the start event. Rule *End*, instead, is enabled when there is at least a token in the incoming edge of the end event, which is then removed.

Rule *AndSplit* is applied when there is at least one token in the incoming edge of an AND-Split gateway; as a result of its application the rule decrements the number of tokens in the incoming edge and increments that in each outgoing edge. Rule *AndJoin* decrements the tokens in each incoming edge and increments the number of tokens of the outgoing edge, when each incoming edge has at least one token. Rule *XorSplit* is applied when a token is available in the incoming edge of an XOR split gateway; the rule decrements the token in the incoming edge and increments the tokens in one of the outgoing edges corresponding to a positive evaluation of the associated expression. Rule *XorJoin* is activated every time there is a token in one of the incoming edges, which is then moved to the outgoing edge. Rule *Task* is activated when there is a token in the incoming edge of a choreography task and moves the token from the incoming edge to the outgoing one. The rule produces a label describing the message exchange. Moreover, the message state is updated with a new value for the involved message name; we abstract from the computation of the message value, which is indeed nondeterministically selected via the *upd* function. Rule *EvtBased* is enabled each time there is a token in the incoming edge of an event-based gateway, which is moved to the activated outgoing edge. A label describing the message exchange is produced and the corresponding value is updated in the message state. Finally, rules Int_1 and Int_2 deal with interleaving.

4.3 From Choreographies to Smart Contracts

We present here the translation of BPMN choreography models into smart contracts. The main challenge is providing a smart contract implementation that is modular in the structure of choreography models (i.e., the smart contract that implements a choreography model follows the structure of the

choreography itself). This aspect, which we call 'modular blockchain-based choreography implementation', is one of the main technical aspects that characterizes this work. It is worth noticing that our approach is agnostic to the blockchain technology used for enacting the choreographies, meaning that it is not based on specific features of the smart contract language supported by the blockchain platform used. In fact, we provide a translation into a smart contract pseudocode, which can then be easily rendered in most mainstream smart contract languages.

The translation of BPMN choreography elements in smart contract terms exploits the translation functions $\langle\!\langle \cdot \rangle\!\rangle$ and $[\![\cdot]\!]$. The former function generates the contract header, containing the definitions of data types and variables, auxiliary functions, and the model-independent code triggering the execution. The latter function, instead, is defined by induction on the syntax of choreographies (see Fig. 4.2) and transforms each element of the BPMN model into a corresponding function of the smart contract that faithfully implements the element's semantics.

The definition of the translation functions $\langle\!\langle \cdot \rangle\!\rangle$ is shown in Listing 4.1, where we assume that the input choreography C involves n parties, m edges (named $\texttt{e}_i$ with $i \in \{1..m\}$, while $\texttt{e}_0$ denotes the spurious edge), k messages (named $\texttt{msg}_j$ with $j \in \{1..k\}$), and h BPMN elements (with identifiers $\texttt{id}_w$ with $w \in \{1..h\}$).

```
1 ⟪C⟫ =
2   enum MsgStatus {DISABLED,ENABLED}
3   enum ChoreographyState {INIT, RUNNING, MSG_WAITING,
        ↪ COMPLETED, DEADLOCKED}
4   private address party_1 = 0x...; ...;
5   private address party_n = 0x...;
6   private ChoreographyState chorState =
        ↪ ChoreographyState.INIT;
7   private int e_0 = 1;
8   private int e_1 = 0; ...;
9   private int e_m = 0;
10  private int enabledMessages = 0;
11  private MsgStatus status_msg_1 = MsgStatus.DISABLED;
12  private Type_11 payload_msg_1_param_1, ..., Type_1r
        ↪ payload_msg_1_param_r;
13  private MsgStatus status_msg_k = MsgStatus.DISABLED;
14  private Type_k1 payload_msg_k_param_1, ..., Type_kt
        ↪ payload_msg_k_param_t;
15  [[C]]
16  private int getMarking()
17  { return e_0+e_1+...+e_m; }
18  private boolean executeOneRule()
19  { return id_1() || ... || id_h(); }
20  public void execute() {
21    chorState = ChoreographyState.RUNNING;
22    while (executeOneRule()) {}
```

```
23    if (enabledMessages > 0)
24    { chorState = ChoreographyState.MSG_WAITING; }
25       else {
26       if (getMarking()>0)
27       { chorState = ChoreographyState.DEADLOCKED; }
28     else { chorState = ChoreographyState.COMPLETED; }
29  } }
```

Listing 4.1: Top-level translation function $\langle\!\langle \cdot \rangle\!\rangle$.

This translation generates the smart contract code that, once deployed in the blockchain, deals with the execution of a specific choreography instance. The code must be instantiated with the addresses of the parties participating in the given choreography instance (see variables $\texttt{party}_i$, line 4 and 5). In this respect, more dynamic solutions could be realized, but this is out of the scope of this work. The smart contract variables store the state of the choreography instance. The execution state function σ is rendered by the variables $\texttt{e}_i$ (lines 7-9), while the message state function γ is rendered by the variables $\texttt{payload\_msg}_j\texttt{\_param}_k$ (lines 11-14). For each message, in addition, a variable (lines 11 and 13) keeps track of whether the reception of the message is `ENABLED` or `DISABLED`.

The execution of the choreography instance is carried out by the `execute()` function (lines 20-29). This is the public function that one has to invoke to activate the choreography instance once the smart contract has been deployed in the blockchain. It iteratively executes one semantic rule at a time on the choreography's elements, thus implementing the interleaving semantics given in Sec. 4.2 (see rules Int_1 and Int_2). This behavior is realized by resorting to the private function `executeOneRule()` (line 18), which sequentially invokes the execution of each choreography's element (using functions $\texttt{id}_i()$ generated by translation function $[\![\cdot]\!]$) until one element is successfully executed. If all elements of the choreography cannot be executed, the instance execution stops, and the motivation underlying this termination is checked. If there are enabled messages, the execution is suspended and will be resumed when the waited message is delivered. Otherwise, the instance terminates successfully (if the marking is empty) or unsuccessfully (if there are deadlocked tokens).

```
1 [[start(id, e0, e)]] = private boolean id() {
2                   if (e0 > 0) { e0--; e++; return true; } else
                        ↪ return false; }
3
4 [[end(id, e)]] = private boolean id() {
5                 if (e > 0){ e--; return true; } else return
                        ↪ false; }
6
7 [[andSplit(id, e, {e1, ..., en})]] = private boolean id() {
8             if (e > 0) { e--; e1++; ...; en++; return true; }
```

```
                ↪ else return false; }
 9
10 [[andJoin(id, {e1, ..., en}, e)]] = private boolean id() {
11                  if (e1 > 0 && ... && en>0) { e1--; ... en--;
                        ↪ e++; return true;
12                  } else return false; }
13
14 [[xorSplit(id, e, {(e1, exp1), ..., (en, expn)})]] = private boolean id() {
15                  if (e > 0 && exp1) { e--; e1++; return true;
16                  } else ...
17                  } else if (e > 0 && expn) { e--; en++; return
                        ↪ true; }
18                  else return false; }
19
20 [[xorJoin(id, {e1, ..., en}, e)]] = private boolean id() {
21                  if (e1 > 0) { e1--; e++; return true; } else
                        ↪ ...
22                  } else if (en > 0){ en--; e++; return true; }
                        ↪ else return false; }
```

Listing 4.2: Inner translation function $[\![\cdot]\!]$: start/end events, AND/XOR gateways.

The definition of the translation functions $[\![\cdot]\!]$ of control flow elements is shown in Listing 4.2 while, Listing 4.3 presents the definition of task element and Listing 4.4 presents the events.

Each choreography element is translated into a function according to the semantics given in Fig. 4.6 and returns a boolean value indicating whether the element is executed or not. In other words, invoking function `id()` corresponds to try to apply the corresponding semantic rule to the choreography element identified by id; if the rule is applicable, the state of the choreography is modified accordingly and the function returns `true`, otherwise it returns `false`.

```
1 [[task(id, e1, p1, msg(param1, ..., paramn), p2, e2)]] =
2   private boolean id() {
3     if (e1 > 0) {
4       e1--; status_msg = MsgStatus.ENABLED;
5       enabledMessages++;
6       emit MessageEvent(p1,p2,msg);
7       return true;
8     } else return false;
9   }
10  public void msg(Type1 param1, ..., Typen param_n){
11    if (status_msg == MsgStatus.ENABLED && msg.sender==p1) {
12      payload_msg_param1 = param1; ...; payload_msg_paramn =
            ↪ paramn;
13      status_msg = MsgStatus.DISABLED;
14      enabledMessages--;
15      e2++;
```

```
16          execute();
17      } }
18
19 [[C1|C2]] = [[C1]]   [[C2]]
```

Listing 4.3: Inner translation function $[\![\cdot]\!]$: tasks and elements composition.

As an example, the function corresponding to a start event (lines 1-2 in Listing 4.2) checks if the spurious edge of the event is marked; in the negative case it returns `false`, because the condition $\sigma(\mathsf{e}_0) > 0$ of the semantic rule is not satisfied and hence the rule is not applicable, otherwise the function moves one token from the spurious edge to the outgoing edge and returns `true`.

```
1  [[eventBased(id, e″, {(p1, msg(Type1 param1, ..., Typen paramn), p2, e),
2          ..., (p3, msg′(Type′1 param′1, ..., Type′h param′h), p4, e′)})]]=
3     private boolean id() {
4       if (e″ > 0) {
5         e″--;
6         status_msg = Status.ENABLED; enabledMessages++; emit
              ↪ MessageEvent(p1,p2,msg);
7         ...
8         status_msg′ = Status.ENABLED; enabledMessages++; emit
              ↪ MessageEvent(p3,p4,msg′);
9         return true;
10      } else return false;
11    }
12    public void msg(Type1 param1, ..., Typen paramn) {
13      if (status_msg == MsgStatus.ENABLED
14            && msg.sender==p1) {
15        payload_msg_param1 = param1; ...; payload_msg_paramn =
              ↪ paramn;
16        status_msg = MsgStatus.DISABLED; enabledMessages--;
              ↪ ...; status_msg′ = MsgStatus.DISABLED;
              ↪ enabledMessages--;
17        e++;
18        execute();
19      } }
20    ...
21    public void msg′(Type′1 param′1, ..., Type′h param′h) {
22      if (status_msg == MsgStatus.ENABLED && msg.sender==p3) {
23        payload_msg′_param′1 = param′1; ... payload_msg′_param′h =
              ↪ param′h;
24        status_msg′ = MsgStatus.DISABLED; enabledMessages--;
25        ...
26        status_msg = MsgStatus.DISABLED; enabledMessages--;
27        e′++;
28        execute();
29      } }
```

Listing 4.4: Event based gateway translation.

The most interesting case of the translation definition is one of the task elements (lines 1-18 in Listing 4.3), which generates two functions. The private function (lines 2-9), if the incoming edge is marked, consumes the incoming token, enables the exchange of the message, and notifies the involved parties by emitting an event (which contains the same information as the transition label produced by the *Task* rule). The public function (lines 10-17), instead, is invoked by the sending party to send the message, whose payload is stored in the contract variables representing the message state function γ (line 12). Then, the message is disabled, the outgoing edge is marked, and the choreography instance execution is resumed. Notably, the message function can be successfully executed only if it is invoked by the expected sending party and if the message exchange for that specific type of message is enabled (line 11). Finally, the translation of the composition of elements (i.e., $C_1|C_2$) is simply rendered as the juxtaposition of the functions resulting from the translations of the composition arguments (line 19). The pseudocode resulting from the translation of the X-ray exam choreography is available as a text file in our artifacts repository [77].

4.4 Evaluation

In this section, we report the evaluation of our translation approach presented in Sec. 4.3. Following the approach, we implemented the smart contracts of the X-ray exam BPMN choreography described in Sec. 4.1.

To demonstrate the proposed translation approach, we choose two different blockchain platforms, Algorand and Polygon. These platforms were selected because they follow very different models. Algorand uses a Pure Proof-of-Stake consensus and Algorand python as smart contract language. Polygon, on the other hand, is EVM-compatible, uses Solidity as smart contract language.

```
1       function start1() private returns (bool)  {
2           if (e0 > 0) {
3               e0--;
4               e1++;
5               return true;
6           }
7           return false;
8       }
9       function takeAppointment() private returns (bool) {
10          if (e1 > 0) {
11              e1--;
12              status_appointment = Status.ENABLED;
13              enabledMessages++;
14              emit MessageEnabled("patient", "radiology",
                    "appointment");
```

```
            return true;
        }
        return false;
    }
    function xor1() private returns (bool) {
        if (e2 > 0) {
            e2--;
            e3++;
            return true;
        } else if (e6 > 0) {
            e6--;
            e3++;
            return true;
        }
        return false;
    }
```

Listing 4.5: Solidity smart contract code (an excerpt)

By providing the implementations of both platforms, we show that our platform agnostic translation approach can handle different programming models, execution rules, and platform-specific constraints. Partial code examples for both platforms are provided in Listing 4.5 (Solidity) and Listing 4.6 (Algorand Python). These functions represent the translated implementation of the initial choreography elements. Presenting the same functions in two different programming languages illustrate the modularity of the translation approach. Specifically, it demonstrates that each choreography element can be independently translated while preserving its semantics, in line with a modular design. This reinforces the claim that the translation framework supports consistent and language-independent mappings. These examples serve as a reference to verify the translation approach. The full source code of the implemented smart contracts is available together with the execution logs at [77].

```
def start1():
    return Seq(
        If(app.state.e0.get() > Int(0)).Then(
            Seq(
                app.state.e0.set(Int(0)),
                app.state.e1.set(app.state.e1.get() + Int(1)),
                Int(1)
            )
        ).Else(
            Int(0)
        )
    )
@Subroutine(TealType.uint64)
def take_appointment():
    return Seq(
        If(app.state.e1.get() > Int(0)).Then(
```

```
            Seq(
                App.globalPut(Bytes("Message"),
                    Bytes("Patient 'radiology': appointment")),
                app.state.e1.set(app.state.e1.get() - Int(1)),
                app.state.status_appointment.set(Int(1)),
                app.state.enabled_messages.set(
                app.state.enabled_messages.get() + Int(1)),
                Int(1)
            )
        ).Else(
            Int(0)
        )
    )
@Subroutine(TealType.uint64)
def xor1():
    return Seq(
        If(app.state.e2.get() > Int(0)).Then(
            Seq(
                app.state.e2.set(app.state.e2.get() - Int(1)),
                app.state.e3.set(app.state.e3.get() + Int(1)),
                Int(1)
            )
        ).ElseIf(app.state.e6.get() > Int(0)).Then(
            Seq(
                app.state.e6.set(app.state.e6.get() - Int(1)),
                app.state.e3.set(app.state.e3.get() + Int(1)),
                Int(1)
            )
        ).Else(
            Int(0)
        )
    )
```

Listing 4.6: Algorand python smart contract code (an excerpt)

The translation approach focuses on preserving the semantics of the original contract while adapting it to the target platform's execution model. This process involves carefully mapping the logic of stateful and stateless operations, handling platform-specific constraints, and ensuring that the resulting contract maintains functional equivalence. By examining the provided listings, readers can observe how platform-specific constructs are translated, how core business logic is preserved, and how subtle differences in execution environments are managed. Overall, this demonstrates not only the feasibility of the translation approach but also its practical applicability across heterogeneous blockchain ecosystems.

4.4.1 Experiments and performance analysis.

For each considered platform, we deployed the X-ray smart contract on a public testnet and executed relevant paths. The choreography execution can be described through four distinct paths based on the values assigned to the variables *accepted* and *registration*, influencing the decision points (i.e., XOR gateways). In the first path, the choreography is executed with both accepted and registration set to true. In the second path, accepted is set to true, while registration is false. The third path has two iterations of the loop, where accepted is initially set to false and then changed to true, while registration remains true. Finally, in the fourth path, the choreography undergoes two loop iterations but terminates with registration set to false.

The execution performances of each path were analyzed in terms of gas used for Polygon and cost for Algorand, compared then with the fiat currency. Regarding the performed experiments, the first path refers to the execution of the choreography with a value of *true* in both the *accepted* and *registration* choices[1]. In the second path instead the choreography was executed with the value of *true* in the *accepted* choice and *false* in the *registration* one[2]. [3]. The third path includes a loop related to the double execution of the *accepted* choice (false, true) while the *registration* is kept *true*[4]. Finally, the last path was executed with the initial loop in the *accepted* choice (false, true) but with the termination of the choreography in the *registration false*[5].

4.4.2 Polygon Experiments

The first experiments were made on the Solidity implementation of the smart contract using the Polygon Amoy testnet. For each executed path, we analyze the performance in terms of units of gas used for the *deployment*, *execution*,

[1]Polygon log available at https://amoy.polygonscan.com/address/0x9f5c6370441c9017bd238ad6e92a8f42189ce709

[2]Algorand log available at https://lora.algokit.io/testnet/application/723197355

[3]Polygon log available at https://amoy.polygonscan.com/address/0xAbeAF334b0b67878Ce704E777F7adeeb3F7270A2
Algorand log available at https://lora.algokit.io/testnet/application/723197874

[4]Polygon log available at https://amoy.polygonscan.com/address/0x3a3017E1456A846699a67f16E92eE6E57e52f577.
Algorand log available at https://lora.algokit.io/testnet/application/723198066

[5]Polygon log available at https://amoy.polygonscan.com/address/0x1718EB5F03047366F80c857C1E53ab04722cAe11.
Algorand log available at https://lora.algokit.io/testnet/application/723198366

and the *total* sum. The log of the executed smart contracts are available at [77][6].

Path	Deploy-cost (units of gas)	Execution-cost (units of gas)	Total-cost (units of gas)
1	3,131,321	1,059,408	4,190,729
2	3,131,321	816,986	3,948,307
3	3,131,321	1,204,517	4,335,838
4	3,131,321	962,095	4,093,416

Table 4.4: Gas used for the execution on Polygon.

The results of the experiments are summarized in Table 4.4, all the deployments consumed 3,131,321 units of gas as the smart contract logic remains the same for all the cases. The execution cost instead, depends on the followed path as demonstrated by the resulting performance. Indeed, the lowest amount of gas used is 816,986 and corresponds to the shortest path where there is no loop and the choreography terminates on the registration task. On the contrary, the highest amount of gas used is 1,204,517 and it corresponds to the execution of the choreography with the first performed loop and the termination on the longest path. Of course, the other paths that perform more iterations of the loop have an increasing cost (i.e., 145,109 units of gas per iteration).

4.4.3 Algorand Experiments

The results of Algorand experiments are reported in Table 4.5. In this case, it is worth noticing that Algorand has a fixed transaction fee of 0.001 ALGO for every transaction, regardless of the type. This leads to the deployment costs of 0.001 ALGO for each path, and the execution only depending on the number of executed messages. Indeed, the lowest execution cost is 0.009 ALGO, corresponding to the shortest path without any loop in the choreography.

The highest cost is 0.013 ALGO refers to the longest execution path with the initial loop and the termination on the last task. (1 start and 12 messages). Algorand has a fixed transaction fee for every transaction, regardless of the type of transaction. The fee is paid only if transaction is included in the block. Algorand has also a notion of minimum balance requirement, which acts like a deposit to rent a space on the blockchain. As the amount of data stored increases, opting into application or ASA,

[6]The complete transaction logs for the smart contracts are available at shttps://lora.algokit.io/testnet/account/GDMVSGQMZKUDMIJ54RCMBGSW7YK6N53QYBS74UC3MIGT7EBR4VKEUAOHIE

Path	Deploy-cost (ALGO)	Execution-cost (ALGO)	Total-cost (ALGO)
1	0.001	0.011	0.012
2	0.001	0.009	0.010
3	0.001	0.013	0.014
4	0.001	0.011	0.012

Table 4.5: Execution costs for different workflow paths on Algorand.

storing data in boxes the minimum balance requirement of associated account increases [7]. The minimal fee prevents spam and remains low enough to allow everyday use, aligning with algorand's high efficiency and scalability. Further it compensates network validators for their work in confirming transactions.

4.4.4 Polygon vs Algorand comparison

To provide an evaluation of the feasibility of the proposed approach, we report here the comparison between the two implemented translations and executed paths. Table 4.6 shows the costs in terms of cryptocurrency and fiat currency for the Polygon and Algorand blockchains.

Path	Polygon		Algorand	
	POL	USD	ALGO	USD
1	0.121784	0.0490	0.012	0.0015
2	0.120777	0.0485	0.010	0.0012
3	0.132792	0.0534	0.014	0.0017
4	0.125395	0.0504	0.012	0.0015

Table 4.6: Comparison of costs for Polygon and Algorand executions.

Considering Polygon, the execution of the four paths costs an average of 0.125187 POL corresponding to 0.0504 USD (according to the exchange value at the time of writing). For Algorand instead, the average cost is 0.001475 ALGO corresponding to 0.000179 USD.

These results highlight the feasibility of the implemented choreography transformation, as the smart contracts deployed and executed exhibit strong performance. It is important to consider that the chosen blockchain platform determines the execution costs, which can vary significantly from one platform to another. For example, using Ethereum could increase the costs. However, this variance is independent of the proposed approach and can be mitigated by selecting the appropriate platform. Another important factor is

the size of the choreography, which directly affects the size of the implemented smart contract. As the number of elements in the choreography increases, the size of the smart contract linearly grows, leading to higher costs. However, given the obtained results, small increases in the choreography model size are likely not to result in significant variations in the performance of the smart contract. Anyway, in practice, large increases in the choreography size are not frequent and the considered case study is part of the average cases [26].

It is worth mentioning that the works reported above mainly focus on aspects related to the generation of smart contract, but it also support the whole life-cycle of choreographies to pseudocode, intermediate representation and finally smart contract.

CHAPTER 5

SUPPORTING RUN-TIME FLEXIBILITY

This chapter of the thesis presents the controlled runtime adaptation execution framework for business process execution, achieving the second objective of this research.

> "*To design and implement controlled runtime adaptation of blockchain-based business processes using Algorand's update capabilities.*"

This chapter focuses on enabling controlled and flexible execution of business processes on the Algorand blockchain, leveraging its native update mechanisms to support dynamic adaptation during runtime without compromising process integrity or security.

The advent of blockchain has enabled the creation of novel decentralized applications where participants have trust and accountability needs. Among the various domains, inter-organizational business processes have found in blockchain a proper solution to create technical infrastructures supporting process execution in decentralized and untrusted environments [85]. By encoding business logic into smart contracts, the latter enforce the proper execution of the process behavior, automatically executing constraints and advancing the state of the process. Blockchain immutability and transparency also play a crucial role, as they allow for keeping a secure and accessible track of past business interactions [23, 24].

Over the years, different approaches have emerged to leverage blockchain for process automation and coordination, typically relying on Ethereum-based platforms [108, 39]. Such solutions specify inter-organizational interactions employing the Business Process Model and Notation (BPMN) standard

[91], which has been established over the years as a dominant solution both in industry and academia. In particular, *BPMN choreographies* allow for modeling inter-organizational interactions, specifying only the message exchange between participants without exposing their internal behavior. In this context, blockchain serves as a target technology, encoding the choreography and its interactions, supporting their distributed execution at runtime [35, 29].

While the adoption of blockchain has enabled a concrete and trustworthy enactment of choreographies, it has also led to several challenges, among which ***flexibility*** is a significant one. When modeling business interactions in terms of a choreography, not all scenarios can be foreseen. Thus, it is crucial to deal with possible factors that can affect a process at runtime, which may lead to the introduction of changes in the initially designed choreography model and, hence, the consequent update of the underlying implementation [31]. This ability to react to unforeseen scenarios is called ***adaptation*** [102].

However, in the context of blockchain-based execution of choreographies, flexibility remains a persistent and open challenge due to the native structure of blockchain [108]. Indeed, the inherent immutability of blockchain typically does not allow for direct updates to smart contract code once deployed, making it difficult to deal with runtime changes. This requires dedicated solutions, opening to new research directions. While updates can be implemented through new smart contract deployments, these come with significant costs and operational overhead, especially if adopting complex patterns [123]. Off-chain solutions have been explored to address this limitation [32], but they introduce their own set of challenges, including security risks and increased complexity in integration.

On the other hand, these limitations can be overcome nowadays by exploiting blockchain architectures that introduce mechanisms for more flexible and adaptive execution [74]. In particular, Algorand [20] is a blockchain platform that aims to address the 'blockchain trilemma', concerning the simultaneous satisfaction of the scalability, security, and decentralization challenges. Designed with a pure proof-of-stake consensus mechanism, Algorand provides instant transaction finality, high throughput, and robust fork resistance, which are key properties for enabling reliable, time-critical business workflows. Traditional blockchain platforms, such as Ethereum, often suffer from issues like unpredictable transaction fees and intricate upgrade procedures, which can limit scalability and reduce the flexibility needed for dynamic business process execution [127]. Instead, Algorand adopts a fixed, low-fee model that provides predictability and cost efficiency for developers and users alike. Furthermore, Algorand offers native features that facilitate updates of smart contracts and storage elements, without the use of proxies, enabling seamless and secure business process evolution and minimizing

operational disruption [8]. These capabilities make Algorand particularly well-suited for implementing flexible choreographies, where adaptability, reliability, and low overhead are critical.

Here, we present how Algorand's design principles can be leveraged to realize **flexible and trusted execution of inter-organizational business processes specified in terms of BPMN choreographies**. To this purpose, we encode BPMN choreographies as smart contracts, deployed and executed on the Algorand blockchain. We show how the adaptation of a choreography at runtime is reflected and enacted on the corresponding smart contract. In fact, a choreography can be adapted to multiple perspectives of the process, and each change leads to a specific update in the underlying smart contract. By relying on the native Algorand update capabilities, we do not need complex and gas-consuming patterns or operations to update the underlying smart contract. In addition, using a native update mechanism allowed us to define a lean approach to control the conditions enabling smart contract updates, thus fostering the definition of update policies for regulating the choreography adaptation. Furthermore, the performance efficiency of Algorand permits dealing with adaptation without incurring high costs.

5.1 Algorand and BPMN Choreographies

Now, we introduce basic notions of the Algorand blockchain, focusing on its peculiarities and technical features. We then describe BPMN choreography diagrams and their elements via a simple example.

5.1.1 Algorand Key Features

Algorand is a decentralized, permissionless blockchain platform designed for scalability, security, and true decentralization [53, 20]. Different to other blockchain technologies such as Ethereum-based ones, it introduces several novelties both at the protocol and implementation layers. As a consensus mechanism, Algorand relies on a Pure Proof-of-Stake (PPoS) protocol, which randomly selects a committee of users to propose and validate blocks depending on their stake amount in the network. This approach ensures fast finality, low latency, and resistance to forks while maintaining decentralization.

Algorand supports smart contracts (also referred to as *applications*) through the Algorand Virtual Machine (AVM). Developers can write smart contracts using Transaction Execution Approval Language (TEAL), a language optimized for safety and performance. To simplify development, Algorand also offers Python-based languages, i.e., PyTEAL and Algorand Python. In this work, we use the latter one.

Another key difference with respect to other platforms is the Algorand storage model, which includes three different types of storage presented in Table 5.1: Local storage, Global storage, and Box storage. Specifically, the local storage is suited for storing user-specific data, the global storage for shared data accessible by all users, and the box storage for application-specific data. In this work, we mainly rely on the global storage, as it provides a persistent on-chain storage that serves as shared memory among business participants to keep exchanged information.

Storage	**Size**	**Funding Account**
Local Storage	2 KB	User account
Box Storage	32 KB	Application account
Global Storage	8 KB	Creator account

Table 5.1: Algorand Storage Capacities and Funding Accounts.

More specifically, the global storage in Algorand is a key-value store directly associated with a smart contract application. Each application can store up to 64 key-value pairs, with each pair having a combined maximum size of 128 bytes (key + value $\leqslant$ 128 bytes), and a total storage limit of 8 KB shared across all pairs. Technically, this storage is maintained as part of the application's on-chain state, where keys are stored as byte slices and values can be either byte slices or uint64 integers.

Storage	**Read**	**Write**	**Delete**
Local Storage	*local_Get*	*local_Put*	*local_Del*
Global Storage	*global_Get*	*global_Put*	*globalDel*
Box Storage	*box_Extract*, *box_Get*	*box_Put*, *box_replace*	*box_Del*

Table 5.2: Algorand Storage Read, Write, and Delete Operations.

To modify global storage in a smart contract, developers must use the appropriate methods within the contract code. Table 5.2 presents the oveview of the methods that can be used to modify the storage on Algorand. Global state can be managed using two approaches:

- *Implicit declaration* (`self.var = UInt64(1)`): This method directly assigns a typed value to the variable, offering concise syntax. It provides limited control over state behavior and obscures the underlying storage mechanism.
- *Explicit declaration* (`self.var = GlobalState(UInt64(1))`): This approach clearly defines the variable as part of the global state. It uses

`.value` for access and modification, and provides a richer API and support for explicit deletions.

Both approaches use the same TEAL opcodes and incur identical minimum balance requirements. However, the explicit method is preferred for clarity, better error handling, and maintainability.

5.1.2 BPMN Choreographies: Running Example

The BPMN standard provides the choreography diagram [91, Ch. 11] as an intuitive and graphical means to describe the interactions that should occur among the multiple participants of a distributed scenario. We illustrate some key elements of this kind of diagram by resorting to the choreography model in Fig. 5.1. We use this simple model throughout the Chapter as a running example.

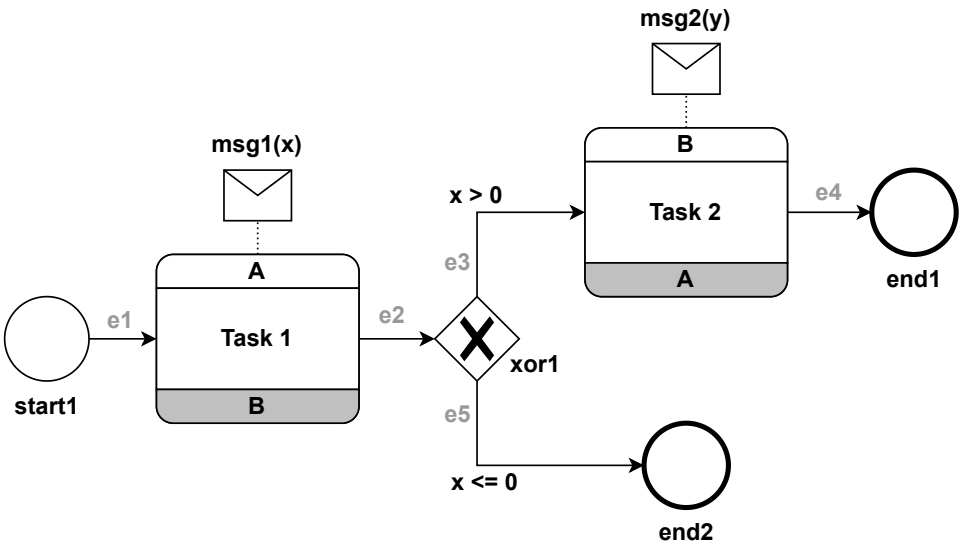


Figure 5.1: Running example: original choreography.

The starting point of the choreography is represented by the *start* event (start1), which is drawn as a circle with an outgoing edge. As in any workflow diagram, *edges* are used to specify the execution flow by connecting choreography elements. Although edge names typically are not visualized in the graphical representation of the diagram, these names are always included in the underlying XML file produced by BPMN modelers; to improve the paper's readability, we reported edge names (using the gray color) in the model in Fig. 5.1.

The start event is connected by edge e1 to a *choreography task* element, which describes a message exchange between two participants involved in the

choreography. The task is represented as rectangles divided into three bands: the central one contains the task's name (Task 1), the white band refers to the initiator participant (A), and the grey one refers to the recipient participant (B). The exchanged message, of type msg1, is graphically represented by a white envelope. Its payload is formed by a single field (x), whose actual content will be used subsequently in the choreography execution to make a choice.

The execution flow of the choreography is controlled by using *gateways*. Gateways act as either split nodes (forking into outgoing edges) or join nodes (merging incoming edges). In the example model, Task 1 is connected by e2 to the *exclusive (XOR) gateway* xor1. This is drawn with a diamond marked with the × symbol and represents a conditional choice based on the value previously exchanged via message msg1. More specifically, this is a XOR-split gateway defining two execution branches to be selected on the basis of the conditions labelling the outgoing edges (i.e., e3 is selected if the value of the field x is greater than 0, otherwise e5 is selected); when executed, the gateway activates exactly one outgoing edge.

If x>0, the choreography execution continues with Task 2, which allows the participant B to reply to the participant A with a message of type msg2. After this message exchange, the choreography terminates by means of an *end* event (end1), which is drawn as a circle with an incoming edge and denotes an ending point of the choreography. If instead x<=0, the choreography execution directly terminates by means of the end2 event.

Real-world distributed processes are rarely static; instead, they evolve due to changes in business requirements, regulatory constraints, or partnership agreements. Therefore, BPMN choreography models must support adaptation, allowing modifications to interaction logic while preserving the overall correctness and interpretability of the workflow. Adaptation typically affects the structure of the choreography, such as refining gateway conditions, introducing additional interaction tasks, or including new participants to reflect operational changes.

We illustrate this concept by adapting our running example, resulting in the updated choreography model shown in Fig. 5.2. This revised model incorporates several structural and logical changes to enhance the interaction flow. A notable change is the introduction of a new participant, C, who is involved in a subsequent interaction. The choreography still begins with the start event start1, followed by the initial message exchange Task 1 where participant A sends message msg1(x) to participant B. However, the conditional logic immediately following this task has been refined. The exclusive (XOR) gateway now evaluates the condition on the payload x differently: edge e3 is taken if x >= 5, leading to a new choreography task, Task 2. In this task, participant B initiates a new message msg4(k) to participant A, indicating an

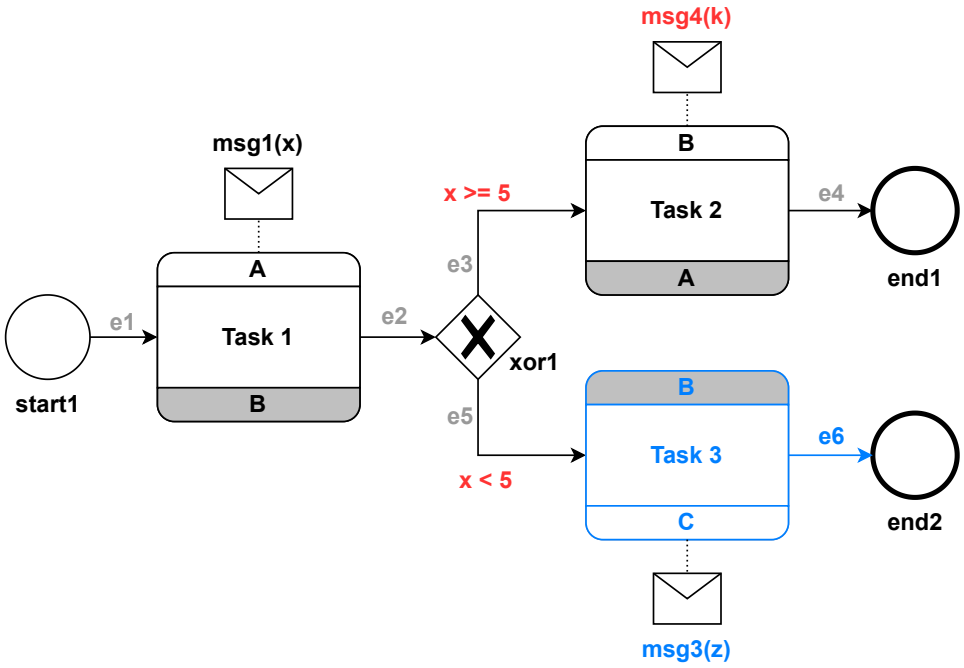


Figure 5.2: Running example: updated choreography.

extended interaction pattern not present in the original model.

Furthermore, the adaptation introduces additional changes, specifically msg3(z) and Task 3. If the condition x >= 5 is not met, the flow proceeds along edge e5 (labeled x < 5) and concludes at end2. This refined gateway logic and the introduction of a new task and termination points demonstrate how a choreography can be adapted to handle more complex business scenarios, incorporate feedback loops, or comply with new procedural rules, all while maintaining a clear and executable workflow structure.

Adaptation may also involve adding new parallel branches to incorporate supporting procedures. For example, before executing Task 2, both participant A and a newly introduced C could be required to provide confirmation messages. This modification would replace the simple XOR decision path with a combination of conditional and parallel gateways, increasing collaboration and validation layers in the process.

5.2 Flexible Execution of Choreographies on Algorand

In this section we show the general idea of how BPMN choreographies can be executed on the Algorand blockchain considering flexibility needs. We start by describing the main components and then we focus on the adaptation of choreographies and the different kind of changes that can be applied.

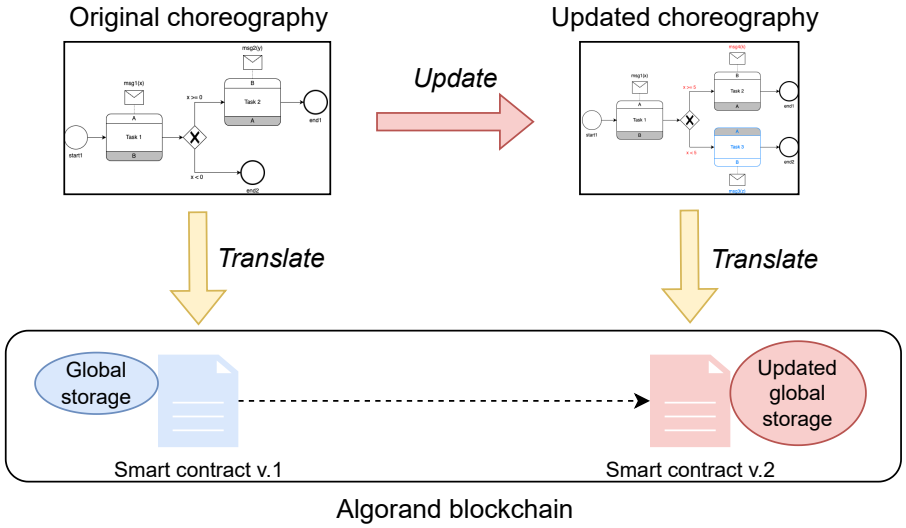


Figure 5.3: Approach to flexible choreography execution: Key phases and components.

Figure 5.3 depicts an overview of the different phases supporting the deployment and update of choreographies. During the **deployment**, a BPMN choreography model is created and translated into blockchain artifacts. In particular, a choreography translates into a smart contract, containing the logic of the interactions to be executed. Additionally, the Algorand global storage is instantiated, containing all relevant information to be stored in a permanent manner, such as data exchanged through choreography messages. At this point, the smart contract can be executed by interested parties sending data according to the deployed logic. During the execution, a choreography may need to adapt to certain changes in the business scenario or the general context. To manage such flexibility need, an **update** is performed on the choreography model, by applying changes to elements such as tasks, messages or participants. This leads to an update of its smart contract implementation, as the choreography is translated into a new version of the smart contract, which is deployed in the blockchain and replaces the existing one. In addition to the smart contract logic, it is also possible to update

the global storage by operating on the relative information (e.g., adding or removing storage variables).

5.2.1 Runtime Adaptation of Choreographies

To ensure a flexible execution of choreographies, the update phase is crucial as it consists of the choreography and its underlying implementation. For this reason, here we focus on the high-level changes that can be applied to the choreography specification and used to derive the new version of the smart contract.

Choreography changes. When adapting a choreography at runtime, several changes can be made to the BPMN model categorized into different types [102, 74]. In this work, we focus on the below operations.

- **Addition**
- **Deletion**
- **Modification**

as they represent atomic types of changes that can be applied to a choreography and that directly reflect on the smart contract and the global storage. These operations can affect the various BPMN choreography elements introduced in Section 5.1, specifically by adding, removing, and modifying (a) tasks, (b) messages, (c) participants, (d) control flow elements, and (e) events.

More specifically, the changes in the adapted model are as follows (modified elements are highlighted in red color, while added ones are in blue): (i) the XOR conditions are modified; (ii) message msg2 is replaced by msg4; (iii) Task 3, exchanging message msg3, is added; (iv) edge e6, connecting Task 3 to end2, is added. It is worth noticing that the shown types of changes in Fig 5.4 represent base operations that can be combined or integrated to derive more complex patterns. For example, choreography elements can be moved, swapped, or replaced. Anyway, thanks to their compositional nature, these patterns do not pose any additional issue concerning the update of the underlying smart contract and global storage.

Controlling adaptation. While updating BPMN choreographies at runtime allows for dynamic adaptation to evolving requirements, some restrictions could be necessary. To this purpose, update policies can be adopted to regulate and constrain changes to the choreography.

A first possibility is the enablement or disablement of changes. In certain contexts, it may be desirable to disable updates for a certain period once the choreography is deployed. For instance, changes may be allowed only during

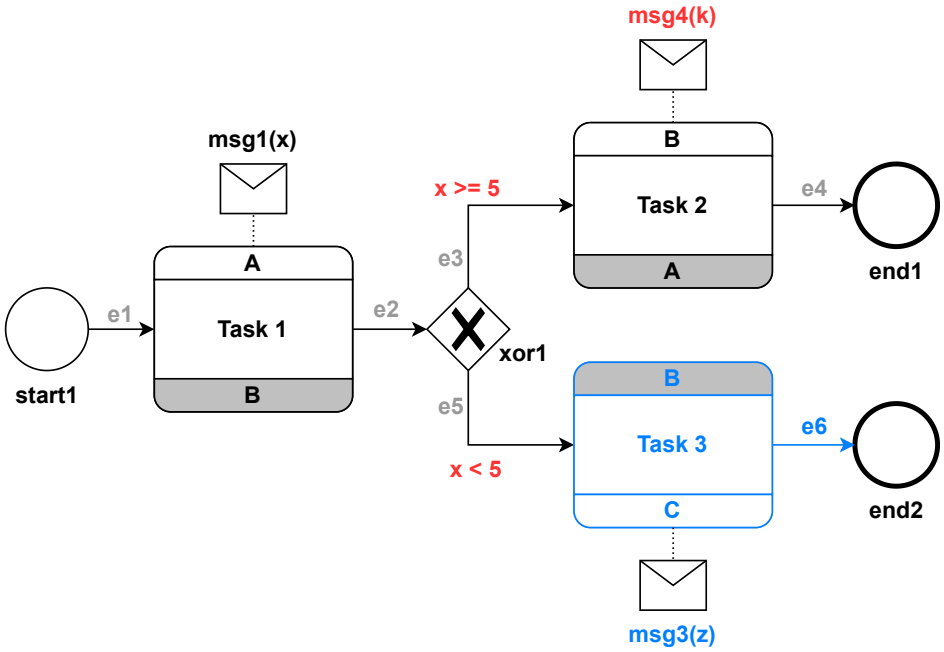


Figure 5.4: Updated choreography with new added task and conditions.

specific reconfiguration periods or under defined conditions, such as when a quorum among participants is reached. Furthermore, a limited number of updates can be allowed for a given choreography; once this limit is reached all update requests will be denied. Another important aspect refers to participant permissions. Not all entities involved in the choreography should necessarily possess the authority to modify its structure. Policies can be used to assign update rights only to authorized participants, such as process owners. In addition, policies can be used to define element-specific restrictions. To safeguard essential parts of the process logic, certain elements may be declared immutable, or some elements act as a commit action that disables the update capability.

These various types of policies can be embedded in the smart contract layer associated with the choreography model. In this way, they will be a publicly visible part of the contract. By doing so, the execution infrastructure can automatically enforce the specified constraints, thereby mitigating the risk of undesired operations. The effectiveness and robustness of this controlled form of update strongly rely on the native update mechanisms of Algorand.

5.3 Implementation of the approach

Here, we present the implementation of our proposal according to the different phases described in Section 5.2. We start by describing how an Algorand smart contract implements a BPMN choreography, then showing how it reflects runtime updates.

5.3.1 Choreography smart contract deployment

The first phase consists of the deployment of the Algorand smart contract encoding the logic of the BPMN choreography model. To this purpose, the smart contract is derived through a translation of the choreography model into smart contract code. As a translation mechanism, in this work we rely on the general approach presented in [80] Chapter 4 . We recall that, this is a *modular* translation, in the sense that the structure of the generated smart contract follows the structure of the choreography from which it originates, thus ensuring a one-to-one correspondence between BPMN choreography elements and smart contract functions. The translation is defined by induction on the syntax of choreographies and transforms each element of the BPMN choreography model into a corresponding function of the smart contract that faithfully implements the element's semantics. The general structure of the smart contract is composed of several functions related to: state initialization, message execution, control flow execution, and contract update. In the following, we show the main parts of the smart contract, focusing on the variables and functions affected by the successive update.

Listing 5.1 shows the `init` and `create` methods used to initialize the global storage of the smart contract. This state contains several variables. The `admin` variable (line 3) represents a privileged account used to manage sensitive operations such as updates or administrative controls in an Algorand smart contract. It is typically declared as `self.admin = GlobalState(Account)`, storing the authorized address directly in the global storage. This variable can be initialized at deployment using `self.admin.value = Txn.sender`(line 23), or assigned to a specific address with `self.admin.value = Account("ADDRESS")`. Access control is then enforced in the other methods through checks like `assert Txn.sender == self.admin.value`. Variables `party_a` and `party_b` (lines 4-5) store the addresses of the two choreography participants, i.e. A and B, respectively. The subsequent list of variables (lines 10-15) represents the marking of the edges e1, ..., e5 connecting the BPMN elements of the choreography (notably, e0 represents a spurious edge for denoting the enabled/disabled status of the start event). The values of these variables represent the choreography marking, which drives the execution flow. Message-specific variables

```
1  def __init__(self) -> None:
2      # Account variables
3      self.admin = GlobalState(Account)
4      self.party_a=GlobalState
5      (Account("ZLVEALW2YM......EDRRXIUY2TVWBUDLA"))
6      self.party_b = GlobalState
7      (Account("6WHFXXVYMB7.....WB3VKJWJLRI7ZBTIY"))
8
9      # Edge marking variables
10     self.e0 = GlobalState(UInt64(0))
11     self.e1 = GlobalState(UInt64(0))
12     self.e2 = GlobalState(UInt64(0))
13     self.e3 = GlobalState(UInt64(0))
14     self.e4 = GlobalState(UInt64(0))
15     self.e5 = GlobalState(UInt64(0))
16
17     # Message-specific variables
18     self.msg1_payload_x = GlobalState(UInt64(0))
19     self.msg2_payload_y = GlobalState(UInt64(0))
20
21     # Locking variable
22     self.locked = GlobalState(UInt64(0))  # 0 = unlocked, 1 =
           locked
23
24 def create(self) -> None:
25     self.admin.value = Txn.sender
26     self.e0.value = UInt64(1)
27     self.locked.value = UInt64(0)
```

Listing 5.1: Initialization of global state for the choreography contract.

(lines 18-19) keep track of the exchanged message data. The `locked` variable (line 22) is used to lock the smart contract methods (by means of assertion `assert self.locked.value == UInt64(0)` that causes method calls to fail) until the global storage is correctly updated. This variable acts as a safety switch: when it is set to false (i.e., value 0), the contract runs normally, instead, during updates it is set to true (i.e., value 1) to block all external activities, preventing inconsistencies; after the global storage updates, it is reset to false to resume execution safely. More details on the update procedure are provided in Section 5.3.2. Finally, the `create` method (lines 24-27) initializes the admin variable (line 25) and starts the process execution at the app creation time by placing one token on the edge **e0** (line 26).

Listing 5.2 shows how the logic of a BPMN choreography model is rendered in terms of Algorand Python smart contract methods. In fact, the translation approach we rely on maps each choreography element (i.e., event, gateway, or task) into a method. Specifically, the reported code refers to the methods corresponding to the first three elements of the running example

```
1  @subroutine
2  def start1(self) -> UInt64:
3      if self.e0.value == UInt64(1):
4          self.e0.value -= UInt64(1)
5          self.e1.value += UInt64(1)
6          return UInt64(1)
7      return UInt64(0)
8
9  @arc4.abimethod
10 def task1(self, msg1_param_x: UInt64) -> None:
11     assert self.locked.value == UInt64(0), "Contract is
           locked"
12     assert self.e1.value > UInt64(0), "Task 1 not active."
13     assert Txn.sender == self.party_a.value, "Only A can
           send msg1"
14     self.e1.value -= UInt64(1)
15     self.msg1_payload_x.value = msg1_param_x
16     self.e2.value += UInt64(1)
17     self.execute()
18
19 @subroutine
20 def xor1(self) -> UInt64:
21     if self.e2.value > UInt64(0):
22         self.e2.value -= UInt64(1)
23         if self.msg1_payload_x.value > UInt64(0):
24             self.e3.value += UInt64(1)
25         else:
26             self.e5.value += UInt64(1)
27         return UInt64(1)
28     return UInt64(0)
```

Listing 5.2: Contract methods corresponding to BPMN choreography elements (an excerpt).

model, i.e., the event start1, the task Task 1, and the XOR-split gateway xor1; the other elements are translated in a similar way. The method corresponding to the start event start1 (lines 1-7) checks if the spurious edge of the event (i.e., e0) is marked; in the negative case it returns false (i.e., value 0), because the condition that triggers the element execution is not satisfied, otherwise the method moves one token from the spurious edge to the outgoing edge (i.e., e1) and returns true (i.e., value 1). Notably, `@subroutine` are methods to encapsulate internal logic, where no user interaction is admitted. These functions are not publicly exposed and can only be called internally by the contract. In contrast, for tasks involving user interaction, we defined `@arc4.abimethod` methods, which serve as the public, externally callable interface of the smart contract.

The method corresponding to the task Task 1 (lines 9-17) allows an ex-

ternal user to exchange the message msg1; this is indeed a directly callable method (see annotation `@arc4.abimethod`). The method accepts as input the payload to be sent, which is then stored inside the global storage in the variable `self.msg1_payload_x` (line 15). Before executing the method, three checks are performed: (i) the contract must be unlocked (line 11); (ii) the task must be enabled according to the control flow of the choreography (i.e., the edge e1 must be marked) (line 12); (iii) the caller account must corresponds to participant A (line 13). If all checks are positive, the method consumes the incoming token (line 14), stores the input data in the global variable (line 15), marks the outgoing edge to advance the control flow (line 16), and resumes the choreography execution (line 17). Finally, the method corresponding to the XOR-split gateway xor1 (lines 19-28) checks if the gateway is enabled (i.e., the incoming edge e2 is marked); in the negative case it returns false, otherwise it executes the logic of the gateway and returns true. The gateway logic consists of consuming the incoming token (line 22) and checking the condition x>0 (line 23), on the basis of which one of the outgoing edges is marked (line 24 or line 26).

The execution of the choreography is carried out by the `execute()` method, which has the same code for any choreography (i.e., it is choreography agnostic). This is the public method that one has to invoke to activate the choreography once the smart contract has been deployed in the blockchain. It iteratively executes one choreography element method at a time until one element is successfully executed (denoted by the return value true); in our running example, the invoked methods are `start1()`, `xor1()`, `end1()`, and `end2()`. Notably, methods `task1()` and `task2()` are not invoked by `execute()`, because they have to be called by external users. If all elements of the choreography cannot be executed, the instance execution stops. In case there are enabled tasks, the execution is suspended and will be resumed when the waited message is delivered (see the call `execute()` as the last instruction of a task method's body). The complete code of the running example choreography is available in an online public repository [76].

Once the Algorand Python smart contract is derived from the choreography model, it has to be compiled to obtain the corresponding TEAL files, containing the approval and clear programs. Finally, these files can be deployed in the Algorand blockchain, creating an Algorand application, via the following command:

```
algokit goal app create
   --creator
XYWE34VCZTEGZJ5PBTGN44PT75ZGUG7T2COJFAYU6I3TUHW5J7TNSYI3ZE
   --approval-prog original_running_example.approval.teal
   --clear-prog original_running_example.clear.teal
   --global-byteslices 16 --global-ints 35 --local-byteslices
      8 --local-ints 8 --extra-pages 3
```

It is worth noticing that the above command specifies the maximum number of byte slices that may be stored in the global/local store (`global-byteslices`/`local-byteslices`), the maximum number of integer values that may be stored in the global/local store (`global-ints`/`local-ints`), and the additional program space for supporting larger TEAL programs (`extra-pages`). The specification of these limits is necessary in order to provide enough space for future extensions of the contract and related stores. However, after the application creation, such values are immutable.

Remark 1. *When deploying the choreography's smart contract, the admin user must take into account the limits imposed by the Algorand platform on the occupancy of the contract and the associated storage. In particular, the user has to foresee how much additional space the contract and its variables may require in the future, as these space limits cannot be modified later. Notably, as shown in Section 5.3.2, variables unused in the updated contract can be removed from the global storage, freeing the space they occupy.*

5.3.2 Choreography smart contract update

The second implemented phase includes the update of the choreography smart contract due to an adaptation of the choreography model. This is achieved by translating the adapted BPMN model into a new version of the choreography smart contract and then updating it in the blockchain.

To this purpose, we rely on the adapted choreography model illustrated in Fig. 5.2. While the translation of the choreography's logic follows the approach previously described, its update is managed by means of two dedicated methods: one for updating the smart contract code and checking update rights, and another one for updating the global storage, initializing new variables, or deleting old ones. Therefore, the update procedure requires two separate method invocations.

Listing 5.3 shows the methods involved in the update. Differently from the previous shown methods, `update()` (line 1) is a `baremethod`, i.e., a low-level method for administrative operations that does not require data type encoding. In particular, it handles the raw action `UpdateApplication`. This method contains all the checks required to manage permissions for update and can trigger additional logic. In our running example, the update method checks if the user performing the contract update is authorized (line 3). Here, we assume that only the admin user can perform the update operation; of course, such privilege could be granted to more than one user, on the basis of the specific authorization policies required by the considered scenario. In addition, the method locks the contract by setting the variable `locked` accordingly (line 4). This prevents executing any methods implementing the

```
1  @baremethod(allow_actions=["UpdateApplication"])
2  def update(self) -> None:
3      assert Txn.sender == self.admin.value, "Only admin can
           update the contract"
4      self.locked.value = UInt64(1)  # Lock the contract
5
6  @arc4.abimethod
7  def update_global_store(self) -> None:
8      assert Txn.sender == self.admin.value, "Only admin can
           update the global store"
9
10     # Added variables
11     self.e6.value = UInt64(0)
12     self.msg4_payload_k.value = UInt64(0)
13     self.msg3_payload_z.value = UInt64(0)
14     self.party_c.value =
           Account("OWSIGW6NHIYM.......5TI3QBHDWIH775ZZLE")
15
16     # Removed variables
17     del self.msg2_payload_y.value
18
19     self.locked.value = UInt64(0)  # Unlock the contract
```

Listing 5.3: Methods managing the smart contract update.

logic of the updated choreography until the global storage is updated as well, in order to avoid any inconsistency.

The admin user can perform the update of the smart contract by means of the following command:

```
algokit goal app update
   --app-id 1001
   --from
XYWE34VCZTEGZJ5PBTGN44PT75ZGUG7T2COJFAYU6I3TUHW5J7TNSYI3ZE
   --approval-prog updated_running_example.approval.teal
   --clear-prog updated_running_example.clear.teal
```

where `1001` is the identifier of the previously deployed Algorand application (generated and returned by the `create` command). The `update` command automatically invokes the `update()` method of the smart contract currently deployed; if it executes successfully, the current contract code is replaced by the new one.

The second method in Listing 5.3 (lines 6-19) permits updating the global storage to include new variables (representing, e.g., the marking of edges or the payload of exchanged messages) and/or delete variables no longer used in the updated contract. Similarly to the `update()` method, it can be successfully executed only by the admin user (line 8). The method performs additional actions that depend on the specific changes applied to the chore-

ography. In this example, it adds variables representing the new edge e6 (line 11), messages msg3 and msg4 (lines 12, 13), and participant C (line 14). Moreover, the method deletes the variable representing the removed message msg2 (line 17). The variables that have to be added or removed can be automatically determined by comparing the original smart contract and the one generated from the choreography after it has been modified. Notably, the replacement of msg2 into msg4 is achieved by deleting the former variable and adding the latter. Finally, the method always unlocks the contract (line 19).

Remark 2. *Although, in principle, unused variables could be kept in the global storage, we decided to delete them in order to free the memory space they occupy. Indeed, we recall that the dimension of the global storage is limited and fixed once and for all at application creation time.*

It is also worth noticing that all variables referred into the the two methods in Listing 5.3 must be declared in the `init()` method. This also applies to deleted variables (`msg2_payload_y` in the example). Anyway, in case of further updates, the variable deleted in this code could be omitted by subsequent versions of the contract code.

Remark 3. *It is worth noticing that we cannot perform the update of both the smart contract and the global storage in a single step. Indeed, if the code performing the update of the global storage (within `update_global_store()`) would be moved inside the `update()` method, it would not be executed when the update of the smart contract is called by means of the `algokit goal app update` command. This is because, the command executes the `update()` method of the currently deployed contract, i.e., the initial one, and not the updated one.*

5.3.3 Update Policies

The update mechanism of the smart contract can be governed through a policy-based control mechanism. A simple policy is illustrated in Listing 5.4.

```
1  def __init__(self) -> None:
2      ...
3      # Update policy
4      self.update_policy = GlobalState(UInt64(0))  # 0 = update
           disabled, 1 = update enabled
5
6  def create(self) -> None:
7      ...
8      self.update_policy.value = UInt64(1)
9
10 @arc4.abimethod
```

```
11 def task1(self, msg1_param_x: UInt64) -> None:
12     ...
13     self.update_policy.value = UInt64(0)  # Disable the update
14     self.execute()
15
16 @baremethod(allow_actions=["UpdateApplication"])
17 def update(self) -> None:
18     assert Txn.sender == self.admin.value, "Only admin can
           update the contract"
19     assert self.update_policy.value == UInt64(1), "Update is
           disabled"  # check the update policy
20     self.locked.value = UInt64(1)
```

Listing 5.4: Update policy example.

The purpose of this policy is to disable the update capability of the choreography once Task 1 has been executed. To achieve this, we rely on variable `update_policy` that acts as a flag: a value of 0 disables updates, while a value of 1 enables them. The variable is initialized to 1 at contract creation (line 8), indicating that updates are initially allowed. The update capability is disabled after the execution of Task 1 (line 13). The `update()` method (line 17) checks the value of `update_policy` (line 19); if updates are disabled, the assertion fails, causing the transaction that invokes `update()` to fail. Other simple update policies, based on the status of the choreography or the authorization rights of the user requesting the update, can be implemented similarly.

5.4 Cost Evaluation

To evaluate the feasibility of the proposed implementation, we present an evaluation of the cost analysis of the executed scenarios on the Algorand blockchain[1], expressed in both cryptocurrency and fiat currency.

More specifically, we have considered different execution scenarios concerning our running example. First, we executed the original choreography depicted in Figure 5.1 (scenarios 1 and 2), then we considered executions where the choreography is replaced by the one shown in Figure 5.2 (scenarios 3, 4, and 5). Due to Algorand's flat and low transaction cost structure, the expenses in *microAlgos* (μA) remain especially low (the fees for every transaction in our experiments are 1000 μA).

[1]We used the Algorand local network for these experiments.

Scenario	Cost (μA)				Total (USD)
	Deploy	Update	Execution	Total	
1	1000	–	3000	4000	0.000717
2	1000	–	2000	3000	0.000537
3	1000	2000	3000	6000	0.0011
4	1000	2000	3000	6000	0.0011
5	1000	2000	3000	6000	0.0011

Table 5.3: Execution costs for the running example.

We detail below the five scenarios, whose results are reported in Table 5.3, also with the USD conversion of total costs[2].

1. **Scenario 1**: The choreography is deployed (1000 μA) and start1 is executed (1000 μA). Then, Task 1 is executed (1000 μA), providing an input value greater than 0, which directs the execution flow to Task 2. After the execution of Task 2 (1000 μA), the choreography terminates with end1.
2. **Scenario 2**: The choreography is deployed (1000 μA) and start1 is executed (1000 μA). Then, Task 1 is executed (1000 μA) providing an input value equal to 0, leading the choreography directly to terminate with end2.
3. **Scenario 3**: After deploying (1000 μA) the choreography and executing start1 (1000 μA), an update is performed via methods `update()` (1000 μA) and `update_global_store()` (1000 μA). Following this, Task 1 is executed (1000 μA) providing an input value greater than 5, which triggers Task 2. The execution of Task 2 (1000 μA) then leads the choreography to terminate with end1.
4. **Scenario 4**: After deploying (1000 μA) the choreography and executing start1 (1000 μA), an update is performed via methods `update()` (1000 μA) and `update_global_store()` (1000 μA). Following this, Task 1 is executed (1000 μA) providing an input value less than 5, which triggers Task 3. The execution of Task 3 (1000 μA) then leads the choreography to terminate with end2.
5. **Scenario 5**: The choreography is deployed (1000 μA) and start1 is executed (1000 μA). Then, Task 1 is executed (1000 μA) providing an input value greater than 5, which directs the execution flow to Task 2. An update is performed via methods `update()` (1000 μA) and `update_global_store()` (1000 μA). The execution of the updated Task 2 (1000 μA) then leads the choreography to terminate with end1.

[2]Based on the ALGO/USD conversion at the time of writing (16/04/2025).

Notably, if the update is performed after Task 1's execution, task Task 3 can never be executed. In fact, the execution of the choreography automatically progresses until it reaches a task, which requires the related task initiator user to send the message by calling the method corresponding to the task. Only during the period from the task enabling and its invocation, the `update()` method can be called to update the contract. Hence, once Task 1 is executed, the execution flow stops at Task 2 and therefore, after the update, Task 3 is unreachable (hence, the corresponding method is disabled).

If we consider choreography-based approaches in the literature, such as ChorChain [30] and FlexChain [33], we can draw a general comparison with our solution, despite the differences in their objectives and implementation architectures. Indeed, we observe that on a performant blockchain like Polygon[3], executing a choreography of 10 messages incurs approximately 0.10 USD for a supply chain scenario and 0.08 USD for an X-ray analysis scenario. In contrast, our implementation, for the same number of tasks, would require approximately 11,000 μA. This corresponds to an estimated total amount of about 0.0026 USD, demonstrating negligible costs and economic viability of our solution.

[3]Based on the POL/USD conversion at the time of writing (19/05/2025).

CHAPTER 6

BPMN-TO-ALGORAND SMART CONTRACT GENERATION TOOL

This chapter elaborates on the design and development of an automated BPMN-to-Algorand smart contract generation tool to address the fourth research objective:

> "*To develop a BPMN-to-Algorand smart contract generation tool that translates BPMN choreography models into executable Algorand smart contracts.*"

This automated framework transforms Business Process Model and Notation (BPMN) choreographies into executable Algorand ARC-4 smart contracts. The system bridges the conceptual gap between high-level business process design and blockchain implementation. It allows organizations and developers to describe multi-party business interactions using BPMN diagrams and to automatically generate verifiable, deployable smart contracts that can execute these workflows on the Algorand blockchain.

The tool[1] was conceived to address the complexity of manually translating business workflows into blockchain logic, which is often prone to human error and inconsistent interpretation. By automating this translation, the system ensures process fidelity, auditability, and repeatability. Moreover, it supports version control, intermediate representations, and interoperability features that make it adaptable to future blockchain environments.

[1]The BPMN-to-Algorand Smart Contract Generation Tool is available on GitHub: `https://github.com/nawaz-malla/bpmn-to-algorand-sc`.

6.1 Objectives and Scope

The central goal of this tool is to provide a fully automated and transparent method for generating blockchain-based smart contracts from BPMN choreography diagrams. It supports the entire lifecycle of process translation, from model parsing and intermediate representation to smart contract generation and deployment.

At its core, the tool parses BPMN choreography models—those that describe interactions between independent participants such as clients, suppliers, or services—and converts them into an intermediate, language-agnostic data model. This intermediate representation (IR) serves as the backbone for generating executable Algorand contracts following the ARC-4 standard, which defines the interaction structure and stateful behavior of Algorand applications. Beyond this, the tool also maintains detailed versioning records so that every modification in the BPMN model leads to the creation of a new, traceable version of the corresponding smart contract and IR.

The system's scope extends beyond mere translation: it is designed to ensure continuity of process execution, support dynamic model evolution, and produce pseudocode suitable for future adaptation into other blockchain or smart contract languages.

6.2 System Architecture

The architecture of the BPMN-to-Algorand tool follows a modular and layered structure, with each component responsible for a specific phase of the transformation pipeline. Figure 6.1 presents the overall architecture of the tool. The foundation of the system lies in the BPMN Parser, which reads and interprets '.bpmn' XML files. This parser identifies key process elements such as participants, messages, tasks, and control gateways, and then constructs a formal internal representation known as the Choreography Intermediate Representation (IR).

The IR serves as an abstract layer between the business model and the blockchain logic. It encapsulates all semantic details—participants, tasks, message flows, and control dependencies—in a structured and JSON-serializable format. This abstraction ensures that the business process can later be converted into smart contract code in any target blockchain language.

Once the IR is generated, it is passed to the Contract Generator module, which produces a fully executable Algorand ARC-4 smart contract written in Python. This contract includes the full set of state variables, task methods, and subroutines necessary to replicate the original BPMN workflow's behavior on-chain. Each participant, task, and decision gateway in the model is

Figure 6.1: Overall Architecture of the BPMN-to-Algorand Smart Contract Translator

translated into an equivalent Algorand construct such as global states, ABI methods, and subroutine functions.

Alongside the contract generator, the Pseudocode Generator creates a platform-independent, algorithmic representation of the workflow. This pseudocode allows the same choreography to be implemented on other platforms, such as Ethereum or Hyperledger, making the tool highly interoperable.

A separate component, the Version Manager, maintains the evolutionary history of all models and contracts. By comparing newly parsed IRs with existing ones using deep structural comparison, it automatically detects modifications to the BPMN model—such as added participants, updated tasks, or new gateways—and stores each new version in an organized directory structure. Each version folder contains the original BPMN file, its intermediate representation, and the generated contract, providing complete traceability of process evolution.

The User Interface (UI) complements these backend modules with a user-friendly front-end built using Streamlit. The interface allows users to upload BPMN files, visualize their intermediate representations, and generate corresponding smart contracts directly from a web browser. It also integrates with Algorand's LoRa Explorer, enabling real-time inspection of deployed contracts and their transaction states.

6.3 Workflow and Execution

The operational workflow of the tool follows a clearly defined sequence of phases. When a user provides a BPMN file, the system first enters the parsing phase, during which the XML structure is analyzed and converted into the internal IR model. Each BPMN participant is recognized as an independent account entity, while tasks and message flows are interpreted as contract methods and state transitions. Gateways such as XOR or AND nodes are mapped to subroutines that handle decision-making logic. Figure 6.2 illustrates an overview of workflow and execution of the BPMN-to-Algorand smart contract generation tool.

Following parsing, the intermediate representation phase stores the IR in a persistent JSON format. This allows for reusability, portability, and verification without the need to reprocess the BPMN file. The IR acts as a single source of truth for both code generation and future interoperability efforts.

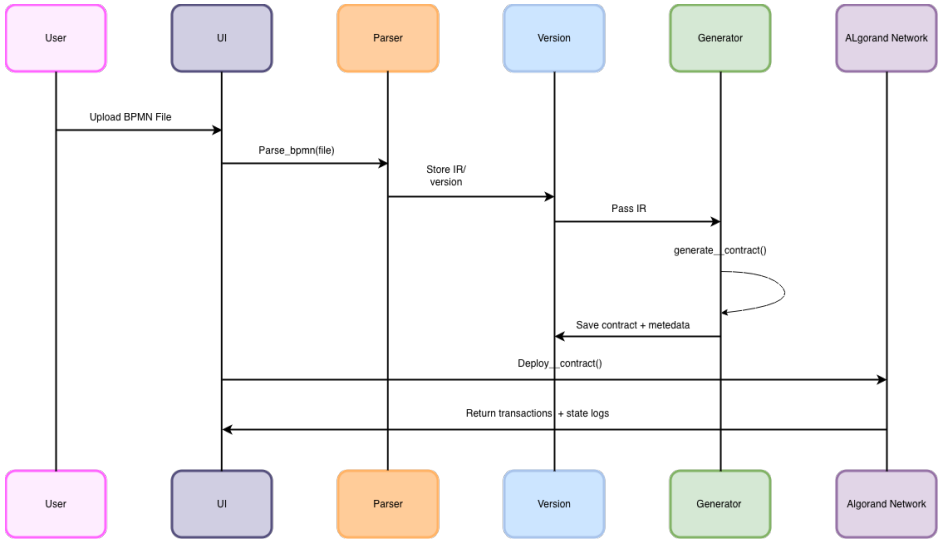


Figure 6.2: Sequence Diagram of BPMN to Smart Contract Translation Workflow

In the contract generation phase, the IR is transformed into a complete Algorand smart contract. The generated code defines all participants, initializes global state variables, and specifies the control flow using process flags that correspond to BPMN edges. Each BPMN task results in a distinct method within the contract, and gateways are represented as conditional subroutines that emulate token flow and branching behavior. The resulting contract can execute deterministically on the Algorand blockchain and includes logging and administrative capabilities such as contract updates and

deletions.

The version management phase ensures the integrity and traceability of process evolution. When a BPMN file is modified and reprocessed, the version manager compares the new IR with previously saved versions. If differences are detected, a new version directory is created to store the updated IR and generated contract. This mechanism enables developers and organizations to track changes in their business processes over time while maintaining a complete historical record.

Finally, during the deployment and monitoring phase, the user can deploy the generated contract to Algorand's local network and observe its execution in real time using the LoRa Explorer integration. The tool provides complete visibility into transaction flow and process state transitions, allowing users to validate the correctness of the generated contract.

6.4 Software Design and Implementation

The tool is implemented entirely in Python 3.13, leveraging the Algopy ARC-4 framework for contract generation and Streamlit for the web interface. XML parsing is performed using Python's built-in 'ElementTree' library, while DeepDiff is used for version comparison. The architecture follows a modular design pattern that separates concerns across parsing, IR modeling, generation, and persistence layers. Data interchange between modules is achieved through JSON files, providing both flexibility and transparency. The system's internal design adheres to an object-oriented, modular structure. Figure 6.3 shows the class diagram representing the relationships between the key components of the tool. The generated contracts follow a

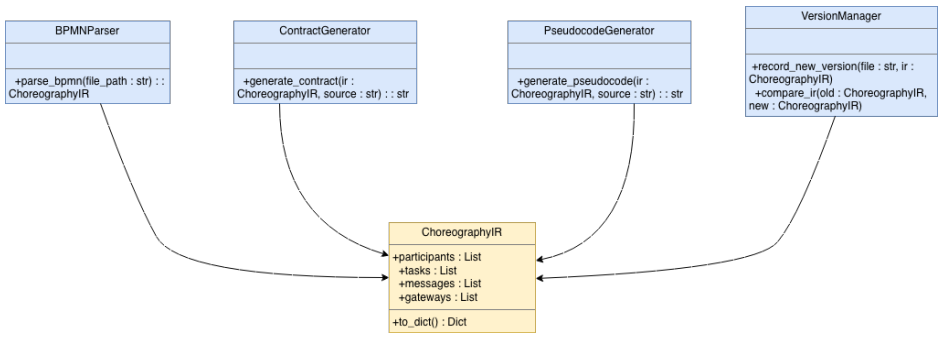


Figure 6.3: Class Diagram of the BPMN-to-Algorand Tool Components

standardized structure consistent with Algorand's ARC-4 protocol. Each contract defines its participants, global states, and message exchanges, and embeds a dynamic execution engine that replicates BPMN token flow. The

engine, composed of subroutines and control edges, automatically fires transitions between states, logging progress until the process reaches completion. Administrative functions such as ‘update‘, ‘unlock‘, and ‘delete‘ ensure secure lifecycle management of the contract on-chain.

6.5 Functional Features

One of the defining characteristics of the BPMN-to-Algorand tool is its adaptability. The contract generator is fully dynamic: regardless of the complexity or structure of the input BPMN file, the tool automatically generates all necessary global states, methods, and control transitions. This dynamic behavior extends to gateways and events, which are translated into executable subroutines that mirror the conditional and parallel behavior of BPMN models.

Every transaction and process step is logged on the blockchain, guaranteeing transparency and traceability. The tool also provides version awareness, enabling users to retrieve and compare older and newer versions of both BPMN and generated contracts. Furthermore, the IR model serves as a foundation for extending the system to other blockchains or programming languages, allowing future researchers or developers to build multi-platform smart contract compilers from the same framework.

6.6 User Interface Design

The Streamlit-based web interface of the BPMN-to-Algorand Smart Contract Generation Tool enhances accessibility, allowing even non-technical users to upload BPMN files, visualize intermediate representations, and generate or inspect Algorand contracts with a single click. The integration with Algorand's LoRa Explorer further enhances its utility by providing direct insights into the state of deployed contracts.

The interface integrates all major functionalities from BPMN model upload and intermediate representation (IR) visualization to pseudocode and smart contract generation within a single, interactive dashboard.

6.6.1 Main Dashboard

This section illustrates the main user interface components and describes the key stages of interaction during the translation process. The main dashboard serves as the central access point for the entire translation workflow. It displays the tool's title, description, and a dedicated upload panel for importing BPMN models.

At the top of the interface, users can also access action buttons for downloading the generated outputs and connecting directly to the Algorand LoRa Explorer for blockchain-level inspection. The design emphasizes clarity and usability, enabling seamless navigation across different stages of the process. Figure 6.4 presents the overview of the main dashboard of the Tool.

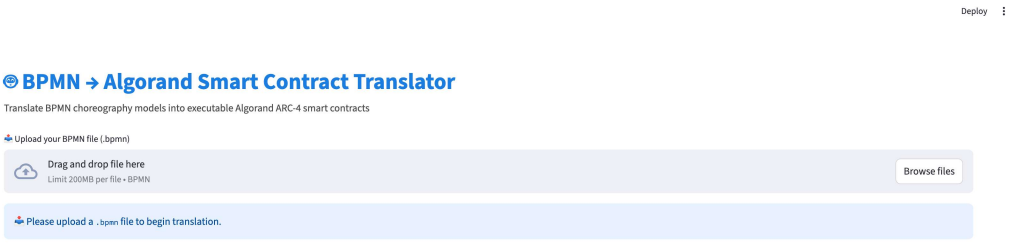


Figure 6.4: Main Dashboard of the Tool

6.6.2 Uploading a BPMN Model

The upload panel supports both drag-and-drop and manual file selection for BPMN diagrams in .bpmn format. Once the user uploads a file, the system validates it and automatically stores it in the /uploads directory for processing. Figure 6.5 shows the process of uploading a BPMN model file through the system interface.

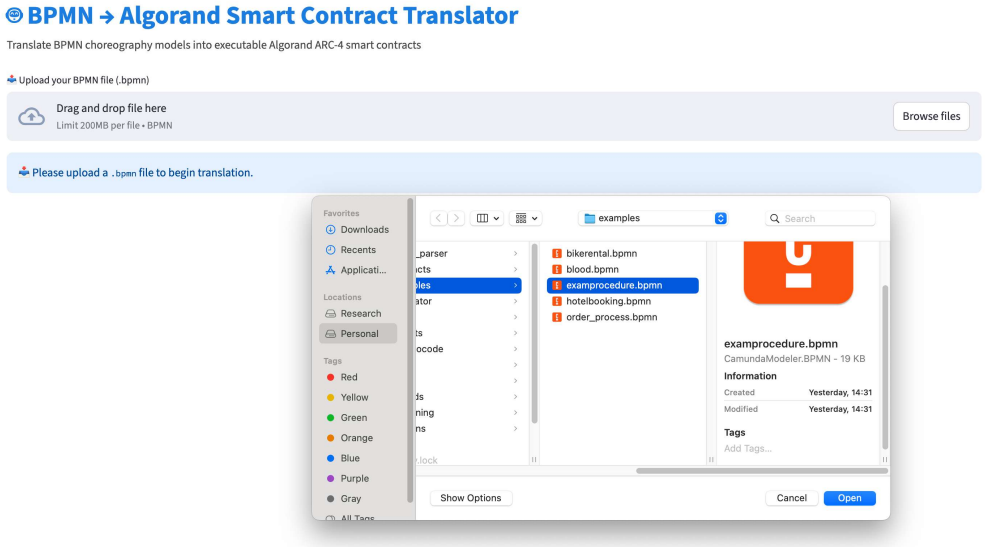


Figure 6.5: Uploading a BPMN Model File

This action triggers the BPMN Parser, which begins extracting structural elements such as participants, tasks, and message flows. Upon successful parsing, the tool transitions to the IR visualization stage, ensuring a smooth and interactive workflow for the user.

6.6.3 Viewing the Intermediate Representation

After the parsing phase, the tool generates a structured Intermediate Representation (IR) of the BPMN model, displayed in the interface as a JSON-based schema. Figure 6.6 presents a visualization of the generated intermediate representation (IR) produced by the tool.

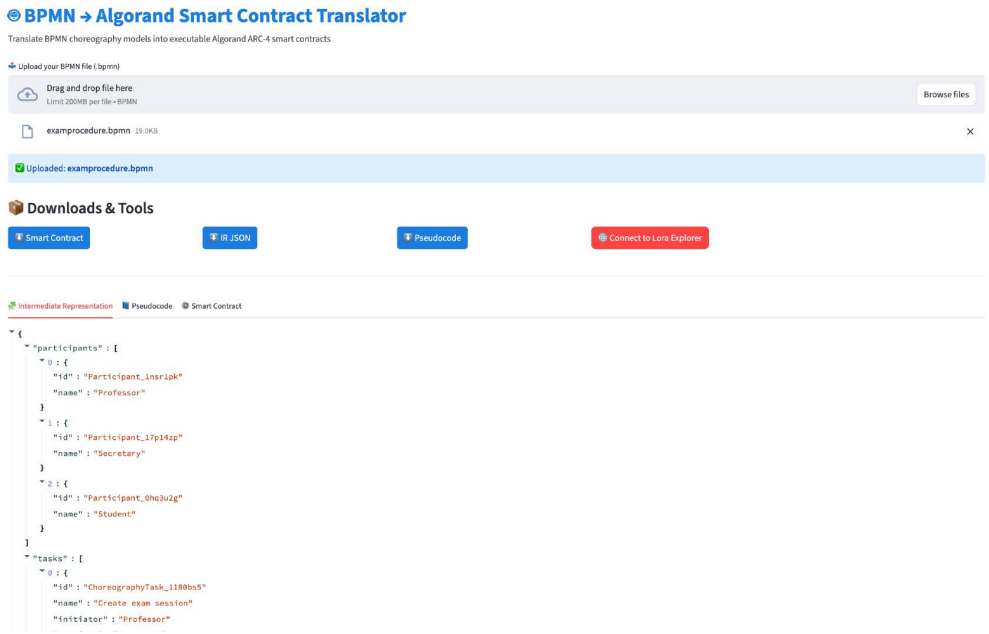


Figure 6.6: Visualization of the Generated Intermediate Representation

The IR outlines key process components including participants, tasks, gateways, and message flows,offering users a detailed view of the internal process structure. This visual feedback ensures that users can verify model accuracy before proceeding to pseudocode or smart contract generation. Additionally, the IR is automatically stored for version control and reuse, supporting long-term process management and reproducibility.

6.6.4 Generating Pseudocode

Once the IR is confirmed, the Pseudocode Generator converts it into an pseudocode representation. The pseudocode serves as a human-readable abstraction of the BPMN logic, reflecting the process flow, participant roles, and decision points. Figure 6.7 presents the Generated Pseudocode.

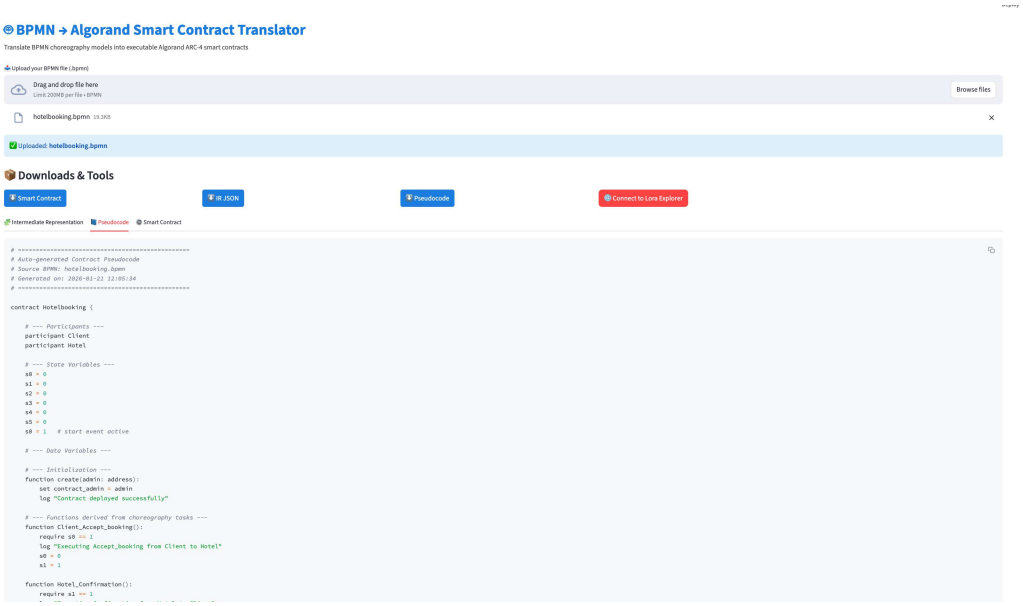


Figure 6.7: Visualization of the Generated Pseudocode

This output acts as a conceptual bridge between the high-level business model and the eventual smart contract code. It also provides cross-platform interoperability, as the same pseudocode can serve as a foundation for implementing the process in other blockchain environments such as Ethereum or Hyperledger.

6.6.5 Generating the Algorand Smart Contract

The Contract Generator module produces a fully executable Algorand ARC-4 smart contract written in Python. The generated contract defines all participants, global state variables, and process control flows corresponding to the BPMN model's tasks and gateways. Figure 6.8 shows the generated Algorand ARC-4 smart contract.

Each task in the BPMN diagram becomes a distinct ABI method, while gateways are implemented as subroutines that manage conditional or parallel execution paths. The generated contract is displayed directly in the user interface, where users can review, edit, or save it locally for deployment.

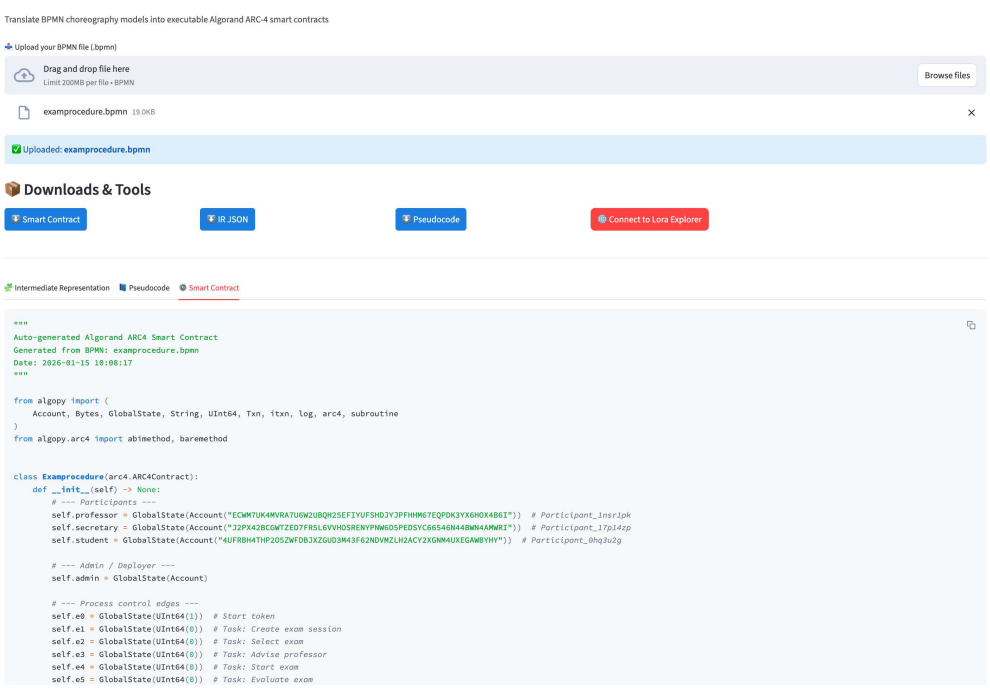


Figure 6.8: Algorand ARC-4 Smart Contract Generated by the Tool

6.6.6 Downloading and Exploring the Results

After the translation process completes, the interface provides download options for all generated artifacts. In addition to local downloads, the interface offers a direct integration with the Algorand LoRa Explorer, which allows users to inspect the deployed contract's state, transaction history, and event logs in real time. This functionality enhances transparency and verifiability, ensuring that the generated contracts can be monitored directly on the blockchain. Figure 6.9 highlights the available download options and the LoRa Explorer integration interface.

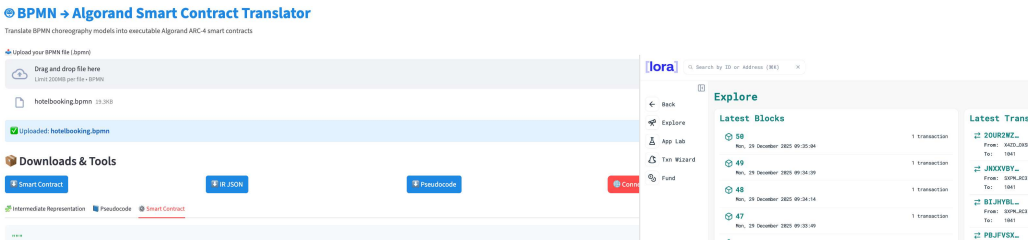


Figure 6.9: Download Options and LoRa Explorer Integration

Overall, the user interface embodies the system's guiding principles of simplicity, transparency, and automation, making the BPMN-to-Algorand

tool both powerful and accessible to users from technical and non-technical backgrounds alike.

6.7 Discussion

The BPMN-to-Algorand Smart Contract Generation Tool represents a significant step forward in the integration of business process modeling and blockchain automation. By seamlessly converting BPMN choreographies into executable Algorand ARC-4 smart contracts, the system unifies high-level process design with decentralized execution. Its modular architecture, version control capabilities, and extensible design make it both a practical engineering tool and a research contribution to the field of model-driven blockchain development.

Through its ability to translate abstract workflow models into real, executable smart contracts, the tool reduces the technical barrier to blockchain adoption while maintaining semantic correctness and full lifecycle traceability. Future work may focus on enhancing the reasoning behind gateway translation, automating data-type inference from BPMN variables, and extending the tool's interoperability to multiple blockchain ecosystems.

CHAPTER 7

SOLIDITY TO ALGORAND SMART CONTRACTS

This chapter elaborates on the AI-assisted translation of Solidity smart contracts into Algorand smart contracts to address the following objective:

> " *To provide an AI-assisted methodology for migrating existing business process implementations to the Algorand blockchain platform.*"

The blockchain ecosystem, however, is far from homogeneous. Leading platforms like Bitcoin [88], Ethereum [18], Solana [15] and Algorand [19] each provide unique architectures, languages, and execution environments. Despite that, so far most of the solutions for blockchain-based execution of business processes relied on Ethereum, thus encoding the business logic of processes into solidity smart contracts. However, the reliance on Ethereum also introduces several limitations for organizations seeking to adopt alternative platforms. Issues such as high transaction fees, network congestion, and evolving protocol changes incentivize the exploration of more scalable, cost-efficient, and secure alternatives.

On the other hand, as discussed in the previous chapters, the Algorand blockchain provides great benefits in terms of performance, costs, and most of all the flexibility. Therefore , to migrate solidity implementations of business processes, we investigate whether current LLMs can help bridge this gap by assisting in the translation process from Solidity (the most popular smart contract language) to Algorand Python.

While recent research has demonstrated the potential of LLMs for code translation across various programming languages, Algorand Python presents unique challenges. As a relatively new language with a specialized syntax for blockchain operations, it is likely underrepresented in the training data

of most LLMs. Furthermore, the fundamental architectural differences between Ethereum's account-based model and Algorand's architecture mean that direct translations often require significant restructuring beyond simple syntax conversion. Specific challenges include differences in state management (global vs. local storage), transaction handling models, and execution contexts between EVM and AVM.

Our exploratory investigation addresses three key questions:

RQ1 To what extent can current LLMs generate syntactically valid and functionally equivalent Algorand Python code from Solidity contracts?

RQ2 Which prompting strategies appear most effective for guiding LLMs in this cross-language translation task?

RQ3 What are the common error patterns and challenges that emerge when using LLMs for blockchain code translation?

To address these questions, we experimented with four leading LLMs—ChatGPT, Claude, Qwen, and DeepSeek, (shown in Table 7.1)–and evaluated their performance on translating 20 Solidity smart contracts using different prompting strategies, including zero-shot, few-shot, chain-of-thought, and combined approaches.

Model Name	**Developer/Organization**	**Version / Access**
ChatGPT	OpenAI	GPT-4 / GPT-3.5
Claude	Anthropic	sonnet 4.5
Qwen	Alibaba Cloud	Qwen 2.5
DeepSeek	DeepSeek AI	DeepSeek-latest version

Table 7.1: Large Language Models (LLMs) Evaluated in the Experiment

We manually assessed the translation quality based on syntax correctness, functional equivalence, and compilation success, tracked error patterns, and documented the effort required to transform LLM-generated code into fully functional Algorand smart contracts. Table 7.2 presents the experimental setup for the translation of smart contracts.

The contributions of this preliminary study include:

1. A collection of corresponding Solidity and Algorand Python smart contracts that can serve as examples for few-shot learning in future LLM applications
2. Practical insights regarding the time, effort, and expertise required when using LLMs to translate between blockchain programming languages, namely Solidity to Algorand.

Aspect	Description
Objective	Evaluate the translation performance of leading Large Language Models (LLMs) on Solidity smart contracts.
Number of Smart Contracts	20 Solidity smart contracts.
Task Type	Translation between Solidity and Algorand Python programming language).
Evaluation Metrics	Accuracy, syntactic correctness, semantic fidelity.

Table 7.2: Experimental Setup for Translating Solidity Smart Contracts with LLMs

3. An analysis of common translation errors and comparative performance metrics across different LLMs and prompting techniques for Solidity to Algorand Python translation
4. A preliminary evaluation framework for LLM-generated smart contract translations, including metrics for measuring syntactic correctness, functional similarity, and required manual correction effort
5. A detailed mapping between Solidity and Algorand Python language constructs that can inform future research and tool development in this area. [1]

While our findings are based on a limited sample of smart contracts and a single developer's expertise, and represent initial explorations rather than comprehensive validation, they provide valuable practical insights for developers seeking to leverage LLMs in blockchain development workflows. Our error pattern analysis and mapping between language constructs offer immediate utility for developers working across these ecosystems, even as LLM capabilities continue to evolve. Moreover, our observations can guide future research efforts, help developers set realistic expectations when using LLMs for code translation, and identify specific areas where current models require improvement or human oversight.

As both LLMs and blockchain technologies continue to evolve, early explorations like ours play an important role in identifying promising research directions, surfacing practical challenges, and developing initial frameworks. Our experience report contributes to the emerging body of knowledge at the intersection of artificial intelligence and blockchain development, offering practical perspectives on how these technologies can be combined to reduce barriers to entry and enable greater interoperability across blockchain ecosys-

[1] All data, code, and translation artifacts described in this paper are available on a GitHub repository : `https://github.com/nawaz-malla/Algosol-Dataset`

tems.

7.1 Methodology

We conducted a preliminary, small-scale evaluation of large language models (LLMs) for translating Solidity smart contracts to Algorand Python. Our approach examined the models' ability to assist in the translation process by generating code that, with appropriate human guidance and verification, could be refined into syntactically valid and functionally equivalent Algorand implementations.

Criterion	**Purpose**
Syntax Validity	Checks if translated code compiles correctly.
Semantic Accuracy	Ensures logic and function equivalence.
Readability	Measures clarity and maintainability.
Prompt Efficiency	Evaluates token cost vs. performance gain.

Table 7.3: Evaluation Framework for Solidity Translation Performance

We measured translation quality through metrics including correct lines of code (LOC) percentage, successfully translated methods, and error patterns across a limited set of 20 manually verified contracts. While small in scale, this focused assessment provides preliminary insights into both the capabilities of current LLMs and the specific translation challenges between these blockchain ecosystems. Table 7.3 presents the evaluation criteria used in this study.

7.1.1 Dataset Collection and Preparation

Our preliminary investigation employed a bidirectional translation approach to evaluate LLM performance in translating between Solidity and Algorand Python. Given the scarcity of Algorand smart contracts in public repositories, we curated a small dataset of 10 contracts through the following process:

1. We selected 6 Algorand Python smart contracts, which represent available exemplar contracts from official Algorand Foundation repositories [50], and 4 Solidity smart contracts from [124]. These contracts encompass different functionality, including marketplace operations, voting systems, auctions, calculations, attendance tracking, data storage, banking, and token handling.

2. The 6 Algorand contracts were manually translated to Solidity by the first author, who has medium-level development experience in both Solidity and Algorand. The translation process focused on preserving functional equivalence while adapting to Ethereum's architectural model. This included mapping Algorand's global state management to Solidity storage variables, converting transaction handling patterns, and adapting Algorand's program logic to Solidity's contract structure.

We included both manually translated Algorand contracts and native Solidity contracts to provide a basic evaluation reference point while still capturing some authentic Solidity development patterns, acknowledging that this preliminary exploration serves as a starting point rather than a comprehensive assessment. For evaluation, we measured several key metrics: (1) percentage of correctly translated lines of code, (2) number of correctly translated methods, (3) number of iterations required to achieve compilable code, and (4) time required for manual correction when necessary. Correctness was assessed through syntax validation, compilation testing, and functional comparison with reference implementations. The evaluation metrics are summarized in Table 7.4.

Metric	**Description**	**Scale**
Translated Lines	Lines of code correctly translated.	Percentage
Translated Methods	Correctly translated functions.	Count
Compilation-Iterations	Attempts to obtain compilable code.	Count
Manual Fix Time	Time needed for manual corrections.	Minutes

Table 7.4: Key metrics used to evaluate Solidity smart contract translation performance.

While we acknowledge limitations in this methodology, particularly the modest sample size and potential biases in the manual translations, it provides a practical framework for generating preliminary insights into LLM capabilities in cross-blockchain smart contract translation.

7.1.2 Solidity Constructs to Algorand Python Mapping

We present a mapping between Solidity and Algorand Python language constructs to bridge the architectural differences between these blockchain platforms. Created through analysis of documentation, example contracts, and developer resources, this mapping identifies equivalent constructs across contract structure, data types, function modifiers, operations, and state man-

agement. A concise overview is presented in Table 7.5 and the full extended mapping is available in the project repository [75].

Category	Solidity	Algorand Python
Contract Structure	contract X { ... }	class X(algopy.ARC4Contract):
	is inheritance	Multiple base classes
Basic Types	uint256, uint8	UInt64
	address	Account
	string	String
	bytes, bytes32	Bytes
	bool	bool
	mapping	GlobalState
Complex Types	Arrays	Box, BoxRef
	Structs	Multiple boxes
	Enums	Constants
Function Modifiers	public	@algopy.arc4.abimethod()
	view	@algopy.arc4.abimethod (read-only=True)
	pure	@algopy.subroutine
	payable	Check Txn.amount in function
Operations	require(x, msg)	assert x, msg
	+=, -=, *=, /=	Direct operators
	revert("msg")	assert False, "msg"
Visibility	public	@algopy.arc4.abimethod()
	private	@algopy.subroutine
	internal	@algopy.subroutine
	external	@algopy.arc4.abimethod()
Events	event X(...)	log(...)
	emit X(...)	log(Bytes(...))

Table 7.5: Feature Mapping: Solidity vs Algorand Python

This mapping serves dual purposes: it provides a structured reference for evaluating translation correctness and forms a key component of our prompting strategies. By incorporating this mapping into our prompts, we aimed to improved LLMs' understanding of platform differences and improve translation accuracy. To our knowledge, no existing studies focus specifically on Solidity-to-Algorand Python translation, making this mapping a novel contribution.

7.1.3 Selected Prompts

Prompt engineering plays an important role in optimizing LLM performance for specialized tasks like code translation. Our prompt selection was guided by established research on LLM capabilities [58, 46] and progressed systematically from basic to more sophisticated techniques. Table 7.6 summarizes the four prompting strategies employed in our study. Detailed prompts are available on our GitHub repository [75]. We selected these prompting tech-

Prompting Strategy	**Description**	**Expected Effect**
Zero-Shot	without examples.	generalization.
Few-Shot	few sample translations.	learning from context.
Chain-of-Thought (CoT)	step-by-step reasoning.	reasoning clarity.
Combined Approach	examples and reasoning.	additive benefits.

Table 7.6: Prompting strategies used in the experiment.

niques based on their demonstrated effectiveness in complex reasoning and code generation tasks.

Zero-shot prompting served as our baseline, testing the models' inherent knowledge of both languages without additional context. This approach, while straightforward, helps establish the fundamental capabilities of each model.

Few-shot prompting leverages the models' ability to learn from examples, providing concrete translation patterns between Solidity and Algorand Python. This approach is particularly valuable for Algorand, which may be underrepresented in the models' training data. Our few-shot prompts included the mapping table (Table 7.5), and two short corresponding smart contract examples that demonstrate basic constructs before presenting the translation task.

Chain-of-Thought (CoT) prompting exploits the models' reasoning capabilities by decomposing the translation process into discrete steps. This technique has shown particular effectiveness for complex tasks requiring multi-step reasoning [58]. We guided each model through a structured analysis of the input contract's features before generating Algorand code.

7.2 Evaluation

This section documents our practical experience using four LLMs—ChatGPT, Claude, Qwen, and DeepSeek—to assist in translating Solidity smart contracts to Algorand Python. Rather than presenting a comparative evaluation of model performance, we share observations from our translation effort of 10 Solidity contracts using a few popular prompting techniques. Our goal was to explore whether current LLMs can serve as useful assistants in cross-blockchain development workflows, specifically for developers seeking to port Ethereum applications to Algorand's ecosystem.

7.2.1 Translation Process and Developer Workflow

Our translation workflow followed a consistent pattern across all contracts and models. We began with careful prompt construction, developing few-shot, chain-of-thought, and combined prompt strategies that incorporated Solidity-to-Algorand feature mappings (Table 7.5). Each LLM generated initial translations, which we then refined through iterative cycles when errors appeared. These refinements continued for up to four iterations, with LLMs attempting to correct issues based on error messages and guidance. When this iterative refinement reached diminishing returns, a developer with knowledge of both ecosystems manually intervened to correct remaining issues. Translations were considered complete only when they successfully compiled in the Algorand development environment. This approach reflects practical development scenarios where LLM assistance reduces initial translation effort, while domain expertise remains essential for resolving complex implementation challenges.

7.3 LLM-Assisted Translation

Our experience, quantified for four different LLM models is quantified in next subsections, revealed several patterns across prompting strategies. Notably, the zero-shot prompt was excluded from this analysis as it consistently generated PyTeal code instead of the intended Algorand Python code, thereby failing to meet the evaluation criteria. The specific models evaluated in this study are listed in Table 7.7, which provides the essential details for each—namely the model name, its developer, and the version accessed during experimentation.

Model Name	Developer	Version Access
ChatGPT	OpenAI	GPT-3.5
Claude	Anthropic	Claude 3.5 sonnet
Qwen	Alibaba Cloud	Qwen 2.5
DeepSeek	DeepSeek AI	DeepSeek-V3

Table 7.7: Large Language Models (LLMs) Evaluated in the Experiment

7.3.1 ChatGPT: GPT 3.5

ChatGPT emerged as a robust generalist in smart contract translation, demonstrating strong capabilities in understanding Solidity patterns and generating syntactically correct Algorand Python code as presented in Table 7.8. The model excelled in translating basic contract structures and common programming patterns, achieving approximately 70% accuracy in initial translation attempts for straightforward contracts. However, ChatGPT struggled with Algorand-specific constructs, particularly the box storage model and ARC4 method decorators, often requiring 3-5 iterative refinements to achieve compilable code.

Task	Few-Shot	CoT	FS+CoT
Hello World	66.66%	55.55%	77.77%
Voting	62.08%	55.00%	60.41%
Auction	68.86%	63.20%	66.98%
Calculator	65.21%	34.78%	56.52%
Attendance	53.33%	34.44%	50.55%
Counter	70.37%	44.44%	62.96%
Tokenpool	59.37%	38.54%	55.20%
Banking	78.00%	48.00%	72.00%
Storage	61.21%	52.12%	59.39%
Marketplace	67.53%	45.45%	62.33%

Table 7.8: Accuracy of ChatGPT by code correctness under different prompting techniques.

7.3.2 Claude: Claude 3.5 Sonnet

Claude demonstrated exceptional performance in understanding the architectural nuances between blockchain platforms, showing particular strength in translating complex state management patterns and contract inheritance

hierarchies. With method-level translation accuracy reaching 75% for intermediate complexity contracts, Claude proved most effective in maintaining functional equivalence across platforms as shown in Table 7.9. The model's structured reasoning capabilities made it particularly responsive to chain-of-thought prompting, allowing it to systematically address translation challenges step by step. However, Claude occasionally over-engineered solutions for simple contracts and showed limitations in handling Algorand's transaction grouping patterns, requiring manual intervention for optimal payment verification implementations.

Task	**Few-Shot**	**CoT**	**FS+CoT**
Hello World	88.88%	55.55%	88.88%
Voting	72.08%	61.25%	67.91%
Auction	76.41%	64.15%	74.52%
Calculator	82.60%	65.21%	78.26%
Attendance	73.33%	57.77%	70.55%
Counter	85.18%	59.25%	77.77%
Tokenpool	67.70%	46.87%	60.41%
Banking	76.00%	58.00%	70.00%
Storage	65.45%	59.39%	62.42%
Marketplace	68.83%	53.24%	61.03%

Table 7.9: Accuracy of claude by code correctness under different prompting techniques.

7.3.3 Qwen: Qwen 2.5

Qwen distinguished itself through efficient translation of core business logic and reliable handling of basic data types and control structures. The model achieved remarkable success with simpler contracts under 50 lines of code, generating compilable Algorand Python with minimal iterations. Qwen's pragmatic approach proved valuable for straightforward translations, though it encountered challenges with complex data structures and advanced Algorand features like box references and global state management. The model responded well to few-shot prompting with concrete examples, demonstrating rapid learning from provided translation patterns while maintaining consistent output quality across multiple contract types as documented in Table 7.10.

Task	Few-Shot	CoT	FS+CoT
Hello World	66.66%	44.44%	55.55%
Voting	52.91%	46.25%	49.58%
Auction	59.43%	52.28%	54.71%
Calculator	60.86%	34.78%	52.17%
Attendance	53.88%	43.88%	47.22%
Counter	62.96%	44.44%	55.55%
Tokenpool	47.91%	40.62%	48.95%
Banking	62.00%	46.00%	54.00%
Storage	52.72%	47.87%	53.93%
Marketplace	55.84%	46.75%	53.24%

Table 7.10: Accuracy of Qwen by code correctness under different prompting techniques.

7.3.4 DeepSeek: DeepSeek V3

DeepSeek showed promising capabilities in specific translation domains, particularly excelling in mathematical operations and algorithmic logic translation, with the corresponding results summarized in Table 7.11.

Task	Few-Shot	CoT	FS+CoT
Hello World	77.77%	55.55%	66.66%
Voting	66.25%	58.33%	64.16%
Auction	68.86%	59.43%	66.98%
Calculator	73.91%	43.47%	65.21%
Attendance	67.77%	55.00%	60.55%
Counter	70.37%	51.85%	66.66%
Tokenpool	60.41%	56.25%	61.45%
Banking	70.00%	54.00%	62.00%
Storage	62.42%	55.75%	60.00%
Marketplace	67.53%	58.44%	64.93%

Table 7.11: Accuracy of DeepSeek by code correctness under different prompting techniques.

The model demonstrated strong performance in translating calculation-intensive contracts and voting mechanisms, achieving near-perfect accuracy for arithmetic and logical operations. However, DeepSeek exhibited significant variability in handling platform-specific features, struggling consistently with Algorand's asset transfer operations and application call verification patterns. Its performance improved dramatically with explicit feature map-

ping guidance, suggesting strong potential with more targeted training and refined prompting strategies.

Our experience, quantified in Table 7.12, revealed several patterns across four different models. The results presents the performance comparison of four different LLMs for the Few-shot prompt which gave the highest accuracy. Notably, the zero-shot prompt was excluded from this analysis as it consistently generated PyTeal code instead of the intended Algorand Python code, thereby failing to meet the evaluation criteria.

Contract Name	**Methods**	**Methods Translated Correctly (%)**			
		Qwen	**ChatGPT**	**DeepSeek**	**Claude**
HelloWorld	2	50.0%	50.0%	100.0%	100%
Voting	13	46.1%	53.8%	61.5%	69.2%
Auction	9	44.4%	55.5%	55.5%	77.7%
Calculator	6	33.3%	50.0%	50.0%	66.6%
Attendance	16	56.25%	62.5%	68.7%	75.0%
Counter	3	0.0%	33.3%	33.3%	66.6%
Tokenpool	8	50.0%	62.5%	62.5%	87.5%
Banking	6	50.0%	50.0%	66.6%	83.3%
Storage	17	47.05%	58.8%	64.7%	70.5%
Marketplace	5	40.0%	60.0%	60.0%	60.0%

Table 7.12: Methods Translated Correctly Across LLMs

Few-shot prompting consistently produced more accurate translations than chain-of-thought approaches alone, suggesting that concrete examples are particularly valuable for this specialized task. Translation accuracy declined predictably with contract complexity for all models and prompting strategies, with average accuracy dropping from approximately 75% for simple contracts to around 60% for complex ones.

We identified common failure points across all models, including Algorand's box storage model for complex data, asset transfer operations, and application call verification patterns. Translations accuracy varied from minor to substantial errors, suggesting that certain Solidity patterns remain challenging to translate regardless of the model or prompting approach used. These observations indicate that current LLMs have useful but limited capabilities in cross-blockchain translation, with clearly defined strengths and weaknesses that developers should consider when planning translation efforts.

7.4 Developer Effort and Intervention

Based on our observation that few-shot prompting yielded better results, we conducted a deeper analysis focusing exclusively on this prompting technique.

Table 7.13 presents method-level translation performance and documents the effort required to achieve compilable Algorand Python code.

Contract Name	**LoC**	**Effort Until Correct Translation**	
		LLM/Manual	**Iterations/Time (min)**
HelloWorld	9	LLM	3
Voting	152	Manual	45
Auction	74	Manual	38
Calculator	47	LLM	4
Attendance	180	Manual	52
Counter	23	LLM	3
Tokenpool	96	Manual	30
Banking	40	LLM	6
Storage	165	Manual	67
Marketplace	77	Manual	23

Table 7.13: Effort Required for Correct Translation of Smart Contracts

. Claude showed good method-level accuracy, with DeepSeek performing well on certain contracts. The effort metrics indicate that simpler contracts (HelloWorld, Calculator, Counter, Banking) could be translated with just LLM refinement through 3-6 iterations. More complex contracts required manual intervention ranging from 23-67 minutes.

We observed a relationship between contract complexity and developer effort. The Storage contract (165 LoC) needed 67 minutes of manual correction, while the Counter contract (23 LoC) required only 3 LLM iterations. Contracts needing manual intervention typically contained complex data structures, state management, and platform-specific logic that presented challenges for automated translation.

7.4.1 Common Translation Challenges

Throughout our translation experience, we encountered recurring error patterns that represent the key challenges in cross-blockchain translation. Table 7.14 shows a preliminary categorization of these errors, which we found consistent across all models. State management discrepancies emerged as a primary challenge, with conceptual differences between Ethereum's global state and Algorand's key-value approach leading to persistent translation errors, particularly with complex data structures. Type system mismatches also proved problematic, as Solidity's static typing system differs significantly from Algorand Python's approach, resulting in frequent type errors that required manual correction.

Category	Error Message
Attribute Errors	"BoxMap[String, VotingRound]" has no attribute "exists" "String" has no attribute "encode"
Name Errors	Name "gtxn" is not defined
Type System Errors	Expression has type "Any" "UInt64" not callable "Bytes" has no attribute "to uint64"
Argument Type Errors	Argument 1 to "join" has incompatible type "reversed[str]" Argument 1 to "join" has incompatible type "list[String]"
Logical Errors	assert not self.rounds.exists(vote_id) asset_transfer.xfer_asset == self.assetid.value
Transaction Errors	assert paymenttx.type_enum == 1 assert payment_tx.receiver == Txn.application_address
Unsupported Operations	Value of type Module is not indexable
Return Type Mismatch	Returning Any from function declared to return "Bytes"
Global/Local State Issues	GlobalState[Account] has no attribute "exists" Expression has type "Any" when accessing global state

Table 7.14: Comprehensive Table of Errors in Algorand Python Smart Contract Execution

Transaction handling represented another significant challenge area. Algorand's transaction model differs fundamentally from Ethereum's, causing consistent errors in payment verification and asset transfers. We also observed frequent API knowledge gaps, with all models occasionally attempting to use non-existent Algorand methods or incorrectly implementing Algorand-specific features. These patterns highlight areas where LLMs currently lack sufficient understanding of blockchain-specific concepts rather than general programming knowledge. Notably, approximately 60-70% of errors were related to syntax issues from the platform-specific constructs related to Algorand blockchain.

7.5 Discussion

Based on our translation experience, we offer several practical guidelines for developers considering LLM-assisted Solidity-to-Algorand migration. First, developers should approach LLM-generated translations as a starting point rather than a finished product, particularly for contracts exceeding 50 lines of code. For optimal results, we recommend decomposing complex contracts into functional components before translation, employing few-shot prompting with explicit feature mappings, and allocating sufficient time for manual verification of state storage implementations, type conversions, and transaction validation logic. This hybrid human-AI approach currently offers the most efficient path for cross-blockchain development, reducing but not eliminating the need for platform-specific expertise.

This work should be interpreted as an initial exploration rather than a comprehensive evaluation, offering a practical perspective on an emerging technological opportunity despite its inherent limitations of relying on a single developer's expertise, a modest set of contracts, and a limited number of translation iterations. As blockchain ecosystems continue to diversify, the need for interoperability solutions becomes increasingly pressing, and our findings demonstrate that LLMs already offer tangible assistance in addressing this challenge. We see this work as a call to action for the broader blockchain and AI communities to invest in purpose-built solutions for cross-chain development. While current general-purpose LLMs provide useful starting points, specialized models trained on blockchain-specific datasets could dramatically improve translation accuracy. By sharing our experiences now, we hope to catalyze these developments and accelerate progress toward more seamless blockchain interoperability.

Beyond this work, we have applied our findings to translate 10 previously untranslated Solidity smart contracts from [125] to Algorand Python (these contracts are part of our artifacts). These translations, covering diverse applications including marketplace functionality, banking operations, voting systems, and token management, are available in our public artifact repository. This practical application of our hybrid LLM-manual approach demonstrates its real-world viability and provides the community with valuable cross-blockchain implementations that can serve as reference points for future development efforts.

PART IV

VALIDATION OF THE APPROACH

CHAPTER 8

VALIDATION OF THE APPROACH

This chapter presents the validation and experimentation conducted to evaluate the performance and cost-efficiency of the proposed approach. This chapter validates the proposed modular, semantics-driven approach for implementing BPMN choreographies as smart contracts across three distinct real-world scenarios addressing the research objective RO5 of this thesis.

> "*To validate the proposed approach, and in particular its economic viability and performance characteristics, through empirical evaluation across diverse business use cases.*"

The experiments were designed around three distinct use cases: X-ray Examination, Bike Rental, and Hotel Booking. These use cases were chosen to represent a wide range of real-world applications, each requiring complex data flows, multiple service integrations, and robust handling of dynamic participant interactions. The validation process involved implementing and deploying smart contracts on the Algorand Lora Explorer Testnet, ensuring that the system could handle the business logic of each use case efficiently while providing cost-efficiency and flexibility in terms of process evolution.

Each use case was implemented using the methodology detailed in Chapter 5, where the choreography models were translated into smart contracts and executed on the Algorand Lora Explorer Testnet. The primary focus of this chapter is to demonstrate how the approach can handle different execution paths, assess transaction costs, and analyze the system's overall feasibility across a variety of use cases.

8.1 Execution Paths and Runtime Behavior

All the deployed smart contracts support multiple execution paths corresponding to the alternative scenarios defined in the choreography models. These execution paths reflect different operational conditions and system evolutions that may arise across the considered use cases. In particular, the following execution paths were evaluated:

- **Path 1 (Shortest path):** An expedited execution scenario in which a reduced set of interactions is sufficient to complete the process. This path models exceptional or optimized conditions where certain checks or interactions can be safely omitted, allowing the system to reach completion with minimal steps.
- **Path 2 (Longest path):** A complete execution scenario in which all predefined interactions and verification steps are performed. This path represents the standard workflow where all participating entities are involved, all mandatory checks are executed, any looped activities (if executed) are for a single iteration only, and the process concludes with the successful delivery of the final service outcome.
- **Path 3 (Update path):** A modification scenario in which the choreography model is updated to reflect changes in the system. These changes may include the introduction of new participants, the addition or removal of specific tasks, or alterations to the interaction logic in response to evolving business requirements, regulatory constraints, or service policies.
- **Path 4 (Post-Update path):** A stabilization and validation scenario executed after the choreography model has been updated. This path is used to assess the correct execution of the revised model and to evaluate the impact of changes on future executions. The update operation incurs a one-time cost of 2000 micro-algo (µAlgo) at the moment the model is updated; this cost is not incurred in subsequent executions of the updated choreography.

Parallel gateways in the choreography enable concurrent execution of independent verification steps, significantly reducing the overall execution time for non-emergency cases. At runtime, the smart contract enforces correct sequencing by enabling only those methods corresponding to currently active choreography tasks. Any attempt to invoke a method out of order or by an unauthorized participant is rejected by the contract.

8.2 Healthcare Use Case

The healthcare domain represents a particularly challenging application area for blockchain-based business process execution due to its strict requirements on correctness, auditability, privacy, and regulatory compliance. In this section, we evaluate the proposed choreography-driven execution approach through an X-ray examination use case. The use case focuses on modeling and enforcing complex interactions among multiple autonomous participants, including patients, radiology departments, and medical wards. Figure 8.1 presents the baseline BPMN choreography model for healthcare use case.

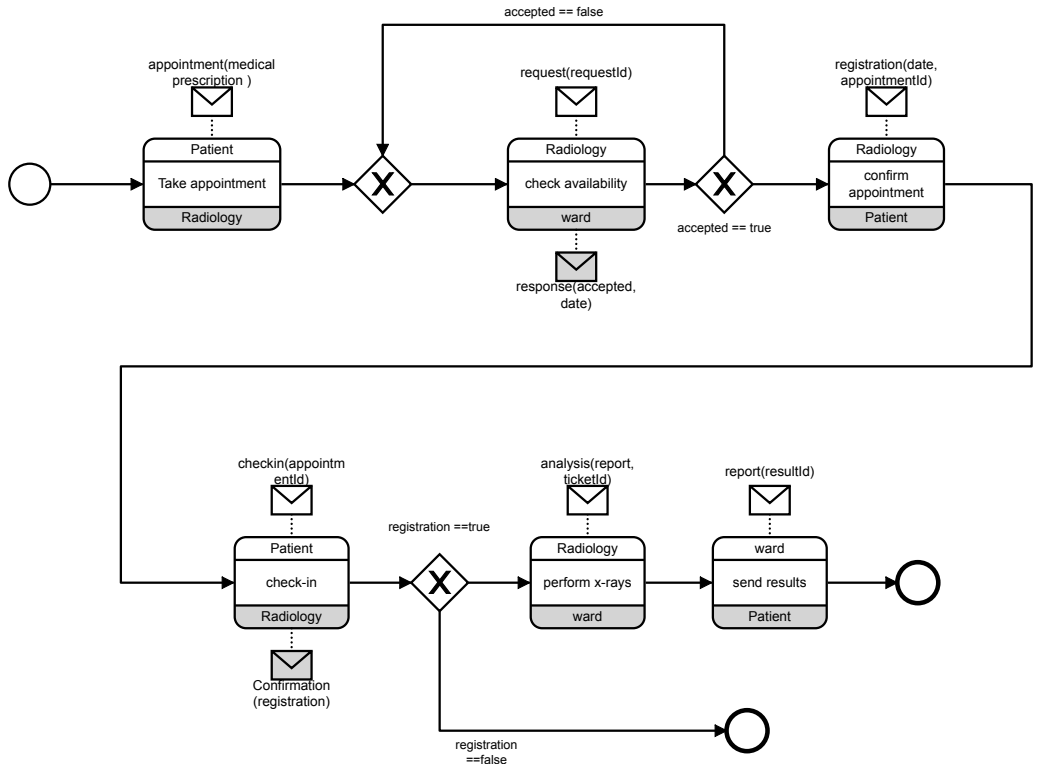


Figure 8.1: Basic X-ray exam choreography diagram.

The primary objective of this use case is to demonstrate that the proposed approach can reliably encode healthcare interaction protocols as executable smart contracts, ensuring that all process steps are executed in compliance with predefined medical procedures. In particular, the approach enforces mandatory pre-examination checks (e.g., vaccination verification and temperature screening), appointment scheduling rules, and validation of medical results, while maintaining transparency and traceability of all interactions.

8.2.1 Choreography Modeling and Implementation

The X-ray examination process was modeled using BPMN choreography diagrams (Figure 8.1), which provide a global view of message-based interactions without exposing the internal logic of individual participants. The choreography involves three main roles: the patient, the radiology department, and the medical ward. Each role interacts with the others exclusively through explicitly defined message exchanges.

The choreography model as shown in Figure 8.1 captures the full lifecycle of an X-ray examination, starting from the appointment request and continuing through availability checks, pre-examination verifications, examination execution, and result delivery. Conditional gateways are used to model decision points, such as the availability of examination slots and the handling of emergency cases. Parallel gateways enable concurrent execution of independent checks, such as vaccination validation and temperature screening[1].

The BPMN choreography was automatically translated into an Algorand Python smart contract using the translation tool. Each choreography task was mapped to a dedicated contract method, while gateways were encoded as routing subroutines enforcing the correct execution order. The resulting smart contract was deployed on the Algorand Lora Explorer Testnet.

8.2.2 Deployment on Algorand

The deployment phase consists of instantiating the generated smart contract on the Algorand blockchain. During deployment, the contract initializes global state variables representing choreography control-flow tokens, participant identities, and shared data fields required during execution.

Deployment on Algorand requires a single transaction, resulting in a predictable and relatively low cost. In this use case, contract deployment consumed approximately 1000 μALGO. Once deployed, the contract becomes a self-contained execution engine that participants can interact with directly, without reliance on any centralized middleware or orchestration service.

This decentralized deployment model is particularly suitable for healthcare environments, where trust among participants cannot always be assumed and auditability of actions is critical.

8.2.3 Process Update and Evolution

Healthcare processes are subject to frequent changes due to new regulations, updated medical protocols, or evolving institutional policies. To address this,

[1]The models were taken from the repository available at `https://bitbucket.org/proslabteam/chorchain/src/master/Examples/` and were later adapted to meet our requirements for experimentation.

the proposed framework supports process evolution through versioned smart contract generation. Specifically, we distinguish between a baseline BPMN choreography model, shown in Figure 8.1, which captures the initial examination workflow, and an updated model show in Figure 8.2, which extends the process with additional pre-examination checks and refined gateway conditions.

When the BPMN choreography model is modified—such as by adding new pre-examination checks or updating gateway conditions, the translation tool generates a new version of the smart contract. This new version can be deployed alongside existing ones, allowing running process instances to complete uninterrupted while new instances adopt the updated logic. The update cost, as shown in Table 8.1, remains modest compared to full redeployment, making iterative refinement economically viable.

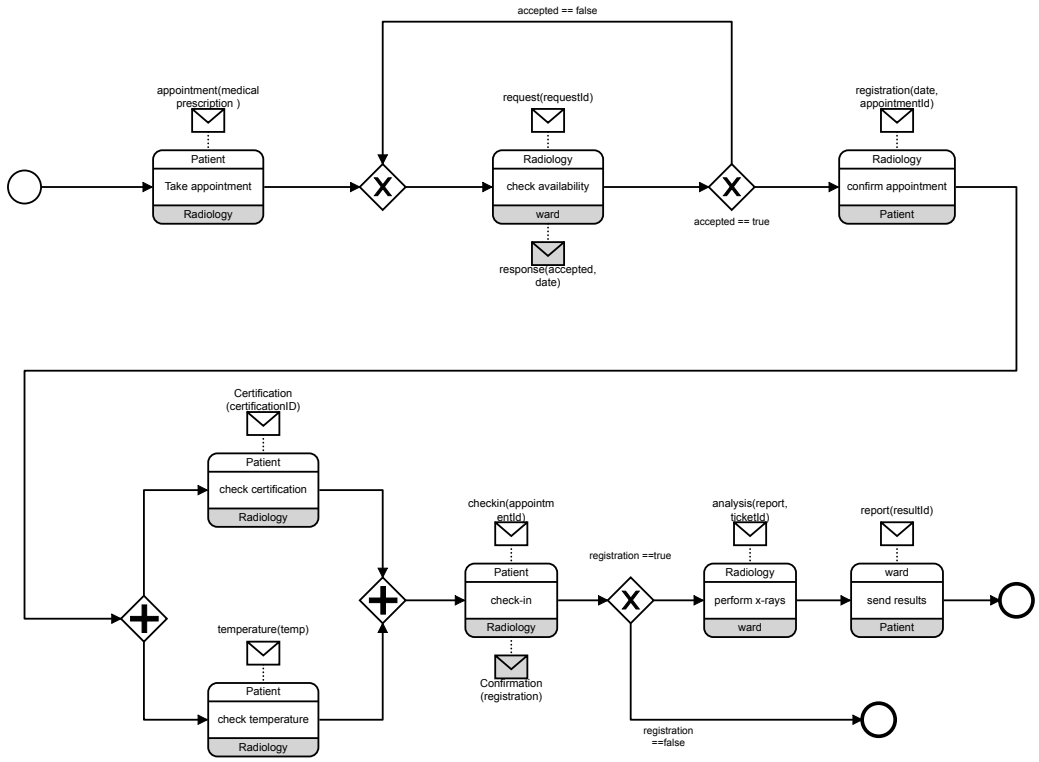


Figure 8.2: Updated X-ray exam choreography diagram.

8.2.4 Feasibility and Discussion

The X-ray examination use case demonstrates the practical feasibility of executing healthcare choreographies on the Algorand blockchain. The results confirm that the proposed approach can faithfully enforce complex interaction protocols, support parallel execution, and accommodate multiple exe-

cution paths with predictable costs.

From a healthcare perspective, the approach provides strong guaranties of transparency, auditability, and compliance while preserving participant autonomy. From a technical perspective, the combination of BPMN choreography modeling, automated smart contract generation, and Algorand's low-cost execution environment offers a scalable and flexible solution for real-world healthcare workflows.

Overall, this use case validates the suitability of the proposed framework for safety-critical, regulation-heavy domains such as healthcare, and provides a solid foundation for its application to other multi-party business processes.

8.2.5 Cost Evaluation

To assess the economic feasibility of the proposed approach, we evaluated the execution costs of the X-ray examination process on the Algorand Lora Explorer Testnet. The results demonstrate that Algorand offers predictable and low-cost execution suitable for healthcare processes with potentially high transaction volumes.

Contract deployment incurred a cost of approximately 1000 μALGO. Execution costs varied depending on the selected execution path. The shortest execution path required approximately 8000 μALGO, while the longest path, involving additional checks and parallel activities, required up to 10000 μALGO. Transaction fees remained fixed at 0.001 μALGO per transaction, ensuring cost predictability.

Scenario	**Cost (μA)**				**Total (USD)**
	Deploy	**Update**	**Execution**	**Total**	
Shortest path	1000	–	7000	8000	0.000952
Longest Path	1000	–	9000	10000	0.0012
Update path	1000	2000	11000	14000	0.0017
Post-Update Path	1000	–	12000	12000	0.0014

Table 8.1: Execution costs for the running example.

Table 8.1 presents the deployment, execution and update costs for four different scenarios[2].

[2]The experiments of the healthcare usecase are available on the testnet.
Shortest Path: `https://lora.algokit.io/testnet/application/754170859`
Longest Path: `https://lora.algokit.io/testnet/application/754173516`
Update path: `https://lora.algokit.io/testnet/application/457180845`
Post-update path: `https://lora.algokit.io/testnet/application/754186175`

The fee is quoted in μALGO (1 ALGO = 1,000,000 μALGO). Using the conversion rate 1 ALGO = 0.12\$, this corresponds to 0.00000012\$ per μALGO, which is an extremely small amount in fiat currency. This highlights that the total cost remains negligible when expressed in USD, despite appearing larger in mALGO units.

8.3 Cross-Domain Validation

Although the healthcare use case demonstrates the applicability of the proposed approach in a safety-critical and regulation-intensive environment, it is not sufficient on its own to claim general feasibility. Healthcare workflows are characterized by strict compliance rules, limited participant roles, and relatively predictable execution patterns. To validate the generality, flexibility, and cost-effectiveness of the approach, we therefore extended the evaluation to two additional use cases drawn from fundamentally different application domains.

Specifically, we selected a *Bike Rental* use case and a *Hotel Room Booking* use case[3]. These domains differ significantly from healthcare in terms of interaction dynamics, decision complexity, frequency of execution, and economic constraints. Together, these three use cases form a representative spectrum of real-world business processes, ranging from safety-critical services to consumer-oriented transactional workflows.

8.3.1 Multi-Domain Evaluation

The core objective of the proposed framework is to enable trustworthy, flexible, and cost-effective execution of choreography-based business processes on blockchain. To convincingly demonstrate this objective, it is essential to evaluate the approach across domains that exhibit different characteristics:

- **Healthcare:** High regulatory requirements, strict execution order, and strong auditability needs.
- **Bike Rental:** Short-lived, consumer-driven interactions with optional services, payments, and post-execution feedback.
- **Hotel Room Booking:** Complex decision logic, multiple alternative execution paths, retries, cancelations, and refunds.

By applying the same modeling, translation, and execution pipeline to these diverse scenarios, we aim to demonstrate that the proposed approach is not domain-specific, but rather a general solution for choreography-driven process execution.

[3]The implementations are available on Github
`https://github.com/nawaz-malla/Phd-Thesis-Usecase`

8.3.2 Validation Dimensions

Across all three use cases, the evaluation focuses on the following validation dimensions:

- **Correctness of Execution:** Ensuring that all interactions strictly follow the BPMN choreography semantics.
- **Cost Predictability:** Measuring deployment, execution, and update costs on Algorand.
- **Flexibility and Updatability:** Assessing how process changes are handled through automated contract regeneration and versioning.
- **Cross-Domain Applicability:** Verifying that the same translation and execution strategy works without domain-specific adaptations.

The healthcare use case primarily validates correctness and compliance, while the Bike Rental and Hotel Room Booking use cases emphasize flexibility, dynamic decision-making, and economic feasibility.

8.4 Bike Rental Use Case

To further validate the proposed approach beyond the healthcare domain, we consider a Bike Rental use case that represents a consumer-oriented, business process with optional services and financial transactions. Unlike healthcare workflows, bike rental processes are executed frequently, are highly cost-sensitive, and involve dynamic decision-making, making them suitable for evaluating both execution efficiency and cost predictability.

8.4.1 Choreography Model

The Bike Rental process was modeled as a BPMN choreography involving three main participants: the customer, the bike rental center, and an insurance provider. The choreography captures interactions such as availability requests, optional insurance selection, payment execution, bike delivery, damage reporting, refund handling, and receipt issuance.

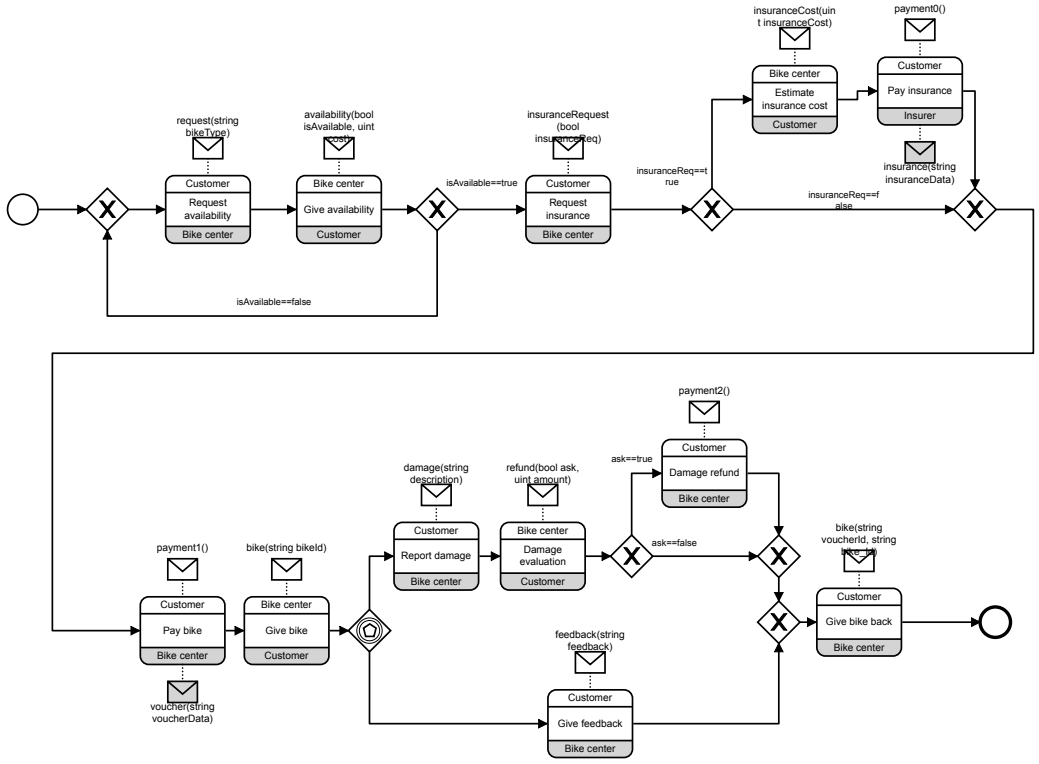


Figure 8.3: Basic Bike rental choreography diagram.

Figure 8.3 illustrates the basic BPMN choreography model for the Bike Rental use case. The model includes multiple conditional paths, allowing customers to proceed with or without insurance, and supports post-rental actions such as feedback submission and refund requests in case of reported damage. Figure 8.4 presents the updated model, which extends the basic choreography by incorporating additional refinements and enhancements, such as refund handling process.

The choreography was automatically translated into an Algorand Python smart contract using our BPMN-to-Algorand translator. Each choreography task and gateway was mapped to a dedicated contract method or routing rule, ensuring faithful preservation of the BPMN semantics.

8.4.2 Execution and Control Flow

The generated smart contract enforces the correct sequencing of interactions and participant roles. Conditional gateways are realized as explicit routing subroutines, enabling the contract to dynamically select execution paths based on runtime data, such as insurance selection or damage reporting.

Optional insurance introduces branching behavior, while refund handling adds a post-execution path that may be triggered only under specific con-

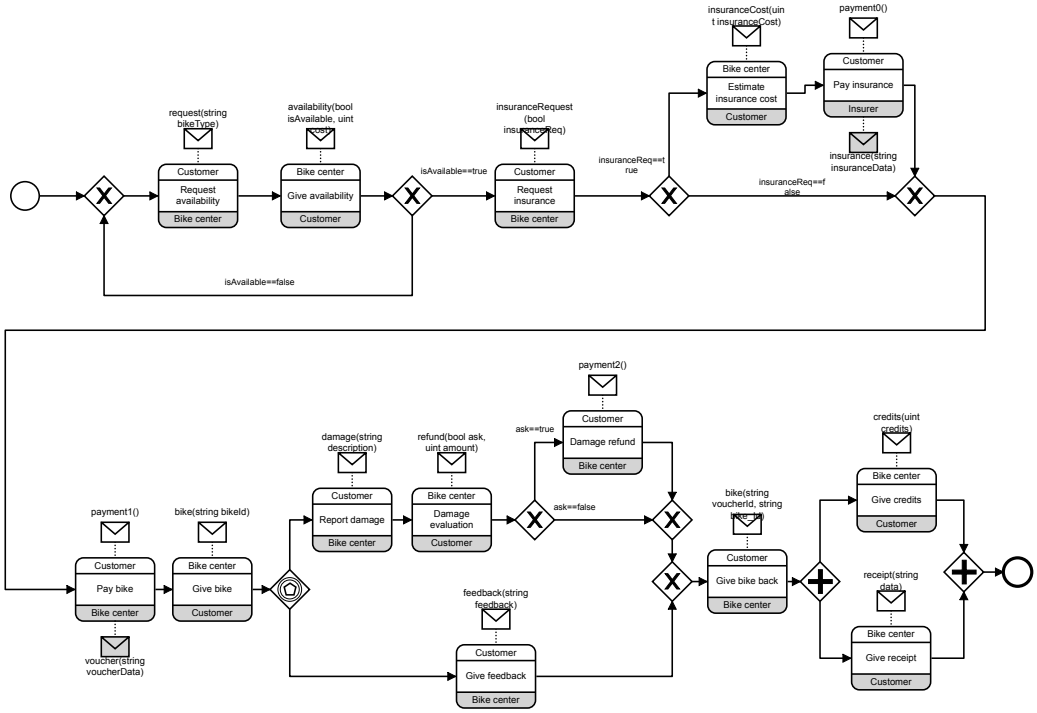


Figure 8.4: Updated Bike rental choreography diagram.

ditions. These constructs validate the ability of the proposed approach to support flexible execution patterns commonly found in real-world consumer services.

8.4.3 Cost Evaluation

The Bike Rental use case was deployed and executed on the Algorand test network to evaluate its economic feasibility. Table 8.2 summarizes the deployment, execution, and update costs[4] for different scenarios[5].

The results show that even with optional services and conditional execution paths, transaction costs remain low and predictable. This confirms that the approach is suitable for high-frequency, low-margin business processes, where cost efficiency is critical.

[4]The experiments of bikerental usecase are available on the testnet.
Shortest Path: `https://lora.algokit.io/testnet/application/753195710`
Longest Path: `https://lora.algokit.io/testnet/application/753191231`
Update path: `https://lora.algokit.io/testnet/application/754560493`
post-Update path: `https://lora.algokit.io/testnet/application/753201098`

[5]The Algo-to-USD conversions were calculated based on the online exchange rate as of 6 January 2026.

Scenario	Cost (μA)				Total (USD)
	Deploy	Update	Execution	Total	
Shortest Path	1000	–	9000	10000	0.0012
Longest Path	1000	–	14000	15000	0.0018
Update Path	1000	2000	16000	19000	0.0023
Post-Update Path	1000	–	11000	12000	0.0014

Table 8.2: Execution costs for the Bike Rental example.

8.5 Hotel Room Booking Use Case

The Hotel Room Booking use case represents a service-oriented workflow with higher control-flow complexity compared to the Bike Rental scenario. This use case is characterized by multiple decision points, retries, cancellations, and refund mechanisms, making it well-suited to evaluate the expressiveness and flexibility of the proposed framework.

8.5.1 Choreography Model

The hotel booking process was modeled as a BPMN choreography involving two primary participants: the client and the hotel. The choreography includes interactions for booking requests, availability checks, client confirmation, payment execution, receipt issuance, retry logic, rejection handling, and refund processing.

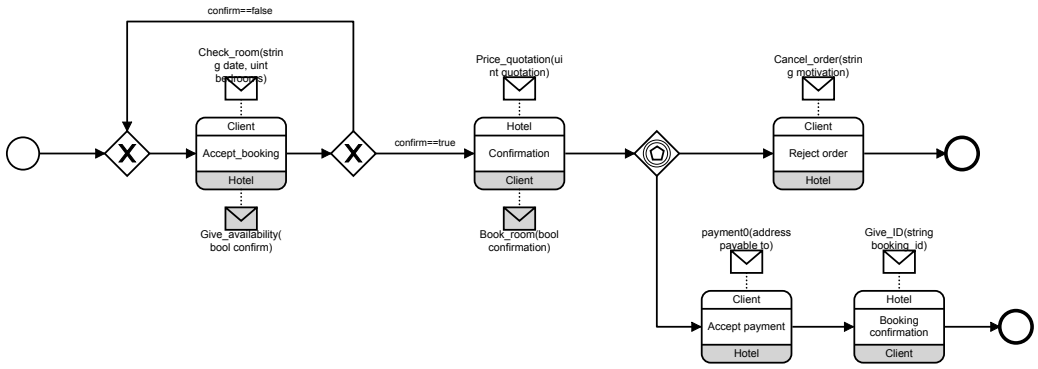


Figure 8.5: Basic Hotel booking choreography diagram.

Figure 8.5 presents the basic BPMN choreography model for the Hotel Room Booking use case. The model contains multiple XOR gateways and looping constructs, allowing the process to return to earlier stages in case of unavailability or client-initiated retries. Figure 8.6 illustrates the updated

model, which extends the basic choreography by introducing additional execution paths to capture alternative booking outcomes and extended interaction scenarios beyond the core flow.

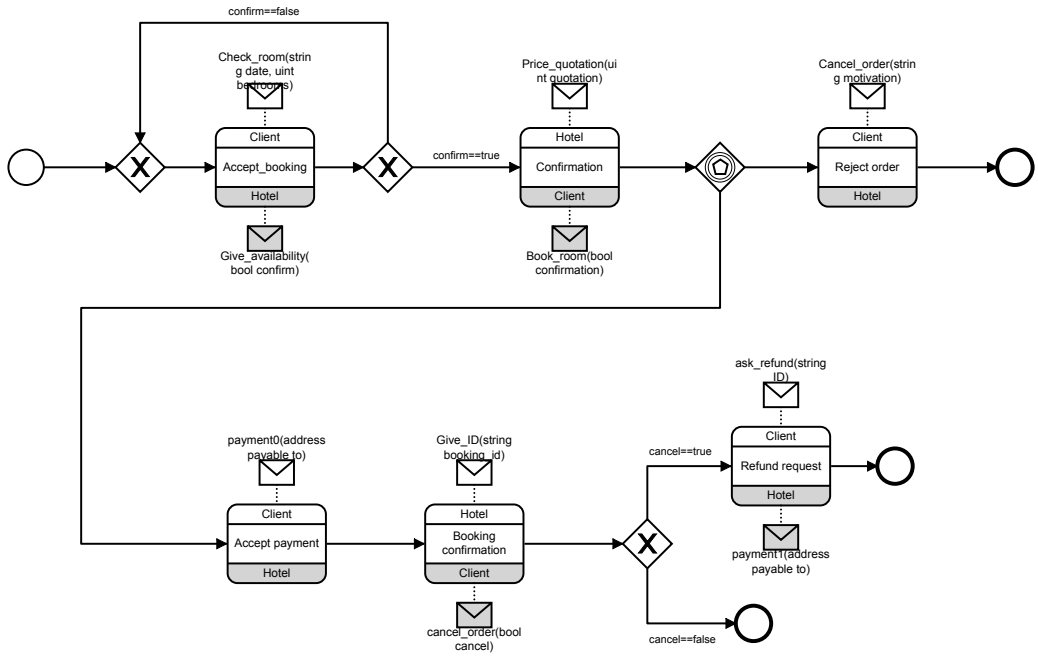


Figure 8.6: Updated Hotel booking choreography diagram.

As with the previous use cases, the choreography was automatically translated into an Algorand smart contract using our toolchain. Each BPMN element was mapped to a modular contract component, facilitating readability, verification, and targeted updates.

8.5.2 Execution Paths and Update Handling

The generated smart contract supports multiple execution paths, including successful booking, rejection, retry, and refund scenarios. XOR gateways are implemented as routing subroutines that evaluate runtime conditions, such as room availability or client confirmation, and direct execution accordingly.

Importantly, the use case demonstrates the updatability of the approach. Changes to the BPMN model, such as adding refund logic or modifying retry conditions, result in the automatic generation of a new contract version, without affecting unrelated process components.

8.5.3 Cost Evaluation

Table 8.3 reports the deployment, execution, and update costs[6] for the Hotel Room Booking use case on Algorand [7].

Scenario	Cost (μA)				Total (USD)
	Deploy	Update	Execution	Total	
Shortest Path	1000	–	8000	9000	0.0011
Longest Path	1000	–	10000	11000	0.0013
Update Path	1000	2000	12000	14000	0.0017
Post-Update Path	1000	–	12000	13000	0.0016

Table 8.3: Execution costs for the Hotel booking example.

Despite the increased complexity of control flow and additional execution paths, the overall costs remain low and predictable. This confirms that the proposed approach scales effectively with process complexity while preserving economic feasibility.

8.6 Cross-Use Case Analysis and Feasibility

Taken together, the three use cases provide strong empirical evidence for the feasibility and general applicability of the proposed approach. Despite operating in distinct domains, all use cases were modeled using BPMN choreographies, translated using the same automated translation tool, and executed on the Algorand blockchain without domain-specific modifications.

The healthcare use case validates the approach in a compliance-driven environment, the bike rental use case demonstrates cost-effective execution for high-frequency consumer services, and the hotel booking use case confirms the ability to handle complex control-flow and financial logic. Across all scenarios, deployment and execution costs remained low and predictable, while process updates were handled efficiently through contract regeneration and versioning.

These results collectively show that the proposed approach is not limited to a single domain or interaction pattern. Instead, it provides a unified and

[6]The experiments of hotelbooking usecase are available on the testnet.
Shortest Path: `https://lora.algokit.io/testnet/application/752906949`
Longest Path: `https://lora.algokit.io/testnet/application/752901566`
Update path: `https://lora.algokit.io/testnet/application/754511028`
post-Update Path: `https://lora.algokit.io/testnet/application/752946714`

[7]The Algo-to-USD conversions were calculated based on the online exchange rate as of 27 January 2026.

scalable solution for trustworthy execution of choreography-based business processes across heterogeneous application contexts.

PART V

CONCLUSIONS & FUTURE WORKS

CHAPTER 9

CONCLUSIONS & FUTURE WORKS

9.1 Concluding Remarks

This thesis addresses the key challenges in the design and execution of inter-organizational business processes, with a focus on leveraging blockchain technology to establish trust, support adaptability, and enable migration. The core challenges revolve around the need for a secure, transparent, and immutable environment for collaborative processes, while also accommodating the dynamic nature of business requirements and the evolving blockchain ecosystem. To tackle these challenges, the thesis defined and pursued five primary research objectives, each contributing a novel solution grounded in a model-driven and automated methodology.

The first objective directly addressed the fundamental requirement of trust in collaborative systems. The thesis proposed a model-driven methodology that translates standard BPMN choreography diagrams into executable smart contracts. By using BPMN, the methodology captures inter-organizational process logic from a high-level, shared perspective. The core innovation lies in the automated translation of these formal models into smart contract code. Once deployed on a blockchain, this smart contract governs the process execution, ensuring that all interactions occur according to the predefined rules and are immutably recorded. This approach shifts trust from individual participants to the technological infrastructure, providing security, distribution, transparency, and enforcement.

Building upon this foundation, the second objective confronted the inherent tension between business process flexibility and blockchain immutability.

To enable processes to adapt to unexpected changes at run-time, the thesis introduced a framework that utilizes Algorand's native stateful smart contract update mechanism. This solution provides a principled approach to process evolution. Rather than relying on complex off-chain logic or multi-contract systems, the framework allows authorized parties to securely propose and approve modifications to the on-chain smart contract logic itself. This leverages Algorand's efficient consensus to maintain trust and transparency while granting controlled flexibility, ensuring the process remains agile without compromising its trustworthy and verifiable nature.

The third objective addressed the challenge of automating the deployment of blockchain-based business processes by bridging high-level process modeling and executable smart contracts. The thesis proposed a BPMN-to-Algorand smart contract generation tool that enables the systematic translation of BPMN choreography models into executable Algorand smart contracts. This automated translation substantially reduces the complexity, development effort, and potential inconsistencies associated with manually implementing smart contracts from process models. As a result, the proposed tool enhances the practicality of deploying process-driven applications on Algorand and supports a model-driven, scalable, and repeatable pathway from business process design to blockchain execution.

The fourth objective tackled the challenge of vendor lock-in and platform dependency within the fragmented blockchain landscape. To facilitate the migration of process implementations across different platforms, the thesis proposed an AI-assisted methodology for translating smart contracts. This approach uses machine learning techniques to analyze the logic, state variables, and interaction patterns of a source smart contract (e.g., one implementing a BPMN process). The AI model then assists in generating equivalent, platform-optimized code for a target blockchain (e.g., from Ethereum to Algorand). This methodology significantly reduces the manual effort, risk of errors, and cost associated with platform migration, thereby promoting interoperability and protecting long-term investments in blockchain-based process solutions.

Finally, to validate the feasibility and practicality of the proposed frameworks, the thesis employed a comprehensive case study of a health-care business process. A series of experiments were conducted to assess costs, correctness of execution, update efficiency, and translation accuracy. Further, the validation was extended to other more complex use cases, like hotel booking and bike rental. The results are promising, demonstrating that the integration of model-driven design, native blockchain update features, and AI-assisted translation creates a sustainable and versatile ecosystem for trusted, adaptable, and portable inter-organizational business processes. Compared with existing solutions, the proposed approach offers practical advantages

beyond those measured in the aforementioned experiments. Traditional contract re-deployment, while straightforward, introduces higher costs and may also lead to the loss of the existing contract state. Similarly, pattern-based solutions, as commonly adopted in Solidity, tend to increase operational complexity and cost, while offering more limited control over the update process. These findings provide additional support for the effectiveness of the proposed approach while also pointing to opportunities for further validation and extension.

9.2 Future Works

The contributions of this thesis open numerous avenues for future research across multiple dimensions of blockchain technology and business process management. In the domain of intelligent development tools, significant opportunities exist for enhancing AI-assisted blockchain development. Future work should focus on creating specialized language models trained exclusively on blockchain codebases and documentation, potentially yielding dramatic improvements in translation accuracy and security awareness. The integration of real-time formal verification with AI-generated code represents another promising direction, where development environments could automatically verify semantic equivalence and security properties during the translation process. Furthermore, research into AI-powered repair mechanisms for identified vulnerabilities could transform how developers address security concerns in cross-platform migrations.

An important direction for further research concerns the systematic evaluation of LLM-based translation approaches in comparison with traditional rule-based and hybrid techniques. While this thesis demonstrates the feasibility of using LLMs to translate Solidity smart contracts into Algorand-compatible implementations, a comprehensive benchmark is currently lacking. Future investigations could therefore focus on defining suitable baselines, developing standardized evaluation metrics (e.g., correctness, completeness, and executability), and performing controlled experiments to assess the relative strengths and limitations of each approach. In addition, hybrid solutions that combine LLM-based generation with rule-based validation or post-processing could be explored to improve reliability and consistency. Such investigations could provide a clearer understanding of the practical advantages, trade-offs, and applicability of LLM-driven translation in comparison to more traditional methods.

The exploration of cross-platform interoperability demands investigation into universal intermediate representations for smart contracts. Such representations could enable truly platform-agnostic development, where business processes specified once could be deployed across multiple blockchain

ecosystems without significant re-engineering. Building on our work with cross-chain translations, future research should address the challenges of multi-platform choreography execution, particularly focusing on atomicity guarantees and consistent state management across heterogeneous blockchain environments.

For enterprise-scale adoption, several critical research directions emerge. The development of sophisticated governance frameworks for decentralized process evolution requires attention to complex multi-stakeholder approval mechanisms and dispute resolution protocols. Large-scale performance evaluation frameworks must be established to benchmark business process execution under realistic load conditions and complexity scenarios. Additionally, industry-specific implementations in domains such as healthcare, supply chain, and finance would provide valuable insights into domain-specific requirements and regulatory considerations.

A further research direction concerns the extension of the proposed approach to more comprehensively support choreography flexibility, particularly with respect to the management and governance of runtime modifications. While this thesis provides an initial step in enabling flexibility in blockchain-based choreographies, additional work is required to define robust mechanisms for controlling and validating changes. In particular, future studies could investigate governance models for coordinating updates among participants, as well as techniques for ensuring the correctness, consistency, and compliance of modified choreography instances. This could include exploring approval workflows, versioning strategies, and formal verifications. In fact, our work on formal semantics and translation methodologies suggests promising directions in verification. Future research could develop automated verification tools for ensuring business process properties before deployment (such as soundness, deadlock-freedom, and compliance with predefined constraints). The integration of model-checking techniques with smart contract generation could enable provably correct implementations of critical business processes. This work, however, requires the formalization of Algorand Python semantics, which is currently missing. There are some formalizations of Solidity (see, e.g., TinySol [12]) that can be used as a source of inspiration. This will allow to formally prove that the translation preserves the semantics of choreographies, meaning that the formal semantics of the smart contract resulting from the translation is equivalent to the semantics of the originating choreography (according to some suitable notion of equivalence). However, this deserves future investigation. Notably, this formal verification will also allow to check that choreography updates preserve the intended semantics and behavioral properties of the initial choreography.

The flexibility mechanisms introduced in this thesis inspire further innovation in smart contract evolution. Research into fine-grained update poli-

cies, version control systems for decentralized applications, and automated regression testing for smart contract updates would significantly enhance the practical utility of runtime modifications. Additionally, exploring the integration of our update mechanism with decentralized autonomous organization (DAO) governance models could yield novel approaches to collective decision-making in business process evolution.

From an experimental perspective, further work could extend the current evaluation by considering additional validation dimensions beyond execution cost and functional correctness. While this thesis focuses on the feasibility and flexibility of implementing business processes on Algorand, a more comprehensive empirical assessment could further strengthen the approach. In particular, future investigations could include stress testing under high transaction loads to evaluate scalability, as well as long-term performance analyses to observe system behavior over extended periods. Moreover, controlled adversarial scenarios such as delayed transactions, network congestion, or conflicting updates could be explored to assess the robustness and resilience of the proposed solution. Incorporating these dimensions could provide a more complete understanding of the system's performance and its suitability for real-world deployment.

In conclusion, while this thesis has made substantial contributions to the field of blockchain-enabled business processes, it also reveals rich opportunities for future investigation. The integration of formal methods, architectural innovation, and AI-assisted development represents a powerful paradigm that will continue to drive advancements in decentralized systems. As blockchain technology matures and finds broader adoption, the principles and frameworks established in this work provide a solid foundation for building more flexible, efficient, and accessible decentralized applications that can truly transform how organizations collaborate in the digital age.

Appendix

I. License

Non-exclusive licence to reproduce the thesis and make it publicly accessible

I, **Nawaz Abdullah Malla**,

1. hereby grant the **University of Florence and the University of Camerino, Italy** a free, non-exclusive permit to reproduce my thesis for the purpose of preservation, including its addition to the Universities' digital archives, until the expiry of the term of copyright:

(Title of the Thesis)
Flexible and Trustworthy Execution of Business Processes,
supervised by
Prof. Francesco Tiezzi.

2. I grant the **Universities** permission to make the thesis specified in point 1 publicly accessible through their web environments, including their digital archives, under the Creative Commons licence **CC BY-NC-ND 3.0**. This licence allows others to reproduce, distribute, and communicate the work to the public, provided that appropriate credit is given to the author. The licence prohibits commercial use of the work and the creation of derivative works until the expiry of the term of copyright.
3. I am aware that the author retains the rights specified in points 1 and 2.
4. I certify that granting this non-exclusive licence does not infringe the intellectual property rights of third parties nor violate any personal data protection regulations.

Nawaz Abdullah Malla
27-02-2026

BIBLIOGRAPHY

[1] Michael Adams, Suriadi Suriadi, Akhil Kumar, and Arthur H. M. ter Hofstede. Flexible integration of blockchain with business process automation: A federated architecture. In *CAiSE Forum*, volume 386 of *LNBIP*, pages 1–13. Springer, 2020.

[2] Michael Adams, Suriadi Suriadi, Akhil Kumar, and Arthur HM ter Hofstede. Flexible integration of blockchain with business process automation: A federated architecture. In *Int. Conference on Advanced Information Systems Engineering*, volume 386 of *LNBIP*, pages 1–13. Springer, 2020.

[3] Karan Aggarwal, Mohammad Salameh, and Abram Hindle. Using machine translation for converting python 2 to python 3 code. Technical report, PeerJ PrePrints, 2015.

[4] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021.

[5] Aitor Aldazabal, Terry Baily, Felix Nanclares, Andrey Sadovykh, Christian Hein, and Tom Ritter. Automated model driven development processes. In *Model Driven Tool and Process Integration*, pages 361 – 375. Fraunhofer IRB Verlag, 2008.

[6] Saiqa Aleem, Sanja Lazarova-Molnar, and Nader Mohamed. Collaborative business process modeling approaches: a review. In *Proc. of the 2012 IEEE 21st International workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 274–279. Citeseer, 2012.

[7] Algorand. Structure - Algorand Developer Portal, 2023.

[8] Algorand Developer. *smart contract update*, 2019.

[9] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards verifying ethereum smart contract bytecode in isabelle/hol. In *Proceedings of the 7th ACM SIGPLAN international conference on certified programs and proofs*, pages 66–77, 2018.

[10] Victor Amaral de Sousa, Corentin Burnay, and Monique Snoeck. B-merode: a model-driven engineering and artifact-centric approach to generate blockchain-based information systems. In *International Conference on Advanced Information Systems Engineering*, pages 117–133. Springer, 2020.

[11] M. Autili, P. Inverardi, and M. Tivoli. Choreos: Large scale choreographies for the future internet. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*, pages 391–394, 2014.

[12] Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. A minimal core calculus for solidity contracts. In *International Workshop on Data Privacy Management*, pages 233–243. Springer, 2019.

[13] Iris Beerepoot, Claudio Di Ciccio, Hajo A Reijers, Stefanie Rinderle-Ma, Wasana Bandara, Andrea Burattin, Diego Calvanese, Tianwa Chen, Izack Cohen, Benoît Depaire, et al. The biggest business process management problems to solve before we die. *Computers in Industry*, 146:103837, 2023.

[14] Sahil Bhatia, Jie Qiu, Niranjan Hasabnis, Sanjit Seshia, and Alvin Cheung. Verified code transpilation with LLMs. *Advances in Neural Information Processing Systems*, 37:41394–41424, 2024.

[15] Grayscale Building Blocks. An introduction to solana. *no. December*, 2021.

[16] Peter Bodorik, Christian Gang Liu, and Dawn Jutla. Tabs: Transforming automatically bpmn models into blockchain smart contracts. *BCRA*, 4(1):100115, 2023.

[17] Giorgio Bruno, Giulia Bruno, and Marcello La Rosa. On collaborations and choreographies. In *Technologies for Collaborative Business Process Management, Proceedings of the 1st International Workshop on Technologies for Collaborative Business Process Management*, pages 3–12. INSTICC Press, 2006.

[18] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3(37):2–1, 2014.

[19] Jing Chen and Silvio Micali. Algorand. *arXiv preprint arXiv:1607.01341*, 2016.

[20] Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theoretical Computer Science*, 777:155–183, 2019.

[21] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. *Advances in neural information processing systems*, 31, 2018.

[22] Michele Chinosi and Alberto Trombetta. Bpmn: An introduction to the standard. *Computer Standards & Interfaces*, 34(1):124–134, 2012.

[23] Claudio Di Ciccio, Giovanni Meroni, and Pierluigi Plebani. Business process monitoring on blockchains: Potentials and challenges. In *Enterprise, Business-Process and Information Systems Modeling*, volume 387 of *LNBIP*, pages 36–51. Springer, 2020.

[24] Claudio Di Ciccio, Giovanni Meroni, and Pierluigi Plebani. On the adoption of blockchain for business process monitoring. *Software and Systems Modeling*, 21(3):915–937, 2022.

[25] Sadrettin Çodur and Burak Erkayman. Blockchain technology from the supply chain perspective: A systematic literature review. *Spectrum of Decision Making and Applications*, 2(1):268–285, 2025.

[26] Ivan Compagnucci, Flavio Corradini, Fabrizio Fornari, and Barbara Re. A study on the usage of the bpmn notation for designing process collaboration, choreography, and conversation models. *Business & Information Systems Engineering*, 66(1):43–66, 2024.

[27] Mario Cortes Cornax, Sophie Dupuy-Chessa, Dominique Rieu, and Marlon Dumas. Evaluating choreographies in BPMN 2.0 using an extended quality framework. In *Business Process Model and Notation - Third International Workshop, BPMN*, volume 95 of *Lecture Notes in Business Information Processing*, pages 103–117. Springer, 2011.

[28] Flavio Corradini, Alessandro Marcelletti, Andrea Morichetta, Andrea Polini, Barbara Re, Emanuele Scala, and Francesco Tiezzi. Model-driven engineering for multi-party business processes on multiple blockchains. *BCRA*, 2(3):100018, 2021.

[29] Flavio Corradini, Alessandro Marcelletti, Andrea Morichetta, Andrea Polini, Barbara Re, and Francesco Tiezzi. A choreography-driven approach for blockchain-based iot applications. In *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pages 255–260. IEEE, 2022.

[30] Flavio Corradini, Alessandro Marcelletti, Andrea Morichetta, Andrea Polini, Barbara Re, and Francesco Tiezzi. Engineering trustable and auditable choreography-based systems using blockchain. *ACM Trans. Manag. Inf. Syst.*, 13(3):31:1–31:53, 2022.

[31] Flavio Corradini, Alessandro Marcelletti, Andrea Morichetta, Andrea Polini, Barbara Re, and Francesco Tiezzi. Flexible execution of multi-

party business processes on blockchain. In *WETSEB*, pages 25–32, 2022.

[32] Flavio Corradini, Alessandro Marcelletti, Andrea Morichetta, Andrea Polini, Barbara Re, and Francesco Tiezzi. A flexible approach to multi-party business process execution on blockchain. *Future Gener. Comput. Syst.*, 147:219–234, 2023.

[33] Flavio Corradini, Alessandro Marcelletti, Andrea Morichetta, Andrea Polini, Barbara Re, and Francesco Tiezzi. A flexible approach to multi-party business process execution on blockchain. *Future Gener. Comput. Syst.*, 147:219–234, 2023.

[34] Flavio Corradini, Alessandro Marcelletti, Andrea Morichetta, Andrea Polini, Barbara Re, and Francesco Tiezzi. A flexible approach to multi-party business process execution on blockchain. *Future Generation Computer Systems*, 147:219–234, 2023.

[35] Flavio Corradini, Alessandro Marcelletti, Andrea Morichetta, Andrea Polini, Barbara Re, Francesco Tiezzi, et al. Chorchain: A model-driven framework for choreography-based systems using blockchain. In *ITBPM@ BPM*, pages 26–32, 2021.

[36] Flavio Corradini, Alessandro Marcelletti, Andrea Morichetta, Andrea Polini, Barbara Re, Francesco Tiezzi, et al. Chorchain: a blockchain-based framework for executing and auditing bpmn choreographies. In *CEUR WORKSHOP PROCEEDINGS*, pages 132–136. CEUR-WS, 2022.

[37] Flavio Corradini, Andrea Morichetta, Andrea Polini, Barbara Re, and Francesco Tiezzi. Collaboration vs. choreography conformance in BPMN. *Log. Methods Comput. Sci.*, 16(4), 2020.

[38] Flavio Corradini, Andrea Morichetta, Barbara Re, and Francesco Tiezzi. Walking through the semantics of exclusive and event-based gateways in BPMN choreographies. In *The Art of Modelling Computational Systems*, volume 11760 of *LNCS*, pages 163–181. Springer, 2019.

[39] Simon Curty, Felix Härer, and Hans-Georg Fill. Design of blockchain-based applications using model-driven engineering and low-code/no-code platforms: a structured literature review. *Software and Systems Modeling*, 22(6):1857–1895, 2023.

[40] Mila Dalla Preda, Saverio Giallorenzo, Ivan Lanese, Jacopo Mauro, and Maurizio Gabbrielli. Aiocj: A choreographic framework for safe adaptive distributed applications. In *International Conference on Software Language Engineering*, pages 161–170. Springer, 2014.

[41] Lívia Maria Bettini de Miranda, Rodrigo Dutra Garcia, Jo Ueyama, Gowri Ramachandran, and Fábio Müller Guerrini. A blockchain-based reputation system for trust management in collaborative multi-stakeholder settings. In *Workshop em Blockchain: Teoria, Tecnologias e Aplicações (WBlockchain)*, pages 41–54. SBC, 2024.

[42] Andrea Delgado, Daniel Calegari, Félix García, and Barbara Weber. Model-driven management of bpmn-based business process families. *Software and Systems Modeling*, 21(6):2517–2553, 2022.

[43] Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. *Fundamentals of Business Process Management, Second Edition*. Springer, 2018.

[44] Amir M Ebrahimi, Bram Adams, Gustavo A Oliva, and Ahmed E Hassan. A large-scale exploratory study on the proxy pattern in ethereum. *Empirical Software Engineering*, 29(4):81, 2024.

[45] Robert Engel, Worarat Krathu, Marco Zapletal, Christian Pichler, RP Jagadeesh Chandra Bose, Wil van der Aalst, Hannes Werthner, and Christian Huemer. Analyzing inter-organizational business processes: process mining and business performance analysis using electronic data interchange messages. *Information Systems and e-Business Management*, 14(3):577–612, 2016.

[46] Tom Brown et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[47] Lishui Fan, Jiakun Liu, Zhongxin Liu, David Lo, Xin Xia, and Shanping Li. Exploring the capabilities of llms for code change related tasks. *ACM Transactions on Software Engineering and Methodology*, 2024.

[48] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15. IEEE, 2019.

[49] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[50] Algorand Foundation. Puya examples, Accessed: 20-Mar-2025, online.

[51] Imen Ben Fraj, Yousra Bendaly Hlaoui, and Leila Ben Ayed. A control system for managing the flexibility in BPMN models of cloud service workflows. In *13th IEEE International Conference on Cloud Computing*, pages 537–543. IEEE, 2020.

[52] Saverio Giallorenzo, Fabrizio Montesi, and Maurizio Gabbrielli. A model for correlation-based choreographic programming. *PeerJ Computer Science*, 10:e1907, 2024.

[53] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th symposium on operating systems principles*, pages 51–68, 2017.

[54] Hao Hao, Dahlia Malkhi, Maofan Yin, and Lizan Zhou. Lyte quorum: Off-chain ready smart contract hosted with choice. *arXiv preprint arXiv:2509.23448*, 2025.

[55] Sindre Grønstøl Haugeland, Phu H Nguyen, Hui Song, and Franck Chauvel. Migrating monoliths to microservices-based customizable multi-tenant cloud-native apps. In *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 170–177. IEEE, 2021.

[56] Nam Huynh and Beiyu Lin. A survey on large language models for code generation. *arXiv preprint arXiv:2503.01245*, 2025.

[57] Jan Mendling et al. Blockchains for business process management - challenges and opportunities. *ACM Trans. Manag. Inf. Syst.*, 9(1):4:1–4:16, 2018.

[58] Jason Wei et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[59] Jiexin Wang et al. Enhancing large language models for secure code generation: A dataset-driven study on vulnerability mitigation. *arXiv preprint arXiv:2310.16263*, 2023.

[60] Jun Jin, Le Yan, Yidan Zou, Jie Li, and Zhen Yu. Research on smart contract verification and generation method based on bpmn. *Mathematics*, 12(14):2158, 2024.

[61] Svetoslav Karaivanov, Veselin Raychev, and Martin T. Vechev. Phrase-based statistical translation of programming languages. In *Onward! @SPLASH*, pages 173–184. ACM, 2014.

[62] Rabimba Karanjai, Lei Xu, and Weidong Shi. Solmover: Smart contract code translation based on concepts. In *AIware*, pages 112–121. ACM, 2024.

[63] Dongsoo Kim, Minsoo Kim, and Hoontae Kim. Dynamic business process management based on process change patterns. In *International Conference on Convergence Information Technology*, pages 1154–1161. IEEE, 2007.

[64] Philipp Klinger, Long Nguyen, and Freimut Bodendorf. Upgradeability concept for collaborative blockchain-based business process execution framework. In *Conference on Blockchain*, volume 12404 of *LNCS*, pages 127–141. Springer, 2020.

[65] Philipp Klinger, Long Nguyen, and Freimut Bodendorf. Upgradeability concept for collaborative blockchain-based business process execution framework. In *ICBC*, volume 12404 of *LNCS*, pages 127–141. Springer, 2020.

[66] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*, 2020.

[67] Aron Laszka, Scott Eisele, Abhishek Dubey, Gabor Karsai, and Karla Kvaternik. Transax: A blockchain-based decentralized forward-trading energy exchanged for transactive microgrids. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 918–927. IEEE, 2018.

[68] Victor Lemaire[1], Tiphaine Henry, Álvaro García-Pérez[1], Walid Gaaloul, and Sara Tucci-Piergiovanni[1]. Leveraging the diamond pattern for scalable and upgradeable. In *Business Process Management Forum: BPM 2025 Forum, Seville, Spain, August 31–September 5, 2025, Proceedings*, page 221. Springer Nature, 2025.

[69] Xiaofan Li, Jin Yang, Jiaqi Chen, Yuzhe Tang, and Xing Gao. Characterizing ethereum upgradable smart contracts and their security implications. In *Proceedings of the ACM Web Conference 2024*, pages 1847–1858, 2024.

[70] Orleans López-Pintado, Luciano García-Bañuelos, Marlon Dumas, Ingo Weber, and Alexander Ponomarev. Caterpillar: A business process execution engine on the ethereum blockchain. *Softw. Pract. Exp.*, 49(7):1162–1193, 2019.

[71] Orlenys López-Pintado, Marlon Dumas, Luciano García-Bañuelos, and Ingo Weber. Controlled flexibility in blockchain-based collaborative business processes. *IS*, 104:101622, 2022.

[72] Orlenys López-Pintado, Marlon Dumas, Luciano García-Bañuelos, and Ingo Weber. Controlled flexibility in blockchain-based collaborative business processes. *Information Systems*, 104:101622, 2022.

[73] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[74] Sidra Malik, HMN Dilum Bandara, Nick RTP van Beest, and Xiwei Xu. Smart contracts' upgradability for flexible business processes. In *BPM Blockchain, RPA, CEE, Educators and Industry Forum*, pages 55–70. Springer, 2024.

[75] Nawaz Malla. GitHub repository: Algosol-Dataset. `https://github.com/nawaz-malla/Algosol-Dataset`, 2025. Accessed: 2025-11-24.

[76] Nawaz Malla. Github repository 'Flexible-Business-Processes'. `https://github.com/nawaz-malla/Flexible-Business-Processes`, 2025.

[77] Nawaz Malla. Online artifacts repository, 2025. `https://github.com/nawaz-malla/bpmn-to-blockchain`.

[78] Nawaz Abdullah Malla, Alessandro Marcelletti, Andrea Morichetta, and Francesco Tiezzi. Unveiling algorand storage peculiarities. In *DLT Workshop*, volume 3791. CEUR-WS.org, 2024.

[79] Nawaz Abdullah Malla, Alessandro Marcelletti, Andrea Morichetta, and Francesco Tiezzi. An algorand-based approach for flexible execution of bpmn choreographies. *CEUR Workshop Proceedings*, 4105, 2025.

[80] Nawaz Abdullah Malla, Alessandro Marcelletti, Andrea Morichetta, and Francesco Tiezzi. A blockchain-based, semantics-driven, modular implementation of bpmn choreographies. In *Society 5.0*, CCIS. Springer, 2025. To appear. Available at `https://github.com/nawaz-malla/Flexible-Business-Processes`.

[81] Nawaz Abdullah Malla, Rumyana Neykova, Giuseppe Destefanis, and Francesco Tiezzi. Llm-based translation of ethereum solidity contracts to algorand python. In *2025 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*, pages 7–12. IEEE, 2025.

[82] Michele Marchesi, Lodovica Marchesi, and Roberto Tonelli. An agile software engineering method to design blockchain applications. In *Proceedings of the 14th Central and Eastern European Software Engineering Conference Russia*, pages 1–8, 2018.

[83] Ricardo Martinho, Dulce Domingos, and João Varajão. Cf4bpmn: a bpmn extension for controlled flexibility in business processes. *Procedia Computer Science*, 64:1232–1239, 2015.

[84] Asma Mejri, Sonia Ayachi Ghanouchi, and Ricardo Martinho. Evaluation of process modeling paradigms enabling flexibility. *Procedia Computer Science*, 64:1043–1050, 2015.

[85] Jan Mendling, Ingo Weber, Wil Van Der Aalst, Jan Vom Brocke, Cristina Cabanillas, Florian Daniel, Søren Debois, Claudio Di Ciccio, Marlon Dumas, Schahram Dustdar, et al. Blockchains for business process management-challenges and opportunities. *ACM Transactions on Management Information Systems (TMIS)*, 9(1):1–16, 2018.

[86] Peter Murray, Nate Welch, and Joe Messerman. Erc-1167: Minimal proxy contract, June 2018. Accessed: 2025-12-23.

[87] Rodrigue Tonga Naha and Kaiwen Zhang. Pupa: Smart contracts for BPMN with time-dependent events and inclusive gateways. In *BPM Blockchain, RPA, and CEE Forum*, volume 459 of *LNBIP*, pages 21–35. Springer, 2022.

[88] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. decentralized business review, 21260. *Satoshi Nakamoto Institute*, 2008.

[89] Next Idea Tech. How to successfully implement a blockchain solution in 2025. Technical report, Next Idea Tech, February 2025. Accessed: 2025-02-06.

[90] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 651–654, 2013.

[91] Object Management Group. Business Process Model and Notation (BPMN V 2.0), 2011. OMG Standard document URL: `http://www.omg.org/spec/BPMN`.

[92] OMG. Business Process Model and Notation, 2011.

[93] OpenZeppelin. Proxy upgrade pattern. Accessed: 2025-12-23.

[94] OpenZeppelin. Beacon proxy, 2025. Accessed: 2025-12-23.

[95] OpenZeppelin. Proxy — openzeppelin contracts 4.x api reference, 2025. Accessed: 2025-12-23.

[96] Gabriel Orlanski, Kefan Xiao, Xavier Garcia, Jeffrey Hui, Joshua Howland, Jonathan Malmaud, Jacob Austin, Rishabh Singh, and Michele Catasta. Measuring the impact of programming language distribution. In *International Conference on Machine Learning*, pages 26619–26645. PMLR, 2023.

[97] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[98] Oscar Pastor. Model-driven development in practice: From requirements. In *Theory and Practice of Computer Science*, volume 10139 of *LNCS*, pages 405–410. Springer, 2017.

[99] Paxos. Understanding the different blockchain types. Technical report, Paxos, May 2025. Accessed: 2025-11-17.

[100] Arun Teja Polcumpally, Krishan Kumar Pandey, Anil Kumar, and Ashutosh Samadhiya. Blockchain governance and trust: A multi-sector thematic systematic review and exploration of future research directions. *Heliyon*, 10(12), 2024.

[101] RareSkills. Uups: Universal upgradeable proxy standard (erc-1822), August 2024. Accessed: 2025-12-23.

[102] Manfred Reichert and Barbara Weber. *Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies*, volume 54. Springer, 2012.

[103] Baptiste Roziere, Jie M Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. Leveraging automated unit tests for unsupervised code translation. *arXiv preprint arXiv:2110.06773*, 2021.

[104] Helen Schonenberg, Ronny Mans, Nick Russell, Nataliya Mulyar, and Wil M. P. van der Aalst. Process flexibility: A survey of contemporary approaches. In *Advances in Enterprise Engineering I*, volume 10 of *Lecture Notes in Business Information Processing*, pages 16–30. Springer, 2008.

[105] Dominik Schultes. Sequalsk—a bidirectional swift-kotlin-transpiler. In *2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, pages 73–83. IEEE, 2021.

[106] Xinzhe Shen, Jiale Luo, Hao Wang, Mingyi Liu, Schahram Dustdar, and Zhongjie Wang. A hybrid bpmn-dmn framework for secure inter-organizational processes and decisions collaboration on permissioned blockchain. *arXiv preprint arXiv:2412.01196*, 2024.

[107] Kamilla Stevenson, Oda Skoglund, Mayank Raikwar, and Danilo Gligoroski. Efficient novel privacy preserving pos protocol proof-of-concept with algorand. In *Proceedings of the 2021 3rd Blockchain and Internet of Things Conference*, pages 44–52, 2021.

[108] Fabian Stiehle and Ingo Weber. Blockchain for business process enactment: A taxonomy and systematic literature review. In *BPM Blockchain, RPA, and CEE Forum*, volume 459 of *LNBIP*, pages 5–20. Springer, 2022.

[109] Fabian Stiehle and Ingo Weber. The cost of executing business processes on next-generation blockchains: The case of algorand. In *BPM Blockchain Forum*, volume 527 of *LNBIP*, pages 89–105. Springer, 2024.

[110] An Binh Tran, Qinghua Lu, and Ingo Weber. Lorikeet: A model-driven engineering tool for blockchain-based business process execution and asset management. In *BPM Demo and Industrial Track*, volume 2196 of *CEUR-WS Proc.*, pages 56–60, 2018.

[111] Gautami Tripathi, Mohd Abdul Ahad, and Gabriella Casalino. A comprehensive review of blockchain technology: Underlying principles and historical background with future challenges. *Decision Analytics Journal*, 9:100344, 2023.

[112] Wil M. P. van der Aalst, Niels Lohmann, Peter Massuthe, Christian Stahl, and Karsten Wolf. Multiparty contracts: Agreeing and implementing interorganizational processes. *Comput. J.*, 53(1):90–106, 2010.

[113] Olegas Vasilecas, Diana Kalibatiene, and Dejan Lavbič. Rule-and context-based dynamic business process modelling and simulation. *Journal of Systems and Software*, 122:1–15, 2016.

[114] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[115] Wattana Viriyasitavat, Li Da Xu, Zhuming Bi, and Assadaporn Sapsomboon. Blockchain-based business process management (bpm) framework for service composition in industry 4.0. *Journal of Intelligent Manufacturing*, 31(7):1737–1748, 2020.

[116] Dingding Wang, Jianting He, Siwei Wu, Yajin Zhou, Lei Wu, and Cong Wang. The dark side of upgrades: Uncovering security risks in smart contract upgrades. *arXiv preprint arXiv:2508.02145*, 2025.

[117] Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. Compilable neural code generation with compiler feedback. *arXiv preprint arXiv:2203.05132*, 2022.

[118] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[119] Ingo Weber, Xiwei Xu, Régis Riveret, Guido Governatori, Alexander Ponomarev, and Jan Mendling. Untrusted business process monitoring and execution using blockchain. In *International conference on business process management*, pages 329–347. Springer, 2016.

[120] Mathias Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer, 2007.

[121] Stephen A White. Introduction to bpmn. *Ibm Cooperation*, 2(0):0, 2004.

[122] Aidan ZH Yang, Yoshiki Takashima, Brandon Paulsen, Josiah Dodds, and Daniel Kroening. Vert: Verified equivalent rust transpilation with large language models as few-shot learners. *arXiv preprint arXiv:2404.18852*, 2024.

[123] Mengya Zhang, Preksha Shukla, Wuqi Zhang, Zhuo Zhang, Pranav Agrawal, Zhiqiang Lin, Xiangyu Zhang, and Xiaokuan Zhang. An Empirical Study of Proxy Smart Contracts at the Ethereum Ecosystem Scale . In *ICSE*, pages 620–620. IEEE Computer Society, 2025.

[124] Zhenguang Liu et al. Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion. In *IJCAI*, pages 2751–2759, 2021.

[125] Zhenguang Liu et al. Rethinking smart contract fuzzing: Fuzzing with invocation ordering and important branch revisiting. *arXiv preprint arXiv:2301.03943*, 2023.

[126] Haozhe Zhou, Amin Milani Fard, and Adetokunbo Makanju. The state of ethereum smart contracts security: Vulnerabilities, countermeasures, and tool support. *Journal of Cybersecurity and Privacy*, 2(2):358–378, 2022.

[127] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach Dinh Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. Smart contract development: Challenges and opportunities. *IEEE transactions on software engineering*, 47(10):2084–2106, 2019.