UNIVERSIDADE DE CAXIAS DO SUL
ÁREA DO CONHECIMENTO DE CIÊNCIAS EXATAS E ENGENHARIAS

RICARDO FERREIRA DE ALENCASTRO GUIMARÃES

GUIA PARA CRIAÇÃO DE SMART CONTRACTS NO ETHEREUM

CAXIAS DO SUL

2024

RICARDO FERREIRA DE ALENCASTRO GUIMARÃES

GUIA PARA CRIAÇÃO DE SMART CONTRACTS NO ETHEREUM

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Orientador: Prof. Dra. Helena Graziottin Ribeiro

CAXIAS DO SUL

2024

RICARDO FERREIRA DE ALENCASTRO GUIMARÃES

GUIA PARA CRIAÇÃO DE SMART CONTRACTS NO ETHEREUM

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Aprovado em 28/11/2024

BANCA EXAMINADORA

_____

Prof. Dra. Helena Graziottin Ribeiro
Universidade de Caxias do Sul - UCS

_____

Prof. Me. Patrícia Montemezzo
Universidade de Caxias do Sul - UCS

_____

Prof. Me. Giovanni Ely Rocco
Universidade de Caxias do Sul - UCS

AGRADECIMENTOS

Agradeço aos meus pais, Ricardo de Alencastro Guimarães e Maristela Ferreira Guimarães, que sempre me apoiaram e estiveram ao meu lado para ajudar nessa etapa importante de minha vida.

A minha orientadora Helena Graziottin Ribeiro, que através de seus ensinamento me permitiu chegar a conclusão deste trabalho. Por saber me apoiar e acreditar em meu potencial.

Também agradeço aos meus amigos que sempre estiveram ao meu lado e me motivaram a conclusão deste trabalho.

“Maior que um temporal é a fé que habita em mim”

Hungria Hip Hop

RESUMO

Os *smart contracts* são uma entre as várias aplicações de *blockchains*. Estes contratos tem sido cada vez mais utilizados por terem sua imutabilidade garantida pela estrutura do *blockchain*. Para criar um *smart contract* é preciso escolher uma implementação de *blockchain*, e utilizar as ferramentas que fazem parte do seu contexto. Este trabalho apresenta uma proposta de um guia para a a criação de *smart contracts* em uma *blockchain*, e um estudo de caso sobre *smart contracts* utilizando Ethereum e Chainlink. Como foco do estudo de caso, utiliza-se o contrato de compra e venda de automóveis com cláusula de arras, desenvolvendo um *smart contract* armazenado na rede do Ethereum e utilizando Chainlink para fornecer dados externos à rede. Desta forma o processo se torna completamente descentralizado e, confiável e pode ser verificado a qualquer momento, mantendo os termos do acordo e as penalidades previstas no caso de descumprimento de alguma das partes que está negociando.

Palavras-chave: *Smart Contracts*. *Blockchain*. Ethereum. Chainlink.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

API *Application Program Interface*

CPF Cadastro de Pessoa Física

DApps *Decentralized applications*

DON *Decentralized Oracle Network*

ETH *Ether*

EVM *Ethereum Virtual Machine*

HTTP *Hypertext Transfer Protocol*

ID Identificador

IP Protocolo da Internet

P2P *Peer-to-Peer*

PoS *Proof-of-Stake*

PoW *Proof-of-Work*

RAM *Random Access Memory*

RENAVAM Registro Nacional de Veículos Automotores

URL *Uniform Resource Locator*

SUMÁRIO

1 INTRODUÇÃO

O desenvolvimento da tecnologia, das comunicações e da Internet proporciona uma interação cada vez maior entre sistemas. Isso acarretou no crescimento dos sistemas distribuídos, que são computadores conectados entre si representados por um sistema único e coerente. (TANENBAUM; STEEN, 2008). Antigamente os computadores pessoais não possuíam um hardware potente, como os que são comercializados atualmente. Esse avanço no mercado trouxe a possibilidade da utilização de máquinas pessoais como cliente e servidor, tirando a responsabilidade de um servidor principal para gerenciar a rede.

Essa evolução da tecnologia proporcionou surgimento de tecnologias como o *blockchain*, que é um sistema distribuído descentralizado. Segundo Sarmah:

> A tecnologia *blockchain* é normalmente associada a criptomoedas como Bitcoin. É um banco de dados de registro de transações que é distribuído, e que é validado e mantido por uma rede de computadores em todo o mundo. (SARMAH, 2018)

Definimos um sistema distribuído por um conjunto de hardwares ou softwares, os chamados nodos da rede. Esses nodos comunicam e realizam ações entre si através do envio de mensagens. (COULOURIS; DOLLIMORE; KINDBERG, 2013) No caso do *blockchain* a solução utilizada é chamada de *Peer-to-Peer* (P2P). Nesse tipo de rede é utilizado o poder computacional dos computadores conectados a ele, para isso é necessário que haja algoritmos de distribuição e recuperação de objetos na rede. É extremamente importante que sempre haja a comunicação entre os pares, para que não haja qualquer tipo de erro decorrente de uma má organização de dados e serviços do sistema.

O Bitcoin é amplamente reconhecido como a tecnologia mais associada ao *blockchain*, principalmente devido à sua popularidade e uso difundido. No entanto, é importante ressaltar que o *blockchain* é uma tecnologia que vai além das criptomoedas. Atualmente, sua utilização abrange uma variedade de aplicativos e sistemas que operam em redes descentralizadas, onde os usuários conectados desempenham um papel fundamental. Esses aplicativos podem executar diferentes funcionalidades e oferecer serviços diversos, aproveitando a segurança e a transparência proporcionadas pelo *blockchain*.

Os contratos estão presente a anos em nosso cotidiano, sendo responsáveis por estabelecerem regras de negócios entre pessoas e/ou empresas. Por sua vez, a evolução dessa tecnologia também trouxe inovação para este meio, permitindo o desenvolvimento de diferentes aplicações como os *smart contracts*. Estes contratos são totalmente eletrônicos e automatizados. São feitos através de linguagens de programação, podendo executar determinada ação de acordo com o que foi definido a ele fazer. (LUU *et al.*, 2016)

Os *smart contracts* são uma das aplicações disponíveis para *blockchains*. Estes contratos estão cada vez mais populares por terem sua imutabilidade garantida através do *blockchain*. Isso significa que uma vez que alguma empresa ou pessoa decidir participar do contrato, está terá que seguir com o que este contrato foi estabelecido fazer.

1.1 OBJETIVOS

O objetivo deste trabalho é fazer a proposta de um guia para a a criação de *smart contracts* no Ethereum, e implementar um estudo de caso através de *smart contract* desenvolvendo um contrato de compra e venda de automóveis com cláusula de arras. As informações de negociação e execuções deste contrato serão gerenciados por uma rede *blockchain*. Com base no objetivo geral, foram elaborados os seguintes objetivos específicos:

- Compreender o funcionamento do *blockchain*.
- Experimentar na prática as tecnologias do Ethereum.
- Compreender o desenvolvimento de *smart contracts*.
- Implementar um estudo de caso utilizando Ethereum.

1.2 ESTRUTURA DO TRABALHO

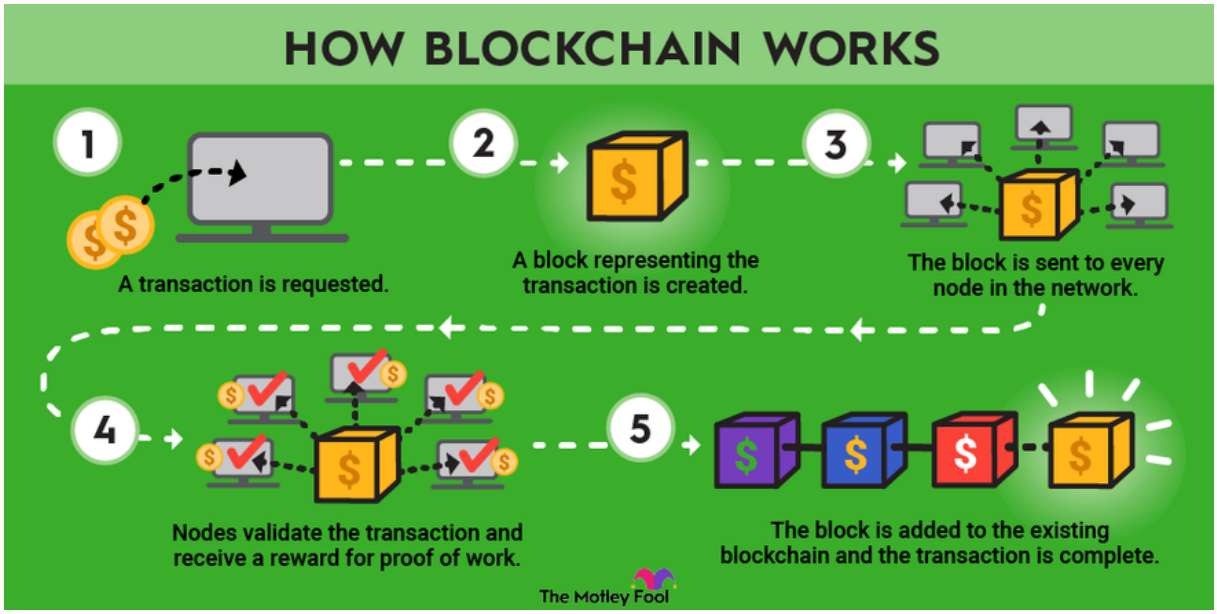
O trabalho está organizado da seguinte forma:

- O Capítulo 2 apresenta o funcionamento de uma rede *blockchain*, explicando o que é um sistema distribuído descentralizado. É apresentado os diferentes tipos de redes disponíveis e suas diferenças, algorítimos de consenso e uma de suas principais redes, o Ethereum.
- No Capítulo 3 é apresentado uma das tecnologias que podem ser utilizadas com *blockchain*, os *smart contracts*. É abordado também como é feito o consumo de dados externos utilizando Chainlink.
- No Capítulo 4 é apresentada a proposta de desenvolvimento, citando as tecnologias que serão utilizadas, a estrutura do *smart contract* a ser desenvolvido utilizando o Ethereum.
- No Capítulo 5 é apresentado o estudo de caso, apresentando a estrutura do *smart contract*, as leis seguidas para o desenvolvimento e os testes para validação do contrato.
- No capítulo 6 é apresentado as conclusões deste trabalho.

2 *BLOCKCHAIN*

O *blockchain* é uma tecnologia que revolucionou a forma como as transações são registradas e verificadas, oferecendo um sistema descentralizado e imutável. Essa inovação se baseia em uma estrutura de dados, composta por blocos interligados. Por ser uma rede dinâmica, novos blocos podem ser adicionado a rede ao longo de sua existência. Conforme ilustrado na Figura 1, quando a inclusão de um bloco é solicitada, a requisição é enviada aos nós da rede, que verificam a validade da transação antes de inserir o bloco na *blockchain*. Uma vez inserido, o bloco torna-se imutável, garantindo a integridade dos dados (NARAYANAN *et al.*, 2016).

Figura 1 – Etapas de funcionamento de uma *blockchain*.



Fonte: <https://www.fool.com/terms/b/blockchain/>

2.1 ORIGEM

Em 2008, Satoshi Nakamoto publicou seu artigo introduzindo o Bitcoin, no qual apresentou o conceito de uma rede descentralizada que registra transações em uma cadeia de blocos (*blockchain*) utilizando *hashing* e o mecanismo de consenso conhecido como *Proof-of-Work* (PoW) (NAKAMOTO, 2008). Essa rede é completamente P2P e *open source*, o que significa que funciona sem a necessidade de uma autoridade central, como empresas ou governos, sendo mantida exclusivamente pelos próprios usuários da rede.

Este trabalho de Nakamoto foi baseado em ideias que já haviam sido estudadas anteriormente. Um dos estudos mais relevantes em relação ao *blockchain* é a pesquisa 'Improving the Efficiency and Reliability of Digital Time-Stamping', realizada por Stuart Haber e W. Scott. Essa pesquisa descreve os fundamentos que sustentam o *blockchain* até os dias de hoje. Ela

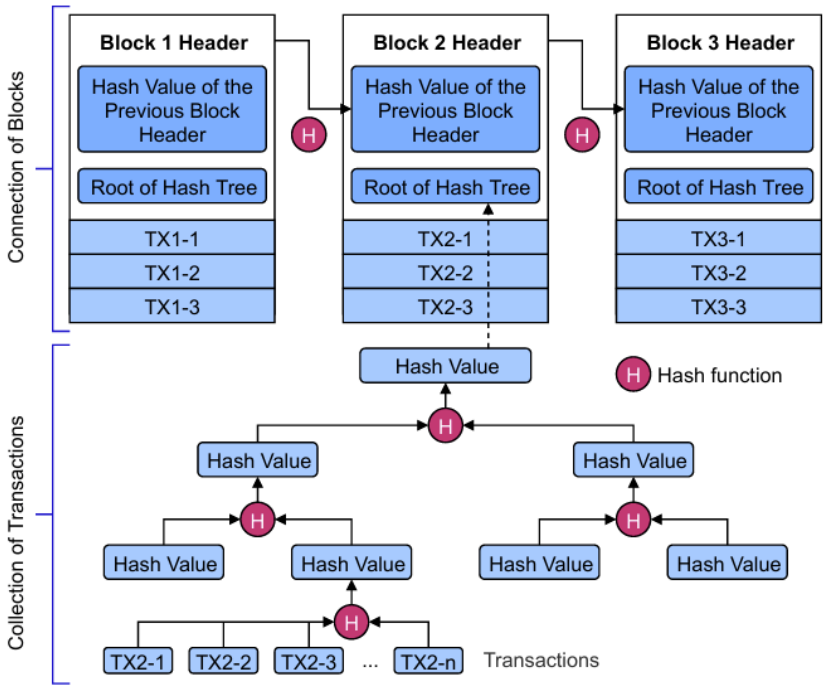
propõe o armazenamento seguro e criptografado de documentos com carimbo de data/hora, utilizando cadeias de blocos lineares. Uma vez registrado, o documento não pode ser alterado, substituído ou excluído. (BAYER; HABER; STORNETTA, 1993).

2.2 ESTRUTURA

O *blockchain* opera de forma independente de qualquer instituição, dependendo apenas de seus usuários conectados à rede. Essa solução utiliza uma arquitetura de rede distribuída do tipo P2P, onde cada nó conectado atua simultaneamente como cliente e servidor. Essa configuração permite a comunicação direta entre os computadores da rede, eliminando a necessidade de um serviço centralizado para seu funcionamento. Essa característica é fundamental para a descentralização do *blockchain* (COULOURIS; DOLLIMORE; KINDBERG, 2013).

Essa tecnologia é composta por uma rede de blocos interligados que armazenam e organizam dados de transações, utilizando a estrutura da árvore de Merkle para garantir a integridade e autenticação das informações. Essa abordagem permite que ramificações específicas sejam acessadas para análise sem a necessidade de consultar todo o conjunto de registros de transações. A árvore de Merkle é um tipo de estrutura de dados utilizada em criptografia que contém uma estrutura hierárquica de informações resumidas sobre um conjunto maior de dados - por exemplo, um arquivo - e é usada para verificar seu conteúdo. Essa estrutura utiliza *hashes* (funções matemáticas que tem como entrada uma chave, e geram como saída um valor único) para autenticar os dados em uma rede distribuída, facilitando a verificação de informações. Essa organização, juntamente com o processamento das transações e a estruturação dos blocos, pode ser representada de forma visual pela Figura 2 para aprimorar o entendimento de seu funcionamento (GADEKALLU *et al.*, 2022).

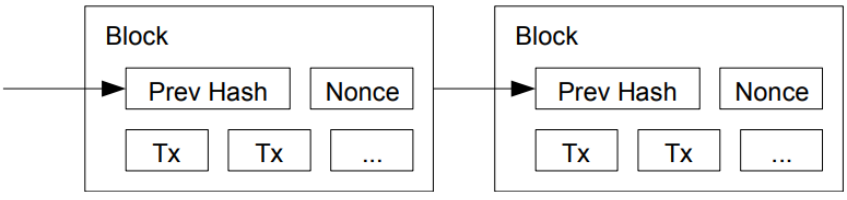
Figura 2 – Estrutura geral do *blockchain*.



Fonte: (GADEKALLU *et al.*, 2022)

A Figura 3 demonstra a estrutura básica de um bloco, esse bloco possui o *hash* do bloco anterior, as transações que ele possui e o *nonce*. O *nonce* não tem significado algum, ele é apenas utilizado como uma variável que os participantes podem alterar para obter diferentes valores de *hash*. O algoritmo de *hash* utilizado no Bitcoin é o Sha-256, esse *hash* deve conter todas as informações do bloco que está sendo validado na rede e o *hash* do bloco anterior. Para gerar um bloco os participantes da rede alteram o valor *nonce* repetidamente, recalculando o *hash* do bloco a cada iteração, na esperança de encontrar um valor de *hash* que satisfaça as condições exigidas. (BAG; RUJ; SAKURAI, 2017).

Figura 3 – Estrutura de um bloco.



Fonte: (NAKAMOTO, 2008)

2.3 TIPOS DE *BLOCKCHAINS*

As redes *blockchain* podem ser classificadas como públicas, privadas ou permissionadas. Cada rede tem sua vantagem e desvantagem, cabe a quem for implementar decidir qual é a melhor.

Redes privadas possuem acesso restrito, permitindo entrada apenas a usuários autorizados pelo administrador da rede. Entidades públicas ou privadas, como bancos, frequentemente optam por seu uso devido ao interesse em protocolos que oferecem maior segurança para suas transações. Para isso, essas organizações podem desenvolver redes próprias ou utilizar redes de terceiros já existentes, como a *blockchain* da empresa R3 CEV[1], que já conta com alguns bancos como clientes (GUEGAN, 2017).

Outro ponto atrativo para que seja utilizada, é a eficiência que um bloco tem para ser gravado. Por ter seus usuários já conhecidos e ter confiança neles, são utilizados algoritmos que tem um custo computacional menor, diferentemente do *blockchain* do Bitcoin por exemplo, que utiliza o PoW como protocolo de consenso. A utilização desses protocolos mais eficientes e com menor uso de poder computacional, acaba se tornando atrativo também por trazer um gasto menor de energia, tornando atrativo sua implementação (CHRISTIDIS; DEVETSIKIOTIS, 2016).

Blockchains privadas possuem uma alta privacidade de seus dados, apesar disso não é simples alterar os dados presentes e por muitas vezes pode não ser possível alterar. Para que haja uma alteração de dados é necessário o consenso da rede, da mesma forma que seria para adicionar um novo bloco (YANG *et al.*, 2020).

Tanto redes públicas quanto privadas são descentralizadas (YANG *et al.*, 2020), embora apresentem diferenças significativas. Uma rede pública, como a do Bitcoin, depende da participação ativa de usuários para funcionar. No início do Bitcoin, por muitas vezes a rede ficava sem participantes, devido à novidade da tecnologia e à ausência de confiança e conhecimento sobre seu funcionamento.

Diferentemente da *blockchain* privada, todos os blocos de uma rede pública são acessíveis, então não há confidencialidade sobre os dados. No caso do Bitcoin, por exemplo, ao se descobrir o endereço de uma carteira, é possível visualizar todo o histórico de transações e o saldo associado a ela.

Para incentivar a participação em redes públicas, os usuários que validam e criam novos blocos são recompensados, em um processo conhecido como mineração. No Ethereum, por exemplo, a inclusão de um novo bloco na rede resulta na remuneração em *Ether* (ETH), como compensação pelo uso computacional empregado no processo (GUEGAN, 2017).

Para que não haja fraude, como o gasto duplo de uma mesma moeda, após um bloco ser inserido na rede, ele não poderá mais ser alterado ou excluído da rede. Essa característica aumenta a confiabilidade da rede, uma vez que suas transações são públicas, permitindo que a comunidade monitore seu funcionamento e identifique rapidamente qualquer irregularidade.

Redes públicas apresentam desempenho inferior quando comparadas às privadas. Isso ocorre porque, ao permitir que qualquer usuário entre ou saia a qualquer momento, independentemente de sua localização no mundo, há um impacto significativo na eficiência para a confir-

[1] <https://r3.com/>

mação de transações, especialmente em comparação com redes privadas e sistemas tradicionais (MOHAN, 2019).

Essencialmente *blockchains* públicas e privadas seguem o mesmo princípio, porém cada uma com suas particularidades, na Tabela 1 pode ser visto um comparativo resumido entre ambas.

Tabela 1 – Comparativo entre *blockchains* públicas e privadas

	Pública	Privada
Acesso	Qualquer usuário	Usuários selecionados
Direito de controle	Descentralizado	Descentralizado
Velocidade da transação	Baixa	Alta
Consenso	Não permissionada	Permissionada
Custo por transação	Alto	Baixo
Privacidade dos dados	Público	Privada a organização
Imutabilidades	Imutável	Parcialmente imutável
Eficiência	Baixa	Alta

Fonte: O Autor (2024).

O Ethereum é uma das redes públicas no qual mais vem se destacando e se consolidando. Trouxe consigo inovações para a tecnologia, com a possibilidade de construção de aplicativos descentralizados. Serviu de inspiração para a aparição de outras plataformas também, como é o caso da Stellar, outra rede pública que utiliza seu próprio algoritmo de consenso, o Stellar Consensus Protocol[2]. Esse projeto tem como foco a velocidade para processar e finalizar transações em sua rede (LAWTON, 2022).

Grandes empresas como a IBM também estão presentes no desenvolvimento de *blockchains*. A IBM Blockchain é uma rede privada que tem como foco empresas como clientes e utiliza o *framework* Hyperledger Fabric[3]. Este *framework* tem como objetivo servir apenas redes de *blockchain* privadas e traz uma grande eficiência por limitar os nodos conectados à rede (R., 2020).

2.4 ETHEREUM

O Ethereum é uma das principais redes de *blockchains* em funcionamento. Anunciado por Vitalik Buterin no final de 2013 e lançado em 2015, trata-se de uma *blockchain* pública e *open-source* (SYED *et al.*, 2019). Vitalik era um membro ativo da comunidade do Bitcoin, vendo o potencial que o Bitcoin tinha, propôs uma nova plataforma que obtivesse sua própria moeda, porem que pudesse ter outras funcionalidades, além de armazenar transações (GERRING, 2016).

[2] <https://stellar.org/papers/stellar-consensus-protocol>
[3] <https://hyperledger-fabric.readthedocs.io/en/release-2.5/>

Inicialmente o Ethereum utilizava PoW como seu algoritmo de consenso, seguindo o utilizado pelo Bitcoin. A partir de 2017 o Ethereum começou a implementação do algoritmo de *Proof-of-Stake* (PoS). Essa mudança no formato de consenso do Ethereum começou paralelamente e apenas em 2022 foi concluída, deixando totalmente o PoW de lado. No PoS os usuários participantes da rede são chamados de validadores e, para criar um novo bloco, é necessário que um validador deposite uma quantia mínima de 32 ETH em um *smart contract* no Ethereum. Caso ocorra uma operação fraudulenta, os ETH depositados podem ser destruídos, e o validador pode até ser banido da rede. Para a criação de um novo bloco, um validador é escolhido aleatoriamente, porem quanto mais ETH possuir, maiores são as chances de ser selecionado, o que contribui para a descentralização da rede. Após a seleção, o validador responsável pela criação do bloco o envia a um comitê de validadores, que verifica e valida o bloco proposto.

A principal diferença em relação ao Bitcoin é que o Ethereum é programável, resultando em uma *blockchain* capaz de manter *smart contracts* e diferentes aplicações, como jogos, redes sociais, entre outras. Todas essas funcionalidades ocorrem dentro da rede Ethereum por meio de sua máquina virtual, que pode operar mediante o pagamento de taxas determinadas pelo *gas*, a unidade de medida utilizada para calcular o custo de uma operação na rede. Quanto mais complexa a operação, maior será o custo de execução (BUTERIN, 2014).

A *Ethereum Virtual Machine* (EVM) é a responsável pela execução de todos contratos e aplicações dentro do Ethereum, executando os *bytecodes* gerados pela sua linguagem de programação Solidity. Os *bytecodes* são armazenados dentro do *blockchain*, por tanto todo nodo que deseja participar da rede deve rodar a EVM para que seja possível executar o que é requisitado pelo cliente.

2.5 APLICAÇÕES

O *blockchain*, além de ser amplamente conhecido por sua aplicação em criptomoedas como Bitcoin e Ethereum, oferece um vasto leque de usos em diversos setores. Sua capacidade de registrar dados de forma imutável e segura tem permitido sua utilização em áreas como certificação digital, verificação da qualidade do produto e sistemas de votação eletrônica (ZĪLE; STRAZDIŅA, 2018).

Outra aplicação crescente do *blockchain* é no desenvolvimento de *Decentralized applications* (DApps), que são aplicativos descentralizados que operam em redes *blockchain*. Esses aplicativos eliminam intermediários ao permitir a interação direta entre usuários, sendo amplamente utilizados em finanças descentralizadas, jogos *play-to-earn* e redes sociais descentralizadas. Essa abordagem aumenta a transparência, a segurança e o controle dos dados pelos próprios usuários, promovendo inovações em várias indústrias (ZĪLE; STRAZDIŅA, 2018).

Outra aplicação relevante do *blockchain* é no uso de *smart contracts*, que permitem automatizar processos de forma segura e eficiente. Esses contratos têm ganhado espaço em

diversos setores por sua capacidade de simplificar e otimizar transações e acordos, destacando-se como uma ferramenta poderosa no contexto das tecnologias descentralizadas.

3 *SMART CONTRACTS*

Este capítulo foi dividido em 3 partes para facilitar o entendimento. A Seção 3.1 o funcionamento e a estrutura de um *smart contract*. A Seção 3.2 explica o funcionamento dos *hybrid smart contracts*, para um contrato ter acesso a dados externos do *blockchain*. A Seção 3.3 comenta sobre o Chainlink, um *oracle* utilizado para fornecer dados externos a *blockchains*.

3.1 ESTRUTURA E FUNCIONALIDADES

Os contratos desempenham um papel fundamental nas transações comerciais e jurídicas, proporcionando um alicerce para a confiança mútua entre as partes envolvidas. Eles estabelecem termos e condições que regem um acordo, delineando as responsabilidades e expectativas de cada parte. No mundo dos negócios, a integridade e a eficácia de um contrato são vitais para garantir o bom funcionamento das operações. Além disso, os contratos não apenas servem como um meio de formalizar acordos, mas também como uma maneira de assegurar que as partes cumpram suas obrigações de maneira justa e equitativa, promovendo, assim, a harmonia nas relações contratuais. A confiabilidade do cumprimento e a garantia de que as partes não se desviem dos termos acordados desempenham um papel central na importância e utilidade dos contratos (TERRA, 2011).

Os contratos inteligentes, ou *smart contracts*, são uma das aplicações das *blockchains*. Os *smart contracts* foram criados com o propósito de trazer uma maior facilidade na hora de realizar as ações contratuais, sem a necessidade de intermédio entre as partes que estão a negociar. O conceito de *smart contract* foi introduzido em 1994 por Nick Szabo, que definiu a tecnologia como uma transação computadorizada que executa os termos definidos no contrato (CHRISTIDIS; DEVETSIKIOTIS, 2016).

Um *smart contract* pode ser definido como um programa que é capaz de tomar decisões que são executadas automaticamente a partir de condições pré definidas em seu código. Uma de suas características mais marcantes é a eliminação da necessidade de um intermediário confiável, como um administrador, para mediar as interações entre as entidades contratantes. Essa autonomia é garantida pela execução descentralizada proporcionada pela rede *blockchain*, que assegura o cumprimento das condições preestabelecidas no contrato. Além de reduzir o atrito nas transações, essa abordagem possibilita um nível elevado de automação, promovendo maior eficiência e confiabilidade no gerenciamento de ativos e na transferência de valores (CASEY, 2018).

Como os *smart contracts* são publicados no *blockchain*, a própria rede assegura e valida o contrato, protegendo de ataques e tornando o intermédio de terceiros desnecessário. A eliminação dessa dependência acarreta em uma eficiência maior por ter suas ações já definidas e um

custo menor do que seria necessário através de meios tradicionais, tendo como único gasto a taxa imposta pela rede para ser executada (ZHENG *et al.*, 2020).

Após a publicação de um contrato na rede, o que for definido em seu escopo não poderá mais ser alterado. Essa imutabilidade traz uma maior segurança para quem for utilizá-lo, pois garante o que foi acordado do início ao fim de seu uso. Tendo isso em vista, é importante que o contrato seja escrito de maneira que garanta a segurança em seu uso. Caso haja algum erro ou alguma mudança seja necessária neste contrato, um novo contrato com as correções feitas deverá ser adicionado à rede para ser utilizado.

Para estabelece um contrato em uma *blockchain*, é preciso utilizar os recursos que ela disponibiliza. Por exemplo, se utilizarmos a Ethereum, em geral a linguagem utilizada para os contratos é a Solidity[1]. A estrutura de um contrato pode ser comparada à de uma classe de uma linguagem de programação como o C#. O Algoritmo 1 representa um contrato simples de exemplo. Nele, o contrato se assemelha a uma classe, mas utiliza a palavra reservada *contract*. Dentro de um contrato, é possível adicionar um construtor, embora seu uso seja opcional, assim como em outras linguagens de programação. O construtor é invocado apenas uma vez, no momento em que o contrato é publicado na *blockchain*, e, após isso, ele não é executado novamente. Ele serve como inicializador do contrato, portanto, se for necessário definir valores iniciais, isso deverá ser feito durante sua execução.

Dentro de um contrato, é possível definir funções. Essas funções são executadas conforme o gatilho estabelecido para elas e desempenham um papel fundamental nos contratos, pois é nelas que toda a lógica e as condições definidas são implementadas. Para garantir maior segurança, os *smart contracts* contam com funções que verificam quem está solicitando a execução de uma função. Por isso, é importante que o programador defina quem pode executá-las, a fim de evitar ações maliciosas (ZHENG *et al.*, 2020). O Algoritmo 1 ilustra o uso de uma função em um contrato. Nesse exemplo, sempre que alguém solicitar a função *getContador()* do contrato, ela retornará o valor atual do contador e, em seguida, o incrementará.

Algoritmo 1 – Estrutura de um *smart contract* escrito em Solidity

```
1
2  pragma solidity ^0.8.0;
3
4  contract Exemplo {
5      uint public variavel;
6      uint contador;
7
8      constructor(uint parametro) internal {
9          variavel = parametro;
10     }
11
12     function getContador() public returns (uint) {
```

[1] <https://docs.soliditylang.org/en/v0.8.20/>

```
13          uint aux = contador;
14          contador++;
15          return aux;
16      }
17  }
```

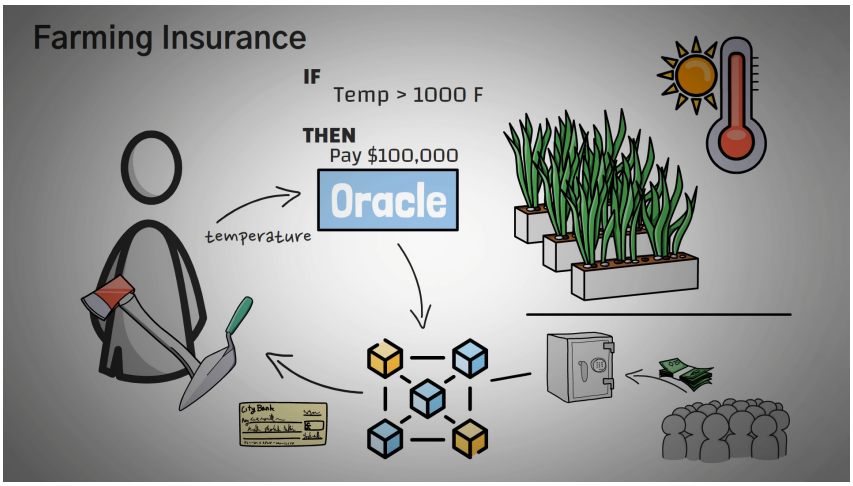
Fonte: O autor (2024).

3.2 *HYBRID SMART CONTRACTS* E *ORACLES*

A popularidade dos *smart contracts* vem crescendo a cada dia devido à sua confiabilidade e segurança. Parte disso se deve ao fato de que o *blockchain* não interage com dados externos, apenas com os que estão dentro de sua própria rede. Isso acontece pois haveria uma inconsistência caso houvesse uma requisição para um serviço, como uma API. O *blockchain* deve garantir que, de um bloco inicial até qualquer outro bloco, os mesmos valores sejam mantidos. Se uma requisição a uma API, fosse incluída dentro de um bloco, o estado poderia ser diferente daquele no momento em que o bloco foi gerado, o que impediria o consenso na rede.

Por conta dessa demanda, surgiram os *hybrid smart contracts*, que recebem informações externas através de *oracles*. Utilizando como exemplo um contrato de seguro agrícola, conforme ilustrado na Figura 4, agricultores fazem pagamentos mensais de um valor pré-determinado. Para que o valor do seguro seja pago, é necessário verificar as condições climáticas da região do agricultor, como 10 dias consecutivos de sol ou um período inteiro de chuva. Com a utilização de um *oracle* o contrato pode receber dados climáticos de fontes como o *Accuweather*, que são verificados e validados pelo *oracle*, permitindo a execução do contrato caso as condições sejam atendidas (CALDARELLI, 2020).

Figura 4 – Fluxo de um *smart contract* de seguro agrícola.

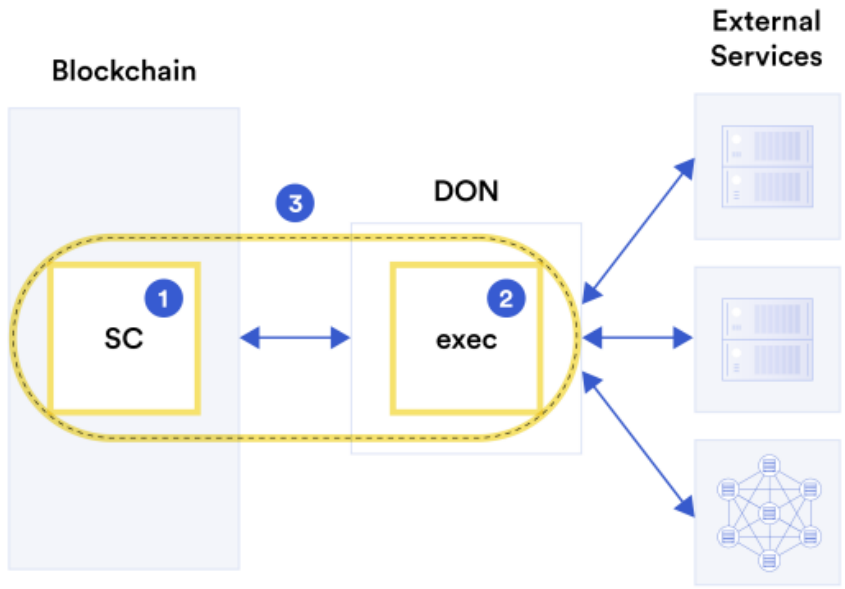


Fonte: O autor (2024)

A Figura 5 mostra que os *hybrid smart contracts* são compostos em duas partes: a pri-

meira é o *smart contract* (representado por SC), no qual roda em uma *blockchain* e a segunda parte em uma rede *oracle* descentralizada (representado por exec). Definimos como *oracle* qualquer dispositivo ou entidade que conecta a *blockchain* a dados externos. Essa tecnologia é responsável por verificar e validar os valores fornecidos a *blockchain* através de serviços externos, para que seja preservado o determinismo da rede (BREIDENBACHA *et al.*, 2021).

Figura 5 – Composição de um *Hybrid Smart Contract*.



Fonte: (BREIDENBACHA *et al.*, 2021)

É possível encontrar várias soluções de *oracles* disponíveis para uso, cada uma adotando diferentes abordagens, como *oracles* centralizados, descentralizados ou até mesmo a inserção manual de dados por seres humanos. A opção mais comum a ser utilizada são *oracles* centralizados, porém essa abordagem pode ser problemática (ELLIS; JUELS; NAZAROV, 2017). O uso de *oracles* centralizados anula as vantagens da descentralização, aumentando o risco à segurança do contrato. Em contratos de alto valor, por exemplo, a chance de serem comprometidos para benefício de agentes maliciosos é considerável (CALDARELLI, 2020).

Tendo em vista esse problema, surgiram as redes *oracles* descentralizadas (do inglês *Decentralized Oracle Network* (DON)). DONs são formadas por nodos de *oracles* que cooperam entre si para validar e completar determinado trabalho, tornando-se uma ferramenta poderosa e flexível para dar suporte a dados externos da *blockchain* (BREIDENBACHA *et al.*, 2021).

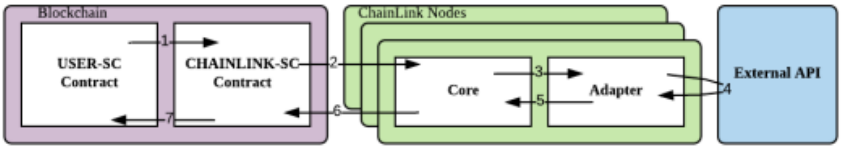
3.3 CHAINLINK

O Chainlink[2] é uma das soluções utilizadas para receber dados externos em um *smart contract*, ele consiste em uma rede descentralizada de *oracles* conectada ao Ethereum. Apesar

[2] <https://chain.link/>

de ter sido construído na rede Ethereum, ele é compatível com *smart contracts* implementados em outras redes *blockchain*. Como pode ser visto na Figura 6 os nodos do Chainlink são responsáveis por receber as requisições de *smart contracts* e possuem duas importantes estruturas: o Chainlink Core, representado na imagem como *core* e o External Adapter, representado na imagem como *adapter*. O Chainlink Core é quem coordena a comunicação entre os contratos inteligentes e os nós, ele recebe os eventos do contrato e encaminha a tarefa para o *adapter*. O External Adapter é o responsável por performar uma requisição e processar a resposta recebida da API, representada na imagem como *External API*, e passar essa informação de volta para o *core*. Após isso o Chainlink Core reporta o dado recebido de volta ao contrato (ELLIS; JUELS; NAZAROV, 2017).

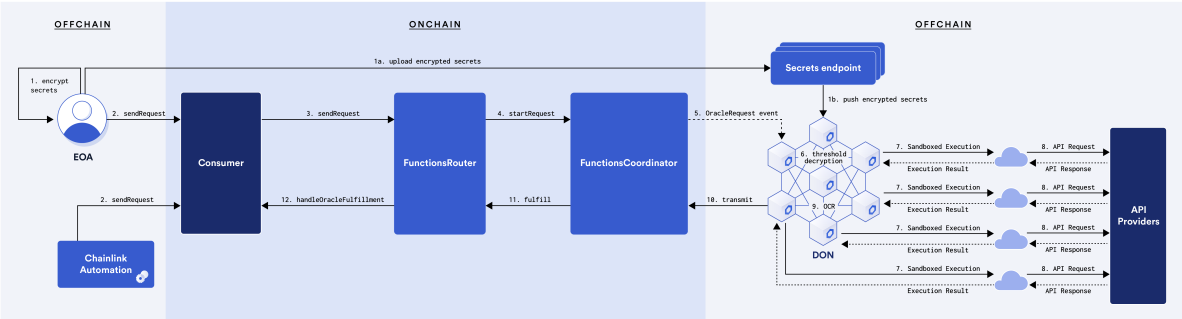
Figura 6 – Fluxo de trabalho do Chainlink.



Fonte: (ELLIS; JUELS; NAZAROV, 2017)

O Chainlink oferece duas abordagens para integração: a Chainlink Any API[3] e a Chainlink Functions. A abordagem Chainlink Functions usa dois *smart contracts* na rede Ethereum para estabelecer a comunicação com um DON. O primeiro *smart contract*, o *FunctionsRouter*, recebe as requisições do contrato e retorna os resultados obtidos, além de enviar as requisições ao segundo *smart contract*, o *FunctionsCoordinator*. Este último atua como uma interface para a DON, onde os *oracles* monitoram os eventos do *FunctionsCoordinator* e transmitem as respostas de volta assim que a execução é finalizada. A Figura 7 demonstra esse fluxo de comunicação.

Figura 7 – Fluxo Chainlink Functions.



Fonte: Disponível em: <https://docs.chain.link/chainlink-functions/resources/architecture>.

[3] https://docs.chain.link/any-api/introduction

Para integrar o Chainlink em um *smart contract*, é necessário importar o cliente Chainlink. No exemplo de código Solidity apresentado no Algoritmo 2, o contrato herda a biblioteca *FunctionsClient* e utiliza suas funcionalidades para buscar e armazenar dados externos. O construtor do contrato define o endereço do *router* para a *FunctionsClient*, especificando o endereço do *FunctionsRouter* — o *smart contract* responsável pelo envio das requisições conforme explicado anteriormente. Esse endereço varia de acordo com a *testnet* ou rede principal utilizada, sendo fundamental verificar o endereço correto para a rede desejada antes de implementá-lo.

O Chainlink Functions utiliza *Javascript* para realizar as requisições HTTP, sendo necessário escrever o código desejado nessa linguagem. No exemplo dado na variável *source* da Algoritmo 2, é mostrado como estruturar essa requisição. Além disso, é essencial registrar uma assinatura para a Chainlink Functions, onde serão adicionados os fundos em LINK, os quais permitirão que as requisições sejam processadas.

Para fazer uma requisição ao Chainlink, utiliza-se o método *_sendRequest()*, onde são fornecidos o Identificador (ID) do DON, o ID da assinatura da Chainlink Function, o limite de *gas* a ser utilizado, e o código em *Javascript* a ser executado. Após a requisição, o método *fulfillRequest* é automaticamente acionado, recebendo e registrando a resposta da operação realizada.

Algoritmo 2 – *Hybrid smart contract* utilizando Chainlink

```
1
2  pragma solidity 0.8.26;
3
4  import {FunctionsClient} from
5  "@chainlink/contracts@1.2.0/src/v0.8/functions/v1_0_0/FunctionsClient.sol";
6  import {FunctionsRequest} from
7  "@chainlink/contracts@1.2.0/src/v0.8/functions/v1_0_0/libraries/FunctionsRe
8  quest.sol";
9
10 contract exemplo is FunctionsClient {
11     using FunctionsRequest for FunctionsRequest.Request;
12
13     address router = 0xb83E47C2bC239B3bf370bc41e1459A34b41238D0;
14     bytes32 donID =
15     0x66756e2d657468657265756d2d7365706f6c69612d3100000000000000000000;
16         uint32 gasLimit = 300000;
17     string public dataReceived;
18
19     string source =
20         "const␣apiResponse␣=␣await␣Functions.makeHttpRequest({"
21         "url:␣`https://apiexemplo.com/cotacao/BTC-USDC`"
22         "});"
23         "if␣(apiResponse.error)␣{"
24         "throw␣Error('Request␣failed');"
```

```
25            "}"
26            "const␣{␣data␣}␣=␣apiResponse;"
27            "return␣Functions.encodeString(data.price);";
28
29        constructor() FunctionsClient(router) {}
30
31        function sendRequest(
32            uint64 subscriptionId
33        ) external returns (bytes32 requestId) {
34            FunctionsRequest.Request memory req;
35            req.initializeRequestForInlineJavaScript(source);
36
37            lastRequestId = _sendRequest(
38                req.encodeCBOR(),
39                subscriptionId,
40                gasLimit,
41                donID
42            );
43        }
44
45        function fulfillRequest(
46            bytes32 requestId,
47            bytes memory response,
48            bytes memory err
49        ) internal override {
50            dataReceived = string(response);
51        }
52    }
```

Fonte: O autor (2024).

4 PROPOSTA DE DESENVOLVIMENTO

O objetivo deste trabalho é propor um *smart contract* de arras para compra e venda de automóveis, modalidade popularmente conhecida como sinal de compra para o vendedor do veículo. Esse tipo de contrato é legalmente reconhecido pela jurisdição brasileira e tratado como uma compra com entrega futura, devendo seguir estritamente o que foi estabelecido e respeitar as leis vigentes (BRASIL, 2002).

Atualmente, os contratos de arras são feitos manualmente e dependem do comprometimento das partes envolvidas, o que pode resultar em disputas judiciais, demandando tempo e recursos de quem foi prejudicado. Já os *smart contracts*, por sua natureza programável, seguem rigorosamente o que foi definido em seu código, garantindo que tanto comprador quanto vendedor cumpram as condições estipuladas.

A proposta deste trabalho é utilizar a rede *blockchain* Ethereum para desenvolver e manter contratos do tipo arras, explorando sua aplicabilidade em transações de compra e venda de veículos. O Ethereum foi escolhido por ser uma das redes mais consolidadas e seguras, oferecendo um ambiente robusto para a criação de *smart contracts* por meio de ferramentas como a EVM. Essa rede possibilita a criação de contratos inteligentes automáticos e autoexecutáveis, eliminando intermediários e reduzindo o risco de inadimplência. Dessa forma, garante-se maior confiança entre as partes envolvidas, ao mesmo tempo em que se minimizam custos e o tempo necessário para resolver possíveis litígios, pois as regras contratuais são executadas automaticamente na rede, tornando o processo mais eficiente e confiável.

4.1 TECNOLOGIAS A SEREM UTILIZADAS

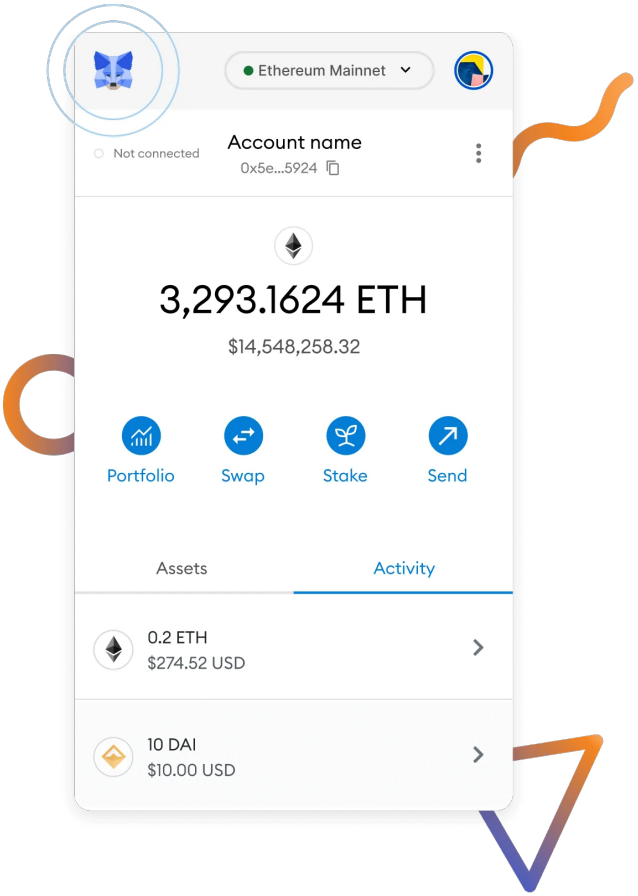
O código do *smart contract* será escrito em Solidity, uma linguagem criada especificamente para contratos inteligentes na rede Ethereum. O ambiente de desenvolvimento a ser utilizado será a Remix IDE[1], uma aplicação que pode ser utilizada em sua versão *web* ou *desktop*, construída especificamente para desenvolver *smart contracts* no Ethereum. O Chainlink será utilizado para fazer a comunicação entre o *smart contract* e os dados recebidos de fora da rede Ethereum. A escolha do Chainlink tem como objetivo assegurar que o contrato está recebendo um valor legítimo, através das validações feitas pela rede. Com essa validação, comprador e vendedor poderão ter a segurança de confiar no contrato e negociar tranquilamente.

As transações do contrato serão realizadas utilizando o Metamask[2]. Como demonstrado pela Figura 8, esse software é uma carteira onde são armazenadas criptomoedas. A escolha do Metamask se dá pelo fato de ser possível integrar um *smart contract* com a carteira, permi-

[1] https://remix-ide.readthedocs.io/en/latest/
[2] <https://docs.metamask.io/>

tindo que, quando for requisitada uma cobrança e o usuário confirmar, o valor seja debitado automaticamente para o destinatário correto. Será utilizado ETH como moeda de troca, pois o *blockchain* a ser utilizado será a rede do Ethereum. Esta rede foi selecionada por ter sido a primeira a implementar o conceito de *smart contract* e por possuir uma vasta documentação para implementação.

Figura 8 – Carteira Metamask.



Fonte: <https://metamask.io/institutions/>

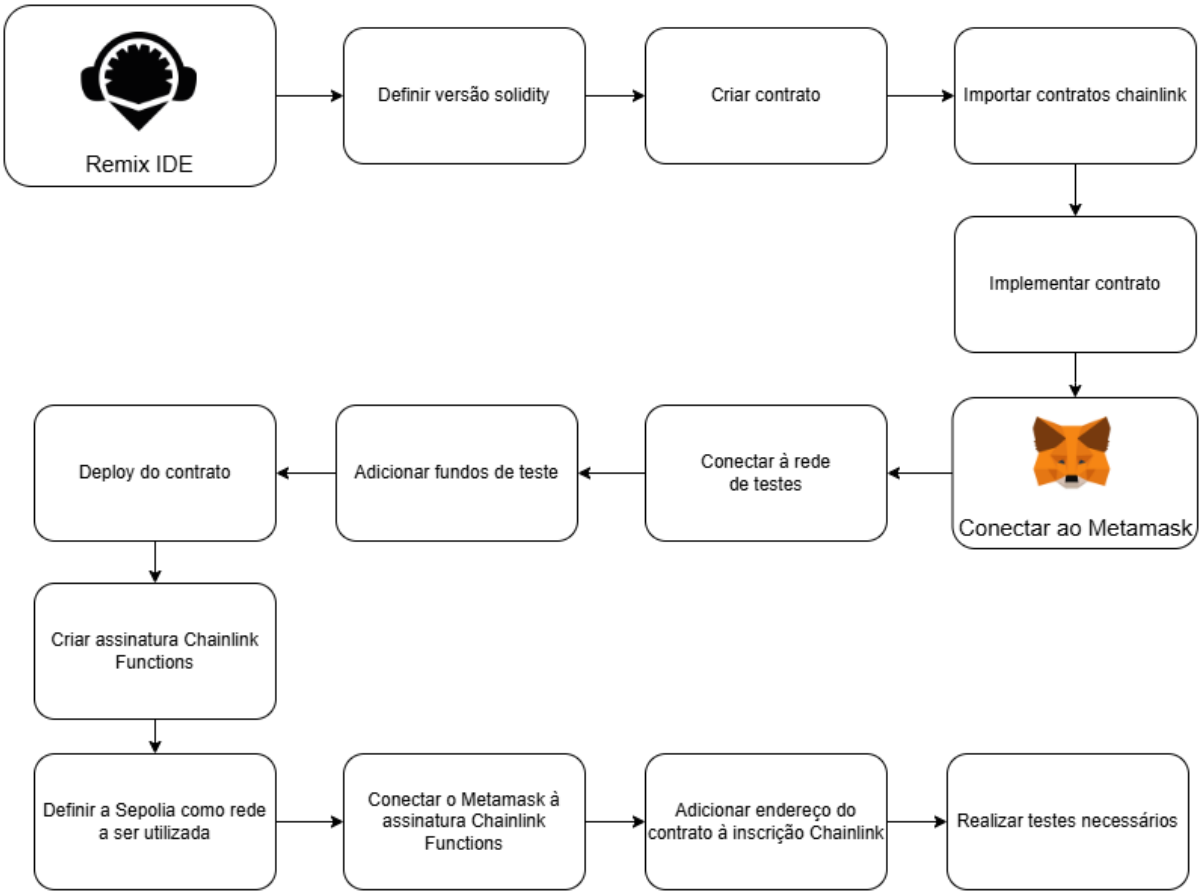
Para testar o contrato, será utilizada a rede Sepolia[3], uma *testnet* do Ethereum destinada exclusivamente para desenvolvimento e testes. *Testnets* como a Sepolia são redes de simulação onde não é necessário o uso de dinheiro real, facilitando a verificação e depuração do contrato antes de sua publicação oficial na rede Ethereum. O Chainlink também oferece suporte a *testnets* e será configurado para operar na mesma rede Sepolia do Ethereum. Além disso, o Metamask permite integração com a rede Sepolia, possibilitando a adição de fundos em ETH e LINK Tokens sem valor real na carteira. Dessa forma, o desenvolvimento e testes do contrato podem ser realizados sem custos adicionais, assegurando um ambiente seguro e realista para ajustes necessários.

[3] <https://sepoliafaucet.com/>

4.2 ETAPAS PARA DESENVOLVER UM *SMART CONTRACT*

O desenvolvimento de um novo *smart contract* com Chainlink pode ser realizado conforme os passos apresentados no fluxograma da Figura 9.

Figura 9 – Fluxograma de desenvolvimento.



Fonte: O autor (2024)

1. Primeiramente, ao criar um novo projeto no ambiente de desenvolvimento, é importante selecionar a versão da linguagem Solidity que será usada para compilar o contrato.

2. Após definir a versão, deve-se criar o arquivo onde o contrato será escrito, utilizando a extensão '.sol'. Esse arquivo deve começar especificando a versão do Solidity, conforme ilustrado no Algoritmo 3.

3. Definida a versão, procede-se com a importação da biblioteca do Chainlink, endereço do *router* e endereço do DON.

4. Em seguida, o código do contrato pode ser desenvolvido de acordo com as funcionalidades desejadas.

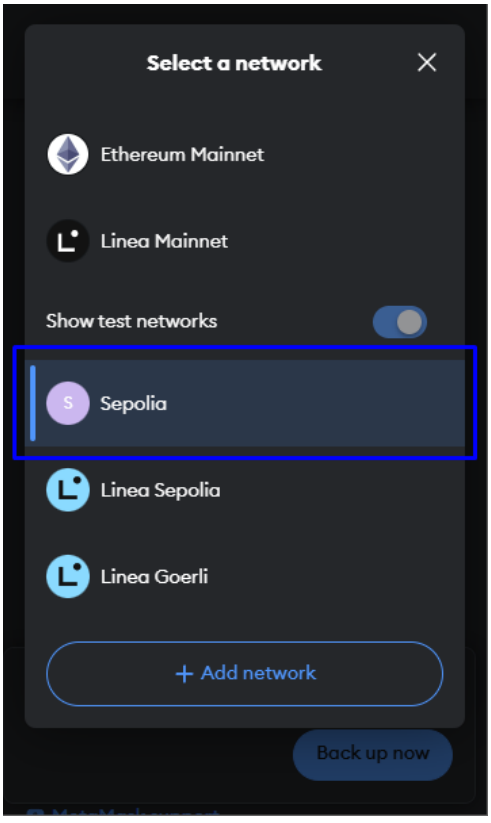
Algoritmo 3 – Estrutura de um *smart contract* escrito em Solidity

```
1
2  pragma solidity 0.8.26;
3
4  import {FunctionsClient} from
5  "@chainlink/contracts@1.2.0/src/v0.8/functions/v1_0_0/FunctionsCli
6  ent.sol";
7
8  import {FunctionsRequest} from
9  "@chainlink/contracts@1.2.0/src/v0.8/functions/v1_0_0/libraries/Fu
10 nctionsRequest.sol";
11
12 address router = 0xb83E47C2bC239B3bf370bc41e1459A34b41238D0;
13
14 bytes32 donId =
15 0x66756e2d657468657265756d2d7365706f6c69612d3100000000000000000000;
```

Fonte: O autor (2024).

5. Para publicar o contrato em uma *testnet*, é necessário utilizar uma carteira Ethereum. Conforme ilustrado no fluxograma da Figura 9, o MetaMask será utilizado como carteira. Como mostrado na Figura 10, para conectar o Metamask a uma rede de testes, basta exibir as redes de teste disponíveis e selecionar a rede Sepolia.

Figura 10 – Metamask conectado a *testnet* Sepolia.



Fonte: O autor (2024)

6. Com a carteira já conectada à rede de testes, é necessário adicionar fundos de teste em ETH e LINK para possibilitar a publicação do contrato. Esses fundos podem ser obtidos através dos Faucets. Por exemplo, ao utilizar o Chainlink Faucet[4], é preciso acessar o site, fazer login e informar o endereço da carteira para a qual os fundos devem ser enviados, conforme ilustrado na Figura 11.

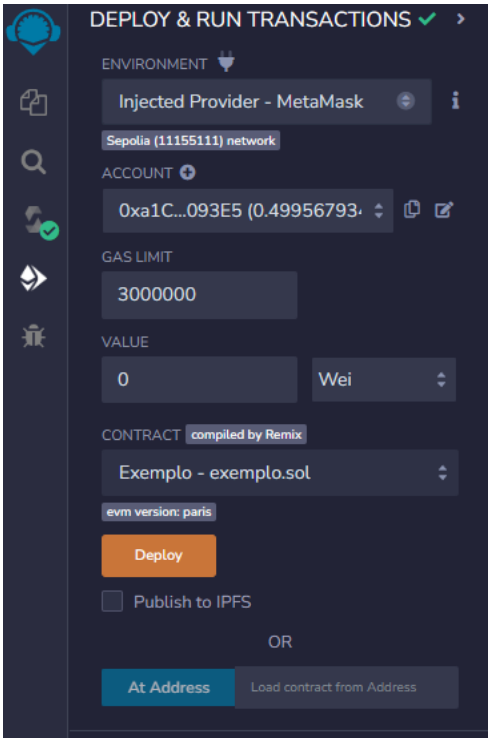
Figura 11 – Área para adicionar fundos a carteira no Chainlink.

Fonte: O autor (2024)

7. Em seguida, conforme ilustrado na Figura 12, deve-se selecionar a opção *Injected Provider* no campo *environment*. Essa configuração permite publicar o contrato na rede de testes Sepolia. Após essa etapa, na opção *contract*, é necessário selecionar o contrato que foi escrito e clicar em *deploy*. O Metamask exibirá então uma mensagem de confirmação para a cobrança da taxa de transação. Após a confirmação do pagamento, o contrato estará publicado na rede *blockchain* e pronto para ser testado.

[4] https://faucets.chain.link/sepolia

Figura 12 – Área de publicação do *smart contract*.



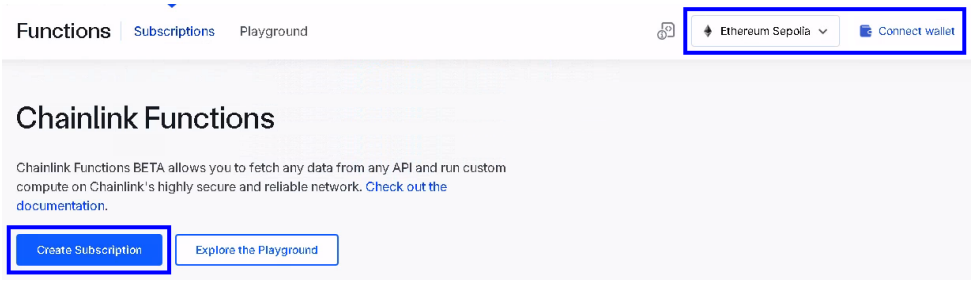
Fonte: O autor(2024)

8. Após a publicação do contrato, é necessário criar a assinatura do Chainlink Functions para possibilitar as requisições na rede, como ilustrado na Figura 13

9. Em sequência, deve-se acessar a página de assinatura e definir a rede Sepolia como rede à ser utilizada.

10. A próxima etapa envolve conectar a carteira contendo a conta de testes ao *website* e, em seguida, clicar em criar assinatura.

Figura 13 – Tela inicial das assinaturas Chainlink Functions.

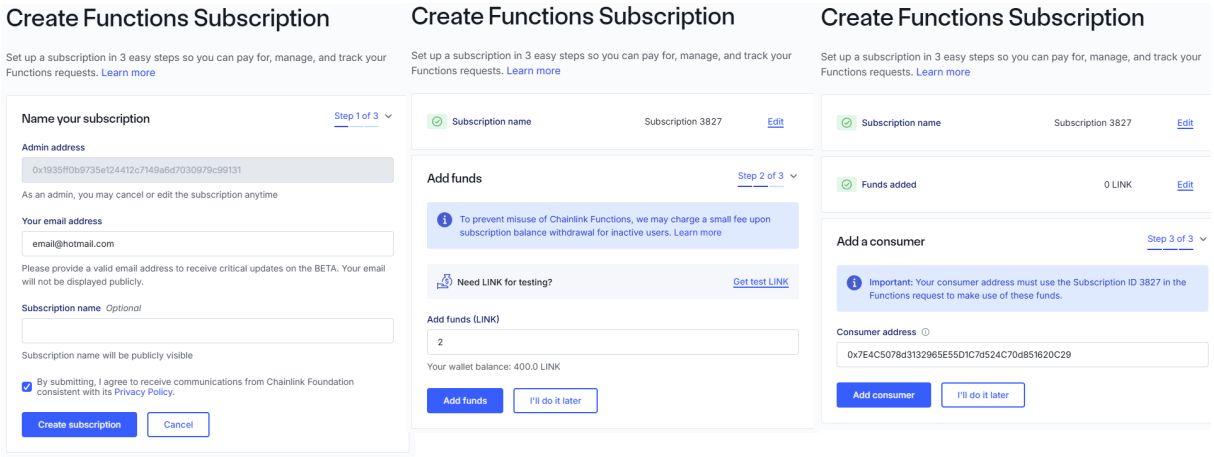


Fonte: O autor(2024)

11. Após concluir as etapas anteriores, o *website* apresentará três passos adicionais, conforme ilustrado na Figura 14. Nessa fase, o Chainlink solicitará a quantidade de fundos LINK

que se deseja adicionar e o endereço do contrato que utilizará a assinatura para realizar requisições na rede Chainlink.

Figura 14 – Passos para criar a assinatura Chainlink Functions.



Fonte: O autor(2024)

Com todas as etapas concluídas, o *smart contract* já está pronto para testes, permitindo validar suas funcionalidades diretamente na *testnet*. Caso seja necessário publicar uma nova versão do contrato, basta adicionar o endereço do contrato na assinatura existente e realizar novos testes conforme necessário.

5 ESTUDO DE CASO

Nesse capítulo será apresentada um estudo de caso da proposta de guia para criação de contrato inteligente apresentada no Capítulo 4. O contrato foi desenvolvido para que qualquer vendedor e comprador que desejarem negociar um automóvel possam utilizá-lo, garantindo segurança e confiança na transação graças à imutabilidade da *blockchain*. Esse contrato registrará todos os dados relevantes e as transações que estão sendo realizadas. Para iniciar a intenção de compra, é necessário fornecer as seguintes informações:

- Registro Nacional de Veículos Automotores (RENAVAM) do veículo a ser negociado.
- Placa do veículo a ser comprado.
- Número do Cadastro de Pessoa Física (CPF) do vendedor.
- Número do CPF do comprador.
- Valor de arras em ETH a ser paga pelo comprador.
- Valor total do veículo em ETH.
- Dias necessários para realizar o pagamento completo.

Após o vendedor informar os dados necessários, o contrato verificará, por meio do código do RENAVAM e da placa do veículo, se o automóvel já está em negociação. Caso não esteja, o processo de compra prosseguirá. Vale ressaltar que todo veículo registrado possui um número único de RENAVAM, que acompanha seu histórico desde a fabricação até a baixa no Detran (BRASIL, 1997). Após essa verificação, o comprador poderá transferir o valor das arras acordado, obtendo um período estipulado para realizar o pagamento restante do veículo.

Para confirmar a compra do veículo, o valor total de venda deve ser depositado, e o vendedor deverá transferir o veículo para o comprador até a data previamente acordada. Para esse processo, foi desenvolvida uma API que simula os dados fornecidos pela API oficial do Denatran[1]. A verificação é realizada com base nas informações do comprador, e a API retorna a confirmação sobre a transferência de propriedade do veículo. Assim, o contrato poderá validar se o veículo já está sob posse do comprador. Confirmada a transferência, o valor total da venda será liberado ao vendedor, junto com o depósito feito por ele inicialmente. Caso contrário, o comprador receberá de volta o valor integral pago, além das arras depositadas pelo vendedor como forma de indenização.

[1] <https://www.gov.br/conecta/catalogo/apis/wsdenatran>

5.1 LEIS SEGUIDAS

A construção desse *smart contract* foi feita com base nos artigos 417 e 420 do Código Civil Brasileiro, que objetivam a garantia de que um negócio venha a ser fechado (BRASIL, 2002). O artigo 417 do Código Civil dispõe:

> Art. 417. Se, por ocasião da conclusão do contrato, uma parte der à outra, a título de arras, dinheiro ou outro bem móvel, deverão as arras, em caso de execução, ser restituídas ou computadas na prestação devida, se do mesmo gênero da principal.

Em conformidade com o artigo mencionado, as arras podem ser pagas em dinheiro, e, neste caso, o *smart contract* armazenará o valor em ETH acordado entre as partes para garantir a intenção de compra. Ambas as partes devem depositar quantias equivalentes, visto que, em caso de descumprimento do contrato, o valor servirá como indenização para a parte prejudicada.

O contrato contará com o direito de arrependimento, para impossibilitar que uma indenização além do valor fixado seja pago. Seguindo o que o artigo 420 dispõe:

> Art. 420. Se no contrato for estipulado o direito de arrependimento para qualquer das partes, as arras ou sinal terão função unicamente indenizatória. Neste caso, quem as deu perdê-las-á em benefício da outra parte; e quem as recebeu devolvê-las-á, mais o equivalente. Em ambos os casos não haverá direito a indenização suplementar.

O *smart contract* irá entender como desistência da negociação qualquer ação tomada que vá contra o que foi determinado a fazer. Em caso de desistência do comprador, o valor depositado por ele ficará em posse do vendedor. Se a desistência ocorrer por parte do vendedor, o comprador terá seu valor estornado e receberá como indenização o valor depositado pelo vendedor para assegurar a veracidade da negociação.

5.2 API DENATRAN

Conforme mencionado, o *smart contract* realiza uma requisição para verificar se o veículo foi transferido para o comprador. Para simular esse serviço, que é pago, foi criada uma API em C# utilizando .NET 8.0, com banco de dados SQL Server 2022 para armazenar informações dos automóveis. Essa API replica o *endpoint* do Denatran, permitindo que o *smart contract* confirme a transferência de posse do veículo de forma segura e eficaz, como faria na consulta ao serviço oficial.

A API e o banco de dados foram hospedados em um notebook com processador Intel Core i3 380M, 6 *gigabytes* de *Random Access Memory* (RAM) e Ubuntu Server como sistema operacional. Para o Chainlink poder acessar a API, foi necessário expor o serviço à Internet. Por conta da rede utilizada para os testes utilizar um endereço de Protocolo da Internet (IP) público

dinâmico, foi utilizado o Cloudflare Tunnel[2], que garante que a API seja acessada externamente, mesmo em uma rede com IP dinâmico.

O *endpoint* criado para a API é do tipo *GET* e inclui três parâmetros obrigatórios em sua rota: CPF, placa e RENAVAM. O formato da *Uniform Resource Locator* (URL) é estruturado da seguinte maneira: "/v1/veiculos/proprietario/cpf/{cpf}/placa/{placa}/renavam/{renavam}".

Os dados retornados pela API são os mesmos armazenados no banco de dados. A API oficial do Denatran retorna inúmeras informação acerca do veículo, porém neste caso, irá apenas retornar os dados necessários para testes. A Tabela 2 mostra os campos utilizados.

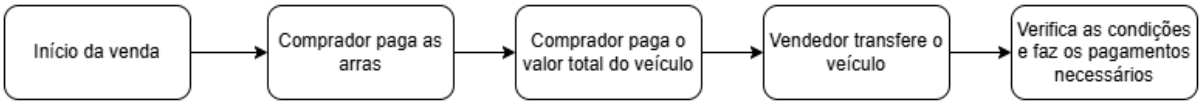
Tabela 2 – Campos armazenados no banco de dados

Campo	**Tipo**
ID	INT
Placa	NVARCHAR(7)
RENAVAM	NVARCHAR(11)
DocumentoProprietario	NVARCHAR(11)

5.3 ESTRUTURA DO *SMART CONTRACT*

O contrato desenvolvido segue como base o fluxograma ilustrado na Figura 15. Ele conta com variáveis globais, listadas na Tabela 3, que são utilizadas ao longo de sua execução. Implementado na versão 0.8.26 do Solidity, o contrato também utiliza a versão 0.8 do Chainlink Functions para funcionalidades adicionais. Embora todas as transações sejam realizadas em ETH, o Solidity utiliza apenas número inteiros. Assim, todos os valores monetários são expressos em Wei, a menor unidade do ETH, onde 1 ETH equivale a 10^{18} Wei.

Figura 15 – Fluxograma base de funcionamento do *smart contract*.



Fonte: O autor(2024)

[2] <https://www.cloudflare.com/pt-br/products/tunnel/>

Tabela 3 – Propriedades globais do contrato.

Campo	Tipo	Descrição
router	address	Endereço do *router* da rede Sepolia
donId	bytes32	Id do *don* da rede Sepolia
gasLimit	uint32	Valor de *gas* para fazer a requisição Chainlink
subscriptionId	uint64	ID da assinatura Chainlink Function
idUltimaRequisicao	bytes32	ID da última requisição feita à Chainlink
idContratoRequisicao	uint256	ID do contrato que foi feita a requisição
ContratoArras	struct	Contem as informações necessárias do contrato
contratosArras	ContratosArras[]	Vetor com todos os contratos sendo negociados
contratoEncontrado	bool	Informa se o veículo está em negociação

Tabela 4 – Propriedades da *struct* ContratoArras

Campo	Tipo
renavam	string
placa	string
carteiraComprador	address
carteiraVendedor	address
cpfComprador	string
cpfVendedor	string
valorArras	uint
valorTotal	uint
dataInicial	uint
prazoDias	uint
pago	bool

1. Para iniciar o contrato, será necessário utilizar a função *iniciarCompra*. Conforme apresentado no Algoritmo 4 é necessário alguns requisitos para que a negociação possa ser criada e adicionada a lista de contratos em andamento. Um detalhe importante dessa função é o uso da palavra reservada *payable*, que permite que as funções que a contenham possam receber valores em ETH. No contexto da função *iniciarCompra*, o vendedor realiza o pagamento das arras diretamente, garantindo assim o início do processo de negociação.

Algoritmo 4 – Função para iniciar a negociação

```
1  function iniciarCompra(string memory _renavam,
2      string memory _placa,
3      string memory _cpfComprador,
4      string memory _cpfVendedor,
5      uint _valorTotal,
6      uint _prazoDias) public payable {
7
8      require(msg.value > 0,
9      "Necessario fazer o pagamento das arras para iniciar");
10     require(_valorTotal > msg.value,
```

```
11      "O valor total precisa ser maior que o das arras");
12
13      require(
14      keccak256(bytes(_cpfComprador)) != keccak256(bytes(_cpfVendedor)),
15      "Comprador e vendedor precisam ter diferentes CPFs");
16
17      encontraContrato(_renavam, _placa);
18      require(!contratoEncontrado, "Veiculo ja em negociacao");
19
20      require(validaCpf(_cpfComprador));
21      require(validaCpf(_cpfVendedor));
22
23      ContratoArras memory contrato;
24      contrato.renavam = _renavam;
25      contrato.placa = _placa;
26      contrato.carteiraVendedor = msg.sender;
27      contrato.cpfComprador = _cpfComprador;
28      contrato.cpfVendedor = _cpfVendedor;
29      contrato.valorArras = msg.value;
30      contrato.valorTotal = _valorTotal;
31      contrato.pago = false;
32      contrato.prazoDias = block.timestamp + (_prazoDias*24*60*60);
33      contrato.dataInicial = block.timestamp;
34
35      contratosArras.push(contrato);
36  }
```

2. Iniciada a negociação, o comprador deve enviar o valor das arras ao contrato. Para isso, utiliza-se a função *enviarArrasComprador*, ilustrada no Algoritmo 5. Essa função verifica se a negociação já foi iniciada e cadastra a carteira do comprador na negociação.

Algoritmo 5 – Função para o pagamento das arras

```
1   function enviarArrasComprador(string memory _renavam,
2       string memory _placa) public payable {
3
4       uint i = encontraContrato(_renavam, _placa);
5
6       require(contratoEncontrado, "Contrato nao encontrado");
7       require(contratosArras[i].valorArras == msg.value,
8       "Valor das arras diferente do pago pelo vendedor");
9       require(contratosArras[i].carteiraComprador != msg.sender,
10      "Carteira ja utilizado para o vendedor");
11      require(contratosArras[i].carteiraComprador == 0x0000000000000000,
12      "Comprador ja informado");
13
14      contratosArras[i].carteiraComprador = msg.sender;
15  }
```

3. Em seguida, o comprador deve fazer o pagamento integral do veículo ao contrato, utilizando a função *pagarValorTotalComprador*. Nessa função, o comprador envia o valor restante do veículo. A função verificase a soma do valor enviado com as arras já depositadas é igual ao valor total do veículo, caso seja, o contrato constará como pago. O Algoritmo 6 mostra as verificações feitas pela função e atribuição de pagamento.

Algoritmo 6 – Função para o pagamento do veículo

```
1  function pagarValorTotalComprador(string memory _renavam,
2      string memory _placa) public payable {
3
4      uint i = encontraContrato(_renavam, _placa);
5
6      require(contratoEncontrado, "Contrato nao encontrado");
7      require(contratosArras[i].carteiraComprador == msg.sender,
8      "Carteira diferente da utilizada pelo comprador");
9
10     uint somaValores = msg.value + contratosArras[i].valorArras;
11     require(contratosArras[i].valorTotal == somaValores,
12     "Valor total diferente do informado");
13
14     contratosArras[i].pago = true;
15 }
```

4. Em seguida, o vendedor deve transferir o veículo para o comprador.

5. A verificação da negociação é realizada pela função *verificaContrato*, que verifica e executa as seguintes situações:

    - Caso o vendedor tenha iniciado a venda e o comprador não tenha pago as arras em até 24 horas, o valor é reembolsado ao vendedor.
    - Se a venda foi iniciada e as arras foram pagas pelo comprador, mas o pagamento total do veículo não foi efetuado até a data acordada, o vendedor recebe o valor depositado por ele, somado às arras enviadas pelo comprador, como indenização.
    - Caso o comprador tenha efetuado o pagamento, uma requisição é feita através do Chainlink para verificar se o veículo já foi transferido.

    Para fazer uma requisição para Chainlink Functions, é necessário escrever um código em *Javascript*. O Algoritmo 7 mostra a requisição feita para a API de testes e verifica se a consulta retorna algum valor. Se houver um valor de retorno, indica que o veículo foi transferido, retornando o código 200, caso contrário, o código 204 será retornado.

Em caso de erro na requisição, o código de erro 500 é retornado. Esses códigos foram definidos conforme os códigos de status *Hypertext Transfer Protocol* (HTTP)[3].

Algoritmo 7 – Requisição em *Javascript*

```
const apiResponse = await Functions.makeHttpRequest({
    url: `https://api.ricardoguimaraes.dev/v1/veiculos/proprietario/
    cpf/${cpf}/placa/${placa}/renavam/${renavam}`});

if (apiResponse.error)
        return Functions.encodeString('500');

const { data } = apiResponse;
const verificaTransferencia = data.renavam !== null ? '200' : '204';

return Functions.encodeString(verificaTransferencia);
```

Feita a requisição, o Chainlink Functions chama automaticamente a função *fulfillRequest*, que verifica, no caso de o veículo ainda não ter sido transferido e o prazo estipulado para a negociação ter se encerrado, transfere-se para o comprador o valor total do veículo, acrescido das arras transferidas pelo vendedor. Caso a negociação seja concluída, o valor do veículo somado às arras depositadas pelo vendedor é pago a ele. O Algoritmo 8 ilustra a implementação realizada.

Algoritmo 8 – Função chamada pelo Chainlink Functions

```
function fulfillRequest(
    bytes32 requestId,
    bytes memory response,
    bytes memory err) internal override {

    bytes memory codigoResponse = response;

    if(contratosArras[idContratoRequisicao].prazoDias < block.timestamp
    && keccak256(codigoResponse) == keccak256((bytes("204")))){
        uint valorReembolso =
        contratosArras[idContratoRequisicao].valorTotal +
        contratosArras[idContratoRequisicao].valorArras;

        (bool sent, bytes memory data) =
        contratosArras[idContratoRequisicao].carteiraComprador
            .call{value: valorReembolso}("");

        require(sent, "Erro ao fazer o pagamento");

        removeContrato(idContratoRequisicao);

```

[3] <https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status>

```
22          return;
23      }
24
25      if(keccak256(codigoResponse) == keccak256((bytes("200")))){
26          uint valorReembolso =
27          contratosArras[idContratoRequisicao].valorTotal +
28          contratosArras[idContratoRequisicao].valorArras;
29
30          (bool sent, bytes memory data) =
31          contratosArras[idContratoRequisicao].carteiraVendedor
32              .call{value: valorReembolso}("");
33
34          require(sent, "Erro ao fazer o pagamento");
35
36          removeContrato(idContratoRequisicao);
37      }
38  }
```

Além das funções mencionadas, o contrato inclui outras duas funções para a desistência da negociação: uma destinada ao vendedor e outra ao comprador. Ambas, quando acionadas, realizam o pagamento da indenização acordada à parte que não desistiu da negociação.

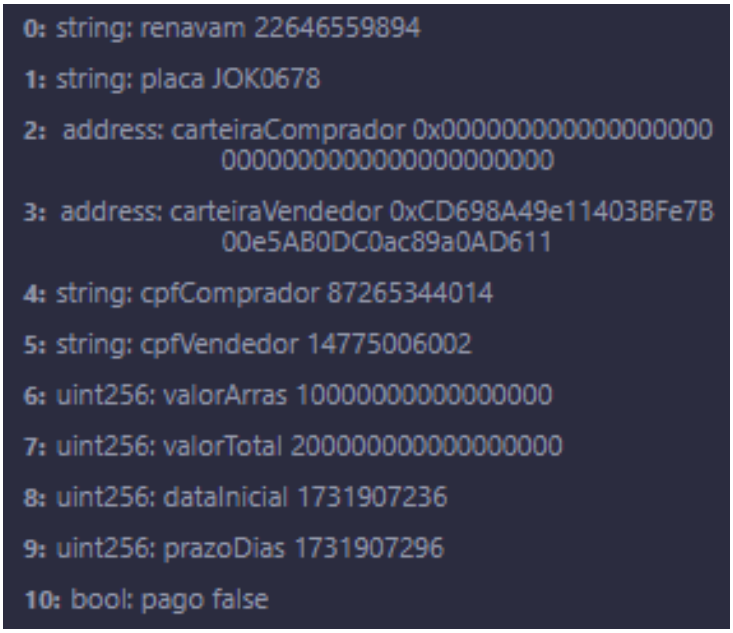
5.4 VALIDAÇÃO

A validação do estudo de caso será realizada por meio de cinco testes simulados, permitindo visualizar o funcionamento do *smart contract*. Para a execução dos testes, será utilizada a versão *web* do Remix IDE, com o navegador Microsoft Edge e o sistema operacional Windows 11 Pro. Os testes podere ser visualizados através do contrato publicado na rede Sepolia, utilizando seu endereço: "0xf27C13c645FaA00b4bc18C130fc03f51495FCCd8".

5.4.1 Cenário 1

O primeiro cenário será o de sucesso da negociação, no qual comprador e vendedor cumpriram todas as exigências. A Figura 16 apresenta os dados iniciais após o vendedor iniciar a compra. Os valores das arras e do total são armazenados em Wei. No entanto, o valor das arras corresponde a 0,01 ETH, enquanto o valor total é equivalente a 0,2 ETH.

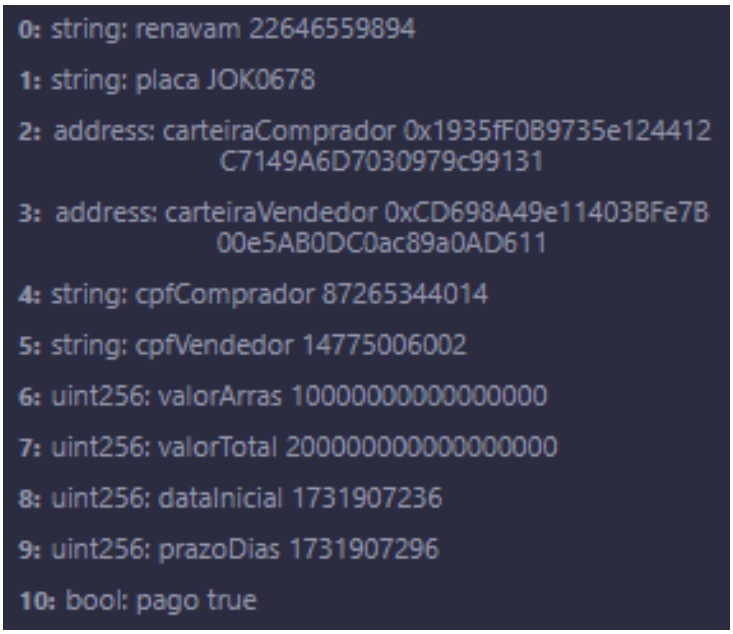
Figura 16 – Valores atribuídos a negociação do cenário 1.



Fonte: O autor(2024)

Após o comprador fazer o pagamento das arras e do valor total, a Figura 17 mostra duas alterações importantes: a primeira é a atribuição do endereço da carteira do comprador e a segunda é que a propriedade "pago"está como verdadeira.

Figura 17 – Valores das variáveis após o pagamento total e das arras.

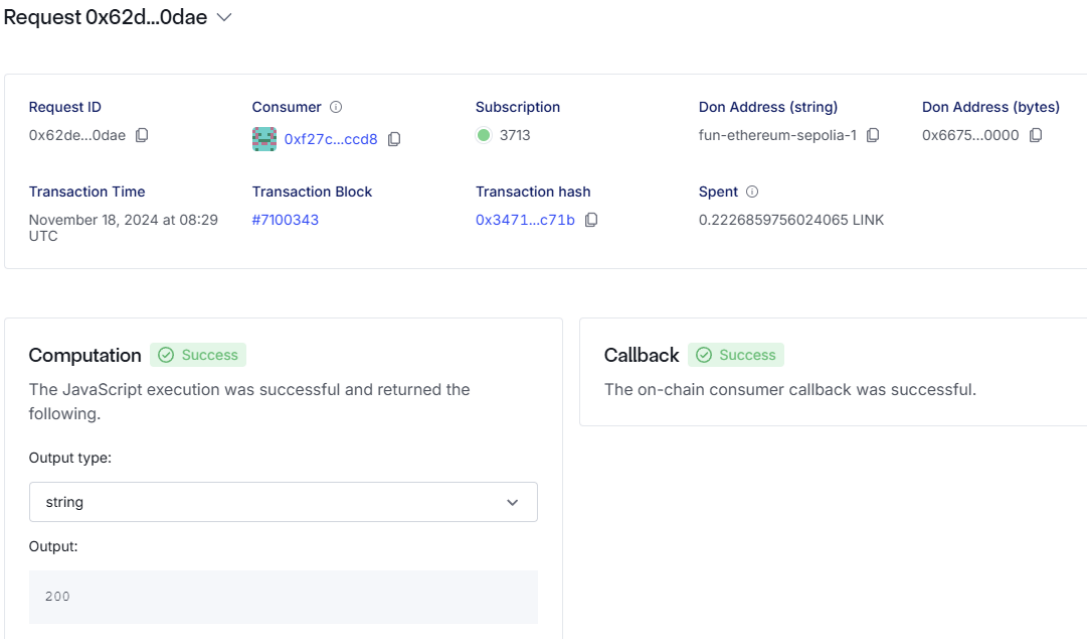


Fonte: O autor(2024)

Para simular a transferência, foram adicionadas ao banco de dados de testes as informações do veículo, juntamente com o CPF do comprador como proprietário. A Figura 18 ilustra a

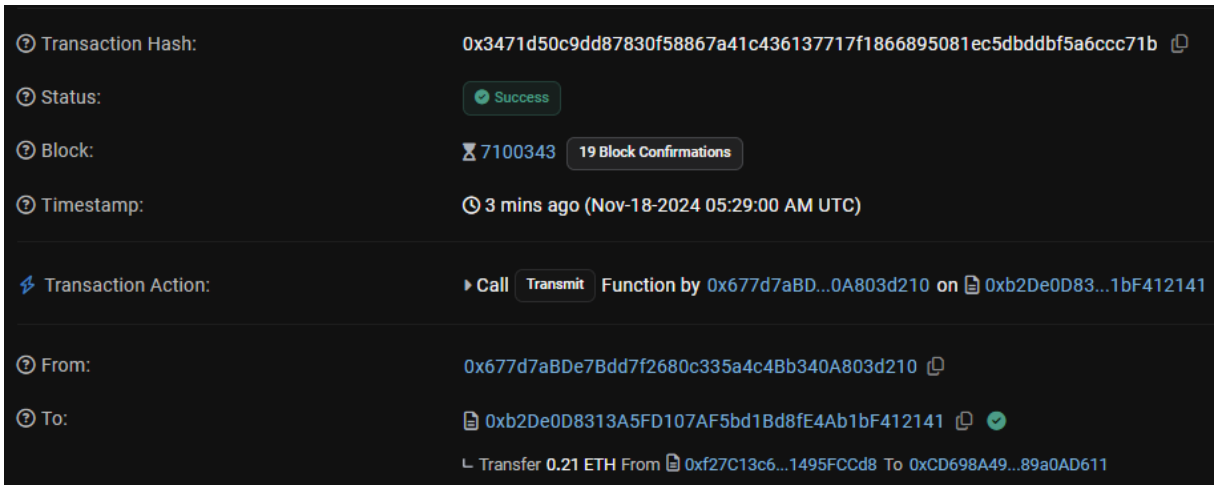
requisição realizada pelo Chainlink Functions e o retorno "200", indicando que a transferência foi concluída com sucesso. Também é possível visualizar na Figura 19 que o valor pago pelo comprador, somado às arras depositadas pelo vendedor, foi transferido para o vendedor.

Figura 18 – Detalhes do retorno da Chainlink Functions no cenário 1.



Fonte: <https://functions.chain.link/sepolia/3713#/side-drawer/request/0x62de76564e7972d84d190149be0e94ac77d8340d0a6bfb623c88e037e72c0dae>

Figura 19 – Detalhes da transação do cenário 1.



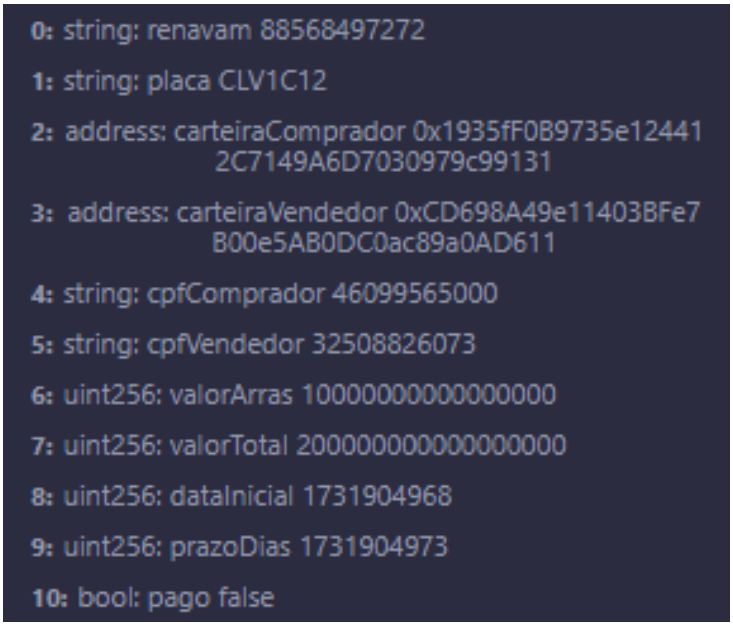
Fonte: <https://sepolia.etherscan.io/tx/0x3471d50c9dd87830f58867a41c436137717f1866895081ec5dbddbf5a6ccc71b>

5.4.2 Cenário 2

O segundo teste será de não pagamento do valor total do veículo por conta do pagador. A caráter de testes, o tempo de prazo utilizado foi de um minuto. A Figura 20 apresenta os

valores atribuídos a negociação. Além disso, é possível visualizar que a carteira do comprador está preenchida, indicando que o valor das arras foi devidamente pago.

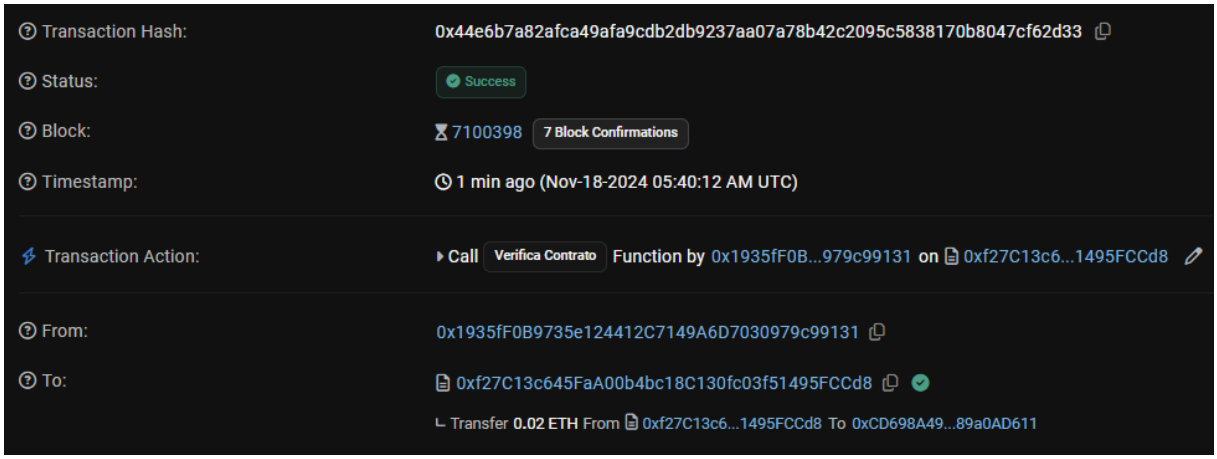
Figura 20 – Valores atribuídos a negociação do cenário 2.



Fonte: O autor(2024)

Após a verificação do estado atual da negociação, a Figura 21 evidencia que o pagamento foi efetuado corretamente, transferindo para a carteira do vendedor o valor das arras depositadas por ele somado ao valor das arras pagas pelo comprador.

Figura 21 – Detalhes da transação do cenário 2.

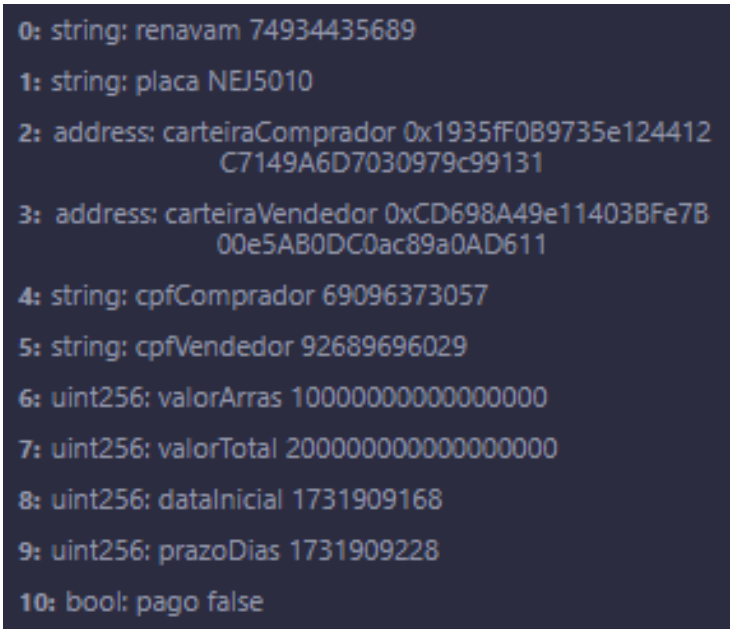


Fonte: <https://sepolia.etherscan.io/tx/0x44e6b7a82afca49afa9cdb2db9237aa07a78b42c2095c5838170b8047cf62d33>

5.4.3 Cenário 3

O terceiro teste simula a situação em que o vendedor não realiza a transferência do veículo. A Figura 22 mostra que o comprador fez o pagamento total do automóvel.

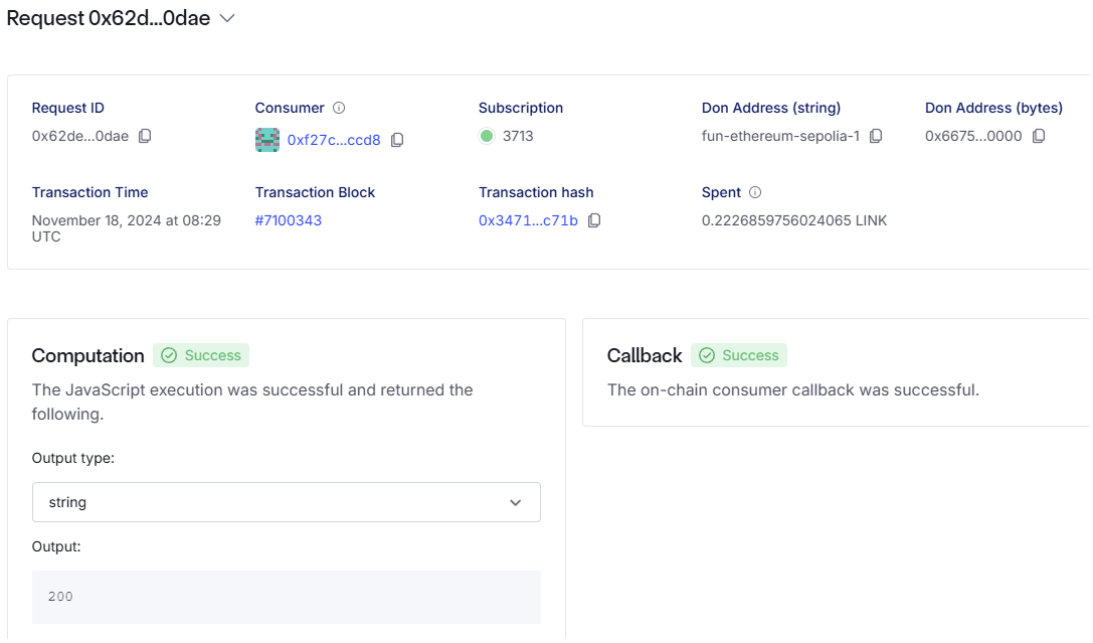
Figura 22 – Valores atribuídos a negociação do cenário 3.



Fonte: O autor(2024)

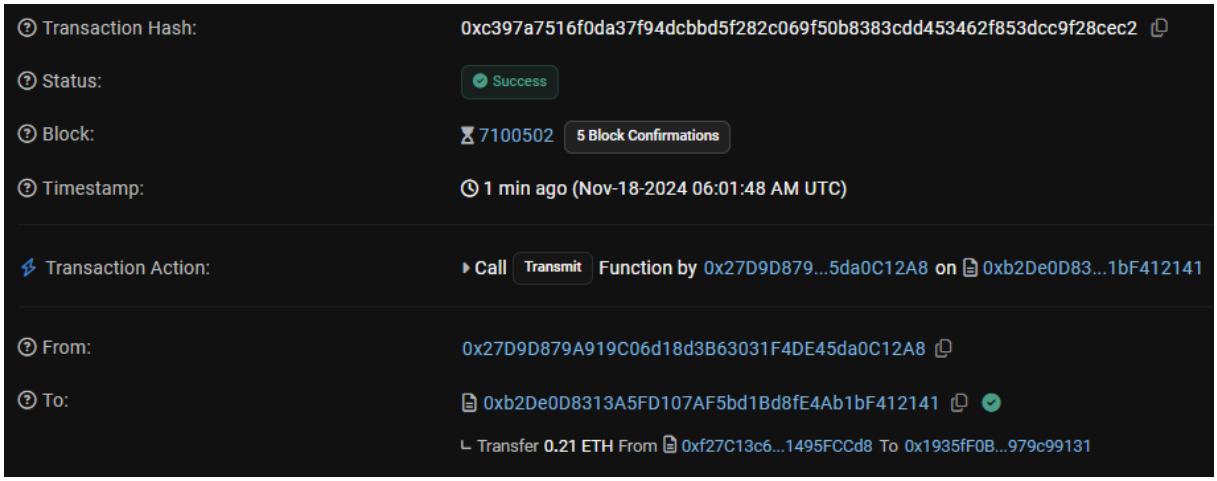
A Figura 23 mostra que após a verificação de que as obrigações do comprador foram cumpridas, porém após a verificação a API, foi constato que o veículo não foi transferido. Sendo assim, conforme ilustra a Figura 24 os valores pagos pelo comprador, somado as arras enviadas pelo vendedor, foram transferidas para ele.

Figura 23 – Detalhes do retorno da Chainlink Functions no cenário 3.



Fonte: <https://functions.chain.link/sepolia/3713#/side-drawer/request/0x9b76e85edfafcc1db9bacdb675f08fd56e5ffbcb053855596c2ac604929997fd>

Figura 24 – Detalhes da transação do cenário 3.

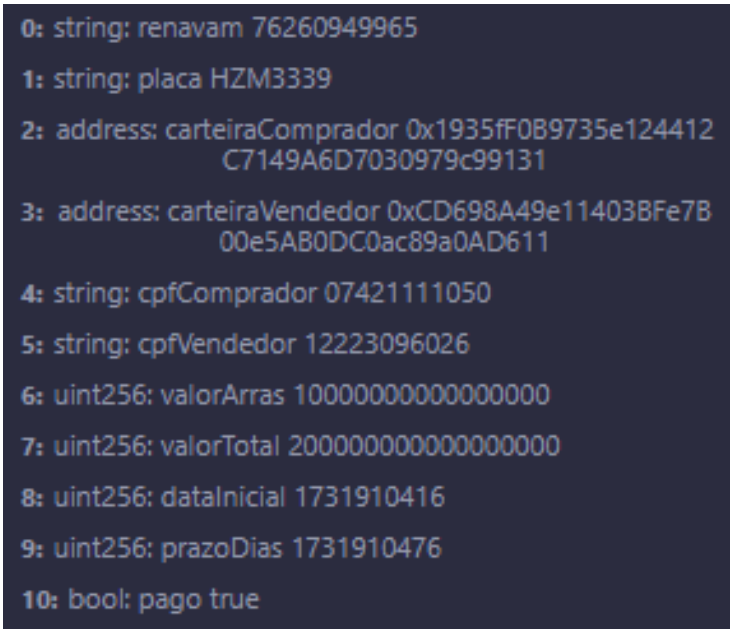


Fonte: <https://sepolia.etherscan.io/tx/0xc397a7516f0da37f94dcbbd5f282c069f50b8383cdd453462f853dcc9f28cec2>

5.4.4 Cenário 4

O quarto cenário simula o cancelamento da negociação por parte do comprador, após o mesmo já ter efetuado o pagamento do veículo. A Figura 25 mostra os valores utilizados para essa situação.

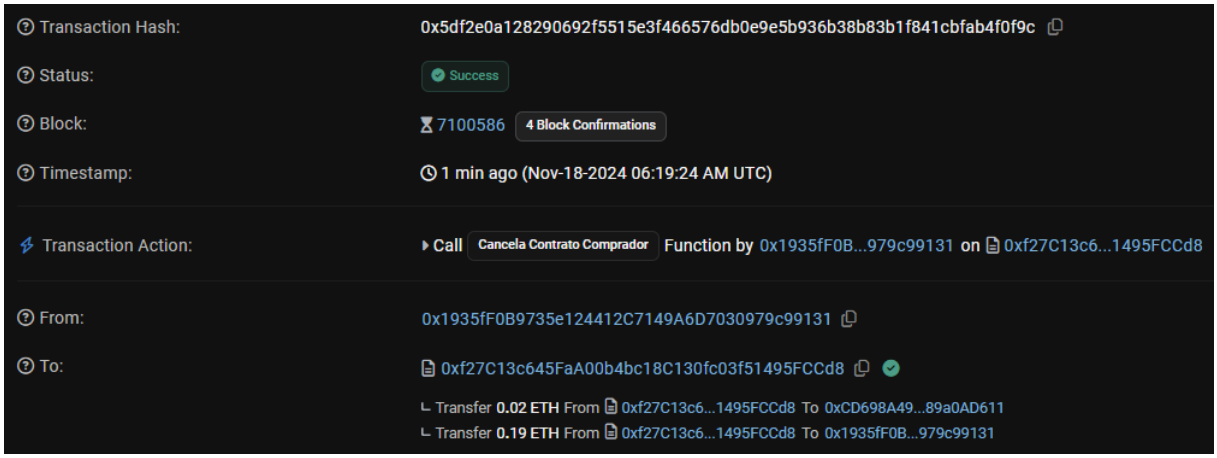
Figura 25 – Valores atribuídos a negociação do cenário 4.



Fonte: O autor(2024)

Nessa situação, conforme apresentado na Figura 26 o vendedor recebeu, como indenização, o valor das arras por ele depositadas somado às arras pagas pelo comprador. Em seguida, o restante do valor depositado pelo comprador foi devolvido à sua conta.

Figura 26 – Detalhes da transação do cenário 4.

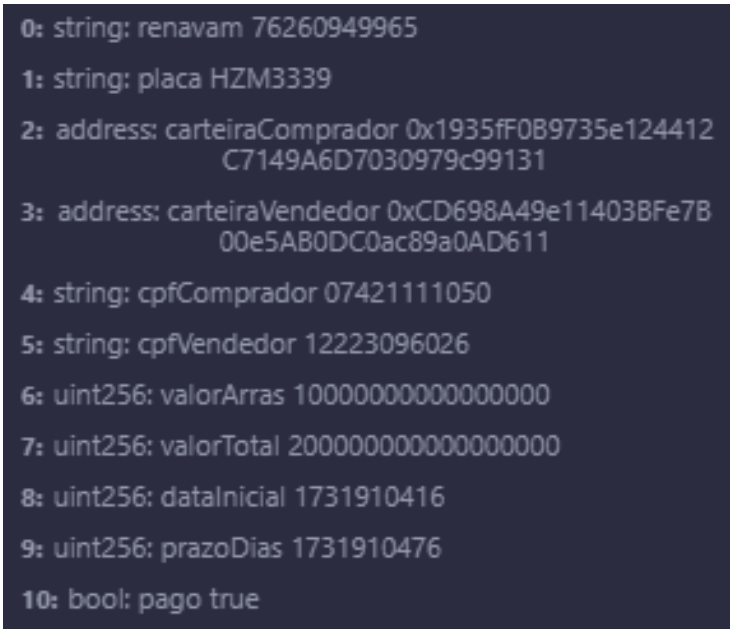


Fonte: <https://sepolia.etherscan.io/tx/0x5df2e0a128290692f5515e3f466576db0e9e5b936b38b83b1f841cbfab4f0f9c>

5.4.5 Cenário 5

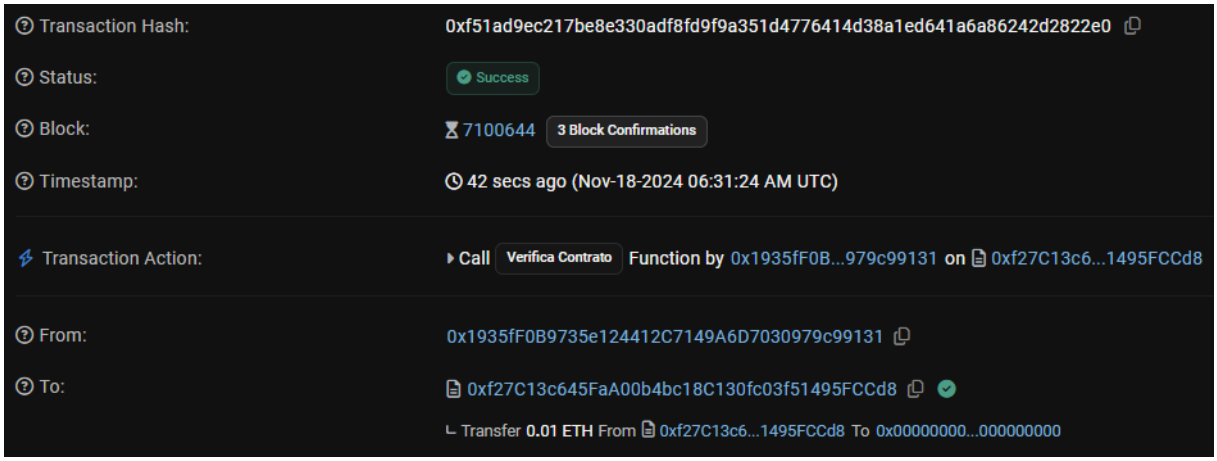
No quinto cenário, o vendedor iniciou a venda, mas, conforme ilustrado na Figura 27 o comprador não efetuou o pagamento das arras. Nesse caso, como mostrado na Figura 28, o valor das arras depositadas pelo vendedor foi reembolsado para sua conta.

Figura 27 – Valores atribuídos a negociação do cenário 5.



Fonte: O autor(2024)

Figura 28 – Detalhes da transação do cenário 5.



Fonte: <https://sepolia.etherscan.io/tx/0xf51ad9ec217be8e330adf8fd9f9a351d4776414d38a1ed641a6a86242d2822e0>

6 CONCLUSÕES

Este trabalho apresenta um estudo detalhado sobre blockchain, Ethereum e *smart contracts*. *Blockchain* é um sistema distribuído que tem no Ethereum e tecnologias associadas uma de suas implementações. O objetivo deste trabalho é apresentar uma proposta de um guia para automatizar a clausula de arras de um contrato de compra e venda de automóveis, utilizando um *smart contract* armazenado na rede do Ethereum. A utilização do *smart contract* torna o processo completamente descentralizado e remove burocracias necessárias no caso de descumprimento de alguma das partes que está negociando, mas garante que os termos do contrato serão executados. No trabalho são apresentadas as etapas para implementação do contrato no Ethereum, utilizando a linguagem Solidity e o Chainlink, e realizados testes para sua validação.

Um dos principais desafios enfrentados foi a escassez de materiais que abordem a aplicação prática de *smart contracts*. Os trabalhos existentes, em sua maioria, concentram-se nos aspectos teóricos e conceituais, sem explorar com profundidade os detalhes da implementação e a integração com ferramentas específicas. Este contexto torna o desenvolvimento refém da documentação oficial das tecnologias empregadas, que, embora robusta, nem sempre é clara ou suficiente para solucionar problemas específicos.

Outro ponto de destaque é a curva de aprendizado associada ao desenvolvimento de contratos inteligentes, especialmente no uso de ferramentas como Solidity, Chainlink e redes de testes como Sepolia.

Apesar das dificuldades, o estudo confirmou o potencial dos *smart contracts* em promover eficiência e confiança nas negociações, eliminando intermediários e assegurando a execução automatizada de cláusulas contratuais.

6.1 TRABALHOS FUTUROS

Neste trabalho, a execução das funcionalidades do contrato publicado foi realizada utilizando o Remix IDE. Para estudos futuros, recomenda-se o desenvolvimento de uma interface gráfica que permita a interação do usuário com o contrato. Essa interface pode ser implementada utilizando o Metamask SDK[1] em *Javascript*, facilitando a comunicação do usuário com o contrato já publicado em uma *blockchain*.

[1] <https://docs.metamask.io/wallet/connect/metamask-sdk/javascript/>

REFERÊNCIAS

BAG, S.; RUJ, S.; SAKURAI, K. Bitcoin block withholding attack: Analysis and mitigation. **IEEE Transactions on Information Forensics and Security**, v. 12, n. 8, p. 1967–1978, 2017. Doi:10.1109/TIFS.2016.2623588.

BAYER, D.; HABER, S.; STORNETTA, W. S. Improving the efficiency and reliability of digital time-stamping. In: CAPOCELLI, R.; SANTIS, A. D.; VACCARO, U. (Ed.). **Sequences II**. New York, NY: Springer New York, 1993. p. 329–334.

BRASIL. Lei nº 9.503, de 23 de setembro de 1997. código de trânsito brasileiro. **Diário Oficial da União**, Brasília, DF, set. 1997. Disponível em: <http://www.planalto.gov.br/ccivil_03/LEIS/L9503.htm>.

____. Lei nº 10.406, de 10 de janeiro de 2002. institui o código civil. **Diário Oficial da União**, Brasília, DF, jan. 2002. Disponível em: <https://www.planalto.gov.br/ccivil_03/leis/2002/l10406.htm>.

BREIDENBACHA, L. *et al.* Chainlink 2.0: Next steps in the evolution of decentralized oracle networks. abr. 2021. Disponível em: <https://research.chain.link/whitepaper-v2.pdf>.

BUTERIN, V. Ethereum whitepaper. 2014. Disponível em: <https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf>.

CALDARELLI, G. Understanding the blockchain oracle problem: A call for action. **Information**, v. 11, n. 11, 2020. ISSN 2078-2489. Doi:10.3390/info11110509. Disponível em: <https://www.mdpi.com/2078-2489/11/11/509>.

CASEY, M. The impact of blockchain technology on finance: A catalyst for change. In: **The impact of blockchain technology on finance : a catalyst for change**. Center for Economic Policy Research, 2018. p. 5. ISBN 9781912179152. Disponível em: <https://worldcat.org/title/1059331326>.

CHRISTIDIS, K.; DEVETSIKIOTIS, M. Blockchains and smart contracts for the internet of things. **IEEE Access**, v. 4, p. 1–1, 01 2016. Doi:10.1109/ACCESS.2016.2566339.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, G. B. T. **Sistemas distribuídos: conceitos e projeto**. 5. ed. Porto Alegre, RS: Bookman, 2013. 17–19;435–437 p.

ELLIS, S.; JUELS, A.; NAZAROV, S. Chainlink: A decentralized oracle network. set. 2017. Disponível em: <https://research.chain.link/whitepaper-v1.pdf>.

GADEKALLU, T. R. *et al.* **Blockchain for the Metaverse: A Review**. 2022. Disponível em: <https://arxiv.org/abs/2203.09738>.

GERRING, T. Cut and try: Building a dream. Ethereum.org, fev. 2016. Disponível em: <https://blog.ethereum.org/2016/02/09/cut-and-try-building-a-dream>.

GUEGAN, D. **Public Blockchain versus Private blockhain**. [S.l.], 2017. Disponível em: <https://ideas.repec.org/p/hal/cesptp/halshs-01524440.html>.

LAWTON, G. Top 9 blockchain platforms to consider in 2022. TechTarget, mar. 2022. Disponível em: <https://www.techtarget.com/searchcio/feature/Top-9-blockchain-platforms-to-consider>.

LUU, L. *et al.* Making smart contracts smarter. In: **Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security**. New York, NY, USA: Association for Computing Machinery, 2016. p. 254–269. ISBN 9781450341394. Doi:10.1145/2976749.2978309. Disponível em: <https://doi.org/10.1145/2976749.2978309>.

MOHAN, C. State of public and private blockchains: Myths and reality. In: **Proceedings of the 2019 International Conference on Management of Data**. New York, NY, USA: Association for Computing Machinery, 2019. (SIGMOD '19), p. 404–411. ISBN 9781450356435. Doi:10.1145/3299869.3314116. Disponível em: <https://doi.org/10.1145/3299869.3314116>.

NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system. maio 2008. Disponível em: <https://bitcoin.org/bitcoin.pdf>.

NARAYANAN, A. *et al.* Bitcoin and cryptocurrency technologies. **Network Security**, v. 2016, n. 8, p. 90–95, 2016. ISSN 1353-4858. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1353485816300745>.

R., B. Blockchain based service: A case study on ibm blockchain services hyperledger fabric. p. 2581–6942, 05 2020. Doi:10.5281/zenodo.3822411.

SARMAH, s. Understanding blockchain technology. v. 8, p. 23–29, 08 2018. Doi:10.5923/j.computer.20180802.02.

SYED, T. A. *et al.* A comparative analysis of blockchain architecture and its applications: Problems and recommendations. **IEEE Access**, v. 7, p. 176838–176869, 2019. Doi:10.1109/ACCESS.2019.2957660.

TANENBAUM, A. S.; STEEN, M. V. **Sistemas distribuídos: principios e paradigmas**. 2. ed. São Paulo, SP: Pearson Prentice Hall, 2008. 1–2 p.

TERRA, E. **Dicionário da língua portuguesa**. Rideel, 2011. 261 p. ISBN 9788533948648. Disponível em: <https://plataforma.bvirtual.com.br>.

YANG, R. *et al.* Public and private blockchain in construction business process and information integration. **Automation in Construction**, v. 118, 10 2020. Doi:10.1016/j.autcon.2020.103276.

ZHENG, Z. *et al.* An overview on smart contracts: Challenges, advances and platforms. **Future Generation Computer Systems**, v. 105, p. 475–491, 2020. ISSN 0167-739X. Doi:10.1016/j.future.2019.12.019. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167739X19316280>.

ZīLE, K.; STRAZDIņA, R. Blockchain use cases and their feasibility. **Applied Computer Systems**, v. 23, n. 1, p. 12–20, 2018. Disponível em: <https://doi.org/10.2478/acss-2018-0002>.