Unlocking Self-Executing SLAs: An Intelligent SLA Extraction, Mapping, & Smart Contract Generation Framework

1st XYZ
ABC
DEF
GHK, LMN
XYZ@ABC

2nd XYZ
DEF
ABC
MNO, PQR
XYZ@ABC

3rd ABC
DEF
LMO
XYZ, DEF
LKJ@PQR

4th DEF
GHQ
MNO
KLM, TUV
XYZ@DEF

***Abstract*—Service Level Agreements (SLAs) which define quantitative metrics and penalties for service providers, have been in existence for decades. However, the lack of automation in their management has been a persistent issue. Smart contracts on blockchain networks offer a promising solution, providing transparent and trusted automation capabilities for such SLAs. In this paper, we introduce an intelligent SLA extraction and mapping framework aimed at addressing the longstanding challenge of efficiently tracking and enforcing Service Level Agreements. Our methodology focuses on ingesting SLAs in both PDF and HTML document formats, reliably extracting key components such as services, objectives, metrics, periods, and remedies. To achieve this, we employ a custom *spaCy named entity recognition model* trained on real-world datasets. Furthermore, we utilize Python-based tools, including Tabula and Beautiful Soup libraries, to capture complex tables representing Service Level Objectives (SLOs). The extracted SLA parameters serve as inputs to our mapping engine, which translates them into three distinct blockchain-based smart contract templates tailored for major vendor platforms i.e., Amazon Managed Blockchain, Kaleido, and Alibaba Cloud services. Comprehensive evaluations conducted across a range of SLAs demonstrate the efficiency of our mapping approach. The mapping engine automatically generates the agreement code of Solidity, CorDapps, and Chaincode, based on the handling terminology, structural variations, and specification differences, which makes it ready for blockchain deployment.**

***Index Terms*—Service Level Agreement, Smart Contract, Blockchain, AI-Driven Natural Language Processing, Data Extraction, Mapping, Blockchain as a service**

I. Introduction

In the era of evolving technology, characterized prominently by the emergence of blockchain-based smart contracts, the conventional natural language-based Service Level Agreement (SLA) documents either in PDF or HTML demands modernization. Although SLAs and Smart Contracts (SCs) formally codify service performance expectations, SLAs lag in embracing digitization to unlock automation, accountability, and analytical insights [1]. As next-generation services demand greater agility and provability, traditional SLAs are largely static and natural-language-bound nature appears outdated. The growing disparity between traditional SLAs and the emerging smart contract landscape inspires us to explore new ways of enhancing SLAs by synergistically connecting them to the latter.

SLAs represent contracts between various parties such as service providers, customers, and resource providers (participants of SLA). The service provider's role is to deliver data and communication services to customers according to the agreed terms, which may also require interacting with other providers to fulfill service commitments. SLAs are typically written in natural language, covering agreement obligations across services [2], [3]. However, bridging the gap between natural language SLAs and smart contract execution requires a framework to systematically map the information extracted from SLAs onto executable smart contract logic. This entails utilizing natural language processing (NLP) techniques to parse SLAs and transform the essential parameters and rules into code. NLP techniques in the field of artificial intelligence (AI) employ machines to analyze unstructured text data such as social media posts, news articles, and reviews [4], [5].

The automated mapping framework serves as a bridge that enables the translation of legacy prose-based agreements into modern software-based smart contract implementation. A smart contract is an autonomous and self-executing code that is deployed on a blockchain. Their operation depends on the validation of predefined agreement conditions. This innovative technology has been successfully integrated into various sectors, including healthcare, finance, supply chains, banking, telecommunication networks, agriculture, and numerous others, showcasing its versatility and transformative potential across diverse industries [6], [7]. Numerous blockchain networks support smart contracts & each platform utilizes its own tailored language and execution environment to write decentralized SC. However, the integration of blockchain as a service (BaaS) provides scalable frameworks to streamline smart contract deployment and management, enabling innovators to focus on value-driven decentralized applications rather than infrastructure [8].

BaaS is a cloud-based platform that utilizes the deployment

The work of XYZ is funded by DEF No. GHQ. Corresponding authors: PQR and LMO.

and management benefits offered by cloud service infrastructure, with the aim of facilitating developers in an efficient and user-friendly environment. The focus is on the effective deployment of blockchain networks, which alleviates concerns about the complex nature of the underlying blockchain architecture [9]. Prominent BaaS platforms actively utilized in the market include Alibaba, Amazon Managed Blockchain, Oracle Blockchain, Kaleido, and Corda Enterprise. Amazon-managed Blockchain is provided by the Amazon Web Service. Amazon Managed Blockchain delivers services for both private and public infrastructure using Hyperledger Fabric and Ethereum, respectively. It delivers fully managed services and handles tasks, such as network setup and network joining. Amazon's Managed Blockchain on AWS simplifies the network configuration with straightforward steps and establishes peer nodes through the Amazon Management Console [10], [11]. Kaleido is another BaaS solution that partners with Amazon, and with Microsoft Azure for on-premise cloud. It supports Enterprise Ethereum, Polygon, Quorum, Corda, Hyperledger Besu, and Hyperledger Fabric [12]. Alibaba BaaS facilitates the utilization of Hyperledger Fabric, Ant Blockchain technologies, and Quorum. This expedites the development of blockchains by simplifying the configuration process and ensuring a swift setup. Moreover, Alibaba BaaS extends support for cryptographic algorithm (CA) certificates to enhance security measures. Alibaba Cloud BaaS delivers advantages such as elevated security, robust stability, user-friendly interfaces, extensive openness, and efficient sharing services for applications based on blockchain technology [13], [14].

In this paper, we introduce a complete framework for mapping SLAs to SCs. More precisely, the framework (and the implemented prototype) identifies the SLA components and generates a corresponding smart contract template. Our novel method for bridging the gap between Service Level Agreements and smart contracts comprises of three essential components. First, we employ NLP techniques to extract information from publicly available PDF and HTML based SLAs following a comprehensive SLA analysis. Then, the smart contracts can be analyzed by selecting a blockchain network from each of the three BaaS platforms (Amazon Managed Blockchain, Kaleido, and Alibaba) and analyzing them according to their structure, functions, and use cases. Finally, we map the extracted SLA components to the templates of the chosen blockchain networks (Ethereum, Corda, and Hyperledger Fabric) within the BaaS platform using our implementation modules.

The subsequent sections of this paper are organized as follows. Section II thoroughly examines existing studies in the field, offering valuable insights. Section III outlines the methodology employed for the proposed solution and provides a detailed account of the approach. Section IV presents a comprehensive evaluation of the proposed solution, shedding light on its practical implications. Finally, Section V serves as the concluding section, summarizing the key findings and suggesting potential directions for future research.

II. Related Works

In reviewing the relevant literature, we discovered that most of the existing research has focused on extracting information from cloud SLAs to create knowledge bases for customers. Several studies have used NLP and semantics to parse SLAs and identify key terms, metrics, and definitions to represent SLAs in a structured, machine-readable format for automated analysis. However, there has been limited exploration of smart contracts for BaaS offerings. Additionally, there is a gap in the research on frameworks, implementation, or tools for mapping natural language SLA components to templates for executable smart contracts for BaaS.

In [15], a framework for extracting definitions, remedies for non-compliance, and contextual words from cloud SLAs was presented. Their framework allows the extraction of key information from SLAs through Beautiful Soup library and structuring them in a graph format to capture relationships between terms, definitions, and remedies. This knowledge graph enables users to efficiently query and analyze SLA terms when evaluating and comparing cloud services. Another study [16] focused on the extraction of cloud SLA metrics and presented a solution grounded in NLP and semantic web technologies. The authors used the information extracted by pattern learning and a noun phrase extractor to create a knowledge base for cloud users. The primary objective is to establish a knowledge base for the management of cloud services. The study conducted by [17] utilized the GATE tool to extract information from 36 SLAs for monitoring purposes. The researchers applied this tool to SLAs from Amazon, Cisco, and Microsoft and employed their information as input features in machine learning techniques, such as Support Vector Machine, Perceptron Algorithm with Uneven Margins, Naive Bayes, K-nearest neighbor, and C4.5, for sentence recognition and retrieval of definitions and formulas.

The work in [18] extracts the information from tables and rules and obligations from six HTML SLA for the end users to manage multiple cloud services. It employed Beautiful Soup library for the table extraction and modal logic formalization for the extraction of rules and obligations. In [19], the authors also employed Beautiful Soup library for the extraction of attribute information from Python data analysis.

III. Methodology

This section discusses the framework, and its working components. The mapping of SLAs to smart contract templates framework is are divided into two key parts as shown in Figure 1. The first part involves data collection of PDF and HTML SLAs, analysis, and classification of these SLAs, along with the extraction of relevant information, such as the name of the SLA, metrics, purpose, compensation clauses, covered service, service provider, and limitations. Natural language processing techniques facilitate the systematic extraction of SLA details after analyzing the SLA document type, into a structured format. The second part analyzes existing smart contract implementations from various BaaS solutions to identify common templates, patterns, and components. This provides a framework for combining modular, adjustable smart contract templates to encode different aspects of SLA procedures and logic.

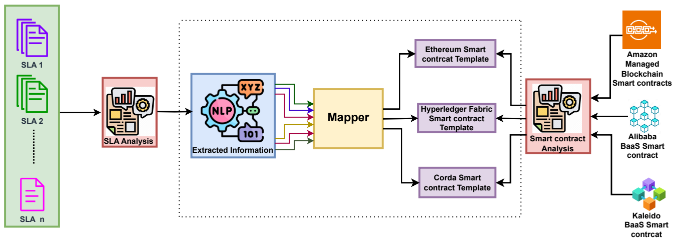

SLA 1
SLA 2
SLA n
SLA Analysis
NLP
XYZ
101
Extracted Information
Mapper
Ethereum Smart contrcat Template
Hyperledger Fabric Smart contract Template
Corda Smart contract Template
Smart contract Analysis
Amazon Managed Blockchain Smart contracts
Alibaba BaaS Smart contract
Kaleido BaaS Smart contrcat

Fig. 1. Proposed framework of the solution.

To enable feasible mapping from SLAs to executable smart contract templates, we developed a specialized framework and then implement it as a prototype. This implementation ingests the extracted structured SLA data and then programmatically inserts relevant details into the appropriate sections of the template of the smart contract. The complete implementation automates the populating of contract variables, service functions, and business logic conditions based on SLA specifications.

A. *Data collection for SLAs*

We initially compiled a corpus of publicly available SLAs i.e, 32 PDF SLAs and 19 HTML SLAs from a diverse range of cloud computing, networking, voice, data, Internet, cybersecurity, VoIP, and fiber service vendors. However, upon careful manual examination of PDF SLAs, we found seven of the sourced documents to be overly broad terms of service agreements, rather than quantifiable SLAs. As likely, we found eleven SLAs from HTML SLAs giving the same broad terms of service agreements. These contained ambiguous language about customer obligations, limitations, and program policies that did not meet our criteria for defined measurable service performance objectives. Hence, we filtered these and focused our study on the remaining 25 PDF and 8 HTML SLAs.

B. *Analysis of SLAs*

This module consists of analysis of both PDF and HTML SLAs. The initial manual examination of PDF SLAs focused on extracting key characteristics, such as purpose, SLOs, length, primary parameters, covered services, and validity period, from the perspective of encoding SLAs into smart contracts. This preliminary analysis revealed wide variation across PDF SLA documents. The lengths ranged from single-page agreements to lengthy 12-15 page contracts. The number and types of defined SLOs also differed based on the nature of the services offered. For instance, the Ziply Fiber Ethernet SLA specifies SLOs for availability, mean time to repair, packet delivery, latency, and jitter, whereas the FPUAnet Dark-Fiber SLA focuses on fiber interruption, mean time to repair, and signal loss SLOs. Some SLAs covered a single service, while others addressed multiple services such as voice, data, and dedicated Internet collectively. Furthermore, this module performs the manual examination of HTML SLAs. These HTML SLAs exhibited key structural differences from PDF contract documents possibly stemming from greater web design flexibility. We observed use of loose formatting preferences with divisions, lists and variable styling rather than the strict numbered clauses seen in PDFs. They incorporate additional hyperlinks for navigation, terms expansion, visual metrics dashboards and interactive configuration wizards unavailable in static PDF text. HTML tables also employ simplified spanning without the intricate constructs spanning pages in contracts. Furthermore, analysis revealed HTML SLAs can encapsulate expansive service coverage spanning organizational offerings under a unified digitized policy framework, contrasting sharply with specificity observed in conventional documents. For example, examined samples like the Huawei Cloud SLA encompassed performance objectives and remedies for sixty two distinctive constituent services such as ranging from computing, storage, CDN to database, security, ModelArts services, Elastic IP and more.

Additionally, we classified these SLAs according to their purpose, such as service delivery, billing, payment, penalties, monitoring, purchasing, help desks, and support. We found that most SLA documents (PDF and HTML) focused on penalty and payment objectives. Additionally, their structure varied significantly in terms of headings, sections, tables, and other formatting elements used to present information. This diversity of SLAs highlights the need to normalize and standardize SLA data to map into smart contract templates effectively. Variance in language, structure, metrics, and parameters motivates the development of natural language processing techniques to systematically extract SLA details from heterogeneous documents in a more uniform manner.

Our methodology capitalizes on the robust capabilities of the spaCy NLP, Tabula and Beautiful Soup libraries in Python to facilitate the extraction of meaningful information from the collection of PDF and HTML SLAs respectively, for seamless integration into smart contracts. Our approach first involves the segmentation of SLA documents into discrete tables and sentences, achieved through adept PDF parsing and extraction. To bridge the gap between SLA components and smart contracts, we employed the spaCy Python library, leveraging its NLP-based capabilities to extract crucial details. The elements targeted for extraction include SLA name, purpose, service provider name, covered services, and validity period. To enhance the efficacy of this process, an annotation file was created for training, derived from diverse 100 SLAs (both PDF and HTML) with the assistance of a Named Entity Recognition (NER) annotator. We also made annotation file for validation from 20 different SLAs (PDF and HTML). Subsequently, the spaCy model was trained using this annotation file to facilitate the extraction of specified entities from the SLAs.

It is noteworthy that the training approach involves not fine-tuning an existing model but rather independently training the spaCy model to suit the specific requirements of SLA extraction. Moreover, for SLO extraction, we implemented the Tabula and Beautiful Soup libraries in Python for PDF and HTML SLAs respectively. Recognizing that SLOs in SLA

```
pragma solidity ^0.8.0;
interface IUniswapRouter {
 function swapExactETHForTokens(uint256 amountOutMin, address[] calldata path, address to, uint256
deadline) external payable returns (uint256[] memory amounts);
 function getAmountsOut(uint256 amountIn, address[] calldata path) external view returns (uint256[]
memory amounts);
}
```

Fig. 2. Solidity purchase-token smart contract structure.

documents are typically presented in a tabulated form, we employ a Tabula and Beautiful Soup libraries to extract these tables. Following the extraction, we employ careful filtration of rows and columns, coupled with pattern matching, to obtain the desired results. The final output is then saved in JSON format, optimizing it for mapping purposes within the broader framework of smart contract integration.

C. Data collection for smart contracts

For smart contract templates, we selected three leading enterprise BaaS solutions: Amazon Managed Blockchain, Microsoft Azure, and Alibaba Cloud. From each BaaS provider, we chose one representative blockchain network for deeper analysis: Ethereum from Amazon, Hyperledger Fabric from Alibaba Cloud, and Corda from Kaleido. This cross section of both platforms and protocols aims to understand the templates that capture common smart contract patterns and components generalized beyond vendor specific implementations. By evaluating diverse BaaS smart contracts for the supply chain, finance, healthcare, and other domains, we can extract reusable templates encapsulating core functions such as agreements, payments, reminders, penalties, and utilities found across many contract types. The ultimate goal is to map modular smart contract building blocks capable of flexibly encoding a wide range of SLA parameters, logic, and workflows through templates.

D. Analysis of smart contracts

Analysis of smart contracts assists in examining three key aspects: the overall contract structure, implemented functions, and domain-specific use case handling. Structural analysis identifies standard sections for initialization, agreements, events, and utilities. Functional analysis catalogs reusable modules for payments, reminders and penalties. Usecase analysis evaluates how contracts encapsulate business logic workflows for supply chain, healthcare, and finance verticals. These dimensions enable understanding for mapping natural language agreements into executable smart contract code.

1) Amazon Managed Blockchain (AMB): We chose five Ethereum smart contracts from the AMB for examination, aiming to understand a standardized template for SLAs encompassing service delivery, monitoring, support, payment, and penalty aspects. During our analysis, we identify shared features and functionalities within these smart contracts, including state variables, events, modifiers, error handling mechanisms, constructors, structures, arrays, function declarations, libraries, inheritance, and enumerations.

```
//contracts/SupplyChain.sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
import "@openzeppelin/contracts/access/Ownable.sol";
contract SupplyChain is Ownable {
    enum Status {
        Ordered,
        Shipped,
        Delivered,
        Cancelled
    }
    struct Item {
        uint id;
        string name;
        Status;
        address orderedBy;
        address approvedBy;
        address deliveredTo;
    }
    mapping(uint => Item) private items;
    uint private itemCount;
```

Fig. 3. Solidity supply-chain smart contract structure.

Examining the structural and operational aspects, as illustrated in Figure 2, the Solidity smart contract snippet[1] pertains to the acquisition of a specific token from Uniswap on the Ethereum platform. The initial line designates the Solidity Compiler. The subsequent lines introduce a specified interface for interaction with the Uniswap router, comprising two sets of function signatures. The `swapExactETHFORTokens` function, delineated with input parameters such as amount out min, path, recipient address, and deadline, is noteworthy. The term `external` signifies that this function is callable from outside the contract, while *payable* indicates its capacity to receive ether and return an array containing a uint256 amount. Another function signature, `getAmountsOut` outlines a function to determine the output amount based on the input amount and trading path. In this instance, the `external` *view* attribute ensures that the function solely provides information, without altering the contract state.

The initial part of the supply chain smart contract[2] is illustrated in Figure 3. Within this context, the `contract` keyword introduces the contract named `SupplyChain` inheriting relevant ownership-related functionalities from the *Ownable* contract. Subsequently, the `enum` keyword establishes an enumeration type labeled *Status* encapsulating potential states (ordered, shipped, delivered, and canceled) for an item within the supply chain. Additionally, a `struct` is employed to represent individual items in the supply chain, encompassing attributes such as the ID, name, status, orderer information, approver details, and delivery recipient. The `mapping` keyword facilitates the association of items with their corresponding struct representations, and the private state variable *itemCount* is employed to track the total number of items in the supply chain.

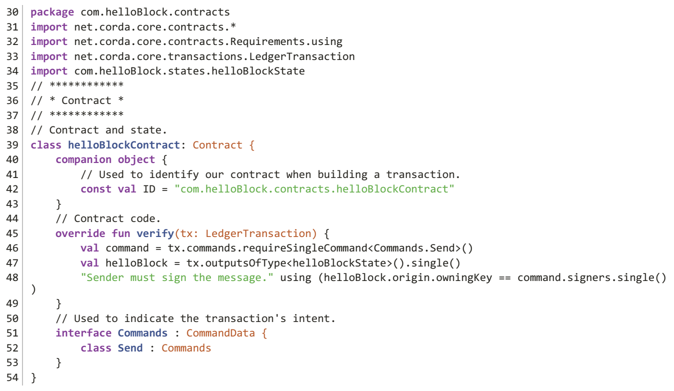
From a usecase standpoint, a smart contract designed for voting addresses this scenario through functions catering to granting voting rights, delegation, casting votes, and determining a winning proposal. The `give right to vote`

[1]https://medium.com/@solidity101/how-to-purchase-a-token-on-the-uniswap-universal-router-with-solidity-3075ce1468d1

[2]https://bizenforce.medium.com/building-a-decentralized-supply-chain-tracking-application-with-smart-contracts-2bc0c132b50a

function empowers the chairperson to assign voting rights to a specific address, while the `delegate` function enables a voter to delegate their vote to another address. The `vote` function facilitates the casting of votes for a particular proposal, and the `winning proposal` function computes the successful proposal by considering all prior votes.

2) Kaleido: We chose five Kotlin-based smart contracts from Corda, utilized as a BaaS in Kaleido. Within CorDapps, there are recurring structural elements, including the contract class, state class, flow classes, service classes, and Command Data class. Moreover, several frequently encountered functions within these CorDapps include constructor, verify(), command handlers, transaction generation, vault operations, flow operators, context accessors, helper functions, and logging functions.



```
package com.helloBlock.contracts
import net.corda.core.contracts.*
import net.corda.core.contracts.Requirements.using
import net.corda.core.transactions.LedgerTransaction
import com.helloBlock.states.helloBlockState
// ************
// * Contract *
// ************
// Contract and state.
class helloBlockContract: Contract {
    companion object {
        // Used to identify our contract when building a transaction.
        const val ID = "com.helloBlock.contracts.helloBlockContract"
    }
    // Contract code.
    override fun verify(tx: LedgerTransaction) {
        val command = tx.commands.requireSingleCommand<Commands.Send>()
        val helloBlock = tx.outputsOfType<helloBlockState>().single()
        "Sender must sign the message." using (helloBlock.origin.owningKey == command.signers.single()
)
    }
    // Used to indicate the transaction's intent.
    interface Commands : CommandData {
        class Send : Commands
    }
}
```

Fig. 4. Corda smart contract structure.

As illustrated in Figure 4, the Corda smart contract[3] for transaction verification is subject to both structural and functional analyses. The `package` statement indicates its placement within the `com.helloBlock.contracts` package, and it imports various Corda contracts-related classes. Subsequently, the code declares a class named `helloBlockContract` implementing the `Contract` interface, thereby signifying its role in defining the contract logic. Within the companion object, a constant *ID* is defined for identifying the contract during transaction construction. The `verify` function, crucial for the transaction verification process, is then implemented. This function checks for the presence of a specific command (*Command.Send*) in the transaction. Simultaneously, it retrieves the single-output state of the *helloBlockState* type from the transaction. Utilizing the `using` function, a condition is enforced, ensuring that the owner key of the *helloBlockState* aligns with the single signer of the command. This validation guarantees that the sender is the correct signatory of the message. Additionally, the contract introduces an interface named `Commands` extending `CommandData` which incorporates an inner class `Send` that implements the *Commands* interface.

[3]https://chainstack.com/picking-an-enterprise-protocol-to-develop-on-introduction/

In the analysis of a use case[4], a scenario involves the initiation of a cake request, construction and completion of a transaction involving multiple participants, and interaction with a counter-party for transaction verification and finalization. The process encompasses the definition of a cake state, formulation of a transaction for a cake request, signing and finalization of the transaction, retrieval of the final cake request state, and reception and verification of a signed transaction from the counter-party. Within the cake state definition, the contract establishes a state that represents a cake of a specified type, baked by one party (the baker) and requested by another party (the customer). The participants involved in this state were explicitly outlined.

In the transaction-building phase for a cake request, the cake request state is initialized and a command indicating the request, along with the corresponding participant's public keys, is generated. Utilizing an identifier, a transaction with a cake-request state as the output is constructed. The subsequent signing and finalizing of the transaction triggers a flow session with the baker. The initial transaction is signed with the signature of the initiating party, followed by the collection of signatures from the involved parties. The transaction was then finalized and recorded on a ledger. The return of the final cake request state provides the output state of the final transaction. In the process of receiving and verifying a signed transaction from the counter-party, a flow is initiated to receive the signed transaction, which is then verified. The finality of the transaction is received, completing the interaction with the counter-party in the transaction verification and finalization process.

3) Alibaba: We chose five GO language-based smart contracts (chaincode) from Hyperledger Fabric, serving as a BaaS on Alibaba Cloud. Within these chaincode, several recurring structures are identified, including the chaincode structure, Init() function, Invoke() function, shim interface, stub structure, and error-handling components. Additionally, the prevalent functions observed in these chaincode include InitLedger(), InvokeHandler(), various asset and user management functions, transaction-related functions, event-related functions, functions for handling JSON, and functions dedicated to error handling.

The snippet in Figure 5 provides part of a Hyperledger Fabric smart contract[5], offering insights from a structural analysis standpoint. The initial line, `package main` designates the GO file as a standalone executable program. The subsequent line commencing with `import` is dedicated to importing essential libraries and packages, encompassing *byte*, *encoding/json*, *fmt*, *strconv*, *strings*, *time*, *the Hyperledger Fabric Chaincode Shim package*, and *protobuf* definitions for facilitating communication with the Hyperledger Fabric peer. Following this, the line `type SimpleChaincode struct` introduces a structure named *SimpleChaincode* serv-

[4]https://simon-maxen.medium.com/have-your-cake-and-eat-it-with-daml-driver-for-corda-523f98de4ae

[5]https://github.com/vanitas92/fabric-external-chaincodes/blob/master/chaincode/marbles02.go

```go
package main
import (
    "bytes"
    "encoding/json"
    "fmt"
    "os"
    "strconv"
    "strings"
    "time"
    "github.com/hyperledger/fabric-chaincode-go/shim"
    pb "github.com/hyperledger/fabric-protos-go/peer"
)
// SimpleChaincode example simple Chaincode implementation
type SimpleChaincode struct{}
// Marble represents the data structure of the assets managed by the chaincode
type Marble struct {
    ObjectType string `json:"docType"`
    Name       string `json:"name"`
    Color      string `json:"color"`
    Size       int    `json:"size"`
    Owner      string `json:"owner"`
}
```

Fig. 5. Hyperledger Fabric smart contract structure.

ing as a representation of the chaincode itself. Additionally, the line `type marble struct` defines a structure named *marble* outlining the data structure for a marble. This structure encapsulates properties such as *Object Type*, *Name*, *Color*, *Size*, and *Owner*.

The content[6] in Figure 6 illustrates the `Invoke` function of the *SimpleChaincode* Functioning as a router, the Invoke function directs various invoke operations based on the provided function name, deferring the actual implementation to specific functions, such as invoke, delete, and query. The determination of the execution operation relies on the value of the function variable derived from the invoked request. The initial line introduces the function as part of the *SimpleChaincode* structure, which serves as the entry point for handling invoke requests. It takes a `shim.ChaincodeStubInterface` as an argument and returns a *pb.Response*. The second line serves the purpose of debugging by printing a message to the console. The third line retrieves the function names and parameters from the invoked request. Subsequent lines scrutinize the function name to ascertain the specific operation (invoke, delete, or query) to be executed. In addition, the final line in the code snippet ensures that if the function name does not match any of the anticipated names, it results in an error response.

Another example[7] is provided for use-case analysis, centering on the management of property assets (*ID*, *color*, *size*, *owner*, and *appraised value*) on a blockchain. This use case encompasses functionalities, such as asset creation, retrieval, update, deletion, existence checks, and ownership transfer. The `InitLedger` function initializes the ledger with a predefined set of assets. `CreateAsset` generates a new asset with the provided details, ensuring that an asset with the given ID does not already exist. `ReadAsset` retrieves and returns the details of a specific asset by its ID. `UpdateAsset` modifies the details of an existing asset with new parameters, first confirming the asset's existence. `DeleteAsset` removes

[6]https://github.com/anandhakumarpalanisamy/bityoga_fabric_test_chaincodes/blob/master/bank_chaincode/go/chaincode_example02.go

[7]https://github.com/hyperledger/fabric-samples/blob/main/asset-transfer-basic/chaincode-go/chaincode/smartcontract.go

```go
func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
    fmt.Println("ex02 Invoke")
    function, args := stub.GetFunctionAndParameters()
    if function == "invoke" {
        // Make payment of X units from A to B
        return t.invoke(stub, args)
    } else if function == "delete" {
        // Deletes an entity from its state
        return t.delete(stub, args)
    } else if function == "query" {
        // the old "Query" is now implemtned in invoke
        return t.query(stub, args)
    }
    return shim.Error("Invalid invoke function name. Expecting \"invoke\" \"delete\" \"query\"")
}
```

Fig. 6. Hyperledger Fabric SC: Invoke of Chaincode.

a specified asset from the ledger and verifies its existence. `AssetExists` checks if an asset with a given ID exists in the ledger. `TransferAsset` facilitates the transfer of asset ownership to a new owner, retrieving the current owner, updating the owner field, and storing the updated asset. Additionally, `GetAllAssets` retrieves all the assets present in the ledger.

IV. Evaluation

In this section, we evaluate our proposed methodology for automated mapping of SLA components to blockchain-based smart contracts. Our methodology comprises two key components. The first component focuses on extracting salient information from natural language SLA documents, which is a prerequisite for coding contract parameters. We implemented a supervised machine learning pipeline utilizing conditional random fields to parse SLA documents and reliably extract aspects such as defined services, performance metrics, measurement periods, and remedy specifications. Through a labeled dataset and cross-validation, our extraction model achieves strong precision and recall, which can correctly identify and capture the required SLA components embedded in unstructured text. The second vital component is the creation of a matching module that can use these extracted SLA parameters and map them to predefined smart contract templates from various blockchain-as-a-service providers. By seamlessly integrating these two components, our methodology aims to completely automate the translation of SLA documents into executable distributed ledger agreements while eliminating friction, costs, and errors associated with manual approaches. We believe that this represents meaningful progress towards SLA migration at scale across enterprises.

Furthermore, to comprehensively evaluate the performance of our spaCy NER model for SLA information extraction, the loss, score and entity scores progression per iterations are plotted as a learning curve in Figure 7. It can be clearly observed that the loss function optimized during model training steadily decreased with an increasing number of iterations. Simultaneously, as the model weights are updated to minimize loss, the entity scores and score calculated on the validation samples for classifying entities increases until convergence.

Table I presents the results of our information extraction model when applied to a sample of real-world SLAs, showcasing six SLAs examples (three PDF SLAs and three HTML SLAs) where it accurately captures entities as well as certain limitations. For the Gigabit Fiber SLA document, we observe

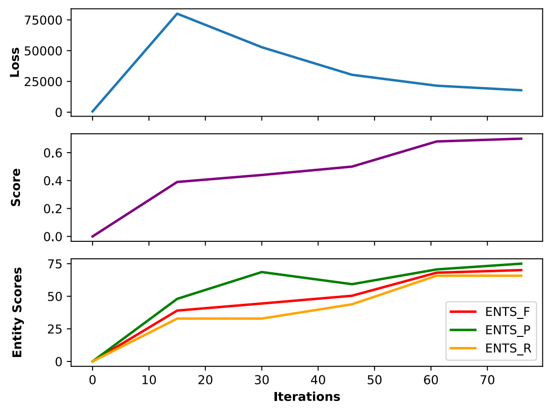
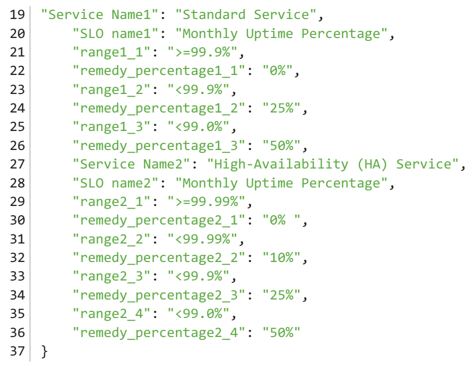


Fig. 7. NER Loss, score and entity score of spaCy NER model.



```
"Service Name1": "Standard Service",
    "SLO name1": "Monthly Uptime Percentage",
    "range1_1": ">=99.9%",
    "remedy_percentage1_1": "0%",
    "range1_2": "<99.9%",
    "remedy_percentage1_2": "25%",
    "range1_3": "<99.0%",
    "remedy_percentage1_3": "50%",
    "Service Name2": "High-Availability (HA) Service",
    "SLO name2": "Monthly Uptime Percentage",
    "range2_1": ">=99.99%",
    "remedy_percentage2_1": "0% ",
    "range2_2": "<99.99%",
    "remedy_percentage2_2": "10%",
    "range2_3": "<99.9%",
    "remedy_percentage2_3": "25%",
    "range2_4": "<99.0%",
    "remedy_percentage2_4": "50%"
}
```

Fig. 8. Json format of Gigabit Fiber SLA.

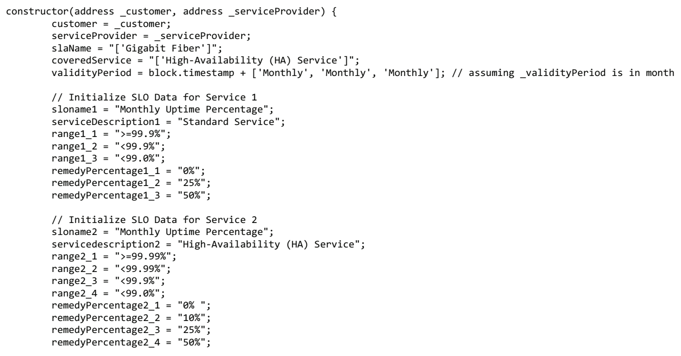
that key components such as the SLA name, purpose, service provider, and validity period are correctly identified from the unstructured text. However, only one covered service is extracted, although two are defined in the document. Additionally, the validity period text was extracted multiple times undesirably. Moving to the Azure Security Center SLA, the prototype accurately captures the SLA name, covered services, provider and validity period but not the purpose. In some examples the extracted data includes some extraneous text under the purpose entity. Analyzing other SLAs, we find a mix of precisely extracted entities and missed entities (written in italic), depending on the textual content and format. However, notably for the Ziply Fiber SLA, Telesystem Dedicated Service Level Agreement, Patton CLoud Services SLA and iTel SLA, our approach extracted all important entities perfectly.

Moreover, SLOs are often represented in tabular formats specifying ranges for metrics such as uptime, latency, and frame loss, and penalty percentages associated with these ranges based on deviation severity. Automatically extracting such structured SLO data is key before mapping them to blockchain smart contracts that encode this programmatic logic. Figure 8 shows an example of the SLO extraction output of the proposed method in a serializable JSON format. It illustrates two covered services under a Gigabit Fiber SLA i.e., a Standard Service and a High Availability service. The figure shows how the method correctly identifies and groups the relevant SLO range, qualifying metrics such as monthly uptime to measure, and remedy percentages as penalties from the tabular representation into accessible key-value pairs at a per-service level. This extracted machine-readable JSON output containing all SLO metrics and remedy data across defined services is well suited for ingestion in the next mapping steps.

Next, we conducted comprehensive evaluations to demonstrate the capability of our proposed mapping system to accurately populate smart contract code templates with the extracted SLA parameters. Owing to space constraints, we present illustrative examples covering the mapping of SLA components to both Ethereum-based and Corda-based contract templates hosted on Amazon Managed Blockchain and Kaleido BaaS platforms. We used the prototype implementation across more than 15 SLA document samples and three different BaaS platforms comprising public and private network configurations. Our results establish good accuracy in correctly populating a fully functional contract from input SLAs spanning definition, terms, and metrics positioned as a significant leap towards automated, trusted self-execution in the IT ecosystem.

A part of the Ethereum smart contract constructor template populated by our framework is shown in Figure 9. It illustrates the effective mapping demonstrated on the Gigabit Fiber SLA document comprising details of two distinct services, that is, a standard service and a high availability service. While our information extraction model only detected one covered service from the free text, it was accurately able to extract both services by analyzing tabular SLO data. The framework mapped basic address identifiers, and validity periods, and correctly preserved the array-based structure defining service-specific SLO criteria with low and high penalty percentages based on monthly uptime deviations. However, directly injecting the extracted text values can lead to downstream issues. Hence, a parse percentage helper function is used in the template that transforms free-form percentage strings into float types required for payment and penalty calculations invoked based on contract monitoring events for true auto-execution.

Figure 10 highlights the mappings of the Corda smart

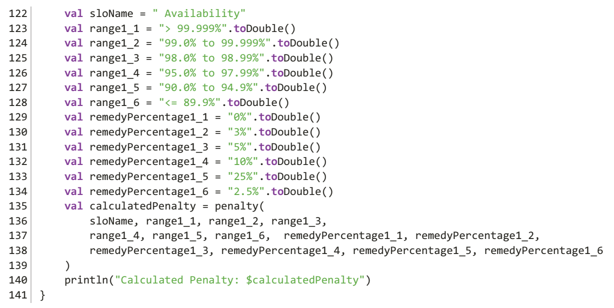


```
constructor(address _customer, address _serviceProvider) {
        customer = _customer;
        serviceProvider = _serviceProvider;
        slaName = "['Gigabit Fiber']";
        coveredService = "['High-Availability (HA) Service']";
        validityPeriod = block.timestamp + ['Monthly', 'Monthly', 'Monthly']; // assuming _validityPeriod is in month

        // Initialize SLO Data for Service 1
        sloname1 = "Monthly Uptime Percentage";
        serviceDescription1 = "Standard Service";
        range1_1 = ">=99.9%";
        range1_2 = "<99.9%";
        range1_3 = "<99.0%";
        remedyPercentage1_1 = "0%";
        remedyPercentage1_2 = "25%";
        remedyPercentage1_3 = "50%";

        // Initialize SLO Data for Service 2
        sloname2 = "Monthly Uptime Percentage";
        servicedescription2 = "High-Availability (HA) Service";
        range2_1 = ">=99.99%";
        range2_2 = "<99.99%";
        range2_3 = "<99.9%";
        range2_4 = "<99.0%";
        remedyPercentage2_1 = "0% ";
        remedyPercentage2_2 = "10%";
        remedyPercentage2_3 = "25%";
        remedyPercentage2_4 = "50%";
```

Fig. 9. Mapped smart contract of Gigabit Fiber SLA.

TABLE I
EXTRACTED INFORMATION (AS IS) FROM PDF AND HTML SLAS BY USING TRAINED-SPACY MODEL.

No.	SLA name	Purpose	Covered Service	Service Provider	Validity Period
1	Gigabit Fiber	SLA states Customer's sole and exclusive remedy for any failure by Gigabit Fiber meet the SLO.	High-Availability (HA) Service	Gigabit Fiber	Monthly
2	Telesystem Dedicated Service Level Agreement	This SLA is the sole and exclusive remedy for any type of disruptions or deficiencies of any kind whatsoever for the Service	Core Data and Voice Services (Type I circuits, ELINE services, Dedicated DIA , SIP Transport, Voice Services; inclusive of equipment managed by Telesystem	Telesystem	Annual
3	Zeta Broadband	*Zeta shall issue credits for each Service*	DIA (Dedicated Internet Access) services, Managed Router Service - Managed Firewall - Shared Web/E-mail Hosting Additional Terms and Conditions Usage.	Zeta Broadband	Month
4	Azure Security Center	*If we do not achieve and maintain the Service Levels for each Service as described in this SLA, then you may be eligible for a credit towards a portion of your monthly service fees*	Azure Services	21Vianet	Monthly
5	Patton Cloud Services SLA	If we do not achieve the Availability as described in this SLA, then you may be eligible for a credit towards a portion of your service fees.	Cloud Service	Patton	Annual
6	iTel SLA's	Customers will be credited to their account a percentage of their Monthly Recurring Charge (MRC) for the month in which the Outage occurred in accordance with the tables in the respective SLA	Data Services	iTel	Monthly



```
    val sloName = " Availability"
    val range1_1 = "> 99.999%".toDouble()
    val range1_2 = "99.0% to 99.999%".toDouble()
    val range1_3 = "98.0% to 98.99%".toDouble()
    val range1_4 = "95.0% to 97.99%".toDouble()
    val range1_5 = "90.0% to 94.9%".toDouble()
    val range1_6 = "<= 89.9%".toDouble()
    val remedyPercentage1_1 = "0%".toDouble()
    val remedyPercentage1_2 = "3%".toDouble()
    val remedyPercentage1_3 = "5%".toDouble()
    val remedyPercentage1_4 = "10%".toDouble()
    val remedyPercentage1_5 = "25%".toDouble()
    val remedyPercentage1_6 = "2.5%".toDouble()
    val calculatedPenalty = penalty(
        sloName, range1_1, range1_2, range1_3,
        range1_4, range1_5, range1_6,  remedyPercentage1_1, remedyPercentage1_2,
        remedyPercentage1_3, remedyPercentage1_4, remedyPercentage1_5, remedyPercentage1_6
    )
    println("Calculated Penalty: $calculatedPenalty")
}
```

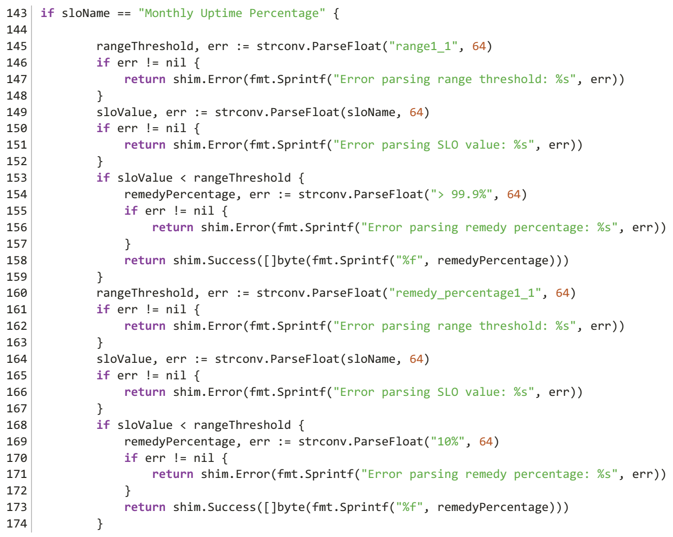
Fig. 10. Mapped smart contract of Telco Services SLA.

contract. It presents the Kotlin code snippet from the penalty-calculation contract function, where the SLA-extracted components are effectively injected into the Kaleido BaaS template. The illustrated data originate from parsing the Voice Telco Services SLA comprising a single SLO around service availability and outlines payment and penalty policies based on the degree of violations. We observe that the information linking relevant contract parties, validity clauses, SLO criteria ranges, and remedy percentages are smoothly mapped across the respective variables and data structures in the target contract.

Figure 11 depicts the mapping of a chaincode smart contract written in the Go programming language, showing specifically the code implementation for penalty and payment functions based on SLOs. By codifying SLO terms extracted from the Azure Security Center's SLA into executable chaincode, this enables automating SLA enforcement. The Azure Security Center SLA defines a monthly uptime percentage SLO along with remedy percentages for violations. We observed the effective mapping of SLA specifications from source JSON format document into chaincode.

V. CONCLUSION

This work contributes an end-to-end automated pipeline that compile complex SLA documents, including PDF and HTML formats, directly into executable smart contract logic. This overcomes key barriers to accessing the benefits of blockchain-based SLA enforcement. Our two-phase methodology first implements an advanced natural language processing approach customized to reliably extract key SLA parameters from text. These parameters encompass covered services, SLA name, purpose, periods, and remedies essential for mapping to code. Furthermore, processing complex specification tables enables capturing service-specific performance thresholds suitable for codification. To demonstrate the effectiveness of our proposed framework, we input the extracted SLA information into our model for mapping to the smart contract templates of three leading BaaS vendors. By evaluating across the pipeline, from information extraction through to final smart contract deployment, our prototype performs accurate policy mapping. This streamlined migration empowers businesses to leverage smart contracts, eliminating reconstructing policies from scratch and thereby fostering increased enterprise blockchain adoption. Future work focuses on expanding target datasets and automating contract generation for broader IT service relationships, as well as enhancing the efficiency and usability of our proposed mapping system.



```
if sloName == "Monthly Uptime Percentage" {

        rangeThreshold, err := strconv.ParseFloat("range1_1", 64)
        if err != nil {
            return shim.Error(fmt.Sprintf("Error parsing range threshold: %s", err))
        }
        sloValue, err := strconv.ParseFloat(sloName, 64)
        if err != nil {
            return shim.Error(fmt.Sprintf("Error parsing SLO value: %s", err))
        }
        if sloValue < rangeThreshold {
            remedyPercentage, err := strconv.ParseFloat("> 99.9%", 64)
            if err != nil {
                return shim.Error(fmt.Sprintf("Error parsing remedy percentage: %s", err))
            }
            return shim.Success([]byte(fmt.Sprintf("%f", remedyPercentage)))
        }
        rangeThreshold, err := strconv.ParseFloat("remedy_percentage1_1", 64)
        if err != nil {
            return shim.Error(fmt.Sprintf("Error parsing range threshold: %s", err))
        }
        sloValue, err := strconv.ParseFloat(sloName, 64)
        if err != nil {
            return shim.Error(fmt.Sprintf("Error parsing SLO value: %s", err))
        }
        if sloValue < rangeThreshold {
            remedyPercentage, err := strconv.ParseFloat("10%", 64)
            if err != nil {
                return shim.Error(fmt.Sprintf("Error parsing remedy percentage: %s", err))
            }
            return shim.Success([]byte(fmt.Sprintf("%f", remedyPercentage)))
        }
```

Fig. 11. Mapped smart contract of Azure Security Center SLA.

REFERENCES

[1] K. Upadhyay, R. Dantu, Y. He, S. Badruddoja, and A. Salau, “Can't understand slas? use the smart contract,” in *2021 Third IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. IEEE, 2021, pp. 129–136.

[2] R. Ranchal and O. Choudhury, “Slam: A framework for sla management in multicloud ecosystem using blockchain,” in *2020 IEEE Cloud Summit*. IEEE, 2020, pp. 33–38.

[3] A. Akbari-Moghanjoughi, J. R. D. A. Amazonas, G. Santos-Boada, and J. Solé-Pareta, “Service level agreements for communication networks: A survey,” *arXiv preprint arXiv:2309.07272*, 2023.

[4] G. Melo, M. Chaves, M. Kolter, and J. H. Schleifenbaum, “Skills requirements of additive manufacturing-a textual analysis of job postings using natural language processing,” in *International Conference on Additive Manufacturing in Products and Applications*. Springer, 2023, pp. 299–316.

[5] N. J. Prottasha, S. A. Murad, A. J. M. Muzahid, M. Rana, M. Kowsher, A. Adhikary, S. Biswas, and A. K. Bairagi, “Impact learning: A learning method from feature's impact and competition,” *Journal of Computational Science*, vol. 69, p. 102011, 2023.

[6] N. Hamdi, C. El Hog, R. Ben Djemaa, and L. Sliman, “A survey on sla management using blockchain based smart contracts,” in *International Conference on Intelligent Systems Design and Applications*. Springer, 2021, pp. 1425–1433.

[7] S. Biswas, K. Sharif, F. Li, Z. Latif, S. S. Kanhere, and S. P. Mohanty, “Interoperability and synchronization management of blockchain-based decentralized e-health systems,” *IEEE Transactions on Engineering Management*, vol. 67, no. 4, pp. 1363–1376, 2020.

[8] W. Zheng, Z. Zheng, X. Chen, K. Dai, P. Li, and R. Chen, “Nutbaas: A blockchain-as-a-service platform,” *Ieee Access*, vol. 7, pp. 134 422–134 433, 2019.

[9] Q. Lu, X. Xu, Y. Liu, I. Weber, L. Zhu, and W. Zhang, “ubaas: A unified blockchain as a service platform,” *Future generation computer systems*, vol. 101, pp. 564–575, 2019.

[10] D. Li, L. Deng, Z. Cai, and A. Souri, “Blockchain as a service models in the internet of things management: Systematic review,” *Transactions on Emerging Telecommunications Technologies*, vol. 33, no. 4, p. e4139, 2022.

[11] K. Gai, J. Guo, L. Zhu, and S. Yu, “Blockchain meets cloud computing: A survey,” *IEEE Communications Surveys & Tutorials*, vol. 22, no. 3, pp. 2009–2030, 2020.

[12] Z. Yang, H. Lei, and W. Qian, “A hybrid formal verification system in coq for ensuring the reliability and security of ethereum-based service smart contracts,” *IEEE Access*, vol. 8, pp. 21 411–21 436, 2020.

[13] S. M. Alshurafa, D. Eleyan, and A. Eleyan, “A survey paper on blockchain as a service platforms,” *International Journal of High Performance Computing and Networking*, vol. 17, no. 1, pp. 8–18, 2021.

[14] S. Biswas, K. Sharif, F. Li, and S. Mohanty, “Blockchain for e-healthcare systems: Easier said than done,” *Computer*, vol. 53, no. 7, pp. 57–67, 2020.

[15] D. N. Ganapathy and K. P. Joshi, “A semantically rich framework to automate cloud service level agreements,” *IEEE Transactions on Services Computing*, vol. 16, no. 1, pp. 53–64, 2022.

[16] S. Mittal, K. P. Joshi, C. Pearce, and A. Joshi, “Automatic extraction of metrics from slas for cloud service management,” in *2016 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2016, pp. 139–142.

[17] L. De Marco, F. Ferrucci, M. T. Kechadi, G. Napoli, and P. Salza, “Towards automatic service level agreements information extraction.” in *CLOSER (2)*, 2016, pp. 59–66.

[18] A. Gupta, S. Mittal, K. P. Joshi, C. Pearce, and A. Joshi, “Streamlining management of multiple cloud services,” in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE, 2016, pp. 481–488.

[19] Y. Li, “Python data analysis and attribute information extraction method based on intelligent decision system,” *Mobile Information Systems*, vol. 2022, 2022.