

ROLLUPS EN ETHEREUM

ETHEREUM ROLLUPS



TRABAJO FIN DE GRADO
CURSO 2024-2025

AUTOR
JAVIER JEREZ REINOSO

DIRECTOR
JESÚS CORREAS FERNÁNDEZ

GRADO EN INGENIERÍA INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

CALIFICACIÓN
10 (MATRÍCULA DE HONOR)

ROLLUPS EN ETHEREUM

ETHEREUM ROLLUPS

TRABAJO DE FIN DE GRADO EN INGENIERÍA INFORMÁTICA

AUTOR

JAVIER JEREZ REINOSO

DIRECTOR

JESÚS CORREAS FERNÁNDEZ

CONVOCATORIA: JUNIO 2025

GRADO EN INGENIERÍA INFORMÁTICA

FACULTAD DE INFORMÁTICA

UNIVERSIDAD COMPLUTENSE DE MADRID

FINAL GRADE

10 (WITH HONORS)

DEDICATION

A todos aquellos que se atrevieron

To all those who dared

ACKNOWLEDGEMENT

En primer lugar, quiero expresar mi más profunda gratitud a mi tutor, Jesús Correas Fernández, por todo el apoyo y la guía que me ha brindado a lo largo del proyecto. Gracias por su ayuda en los momentos de incertidumbre y por las reuniones de más de tres horas cuando la solución no estaba a la vista. Muchas gracias por su vocación y dedicación, no solo durante la realización del proyecto, sino también a lo largo del grado durante estos años.

Quiero agradecer también especialmente a mi familia y amigos por el apoyo recibido durante los quebraderos de cabeza, y en particular a aquellas personas que han estado a mi lado todos los días, de una forma u otra, ya sea cerca o en la distancia.

A todos, muchas gracias.

RESUMEN

ROLLUPS EN ETHEREUM

Este TFG compara diferentes soluciones que utilizan la capa 2 para la escalabilidad de la blockchain de Ethereum, como solución al *blockchain trilemma* planteado por V. Buterin: la dificultad de los sistemas de blockchain actuales para resolver simultáneamente los problemas de escalabilidad, descentralización y seguridad. En particular, se presenta la solución de los *optimistic rollups*, o rollups optimistas, como la alternativa más prometedora en la actualidad para incrementar la escalabilidad sin comprometer la descentralización ni la seguridad del sistema.

Para ello, se estudian los diferentes componentes que conforman esta solución, así como el flujo de información y ejecución entre ellos. Además, se presentan también otros elementos técnicos esenciales del sistema para su correcta operatividad.

Las principales funcionalidades de los rollups optimistas se explican detalladamente. Tanto su eficacia como sus beneficios se justifican mediante cálculos basados en datos reales. Adicionalmente, se presenta la red de pruebas *Optimism* que ha sido utilizada como entorno experimental para desarrollar distintos tipos de pruebas y así evaluar el rendimiento de los rollups optimistas en diferentes contextos.

Por último, los resultados obtenidos de los distintos experimentos se analizan debidamente y se comparan con datos de la red principal de Ethereum para demostrar el rendimiento de esta solución de capa 2.

Palabras clave

Ethereum, blockchain, rollup optimista, capa 2, escalabilidad, contrato inteligente, trilemma, gas, red de pruebas.

ABSTRACT

ETHEREUM ROLLUPS

This final degree project presents the different Layer 2 solutions that increase the scalability of the Ethereum blockchain, as a solution to the Blockchain Trilemma proposed by V. Buterin: the difficulty of current blockchain systems to simultaneously address the problems of scalability, decentralization and security. In particular, the *optimistic rollups* solution is presented as the current most promising alternative to increase the scalability without compromising decentralization nor security of the system.

To do so, different components that define this solution will be studied, as well as its flow of information and execution. Furthermore, other essential technical elements required by rollup systems to work properly will be presented.

The main functionalities of *optimistic rollups* are explained in detail. Both efficiency and benefits are justified through computations based on real data. Additionally, the *Optimism* testnet is introduced, which is the experimental environment used to carry out the different types of tests and thus to evaluate the performance of *optimistic rollups* in different contexts.

Finally, the results obtained from the different experiments are meticulously analyzed and compared with data from the Ethereum mainnet in order to demonstrate the performance of this Layer 2 solution.

Keywords

Ethereum, blockchain, optimistic rollup, Layer 2, scalability, smart contract, trilemma, gas, testnet

INDEX OF CONTENTS

Chapter 1.	Introduction.....	19
1.1	Motivation	19
1.2	Goals.....	19
1.3	Work schedule.....	20
Chapter 2.	Technology Introduction.....	23
2.1	General Background: What is a blockchain?	23
2.2	Ethereum	24
2.3	Blockchain Trilemma.....	25
2.3.1	Decentralization	25
2.3.2	Security.....	26
2.3.3	Scalability.....	26
2.4	The scalability problem	26
2.5	Layer 2 solutions	28
2.5.1	State Channels	28
2.5.2	Plasma Chains	30
2.5.3	Optimistic rollups	31
2.5.4	Zero-knowledge Rollups.....	31
2.6	Conclusions	32
Chapter 3.	Optimistic rollups.....	35
3.1	Topology overview	35
3.1.1	User	37
3.1.2	Bridge / Layer 1 smart contract	37

3.1.3 Sequencer	37
3.1.4 Layer 1 Contracts	38
3.1.5 Operator	39
3.1.6 Fraud Provers	41
3.1.7 Asserter.....	41
3.1.8 Liquidity Provider (LP)	41
3.2 Full Operation Flow	42
3.2.1 Entering the rollup	42
3.2.2 Relation between Operator and Sequencer	42
3.2.3 Exiting the rollup	44
3.2.4 Verification: The “Optimistic” Assumption.....	44
3.2.5 Challenge Period	44
3.2.6 Finalization and return.....	46
3.3 Additional Concepts and Technical Considerations	46
3.3.1 Batch structure and contents	46
3.3.2 Calldata vs Blobs.....	47
3.3.3 Merkle Trees and State Commitments	48
3.3.4 Honest node assumption.....	51
3.4 Performance benefits and metrics.....	51
3.4.1 Increased throughput (TPS)	51
3.4.2 Efficient data storage	53
3.4.3 Merkle tree compression	55
3.4.4 Fee payments.....	56
Chapter 4. Installation Environment	59
4.1 Main architecture differences	59

4.1.1 Op-geth	60
4.1.2 Op-node	60
4.1.3 Op-batcher.....	60
4.1.4 Op-deployer	60
4.1.5 Op-proposer	60
4.1.6 Other components.....	60
4.2 Implementation in Go of key components	61
4.2.1 Optimism Sequencer	62
4.3 OP Stack bridge contract	64
Chapter 5. Test Environment and Evaluation	67
5.1 Preparation.....	68
5.2 Plain transactions	71
5.2.1 Sepolia Layer 1	71
5.2.2 Optimism testnet	73
5.2.3 Arbitrum Sepolia	76
5.2.4 Results.....	79
5.3 Smart contract deployment with multiple plain transactions.....	82
5.3.1 Experimental smart contract.....	82
5.3.2 Sepolia Layer 1	83
5.3.3 Optimism testnet	87
5.3.4 Comparative Analysis	91
5.4 Other interesting use cases	93
Chapter 6. Conclusions and future work	95
6.1 Analysis of results	95
6.2 Problems and issues	96

6.2.1 Ganache	96
6.2.2 OP Stack installation.....	97
6.2.3 Arbitrum Sepolia testnet	99
6.2.4 Final deduction	100
6.3 Future Work.....	101

INDEX OF FIGURES

FIGURE 2.1: TOTAL GAS USED PER BLOCK OVER TIME [11]	27
FIGURE 2.2: STATE CHANNEL REPRESENTATION DIAGRAM [16]	29
FIGURE 2.3: EXAMPLE OF A PLASMA TREE STRUCTURE [18]	30
FIGURE 2.4: TOTAL VALUE LOCKED (IN USD) COMPARISON BETWEEN OPTIMISTIC AND ZK ROLLUPS IN JUNE 2024 [21]	33
FIGURE 3.1: REPRESENTATION OF THE OPTIMISTIC ROLLUP ARCHITECTURE	36
FIGURE 3.2: REPRESENTATION OF A MERKLE TREE [22]	48
FIGURE 3.3: AN EXAMPLE OF A MERKLE PROOF [22]	49
FIGURE 3.4: AN EXAMPLE OF A NODE MISCALCULATION IN A MERKLE TREE [21]	50
FIGURE 3.5: COMPARISON OF DATA GENERATED BY THE SAME TRANSACTION BETWEEN LAYER 1 AND LAYER 2 [19]	52
FIGURE 3.6: THIS GRAPH SHOWS THE DIFFERENCES BETWEEN BLOB AND CALldata CONSUMPTION DURING APRIL 2025 [25]	54
FIGURE 3.7: TRANSACTION FEE DIFFERENCES BETWEEN SEVERAL BLOCKCHAIN NETWORKS ON APRIL 30, 2025, INCLUDING FEES FOR ETHEREUM MAINNET TRANSACTIONS [29]	57
FIGURE 4.1: OPTIMISM SEPOLIA TESTNET ARCHITECTURE REPRESENTATION [30]	59
FIGURE 4.2: PROGRAMMING LANGUAGES USED IN THE OPTIMISM REPOSITORY [32]	61
FIGURE 4.3: A CODE EXCERPT FROM CHANNEL.GO THAT DEFINES THE STRUCTURE OF AN OP STACK "CHANNEL" [33]	63
FIGURE 4.4: A CODE EXCERPT FROM DRIVER.GO THAT DEFINES THE OP STACK "PIPELINE" [34]	63
FIGURE 4.5: A CODE SNAPSHOT FROM L1STANDARDBRIDGE.SOL SMART CONTRACT THAT DEFINES DEPOSITETHTO() METHOD [35]	64
FIGURE 4.6: A CODE SNAPSHOT FROM L1STANDARDBRIDGE.SOL SMART CONTRACT THAT DEFINES ETHDEPOSITINITIATED() EVENT [35]	65
FIGURE 5.1: ETHEREUM MAINNET GAS PRICE METRICS [42]	69
FIGURE 5.2: OPTIMISM MAINNET GAS PRICE METRICS [43]	69
FIGURE 5.3: ARBITRUM MAINNET GAS PRICE METRICS [44]	70
FIGURE 5.4: TRANSACTION INFORMATION FROM A LAYER 1 TO LAYER 1 TRANSACTION THROUGH METAMASK	72
FIGURE 5.5: AN ETHERSCAN ANALYSIS OF A DIRECT TRANSACTION ON THE SEPOLIA TESTNET [46]	73

FIGURE 5.6: A PLAIN TRANSACTION FROM LAYER 1 MAINNET TO THE OPTIMISM BRIDGE CONTRACT (LEFT) AND TRANSACTION DETAILS FROM A TRANSACTION TO THE OPTIMISM BRIDGE CONTRACT (RIGHT) ..	74
FIGURE 5.7: AN ETH TRANSFER TO A SECOND ADDRESS IN THE OP SEPOLIA TESTNET (LEFT) AND DETAILS OF A TRANSACTION TO THE OPTIMISM BRIDGE CONTRACT (RIGHT)	75
FIGURE 5.8: AN ETHERSCAN ANALYSIS OF A DIRECT TRANSACTION ON THE OP SEPOLIA TESTNET	76
FIGURE 5.9: TRANSACTION DETAILS FROM A TRANSACTION TO THE ARBITRUM BRIDGE CONTRACT (LEFT) AND THE ARBITRUM BRIDGE CONTRACT USER INTERFACE (RIGHT)	77
FIGURE 5.10: TRANSACTION REQUEST TO WITHDRAW FROM ARBITRUM SEPOLIA TESTNET THROUGH METAMASK (LEFT) AND ARBITRUM BRIDGE CONTRACT TRANSACTION FROM LAYER 2 ARBITRUM SEPOLIA TESTNET TO LAYER 1 SEPOLIA TESTNET (RIGHT)	78
FIGURE 5.11: SECOND TRANSACTION WITHDRAW MADE LAYER 1 SEPOLIA TESTNET THROUGH METAMASK (LEFT) AND ARBITRUM BRIDGE CONTRACT CONFIRMATION OF WITHDRAW (RIGHT)	78
FIGURE 5.12: SMART CONTRACT FOR SENDING MULTIPLE TRANSACTIONS	82
FIGURE 5.13: DEPLOYMENT COSTS OF A SMART CONTRACT ON SEPOLIA TESTNET FROM METAMASK	83
FIGURE 5.14: AN ETHERSCAN SNAPSHOT OF THE EXECUTION OF A SINGLE TRANSACTION SMART CONTRACT ON SEPOLIA TESTNET	84
FIGURE 5.15: AN ETHERSCAN SNAPSHOT OF THE EXECUTION OF A THOUSAND TRANSACTIONS SMART CONTRACT ON SEPOLIA TESTNET	85
FIGURE 5.16: DEPLOYMENT COSTS OF A SMART CONTRACT ON OPTIMISM SEPOLIA FROM METAMASK	87
FIGURE 5.17: AN ETHERSCAN SNAPSHOT OF THE EXECUTION OF A SINGLE TRANSACTION SMART CONTRACT ON OPTIMISM SEPOLIA TESTNET	88
FIGURE 5.18: DEPLOYMENT COSTS OF A SMART CONTRACT FOR A THOUSAND TRANSACTIONS ON OPTIMISM SEPOLIA FROM METAMASK	89
FIGURE 5.19: AN ETHERSCAN SNAPSHOT OF THE EXECUTION OF A THOUSAND TRANSACTIONS SMART CONTRACT ON OPTIMISM SEPOLIA TESTNET	90
FIGURE 5.20: A SNAPSHOT OF A SMART CONTRACT DEPLOYED ON LAYER 2 WITH AN IMPLEMENTED RECEIVE() METHOD.....	93
FIGURE 5.21: A SNAPSHOT OF THE REMIX ENVIRONMENT EXECUTING THE DEPOSITETHTo() INTERFACE [49]	94
FIGURE 6.1: WARNING MESSAGE SHOWN AT THE BEGINNING OF THE OPTIMISM TUTORIAL.....	97
FIGURE 6.2: ERROR FOUND IN FILE 'HEX-STRINGS.TS', WHICH HAD TO BE MODIFIED TO BE SOLVED	98
FIGURE 6.3: A SNAPSHOT OF A GITHUB DISCUSSION TRYING TO SOLVE THE OP-NODE ISSUE	99

FIGURE 6.4: AN ERROR OCCURS WHEN TRYING TO CONNECT METAMASK TO THE ARBITRUM SEPOLIA RPC	
.....	100

INDEX OF TABLES

TABLE 5.1: CALCULATIONS METRICS AND DATA AT DATE MAY 3, 2025.....71

TABLE 5.2: COSTS DIFFERENCES WHEN EXECUTING A THOUSAND TRANSACTIONS THROUGH A SMART
CONTRACT ON LAYER 1 AND ON LAYER 292

Chapter 1. Introduction

1.1 Motivation

Every blockchain system should maximize three fundamental characteristics: decentralization, security and scalability. This is known as the Scalability Trilemma since it was coined by Vitalik Buterin in 2017 [1]. Current systems mainly focus on the first two aspects due to its difficulty to maximize the three properties at the same time, thus scalability is the weak part of these systems. Moreover, as networks usage increases, blockchains like Ethereum are at their maximum technological capacity of transactions made per second.

Therefore, a solution that allows an increase in network performance without sacrificing other fundamental properties is urgently needed. To do so, after some previous proposals achieved limited success, *Optimistic Rollups* appeared to be the best current solution to solve this problem.

1.2 Goals

The main goal of this project is to deeply understand how *optimistic rollups* work, from both conceptual and practical point of view, as well as understanding other alternative solutions to the scalability problem in Ethereum.

We will study the different components of the optimistic rollup architecture and how they interact with each other. To do so, it is important to understand the differences of blobs and calldata and empirically justify how these new systems increase the overall performance of Ethereum transactions.

An experimental study that tests different real-world use cases, including the use of smart contracts, will be conducted in order to demonstrate the efficiency and scalability of the solution, explaining the key parts of the environment architecture used to perform the experiments.

This project also aims to broaden the knowledge acquired throughout the degree, applying it to understand complex and emerging technologies in the blockchain space, which is an appealing relevant area of interest for my professional path.

1.3 Work schedule

During the first month, on September, I learnt about the main Layer 2 scalability solutions. As an introduction, I followed a seminal work posted by Vitalik Buterin [2] in order to understand the basis of the possible solutions. Every week I scheduled a meeting with my tutor in order to discuss the functioning and discoveries of the topology of each solution, writing down the key properties of each solution with their corresponding explanation of the system.

During October and November, I focused on deeply understanding the fundamental concepts of *optimistic rollups* while I searched for free and public faucets where I could test these solutions so as to compare them and verify their efficiency as far as scalability is concerned. Unfortunately, no faucets were found so we decided to build our own environment with a tool called *ganache* to build a local blockchain to perform the tests.

On December and January, I started writing down the key aspects of *optimistic rollups*, creating the first diagrams of the complete structure and understanding how the environment should be created to meet all the protocols required.

On February, we continued having meetings every week, but we discovered that the *ganache* solution was not possible because I could not communicate two separate *ganache* blockchains at the same time and the *optimistic rollup* protocol was too complex to be replicated from scratch.

After doing more research, we discovered at the beginning of March a tutorial from the *Optimism* official web page that explained how to install your own rollup locally to perform tests using their repository. I dedicated March to explain in detail the components of the *optimistic rollup* topology and how they communicated meanwhile

I did the necessary installations of virtual machines, dependencies and components of the tutorial.

However, at the end of the tutorial some errors happened that could not be solved, I discussed them in github Q&A of the repository but no solution to the problem was found. Therefore, we decided to study optimistic rollups using several public testnets, namely *Sepolia* for Layer 1, and *Optimism* and *Arbitrum Sepolia testnets* for Layer 2. The problems found in setting up a local testnet brought the opportunity to perform the tests in real public networks, that resulted in a more realistic scenario than the one we initially thought.

During April and May, I spent all the remaining time writing the thesis report and documenting all the information discovered from the experiments, obtaining screenshots and collecting and creating diagrams that helped me explain *optimistic rollups* as well as the other Layer 2 solutions. I also justified the benefits of *optimistic rollups* through calculations based on the collected results and data.

Chapter 2. Technology Introduction

In this first section, some key concepts are introduced for a better understanding of the topic, including the essential factors of the architecture as well as the main problems and issues that this system can encounter and the corresponding solutions.

2.1 General Background: What is a blockchain?

A blockchain is decentralized system that stores data through a network of nodes, which are computers that participate in the blockchain validating and storing data. The main properties of the blockchain are that it is *public* and *immutable*, providing a high level of security to the system.

The original motivation of blockchain systems is the decentralized recording of digital currency, without any physical or centralized support. This concept was firstly introduced by *Bitcoin*, and later it was adopted by other systems.

Furthermore, transactions are grouped into blocks, where each block is linked to the previous one using cryptographic technique, making a chain, thus the name “block chain”.

A *consensus protocol* is also needed so as to decide the next block to be added. The main mechanisms are [3]:

- **Proof of Work (PoW):** This mechanism is based on the existence of “miners”, which are nodes that solve complex mathematical operations in order to validate transactions and creating new blocks, ensuring security to the system but consuming a lot of energy.
- **Proof of Stake (PoS):** In this case, validator nodes are chosen depending on the amount of cryptocurrency the “stake”, being more efficient as far as energy consumption is concerned and allowing better scalability since no complex puzzles are needed to be solved.

Since every node in the network verifies each block, it is a safe system that is protected from numerous attacks [4]. Some of them are:

- **Double Spending:** An attacker sends two conflicting transactions at the same time, but one of them is sent with a higher transaction fee, in such a way that it is confirmed before the second one in such a way the attacker used the same funds to make both transactions.
- **Sybil attacks:** In this case, an attacker creates a lot of fake nodes in order to gain influence throughout the network, overwhelming the network and modifying the regular transaction validation process.
- **51% attacks:** If an attacker manages to have more than 50% control of the system, the attacker will be able to invalidate transactions, reorganize the entire blockchain or even disrupt the transaction history of the blockchain.

However, the strength of the system relies on having a large decentralized network of nodes, making it really difficult for attackers to manipulate the system.

2.2 Ethereum

In this section, we will focus on Ethereum blockchain, which is the current most used blockchain designed to, besides from doing payments from one address to another, be programmable through Smart Contracts in such a way logic can be added to the system and decentralized applications (dApps) can be built [5].

In order to perform the different operations on the Ethereum network, the embedded computer of Ethereum, also known as *Ethereum Virtual Machine (EVM)*, is included to execute the contract code safely throughout the system nodes. These computations are paid with a cryptocurrency called *ether*, which is decentralized and issued in a controlled manner to the stakers that secure the network [5].

Smart Contracts are computer programs that are deployed and executed inside the Ethereum blockchain. They include code and some data that represents its state. Once they are deployed, they cannot be taken down from Ethereum since they are no longer controlled by a user. They are triggered by transactions made by users, which will

execute the contract. The execution of a Smart Contract will consume gas units, even if the execution did not succeed [6].

However, this feature arises the risk of denial-of-service (DoS) attacks, for example, if a malicious contract runs indefinitely. This issue is solved with a mechanism called “gas”. Gas is the unit that measures the amount of “effort” required for a specific operation on the Ethereum network. It is the fuel that allows the user to interact with the network, preventing the system from infinite computational loops or spam by means of a gas fee that must be paid for every unit of gas consumed, limiting the computation a transaction can consume [7].

Blockchain systems strongly rely on the use of cryptographic hashes. A cryptographic hash is a mathematical function that maps any piece of data to a bit array of fixed length. It is quick to compute, irreversible and infeasible to produce two pieces of data with the same hash.

Merkle trees organize many operations and transactions made in the blockchain using these hashes. The *Merkle root* is a single hash value that summarizes all the data below it. This happens because even a small change in the data changes its hash, making it a reliable way to represent and verify blockchain operations.

2.3 Blockchain Trilemma

This trilemma, proposed by *Vitalik Buterin*, the co-founder of Ethereum, states that for a blockchain, it is difficult to simultaneously achieve decentralization, scalability and security since optimizing one property usually limits at least one of the others [8] [9].

2.3.1 Decentralization

This property is referred to how the control of the network is distributed among all the participants, in such a way a single party does not have the control of the network. For instance, Ethereum allows anyone to participate in the network as a validator, although the level of decentralization of a system is principally subjective.

For example, if we are focused on obtaining both scalability and security in the system, the network would have to handle large volume of transactions in a safe way, requiring nodes to have high computational and storage capacity and therefore limiting the number of nodes that can participate in the network. This implies having a more centralized system, which, for example, may increase the censorship or collusion between malicious nodes.

2.3.2 Security

Security is also a key factor to be taken into consideration. Blockchain systems must be safe against different types of attacks, and this security requires of *consensus protocols* which are often expensive and limit the number of transactions processed per second.

If we focused on scalability and decentralization are prioritised, we may achieve it by making the system lighter, sacrificing the security mechanisms, thus the security of the system can be compromised.

2.3.3 Scalability

Finally, scalability stands for the capacity of the network to handle efficiently a growing number of transactions, and how quick they are handled.

This is the more common case for existing blockchain systems, where decentralization and security are prioritised as currently happens in Ethereum. The main consequence is having a low transaction throughput and high fees in periods of high congestion.

2.4 The scalability problem

As explained in the previous section, Ethereum currently prioritises decentralization and security since both of them are closely related, thus scalability has been left aside. Partly because the system has grown so much that, as the number of applications and transactions on the network has increased, the transaction fees have also increased, being really expensive to perform operations during congestion.

A key concept to take into account is throughput, which is the number of transactions a blockchain can process given a certain amount of time, often measured in *transactions per seconds* (TPS).

Currently, the Ethereum mainnet supports up to 15 TPS, but with the ongoing growth of the system, it is starting to get limited. Therefore, it is essential to increase the network capacity as far as speed and throughput are concerned. To do so, some “scaling solutions” are needed, but taking the trilemma into account by maintaining the decentralization and the security of the system. For these reasons, several solutions have been recently proposed, collectively known as Layer 2 network solutions [10].

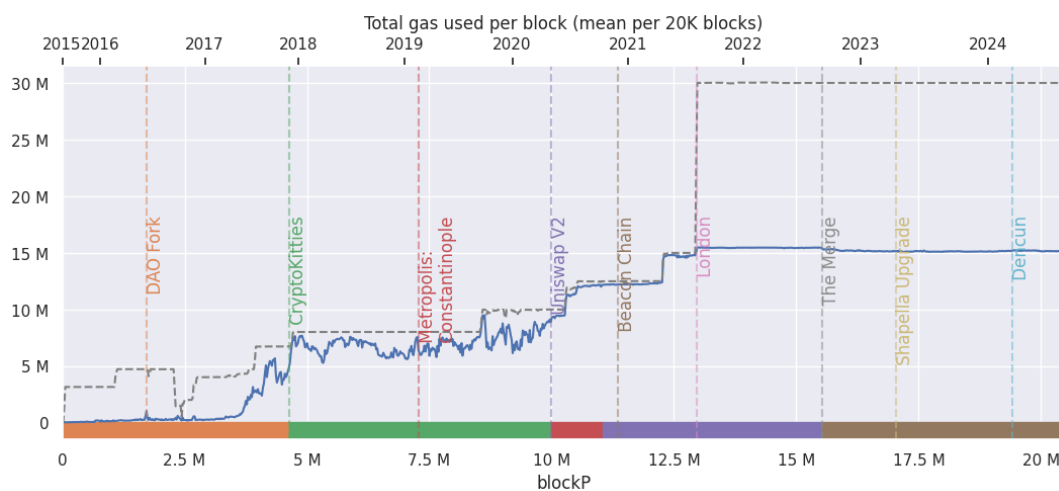


Figure 2.1: Total gas used per block over time [11]

The graph in **Figure 2.1** shows the evolution of the average gas used per block in Ethereum over time, indicating the network congestion and transaction demand. As shown in the graph, gas consumption has continuously been increasing over time, reaching up to 30 million gas units per block since mid-2021. This represents the urgent need of effective Layer 2 solutions that relieve the current congestion problem.

2.5 Layer 2 solutions

A Layer 2 is a separate blockchain that extends the Ethereum mainnet, known as Layer 1, in order to improve the system performance and scalability, inheriting the security guaranteed by the Layer 1.

Layer 2 initiatives appeared as a safer and more scalable approach after a number of previous proposals tried to address the issues posed by high fees and the lack of scalability of the Ethereum mainnet, as for example transaction batching schemes [12] and sidechains [13].

Layer 2 handles transactions off-chain, and to be considered a Layer 2 solution, it must derive security or data availability directly from Ethereum. There are different types of Layer 2 solutions with their own benefits and trade-offs [14].

In what follows, Layer 2 transactions will be referred as off-chain transactions, since they are processed outside Ethereum, which is the Layer 1 network where the mainnet transactions are executed.

2.5.1 State Channels

In the state channels solutions [2] [15], two (or more) Ethereum users that want to operate in a state channel interact with a smart contract known as *multisig* smart contract, committing funds to the channel. These funds are blocked in the contract and their equivalent are then minted on a separate Layer 2 blockchain, so that users can interact with each other on Layer 2 and perform operations.

For each transaction done, each participant signs the final state. A *nonce* is used in each update in order to identify the order of states and avoid attacks. Once they are finished, both participants have signed the last transaction, so the final state is published to the mainnet by one of the participants.

The smart contract now checks both signatures, hence its name *multisig*, and distributes the locked funds according to the final state submitted as shown in **Figure 2.2**.

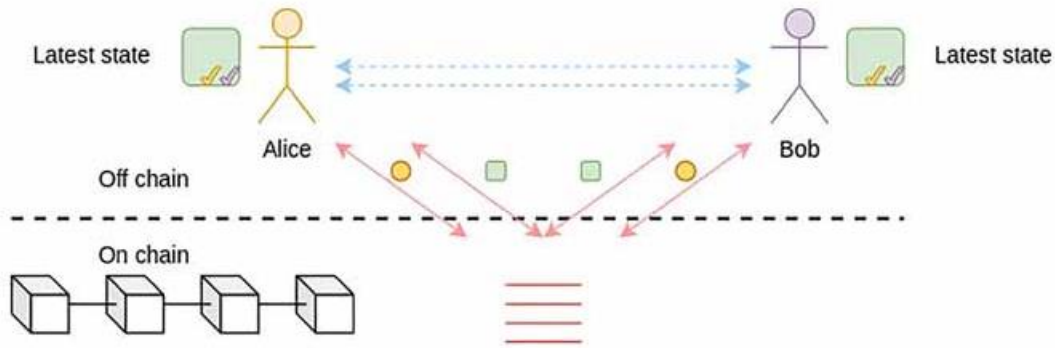


Figure 2.2: State channel representation diagram [16]

In this way, users will be able to interact off-chain while having the security of the mainnet. However, disputes may arise for several reasons such as participants proposing invalid state transitions, finalizing a channel by submitting an old state to the mainnet, participants refusing to sign a state, or even participants going offline, and therefore not being able to sign transactions and the final state.

If the request to process the channel exit is signed by all members involved, the transaction is executed immediately on chain. However, if only one member has signed the last state and decides to submit it on the mainnet, there will be a “challenge period” where all the other members can challenge the request in order to avoid fraudulent actions.

With this approach, the changes are executed and validated by the participants, minimizing the computation of Ethereum Layer 1 since the interested parties of the contract are the responsible for performing the execution and validating the results, increasing transaction speed and solving the scalability problem.

As far as the trilemma is concerned, the security of the state channels has several issues. This solution works like a small mainnet with restricted participants, so if there is a malicious user, the probabilities of security issues increase.

2.5.2 Plasma Chains

A plasma chain [2] [17] is a separate blockchain that executes the transactions off-chain, validating the block with its own mechanism. Firstly, the user deposits the assets in the *plasma chain* through a Layer 1 smart contract, known as *bridge contract*, so now the transactions can be executed off-chain. To ensure the security of the system, there is a centralized actor called the *operator*, which produces blocks and generates *batches* that contain all the *plasma transactions*. Furthermore, state commitments made in the *plasma chain* are published periodically to the Ethereum mainnet in the form of *Merkle roots*, which are nodes that represent all the transactions in a block and are derived from a *Merkle tree* (see **Section 3.3.3** below), which is built with all the computed operations as shown in **Figure 2.3**.

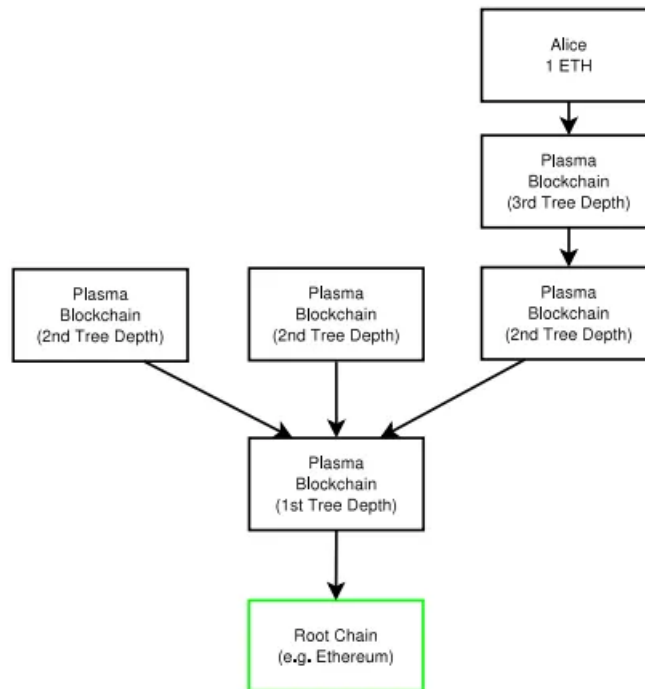


Figure 2.3: Example of a Plasma tree structure [18]

When a user decides to withdraw their assets from the *plasma chain*, a “challenge period” is needed in order to ensure the security of the system, where anyone can challenge a request by proposing a *fraud proof*, which is a mechanism based on *Merkle proof* in case a participant made a malicious request.

The main issue of this solution is the so-called *Mass Exit Problem*. It happens when, given the case that a malicious *operator* exists, since it is a centralized component, it may not publish all the necessary data to the mainnet to perform *fraud proofs*. Therefore, all users must post the last valid state of the chain so as to exit their money, provoking a *mass exit* that drives to a congestion on the mainnet since all the *plasma chain* must be updated at once. If this *mass exit* is not well coordinated, the *operator* might steal the funds of the participants in the *plasma chain*. Therefore, having this centralized component reduces the decentralization of the system as well as its safety.

2.5.3 Optimistic rollups

Optimistic rollups [2] [19] are the most popular Layer 2 solution. They bundle transactions into *batches* and execute them off-chain on Layer 2 and periodically submit state data to the Ethereum mainnet with the use of *batches*, which are compressed data of the operations performed on Layer 2. This solution assumes the executed transactions on Layer 2 are valid, that is why they are called 'optimistic', since they rely on a "challenge period" in which any participant can submit a *fraud proof* if an invalid transaction is detected.

This mechanism allows the increase of throughput while inheriting the security of the Ethereum mainnet. In this way, despite the fact that the challenge period can last up to 7 days, the decentralization and the security of the system are preserved since the computations and operations are performed off-chain and the latter depends on the Ethereum mainnet. For these reasons, *optimistic rollups* seem a good approach to the Ethereum scalability problem. We will study optimistic rollups in detail in the following chapters.

2.5.4 Zero-knowledge Rollups

Zero-knowledge rollups are a variation of *optimistic rollups* and share with them most of its characteristics, [2] [20]. They also process transactions off-chain, with the exception that the validation in the mainnet is cryptographic, known as *validity proofs*. In other words, instead of having a "challenge period", a cryptographic proof ensures that

the resulting state of the rollup is actually the result of executing all the transactions in the *batch*.

Furthermore, *validity proofs* allow participants to prove the correctness of a state without revealing the data it contains, this is why they are called *zero knowledge proofs*. Therefore, withdraw from *zk-rollups* to Layer 1 is straightforward and immediate. Once the smart contract on Layer 1 verifies the correctness of the transaction data, the exit transaction and the return of funds is immediately executed.

There are two types of *ZK-rollups*. The *ZK-SNARKs*, which generate proofs that are quick to verify but require of a *trusted setup* to make it a safe process. And the *ZK-STARKs*, that do not require of a *trusted setup*, instead, they are more expensive to verify on Ethereum, but they are also useful for high volume applications.

Despite the fact that *ZK-rollups* provide security and scalability, the main issues of the approach are its technical complexity and high computational costs that makes them difficult to implement and therefore they are still on development. However, they might be the best Layer 2 solution on the long term.

2.6 Conclusions

Among all the Layer 2 solutions, *Optimistic rollups* currently are the most balanced solution as far as the trilemma is concerned, taking into account scalability, security, decentralization and ease of implementation. While some implementations such as *state channels* or *plasma chains* have decentralization issues and thus, they are no longer used, other solutions such as *zk-rollups* are still under development due to its complexity.

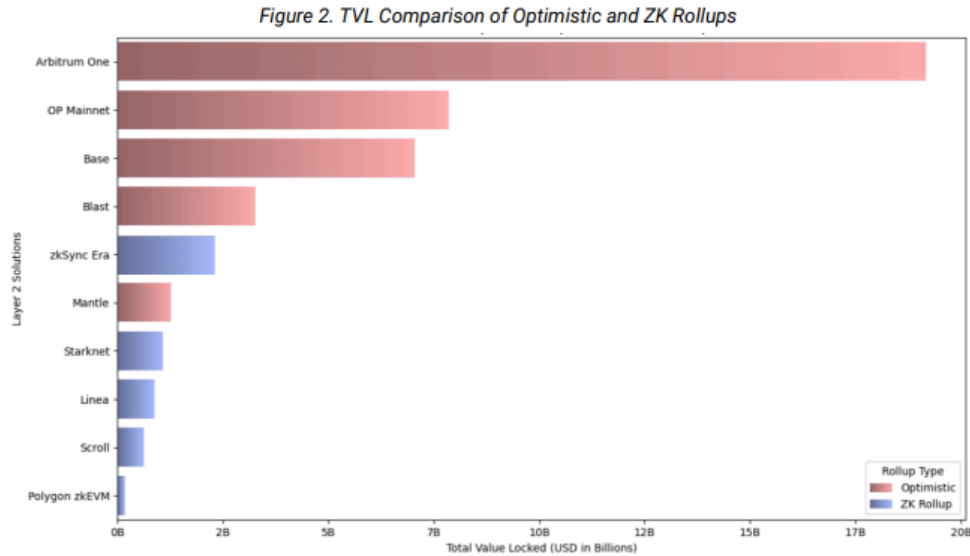


Figure 2.4: Total Value Locked (in USD) comparison between Optimistic and ZK rollups in June 2024 [21]

In the context of **Figure 2.4**, *Total Value Locked (TVL)* stands for the total amount of assets that are deposited into a protocol. A higher TVL usually indicates stronger activity and reliability in the system. In this case, optimistic rollups dominate in terms of TVL with solutions such as *Arbitrum one* or *Op Mainnet* compared to other ZK solutions. As a result, *optimistic rollups* is the current most favourable solution to the Ethereum scalability problem.

Chapter 3. Optimistic rollups

In this section, the *Optimistic rollups* scalability solution is explained in detail, including the architecture, the security properties and how the scalability of the blockchain is improved.

The content of this section is principally based on the official Ethereum documentation [19] so as to ensure technical accuracy in the explanation of *optimistic rollups* solution.

3.1 Topology overview

This part introduces the overall topology of *optimistic rollups*. It presents the main components involved during the process and provides a high-level overview of how they are related to each other and interact with the system **Figure 3.1**.

Optimistic rollups are a type of Layer 2 solution that interacts with the Layer 1 mainnet by submitting state updates periodically. Transactions submitted from Layer 1 are executed off-chain and the resulting state changes are sent to Layer 1 in the form of *batches*. For the system to work, these *batches* are assumed to be valid by default, but they can be challenged within a specific time window (usually 7 days). If the computation is correct and no issues are found, the process is finalized. Otherwise, if the computation is not correct, all the *batches* published after the incorrect *batch*, including it, are reverted.

The diagram in **Figure 3.1** represents all the components involved in the topology of the system, including those belonging to Layer 1 such as essential smart contracts like the *bridge contract* or the *rollup contract*, as well as the Layer 2 components, such as the *sequencer* or the *operator*, which are in charge of receiving, executing, processing and publishing operations on Layer 2. The relationship and data flow between these components is explained in **Section 3.2**. Finally, the last sections provide further explanation of other essential aspects of the system, including the computation of

specific operations that demonstrate the efficiency and functionality of the architecture.

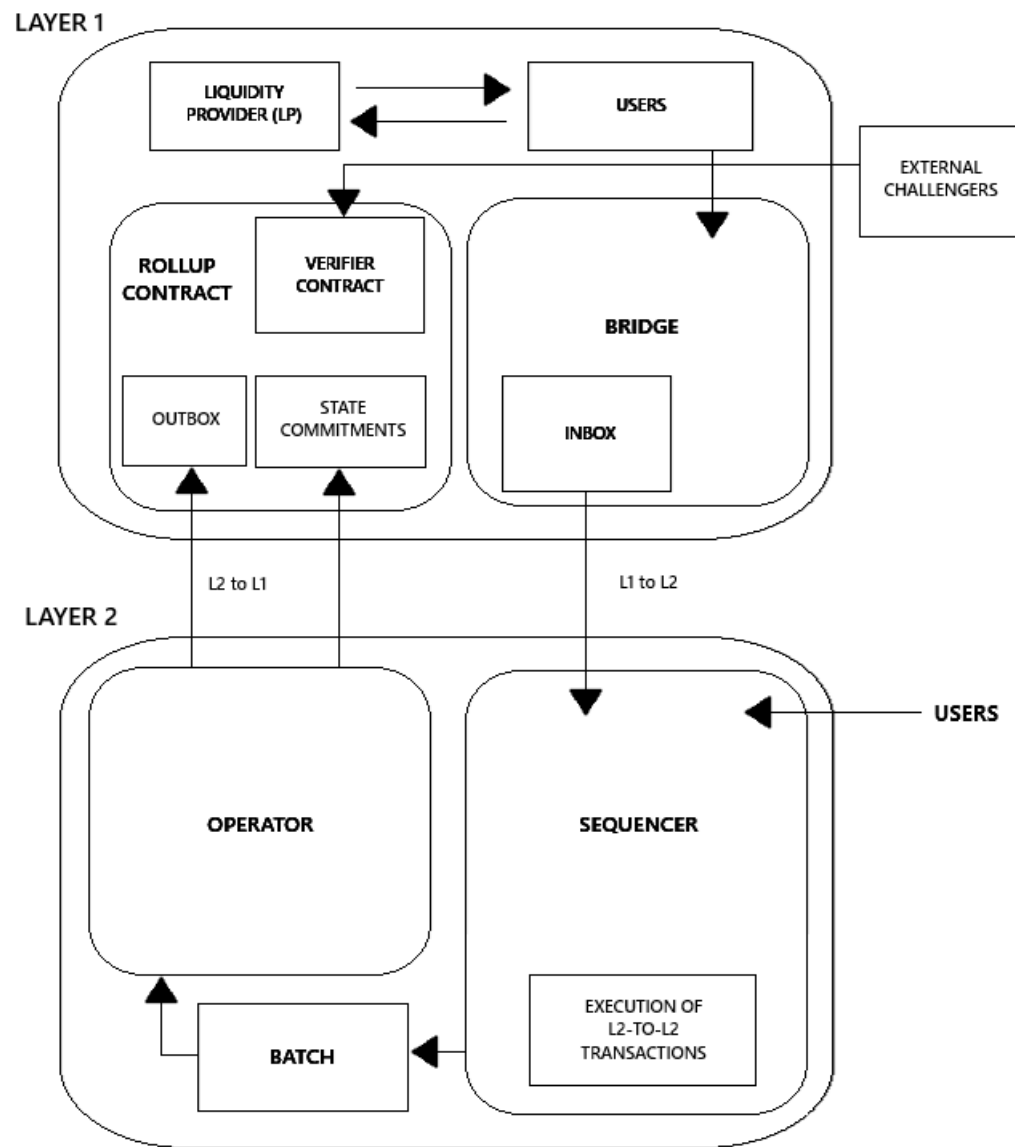


Figure 3.1: Representation of the Optimistic Rollup architecture

3.1.1 User

Users are the primary actors who interact with the optimistic rollup system to perform blockchain operations such as transferring assets, invoking smart contract functions or withdrawing funds. They play a key role in the interaction with Layer 2 and initiating the communication between both layers.

To use an optimistic rollup, *users* must first deposit ETH, ERC-20 tokens or other supported assets into the rollup *bridge contract*. Once on Layer 2, they can submit transactions to the rollup either through an *operator* or directly to the Layer 1 *bridge contract* in case censorship or delays occur. They can also interact with smart contracts deployed on Layer 2 and, in some cases, program Layer 1 contracts to communicate with Layer 2 contracts through bridging mechanisms.

Additionally, *users* are also responsible for initiating withdrawals. To do this, they can use transaction data published on Ethereum to construct *Merkle proofs* that verify their transactions in a valid rollup *batch*.

3.1.2 Bridge / Layer 1 smart contract

Every time *users* make a deposit of ETH, ERC-20 tokens or other supported assets in the rollups *bridge contract* in Layer 1, the assets are locked in the contract. The contract then initiates the transfer of those assets to the Layer 2. Hence, an equivalent amount of assets is then minted to the optimistic rollup and sent to the *users* chosen address on Layer 2, so it is not necessarily the same address as the sender address on Layer 1.

On the other hand, when a user wants to withdraw funds from Layer 2, the assets on Layer 2 are burnt. Then, once the *challenge period* is finished and no fraud was detected, the *bridge contract* on Layer 1 will free the corresponding locked assets to the user.

3.1.3 Sequencer

The *sequencer* is conceptually similar to the *operator*: both processes transactions and produce rollup blocks. However, there are some key differences. For instance, the *sequencer* has a greater control over the ordering of transactions, priority access to the

rollup chain and it is often the only entity authorized to submit transactions to the on-chain contract.

Users generate transactions directly to Layer 2, which are usually queued until the *sequencer* includes them in a *batch*. Although these transactions rely on the *sequencer* to be processed, users can submit those transactions to the Layer 1 *bridge contract* as well, ensuring inclusion. In this way, if the *sequencer*, which is periodically checking for new transactions, ignores the transaction from Layer 2 and continues producing blocks instead, it will eventually submit an incorrect state root. This discrepancy can be challenged later thus the *sequencer* will be penalized through *fraud proofs*.

Despite the fact that both *operators* and *sequencers* represent separate roles, current implementations merge them into a single centralized actor for simplicity. Moreover, some optimistic rollups may refuse to use permissionless *operators*, relying instead in a single *sequencer* to execute the Layer 2 processes.

The previous diagram in **Figure 3.1** shows the main differences between the *sequencer* and the *operator*. While the former is essentially responsible for receiving and ordering the user transactions in Layer 2, the latter is in charge of publishing transaction *batches* and submitting the resulting states to the Layer 1.

3.1.4 Layer 1 Contracts

The logic and the protocol of optimistic rollups are defined by a set of smart contracts deployed on the Ethereum mainnet. These contracts are essential for the functionality and security of the system, as they coordinate the interaction between Layer 1 and Layer 2, store rollup blocks, track user deposits and manage state updates. The key contracts are:

- **Rollup Contract:** This contract is responsible for receiving and storing rollups blocks, including transaction *batches* and the corresponding state roots. Besides, the state root or *Merkle root*, which is the rollup latest state, is hashed and stored in this contract. It also accepts these commitments immediately but can later delete invalid state roots to restore the rollup to its correct state. This contract is also where *fraud proofs* are submitted during the *challenge period*.

- **Bridge Contract:** Its function handles safely the asset transactions in both ways between Layer 1 and Layer 2. It is described in detail in **Section 3.1.2**.
- **Inbox/Outbox mechanism:** Some implementations include *inbox* and *outbox* structures in order to exchange asynchronous messages between both layers. For example, when a user generates a deposit transaction from Layer 1, it is queued in the *inbox* until it is submitted by the *sequencer* to the Layer 2. Conversely, *outbox* enables communication in the same way but from Layer 2 to Layer 1. These mechanisms are essential for ensuring censorship resistance, where users interact directly with the Layer 1 *rollup contract* in case a malicious *sequencer* does not let the user publish transactions on Layer 2.

Optimistic rollups also support *cross-chain function calls*, where Layer 1 smart contracts can interact with Layer 2 smart contracts using bridging contracts to send messages and pass data between them. For instance, this allows us to program a Layer 1 contract to invoke functions belonging to contracts on Layer 2, improving the compatibility between layers.

In fact, developers can also migrate existing smart contracts on Ethereum to Layer 2 rollups with minimal changes. Furthermore, Ethereum smart contracts can act as users by using *bridge* contracts to communicate with Layer 2.

3.1.5 Operator

The first key component is the *Operator*, also known as *validator* or *aggregator*, which is a centralized entity whose responsibility is to aggregate multiple off-chain transactions into larger *batches* before being submitted to Ethereum. It is placed on Layer 2, and it is the key part of the communication from the Layer 2 to the mainnet.

It is important to bear in mind that while rollup systems are often centralized in practice (with a single operator), the design can conceptually support multiple operators depending on the specific rollup implementation and its level of decentralization.

In addition, *operators* are forced to publish the data corresponding to each Ethereum state update. With that information, even if the *operator* goes offline or refuses

to continue submitting transaction *batches*, other nodes can reconstruct the latest rollup state from the available data and resume the block production. However, if there are no *honest nodes*, a malicious *operator* could exploit the system by submitting invalid blocks or incorrect state commitments so as to steal user funds.

In order to become an *operator*, a bond must be provided before producing blocks, which can be slashed if the *validator* does not act honestly and its behaviour is not the one expected.

Every state transition produces a new rollup state, and the operator commits to it by computing a new state root. For each *batch*, it will submit both the old and the new state roots, as well as a *Merkle root* representing the transaction *batch*, allowing anyone to prove that a specific transaction is included in the batch on Layer 1 using a *Merkle Proof*.

The rollup contract on Ethereum accepts new state roots as soon as they are posted by the *operator*. However, it can also remove incorrect state roots and revert the rollup to its previous valid state.

Operators receive fees for their services, known as *Layer 2 operator fees*, which are an amount paid to them as a compensation for the computational costs of processing transactions, just like *gas fees* on Ethereum. However, these are much lower transaction fees since the Layer 2 have higher processing capacities and a better scalability system than the Layer 1, with less congestion than the Ethereum mainnet.

Besides, *operators* are also involved in the *Proof of Fraud* mechanism, as it is seen in **Section 3.1.6** *validators* on the optimistic rollup chain are expected to execute independently the submitted transactions using their own copy of the rollup state. If the resulting state is different from the one proposed by the *operator*, they can start a *challenge period* where a *fraud proof* is submitted in order to demonstrate the invalidity of the *batch*. If the *operator* is proven to be dishonest, it will be penalized by having its bond slashed.

This component is also closely related to censorship risks, mainly because it controls the Layer 2 block production process. As a result, it has the capability to exclude users by either selectively refusing to produce blocks that include certain transactions or by

completely ending its activity. Furthermore, its centralized nature allows it to influence transaction ordering, which could be exploited to delay specific transactions.

3.1.6 Fraud Provers

Also known as *challengers*, the main goal of this actor is to verify the rollup state. They play a key role in the *fraud proof* mechanism by checking the data published on Layer 1 and disputing invalid state transitions if necessary.

In practice, anyone can become a *challenger* of the topology. However, these entities, such as private full nodes or watchdog services, must be able to run rollup software that can reconstruct and verify the rollup state.

Depending on the design of the rollup and the fraud proving scheme it uses, *challengers* can participate in different mechanisms, such as *single-round* interactive proving or *multi-round* interactive proving, which are studied in detail in **Section 3.2.5**.

Challengers are economically rewarded for detecting fraud. If a challenger proves that the rollup operator submitted an invalid state, part of the operator's security deposit is slashed. The *challenger* receives a reward from that portion, while the rest is permanently burnt. This burning mechanism helps prevent collusion between operators, making sure coordinated fraudulent challenges still result in a financial loss.

3.1.7 Asserter

In the context of a challenge, the *asserter* is the actor responsible for submitting state transitions to defend the current state from challenges made by other actors. This role is typically played by the rollup operator, who asserts that a certain new state of the rollup is valid based on the off-chain execution of transactions. These assertions can be disputed if they are considered incorrect.

3.1.8 Liquidity Provider (LP)

When the user is exiting the optimistic rollup, users can rely on a *Liquidity provider* (LP) in order to avoid waiting a week to withdraw the funds from the Layer 1.

Liquidity providers assume ownership of a pending Layer 2 withdrawal request to independently verify it by executing the chain themselves in order to make sure that the transaction will be eventually confirmed. If the computation is correct, it pays the *user* the amount of the transaction on Layer 1 in exchange for a fee.

3.2 Full Operation Flow

This section relates all previously defined components and actors of the topology to explain how the optimistic rollup system operates as a whole.

3.2.1 Entering the rollup

This is the initial phase of the process, where a *user* wants to access the rollup in Layer 2. To do so, the *user* must deposit ETH, ERC-20 tokens or other accepted assets into the *bridge contract*, which will lock the assets on Layer 1 and triggers the minting of equivalent tokens on Layer 2.

Then, this transfer is queued to the *inbox* in Layer 1, where the *sequencer* continuously checks it in order to obtain any user generated transaction from Layer 1. Once the *sequencer* processes the transaction, it includes it in a new Layer 2 block, which will be used later by the operator to submit the transaction to the rollup contract. As a result, the rollup mints the corresponding amount of assets on Layer 2 and sends them to the users chosen address, which can be chosen by the user and does not need to match the one of the senders from Layer 1.

At this point, the user has received the equivalent assets in the optimistic rollup on Layer 2 and is now ready to begin making transactions within the rollup environment.

3.2.2 Relation between Operator and Sequencer

Now the user can perform transactions on Layer 2. All *user* transactions made to the rollup are submitted directly to the sequencer through its RPC node or endpoint. The *user* transactions can be sent either by users directly to the optimistic rollup or to the *bridge contract* on Layer 1, which will be kept in the *inbox* until the *sequencer* includes them.

Transactions are usually stored in the inbox by order of arrival, although this could vary depending on the system design, and they are maintained in a private transaction queue, where it handles the ordering and execution of those transactions over the current rollup state, updating contracts, address balances, etc.

All transactions and their results are then grouped in Layer 2 blocks after their computation, just like the mainnet does, where each block contains an update of the rollup state.

After a certain amount of time, the rollup groups several Layer 2 blocks into a *batch*, which contains the compressed transactions made in those blocks, the necessary data to reconstruct the rollup state and the *state root*.

At this point, the operator takes part in the process since it calculates the *Merkle root* containing all transactions. To do so, the *operator* takes all transactions from the *batch* and organizes them in a binary tree data structure, also known as *Merkle tree*, where each leaf is either a transaction or part of the rollup state, and the parent node is the hash of its two children. In this way, the *Merkle root* represents the *batch* content in a compact form.

The operator then publishes on Layer 1 the old state, the new state and the batch data for which the Merkle root has been calculated. Besides, the data included in the Layer 1 transaction issued depends on the particular implementation of the rollup, as we will see in **Section 3.3.2**.

When a batch is sent to Layer 1, the operator includes a summary of the batch with a Merkle root, which is the top hash of the Merkle tree built from the transactions in the batch and summarizes effectively all the transactions executed across the blocks. It also includes both state roots (the old and the new one) and finally the *blob*, with all the information of the computed transactions.

The *operator* finally publishes the *batch* to the Layer 1 *rollup contract*, where *fraud proofs* are submitted during the challenge period.

3.2.3 Exiting the rollup

Now that the user wants to move its funds back to the Ethereum mainnet, a withdrawal request is initiated on the rollup, which is processed on Layer 2, and the corresponding assets are burnt. This request is treated as any other transaction on Layer 2, in other words, it is included in the next Layer 2 block and then bundled into a *batch*, ready to be sent to Layer 1.

Once the batch that contains the *exit* transaction is posted to Ethereum, the user can generate a *Merkle proof* on Layer 1 to verify that their withdrawal was included in the published data. However, due to the *fraud proof* process, the user must wait until the *challenge period* ends so as to finalize the withdrawal and obtain its funds on Layer 1 mainnet.

3.2.4 Verification: The “Optimistic” Assumption

Optimistic rollups are considered “optimistic” because they assume that all off-chain transactions are valid by default. In other words, they do not verify the correctness of each *batch* before submitting it to the mainnet.

In this way, when a *batch* is submitted on Ethereum, it is accepted as valid, unless someone proves otherwise. Hence, the security of the system depends on the existence of at least one honest *validator* capable of detecting and disputing fraudulent activity during the next *challenge period*.

3.2.5 Challenge Period

The last part to finalize the process is the so-called *challenge period*. To ensure the correctness of the rollup state and maintain the security, the protocol allows a specific time window, known as the *challenge period*, where any participant can dispute the validity of a *batch*.

This process is known as *fraud proving* and only takes part in when a participant believes that a submitted *batch* includes an invalid state transition. If this happens, the system protocol initiates a verification mechanism to solve the conflict between the *asserter* and the *challenger*.

There are two types of *fraud proof* mechanisms used in optimistic rollups:

Single-round interactive proving:

In this method, the protocol executes again the disputed transaction directly on the Layer 1 mainnet through a *verifier contract*. In this way, it computes the resulting state and compares it with the one provided by the *asserter*. If the resulting state does not match, the assertion is proven fraudulent, and the *operator* is penalized by having its bond slashed.

The main disadvantage of this mechanism is that it is very costly in terms of gas consumption, and it requires publishing state data for each transaction, which increases the amount of data stored on Layer 1.

Multi-round interactive proving:

To improve efficiency, a more advanced mechanism is required, where the system reduces the load on Ethereum by resolving dispute through a *back-and-forth* process between the *asserter* and the *challenger* that is coordinated by the *verifier contract*.

1. The flow of this method starts with the *asserter* dividing the entire disputed computation into two equal parts.
2. The *challenger* now selects the half that thinks that contains the invalid state transition.
3. This dividing process, known as *bisection protocol*, continues until both parties disagree about a single computation step. When that step is reached, the Layer 1 resolves the dispute by executing the instruction and evaluating its result.
4. At this point, the *asserter* must submit a *one-step proof* that demonstrates the validity of that single operation.

If the *asserter* does not provide the proof or it results to be invalid, it will automatically lose the challenge. Hence, part of its bond is awarded to the *challenger*, and the rest is burnt to avoid malicious collusion between nodes. Furthermore, the rollup

protocol reverts the state to the last known valid root of the previous assertion, getting rid of the malicious *batch* and continuing from the correct state.

This mechanism ensures that, even in the presence of malicious nodes, the chain will eventually confirm only valid transactions, providing economic incentives for honest behaviour and penalizing incorrect submissions.

The multi-round proving clearly is a much more efficient method since the mainnet only has to verify a single computation step instead of executing again the entire batch. Besides, it also reduces the need to publish large amounts of data on chain and helps reduce gas consumption.

3.2.6 Finalization and return

Once the *challenge period* has finished and assuming no successful disputes have occurred, the *batch* is considered finalized and the *user* can now safely withdraw its assets on the mainnet.

3.3 Additional Concepts and Technical Considerations

3.3.1 Batch structure and contents

The *batch* structure is used in the system to transfer essential information from the Layer 2 rollup to the Ethereum mainnet. It contains several key pieces of information to guarantee correctness and enable the ability to prove or dispute the validity of the off-chain computations:

- **Old state root:** A hash that represents the state of the rollup before executing the transactions in the *batch*.
- **New state root:** A hash that represents the state of the rollup after executing the transactions in the *batch*.
- **Merkle root:** This is the top hash of a *Merkle tree* built from all the transactions in the *batch*. It summarizes all the transactions executed in the *batch*, allowing anyone to check if a transaction is part of the *batch*.

- **Transactions data:** This contains the detailed information of all the transactions in the *batch* needed to reconstruct and verify the state of the rollup, particularly by *operators* and *validators*. Depending on the implementation, this data is published either as *calldata* or as a *blob*, which improves the efficiency and reduces costs.

Together, these components make the *batch* a piece of evidence that indicates the initial state of the rollup, the final state of the rollup and what has been executed.

3.3.2 *Calldata vs Blobs*

Every time optimistic rollups publish *batches* to the Ethereum mainnet, it is necessary to store the data in order to reconstruct the rollup state in the case a dispute happens. Initially, this was done using *calldata*, but with the introduction of *EIP-4844* rollups can now use a more efficient mechanism known as *blobs*.

Calldata:

The traditional mechanism for publishing data on Ethereum is by means of contract function parameters that contain the data to be published. These parameters are stored in the transaction in a region named "calldata".

However, the main disadvantage of *calldata* is its high gas costs since the information is stored permanently on Ethereum and writing large amounts of data on Layer 1 is an expensive and not scalable practice in the long term.

Blobs:

The introduction of *EIP-4844*, also known as *proto-danksharding*, brought a cost-effective solution for including data on chain called *blobs* or "*binary large objects*". *Blobs* are a new kind of data container that allows rollups post large amounts of data on Layer 1 at a lower cost. This happens because, unlike *calldata*, *blobs* are not stored permanently on the mainnet and its data is available off-chain. In fact, the data is available for a limited time, typically 18 days more or less, which is more than enough

time to verify transactions and *fraud proofs* during the *challenge period*. After that time, the content of the *blob* is erased from the Ethereum nodes.

For this reason, rollups are now adopting *blobs* as the main transaction data type to post data on the Ethereum mainnet. Since *fraud proofs* only need the transaction data during the *challenge period* thus having them permanently on Layer 1 is unnecessary and very expensive.

3.3.3 Merkle Trees and State Commitments

Merkle trees are a type of binary tree where each *leaf* node contains a hash of data and each *parent* node contains the hash of its two *child* nodes, resulting in a single top hash known as *Merkle root* that represents the whole structure. They are used as a fundamental data structure to organize large amounts of data efficiently.

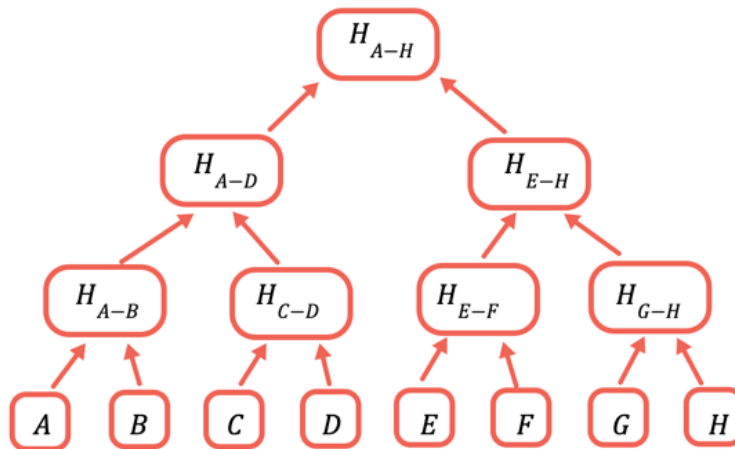


Figure 3.2: Representation of a Merkle Tree [22]

The tree showed in **Figure 3.2** represents a *Merkle tree* where each parent node is built from the hash of its two child nodes. Each intermediate node in the tree is the cryptographic hash of its child nodes. For instance, H_{A-B} contains the hash of the concatenation of the contents of nodes A and B, and H_{A-D} in turn contains the hash of the concatenation of the contents of nodes H_{A-B} and H_{C-D} . The top node of the tree, H_{A-H} , represents all the structure since it is the *Merkle root* of the tree [22].

This structure shown in **Figure 3.3** allows users to verify the inclusion of a piece of data without computing all the nodes of the tree, this is known as *Merkle proof*, which is defined as the minimum set of hashes that allows anyone to verify a specific element of the tree by recomputing the Merkle root. Since the nodes are hashes, any change made to any of the leaf nodes of the tree would produce cascade changes to its ancestors until the top node of the tree.

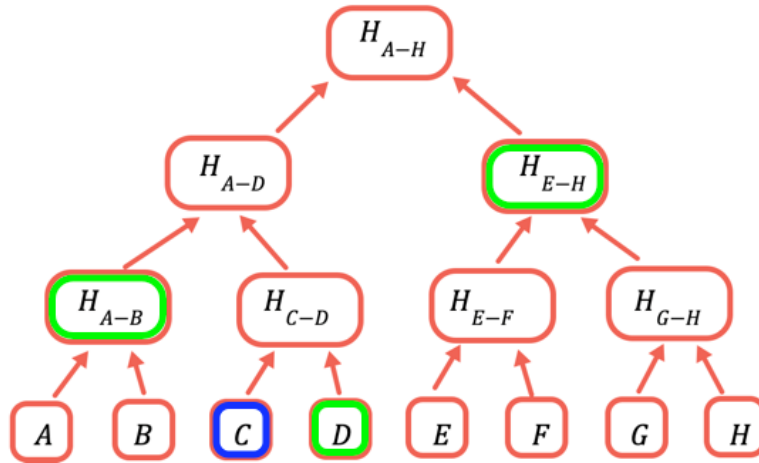


Figure 3.3: An example of a Merkle proof [22]

In this example **Figure 3.3**, we want to prove that node 'C' is correct. To do so, a *Merkle proof* is needed, and it is formed by the hash of 'D', H_{A-B} and H_{E-H} . The three of them constitute a *Merkle proof* to prove that 'C' is correct by computing the root of the tree with this set of hashes, assuming 'C' is correct, and if the same value as H_{A-H} is obtained, then the *Merkle proof* is correct, proving the correctness of node 'C' [22].

The main goal is that the entire rollup state ends up being represented by the *Merkle tree*, this is known as *state tree*, where its *Merkle root*, the *state root*, is stored in the *rollup contract* in Ethereum. Every time a transaction of the *batch* is executed, a new *state root* is produced. State commitments, especially *state roots*, are necessary for proving the correctness of state changes in an optimistic rollup and the *operator* is the

component responsible of submitting the old and the new state to the mainnet when posting *batches*.

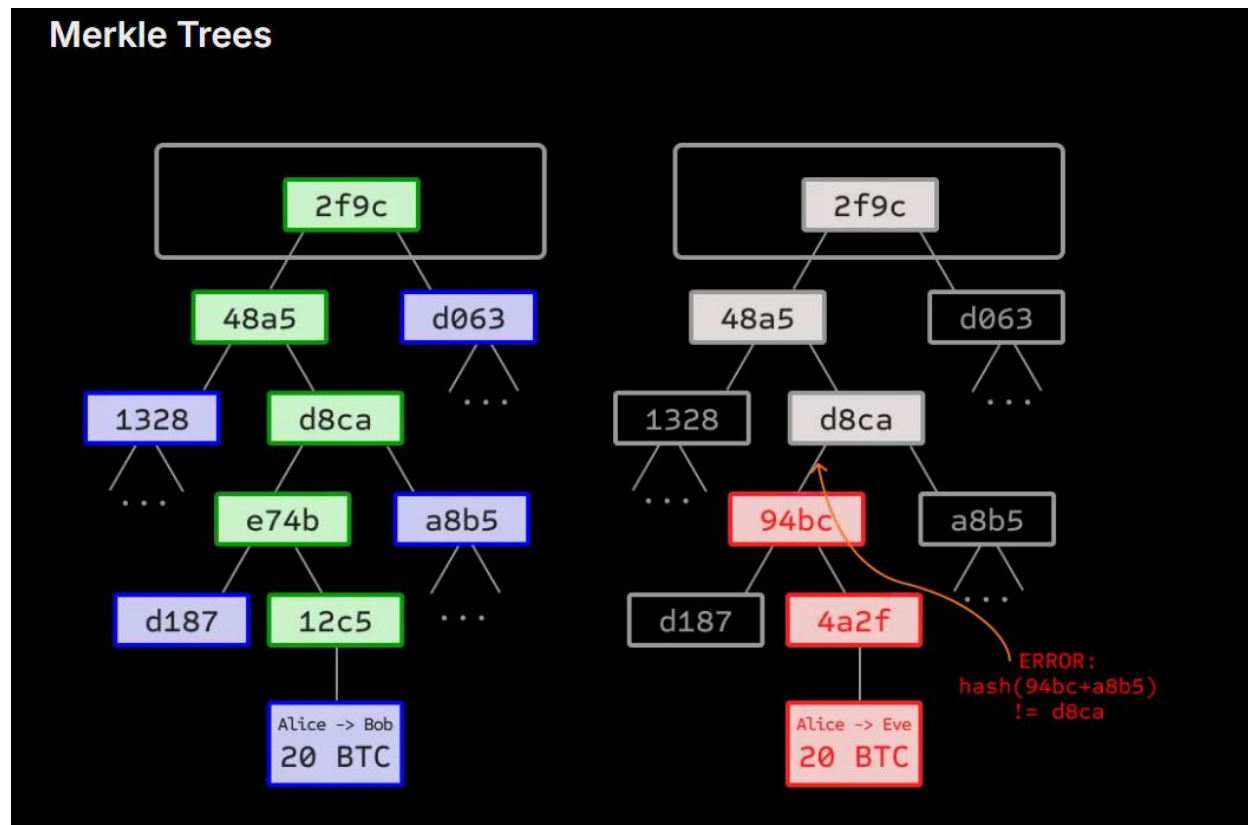


Figure 3.4: An example of a node miscalculation in a Merkle Tree [21]

In the case depicted in **Figure 3.4**, the tree on the left represents some nodes of the original tree structure, where Alice has performed a transaction to Bob. However, on the tree on the right, Alice makes a transaction to Eve instead, having as a result a different hash for that transaction and therefore being unable to obtain the same root node as in the correct tree on the left since the hashes of some nodes of the tree will be different [22].

To sum up, *Merkle trees* improve the efficiency and scalability of the rollups by providing *Merkle roots* instead of having to store or publish all the transaction histories from Layer 2, allowing the system to verify data in an efficient and scalable way.

3.3.4 Honest node assumption

This is a security property of optimistic rollups necessary for the *challenge period*. For instance, the validity of the chain relies on the existence of at least one *honest node*. The *honest node* can advance the chain correctly by either posting valid assertions or disputing invalid assertions. Whatever the case is, malicious nodes who enter into disputes with the *honest node* will always lose their stakes during the fraud proving process.

3.4 Performance benefits and metrics

In this section I will discuss the most relevant benefits and advantages of using optimistic rollups in terms of scalability, cost efficiency and transfer data compared to directly performing the same transactions on Layer 1 mainnet.

3.4.1 Increased throughput (TPS)

One of the scalability problems to be solved is the number of transactions processed per second, also known as *TPS*. To increase this value, the aim is to compress the transaction data before publishing it to the Ethereum mainnet, reducing the size and cost of each operation.

First of all, some important data must be taken into account. Currently, Ethereum blocks have a fixed gas limit of around 15 million gas per block. According to EIP-1559 [23] [19] blocks can use more than 15 million gas units but the transactions cost increases due to congestion. We consider the standard block limit in these estimations.

Posting data on chain consumes an amount of gas that depends on the number of bytes to be published. For instance, a non-zero byte in calldata costs 16 units of gas in the current version of the EVM, whereas zero-byte costs 4 units of gas [24]. As a result, assuming the contract execution cost is zero, it can be deduced that in the worst case is around:

$$\frac{15,000,000 \text{ units of gas} - 21,000 \text{ transaction gas units}}{16 \text{ units of gas} / 1 \text{ byte of calldata}} \cong 936,188 \text{ bytes}$$

Therefore, 936,188 bytes of transaction data can fit in each Ethereum block.

Parameter	Ethereum (L1)	Rollup (L2)
Nonce	~3	0
Gasprice	~8	0-0.5
Gas	3	0-0.5
To	21	4
Value	9	~3
Signature	~68 (2 + 33 + 33)	~0.5
From	0 (recovered from sig)	4
Total	~112 bytes	~12 bytes

Figure 3.5: Comparison of data generated by the same transaction between Layer 1 and Layer 2 [19]

In addition, as shown in **Figure 3.5**, a simple transaction such as sending *ether* from one address to another requires around 112 bytes when it is directly executed on Layer 1. However, if the same transaction is compressed and posted through an optimistic rollup, it would only require around 12 bytes. Taking all these factors mentioned into account, this can lead to several interesting deductions.

For example, given that a rollup transaction is 12 bytes:

$$\frac{936,188 \text{ bytes in a block}}{12 \text{ bytes/transaction}} \cong 78,016 \text{ transactions}$$

The result is that up to 78,016 rollup transactions could theoretically fit in a single Ethereum block.

Moreover, given a block time of 15 seconds, the theoretical throughput would be:

$$\frac{78,016 \text{ rollup transactions}}{15 \text{ seconds}} = 5,201 \text{ transactions/second}$$

However, it is important to bear in mind that this only represents an upper bound since in practice rollup transactions cannot occupy an entire Ethereum block. Currently, rollup transactions already achieve a throughput value of 2,000 *TPS* [19], depending on the system and the network congestion.

Finally, the introduction of *EIP-4844* and *danksharding* combined with the use of *blobs* is expected to improve the scalability of the system by increasing the *TPS* capacity and reducing costs.

3.4.2 Efficient data storage

As mentioned in **Section 3.3.2**, *blobs* are a new method for storing data off-chain. The key factor is that *blobs* are published on the mainnet for a maximum of 18 days, while *calldata* is stored permanently on Layer 1 and it is immutable, increasing the gas costs associated with data storage.

According to *Blobscan*, which is a platform that tracks and compares *blobs* and *calldata* usage, over 53,270 *ether* [25] has been saved in transaction fees since the introduction of *blobs*. This is partly because more than 6.7 million *blobs* have been posted, with a total size of 815.37 GiB, demonstrating that *blobs* enable high throughput data availability.

In addition, *blobs* usage remains consistently high since the equivalent *calldata* of gas usage averages 40 billion units of gas each day and around 9,000 *blob* related transactions are made every day [25].

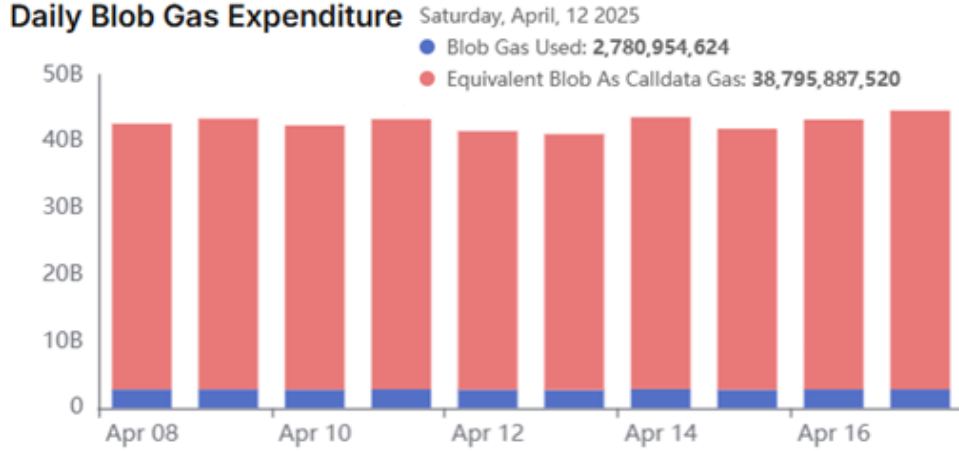


Figure 3.6: This graph shows the differences between blob and calldata consumption during April 2025 [25]

The graph in **Figure 3.6** shows the amount of gas used by *blobs* compared to the amount of gas that would have been consumed if *calldata* had been used instead for a period of 10 days during the month of April 2025. It is easy to deduce that the use of *calldata* means a significantly higher gas cost to store data. In particular, the gas usage for April 12, 2025, is highlighted for comparison in **Figure 3.6**.

We can see in this graph that, on April 12, 2025, *blobs* consumed 2,780,954,624 units of gas, while the *calldata* equivalent would have required 38,795,887,520 gas units. Therefore, we can deduce that *blobs* consume $\frac{38,795,887,520 \text{ units of gas}}{2,780,954,624 \text{ units of gas}} = 13.95$ times less gas than *calldata* for the same volume of data on an average day. Besides, the gas savings on only that day sum up to $38,795,887,520 - 2,780,954,624 = 36,014,932,896$ units of gas.

This reduction can also be expressed in the following way:

$$\text{gas reduction} = \left(1 - \left(\frac{\text{blobs}}{\text{calldata}}\right)\right) \times 100 = \left(1 - \left(\frac{2.78 \text{ billion}}{28.80 \text{ billion}}\right)\right) \times 100 \approx 93\%$$

Hence, *blobs* offer a reduction of 93% gas used compared to *calldata*. This decrease can be translated to lower fees for users, making rollups more financially affordable for them. As a result, *blobs* are a solid replacement for *calldata*, allowing a cheaper, more scalable and more efficient data publication to Ethereum Layer 1, especially on the context of optimistic rollups.

3.4.3 Merkle tree compression

Another key factor that reduces the amount of gas consumed is the use of *Merkle tree* compression techniques. This is mainly because *Merkle trees* represent a set of transactions through a single *root hash*, reducing the amount of data to be stored. Furthermore, the transaction data is published through *blobs*, making the system more efficient as far as gas consumption is concerned. To study the benefits that *Merkle trees* provide to the system in terms of gas optimization, some data will be considered.

First of all, it is known that the size of a rollup *blob* is 128 kiB [25] and that transaction Ethereum *hashes* have a size of 32 bytes, or 256 bits, since they are usually generated with *SHA-256* or *Keccak-256* [26]. According to [24], representing a single byte on the Ethereum mainnet has a cost of 16 bytes in the worst case and the cost of a *hash* is:

$$\text{hash gas cost} = 32 \text{ bytes}/_{\text{hash}} \times 16 \text{ gas}/_{\text{byte}} = 512 \text{ gas}/_{\text{hash}}$$

Furthermore, an average simple transaction on Layer 1 costs to 21,000 gas [27] and these transactions usually have an average size of 100 bytes. This value can be deduced from having two addresses of 20 bytes, the origin and the destination, the value of the transaction of 32 bytes and other transaction fields such as the gas price, the gas, etc, that can add up to 100 bytes.

Finally, taking into account the *EIP-4844* [28] update, each *blob* contains 4,096 field elements of 32 bytes each, this means that, in the context of *Merkle trees*, a *blob* can store up to 4,096 transactions. Furthermore, each Ethereum *block* or *batch* can hold a maximum of 16 *blobs*. This means that:

$$4,096 \text{ transactions}/_{\text{blob}} \times 16 \text{ blobs}/_{\text{block}} = 65,536 \text{ transactions}/_{\text{block}}$$

Which translated to gas consumed is:

$$65,536 \text{ transactions} \times 512 \text{ gas}/_{\text{transaction}} = 33,554,432 \text{ units of gas}$$

If those transactions were performed directly on Layer 1, the gas cost for each block would be:

$$65,536 \text{ transactions} \times 21,000 \text{ }^{gas}/_{transaction} = 1,376,256,000 \text{ units of gas}$$

With these results, the final total savings can be calculated:

$$Savings = \left(1 - \frac{Merkle \text{ tree gas cost per block}}{Layer \ 1 \text{ gas cost per block}}\right) \times 100 = \left(1 - \frac{33,554,432}{1,376,256,000}\right) \times 100 \approx 98\%$$

Which is the same calculation as just comparing the gas consumed for a single transaction:

$$Savings = \left(1 - \frac{Merkle \text{ tree gas cost}}{Layer \ 1 \text{ gas cost}}\right) \times 100 = \left(1 - \frac{512}{21,000}\right) \times 100 \approx 98\%$$

This means that every transaction that is executed and published directly on Ethereum Layer 1 consumes approximately 21,000 gas units, whereas optimistic rollups can represent each transaction on Layer 1 using only its *hash*, consuming just 512 gas units each. Thus, this results in an approximate reduction of 98% in terms of gas costs. Note that this estimation does not include the additional cost from periodic L1 publications of the rollups state, which is discussed in **Section 3.4.4**. This compression mechanism is essential for the scalability of the system since this reduction makes it possible to handle a larger number of transactions at a much lower cost.

3.4.4 Fee payments

For this system to work correctly, it is essential that all the components involved in the rollup topology receive a predefined financial compensation in the form of *fees* in order to carry out their roles. This is the case, for example, of *operators* and *sequencers*.

Therefore, when a user initiates a simple transaction on Layer 2, a small *fee* is paid when submitting the transaction. It is important to bear in mind that these *fees* on Layer 2 are significantly lower than those on Ethereum Layer 1 mainnet.

These fees include, for each simple transaction, its ordering and execution on Layer 2, the process of building the rollup blocks, the update of the system state and the publication of the *batch* on Layer 1 mainnet. The congestion of the network will be also taken into account in the final fee calculation.

According to [29], executing simple transactions such as sending ETH from one address to another through rollups is significantly cheaper than using the Ethereum mainnet.
























Name	Send ETH	Swap tokens
 Metis Network 	\$0.04	\$0.18 
 Loopring	\$0.04	\$0.59 
 zkSync Era	\$0.07	- 
 zkSync Lite	\$0.09	\$0.22 
 Optimism	\$0.09	\$0.18 
 Arbitrum One	\$0.09	\$0.27 
 Boba Network	\$0.15	\$0.17 
 DeGate	\$0.16	\$0.18 
 StarkNet	\$0.19	\$0.57 
 Polygon zkEVM	\$0.19	\$2.75 
 Ethereum	\$1.10	\$5.48 

Figure 3.7: Transaction fee differences between several blockchain networks on April 30, 2025, including fees for Ethereum mainnet transactions [29]

For example, as we can see in **Figure 3.7**, sending ETH trough *Optimism* or *Arbitrum* has a cost of 0.09\$, while performing the exact same transaction on Ethereum mainnet has a cost of 1.10\$. This fee reduction can also be represented as:

$$Savings = \left(1 - \frac{L2 \text{ fee price}}{L1 \text{ fee price}}\right) \times 100 = \left(1 - \frac{0.09}{1.10}\right) \times 100 \approx 92\%$$

Hence, executing transactions on Layer 2 rollups such as *Optimism* or *Arbitrum* provides users around 92% of savings compared to performing the transactions on the Ethereum mainnet.

Furthermore, the platform also provides fee information for more complex transactions such as *swapping* tokens, which is a process that involves *smart contracts* and are therefore more expensive due to the required computational logic. According to the same table, these transactions for *Optimism* cost 0.18\$ and 0.27\$ for *Arbitrum*, which results in an average of 0.225\$. On the other hand, the same transaction has a cost on the Ethereum mainnet of 5.48\$. If we perform the calculations for these types of transactions, we obtain that:

$$Savings = \left(1 - \frac{L2 \text{ fee price}}{L1 \text{ fee price}}\right) \times 100 = \left(1 - \frac{0.225}{5.48}\right) \times 100 \approx 96\%$$

In conclusion, Layer 2 rollups are not only cheaper and more efficient, but they also require significantly lower fees to perform the transactions. In this way, users are able to execute transactions and publish them on the Ethereum mainnet through *blobs*, with the cost of only a few cents.

Chapter 4. Installation Environment

In order to prove the efficiency of *Optimistic rollups*, some experiments will be performed in the Layer 2 *Optimism Sepolia testnet*. To do so, it is important to bear in mind that this testnet differs in some aspects from the standard *Optimistic rollups* architecture explained in **Section 3.1**. Therefore, the *Optimism* architecture is implemented with some modifications [30].

4.1 Main architecture differences

The *Optimism Sepolia testnet* works with a similar structure than the standard *Optimistic rollups*, [31] [30]. The main components of Optimism are shown in **Figure 4.1**.

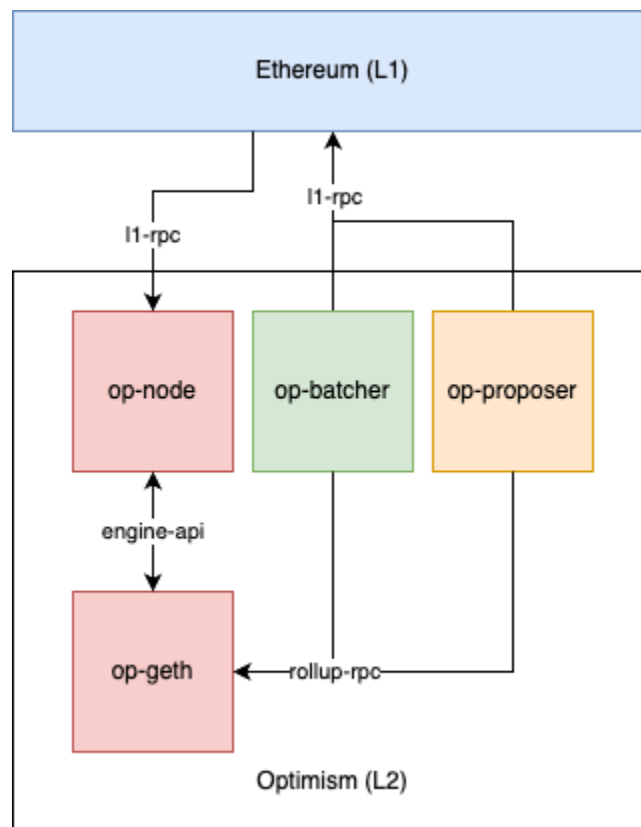


Figure 4.1: Optimism Sepolia testnet architecture representation [30]

4.1.1 Op-geth

It replicates the Ethereum application environment, implementing the execution layer of Layer 2.

4.1.2 Op-node

It is used with *op-geth* to connect the rollup with the Layer 2, it works as the Layer 2 consensus layer.

4.1.3 Op-batcher

This component is similar to the *operator* component in **Section 3.1.5**. It is in charge of submitting the Layer 2 data provided by the *sequencer* to Layer 1 so that anyone can later challenge the state on the mainnet, providing the minimum data as possible to replicate the Layer 2 blocks.

4.1.4 Op-deployer

This element is not taken into account in the topology shown in **Figure 4.1** This is because it is used by *Optimism* as a deployment tool, which compares the current state of the chain with the configuration it should have in order to apply the necessary changes for it to work properly.

4.1.5 Op-proposer

It is responsible for publishing output roots from Layer 2 to Layer 1. It is similar to the *state commitments* block in **Figure 3.1**.

4.1.6 Other components

The relation and workflow of all the explained components is shown in **Figure 4.1**, where all of them belong to the Layer 2.

Besides the main component just described, the system also requires of other components which are already implemented in the system, but they are key for everything to work properly.

A *sequencer* is also necessary in the topology. As in the standard implementation, it is responsible for creating new Layer 2 blocks together with the *op-batcher* and the *op-proposer*. For security reasons, it does not have connection to the internet, in fact, it is only connected to other internal peers.

An *Op-conductor* is also required. It manages and coordinates the services and components of the network so that they work when they are needed.

Besides, an *Op-challenger* that is periodically checking the Layer 2 state and challenging the incorrect information that has been submitted to the mainnet. In this way, the Layer 2 is maintained and safe.

As we can see, the *OP Stack* has a modular design, allowing the construction of different chains that relates components that follow the same protocol.

4.2 Implementation in Go of key components

The Optimism implementation is not all written in Solidity, in fact, most of it is written in Go as it is shown in the official *Optimism* repository as shown in **Figure 4.2**.

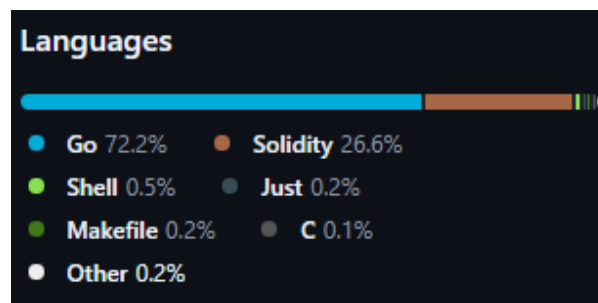


Figure 4.2: Programming languages used in the Optimism repository [32]

This is mainly because smart contracts only run on the *Ethereum Virtual Machine* (EVM) since they are written in *Solidity*, so they can only execute logic on chain. However, the components of the system need to perform off-chain operations in order to communicate with the users and allow communication between both layers.

Some examples of these operations are: reading events from Layer 1, like the *sequencer* from Layer 2 does with the *inbox contract* from Layer 1 to read the requests

that users make to use the rollup, or the *operator* (*op-batcher* in this topology) that publishes *batches* from Layer2 to Layer 1. Therefore, apart from the smart contract that run on Ethereum, the key components explained in section **Section 4.1** are not written in *Solidity*.

4.2.1 Optimism Sequencer

For example, the *sequencer* in the standard topology in **Section 3.1.3** periodically checks the *inbox contract* in Layer 1 in order to process transactions from users that want to mint their assets to the Layer 2. In *Optimism*, this process of communication is done through the following scripts from the *Optimism repository*.

As we already know, smart contracts cannot initiate or call for actions off-chain. Therefore, off-chain components are needed in order to communicate operations and interpret events between layers. This communication between layers from Layer 1 to Layer 2 is principally made by the *sequencer*.

The *op-node* is the component responsible for deriving data from Layer 1 to the Layer 2 state. In the script *channel.go* from the *op-node* component, the node processes and decodes data from Layer 1 by interacting with the “channels”. These “channels” include the deposit requests from the users as shown in *map[uint64]Frame* in **Figure 4.3**, which will be transferred to Layer 2, reproducing the user assets from Layer 1 to Layer 2.


```

// A Channel is a set of batches that are split into at least one, but possibly multiple frames.
// Frames are allowed to be ingested out of order, unless the channel is set to follow Holocene
// rules.
// Each frame is ingested one by one. Once a frame with `closed` is added to the channel, the
// channel may mark itself as ready for reading once all intervening frames have been added
type Channel struct {
    // id of the channel
    id          ChannelID
    openBlock   eth.L1BlockRef
    requireInOrder bool

    // estimated memory size, used to drop the channel if we have too much data
    size uint64

    // true if we have buffered the last frame
    closed bool

    // highestFrameNumber is the highest frame number yet seen.
    highestFrameNumber uint16

    // endFrameNumber is the frame number of the frame where `isLast` is true
    // No other frame number must be larger than this.
    endFrameNumber uint16

    // Store a map of frame number -> frame for constant time ordering
    inputs map[uint64]Frame

    highestL1InclusionBlock eth.L1BlockRef
}

```

Figure 4.3: A code excerpt from `channel.go` that defines the structure of an OP Stack “channel” [33]

Furthermore, the script `driver.go` from the `op-program` directory is used to interpret the data sent from Layer 1, continuously fetching and “deriving” new requests from Layer 1 for executing new transactions on Layer 2 through a “pipeline”, as shown in **Figure 4.4**.

```

func NewDriver(logger log.Logger, cfg *rollup.Config, l1Source derive.L1Fetcher,
    l1BlobsSource derive.L1BlobsFetcher, l2Source engine.Engine, targetBlockNum uint64) *Driver {

    d := &Driver{
        logger: logger,
    }

    pipeline := derive.NewDerivationPipeline(logger, cfg, l1Source, l1BlobsSource, altda.Disabled, l2Source, metrics.NoopMetrics, false)
    pipelineDeriver := derive.NewPipelineDeriver(context.Background(), pipeline)
    pipelineDeriver.AttachEmitter(d)
}

```

Figure 4.4: A code excerpt from `driver.go` that defines the OP Stack “pipeline” [34]

In this way, the *sequencer* must operate outside the blockchain itself working as agents that observe and process on-chain events for the sake of communication between layers, which is a feature Solidity smart contracts cannot do on their own.

4.3 OP Stack bridge contract

The *Optimism Mainnet* includes *bridge* contract so that users can transfer assets from Layer 1 to Layer 2. In the *OP Stack* implementation, this is done through the *L1StandardBridge.sol* from the *OP repository* [35].

In the smart contract, the method *depositETHTo()* allows the user to deposit ETH from Layer 1 to a specific address in Layer 2. This is made through a call to an internal function named *_initiateETHDeposit()*, which prepares the message to be passed to the Layer 2 as shown in **Figure 4.5**.

```
/// @custom:legacy
/// @notice Deposits some amount of ETH into a target account on L2.
///      Note that if ETH is sent to a contract on L2 and the call fails, then that ETH will
///      be locked in the L2StandardBridge. ETH may be recoverable if the call can be
///      successfully replayed by increasing the amount of gas supplied to the call. If the
///      call will fail for any amount of gas, then the ETH will be locked permanently.
/// @param _to      Address of the recipient on L2.
/// @param _minGasLimit Minimum gas limit for the deposit message on L2.
/// @param _extraData Optional data to forward to L2.
///      Data supplied here will not be used to execute any code on L2 and is
///      only emitted as extra data for the convenience of off-chain tooling.
function depositETHTo(address _to, uint32 _minGasLimit, bytes calldata _extraData) external payable {
    _initiateETHDeposit(msg.sender, _to, _minGasLimit, _extraData);
}
```

Figure 4.5: A code snapshot from *L1StandardBridge.sol* smart contract that defines *depositETHTo()* method [35]

Another key factor of the smart contract is the *ETHDepositInitiated()* event that is emitted when a deposit is initiated as shown in **Figure 4.6**. In this way, other system components such as the *sequencer* will monitor these Layer 1 to Layer 2 transfer.

```
contract L1StandardBridge is StandardBridge, ProxyAdminOwnedBase, ReinitializableBase, ISemver {  
    /// @custom:legacy  
    /// @notice Emitted whenever a deposit of ETH from L1 into L2 is initiated.  
    /// @param from      Address of the depositor.  
    /// @param to        Address of the recipient on L2.  
    /// @param amount     Amount of ETH deposited.  
    /// @param extraData  Extra data attached to the deposit.  
    event ETHDepositInitiated(address indexed from, address indexed to, uint256 amount, bytes extraData);  
}
```

Figure 4.6: A code snapshot from *L1StandardBridge.sol* smart contract that defines *ETHDepositInitiated()* event [35]

Chapter 5. Test Environment and Evaluation

The main goal of this experimental section is to evaluate empirically the performance and efficiency of Optimistic rollups on Layer 2 compared to the same executions and transactions on Ethereum Layer 1, providing the necessary information and results in order to answer if rollups are a proper scaling solution for Ethereum or not.

To do so, the evaluation is carried out using public Ethereum testnets. Testnets are blockchain networks that replicate the behaviour of a real mainnet environment but using tokens that has no value. Since they are real networks, testnets allows developers to test smart contracts and applications in a realistic and safe environment. For example, the *Ethereum Sepolia testnet* currently works with 1,700 validator nodes [36], making it a realistic and reliable environment.

For each experimental evaluation some common metrics will be compared such as gas consumption, total price, fees paid, timestamp, etc, for equivalent operations. In this case, the testnets used are the *Ethereum Sepolia testnet as the Layer 1 mainnet*, while for the Layer 2 networks the *Optimism testnet* and *Arbitrum Sepolia testnet* were considered so as to have more comparison possibilities and therefore more precision in the evaluation process.

The aim is to demonstrate that Optimistic rollups can significantly reduce transaction costs, allowing a higher throughput compared to the mainnet in a safe way, as explained in **Section 3.4**, by performing a set of controlled and reproducible tests.

To this end, we first describe the preparation of the experiments in **Section 5.1**. Then, in the first experiment, described in **Section 5.2**, we consider the case of plain ether transactions with no contract addresses involved. The second experiment, in **Section 5.3**, evaluates the case of a simple contract that performs a number of plain transactions. Finally, the third experiment included in **Section 5.4** evaluates of a more complex contract deployed on the Layer 2 network.

5.1 Preparation

Before conducting the experiments comparing the different performances between Layer 1 and Layer 2, a test environment was set up by means of public Ethereum testnets and developer tools. All the operations of the experiments were executed using *Sepolia testnet* as the Layer 1 mainnet, with the benefits of performing free of cost operations and real validators nodes in the system as in the mainnet, where all developers can perform testing operations without congestion issues.

To perform operations in the testnet, users need Sepolia ETH, which is the token used in the *Ethereum Sepolia testnet*. Since these tokens do not have a real value, they are freely distributed through online faucets, which usually require users to complete a human verification process to prevent automated activity.

In addition, a public faucet was used to “mine” the required *Sepolia ETH* in order to later execute the transactions on Layer 2, which can be found in [37]. However, to perform transactions on Layer 2, the use of a *bridge contract* is necessary to mint the *Sepolia ETH* from one layer to the other. For instance, the *Optimism testnet bridge contract* can be found in [38], while the *Arbitrum Sepolia bridge contract* can be found in [39], allowing us to interact with the Layer 2.

Finally, in order to interact with the networks and perform operations with the *Sepolia ETH*, the *Optimism testnet ETH* and the *Arbitrum Sepolia ETH*, a *Metamask* wallet is used [40], where two different accounts are needed in order to execute plain transactions between them on Layer 1 and on Layer 2. Both *Optimism* and *Arbitrum* testnet networks were manually added to *Metamask* using the necessary RPC settings in [41].

In order to perform consistent and fair comparisons between the different networks, all calculations in this section are made with the same transaction costs in the same date, May 3, 2025. This piece of data can be seen in **Figure 5.1**, **Figure 5.2** and **Figure 5.3** for each network, respectively.

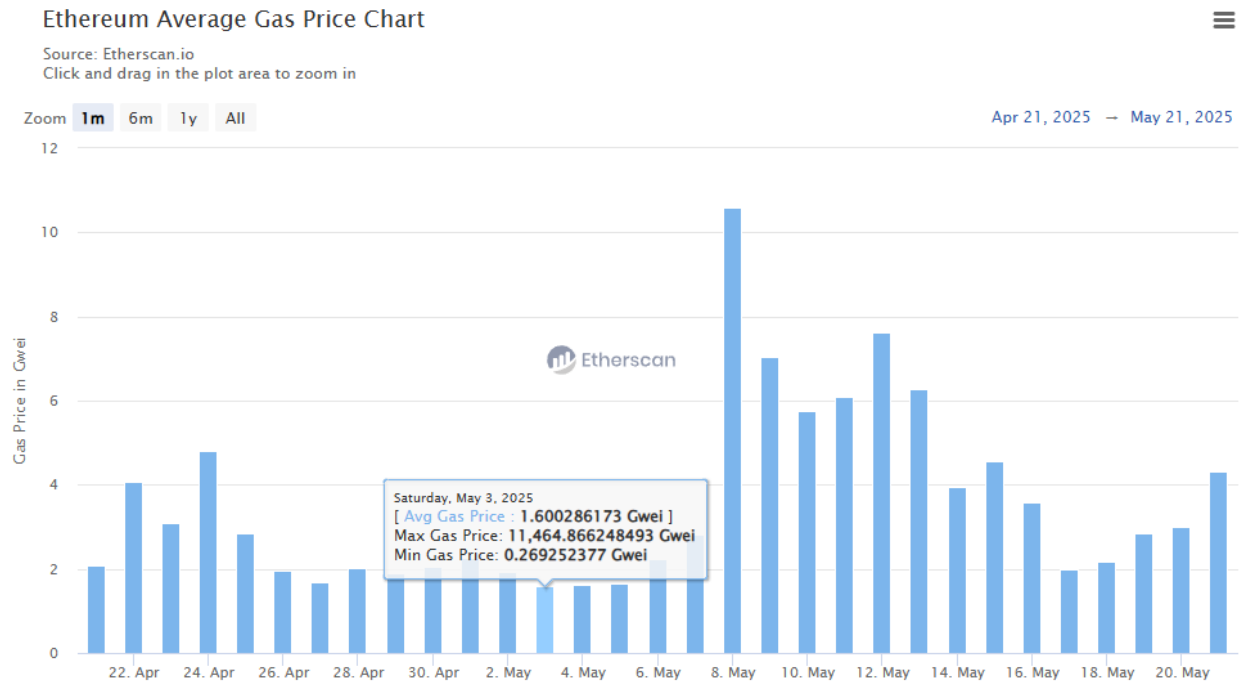


Figure 5.1: Ethereum Mainnet gas price metrics [42]

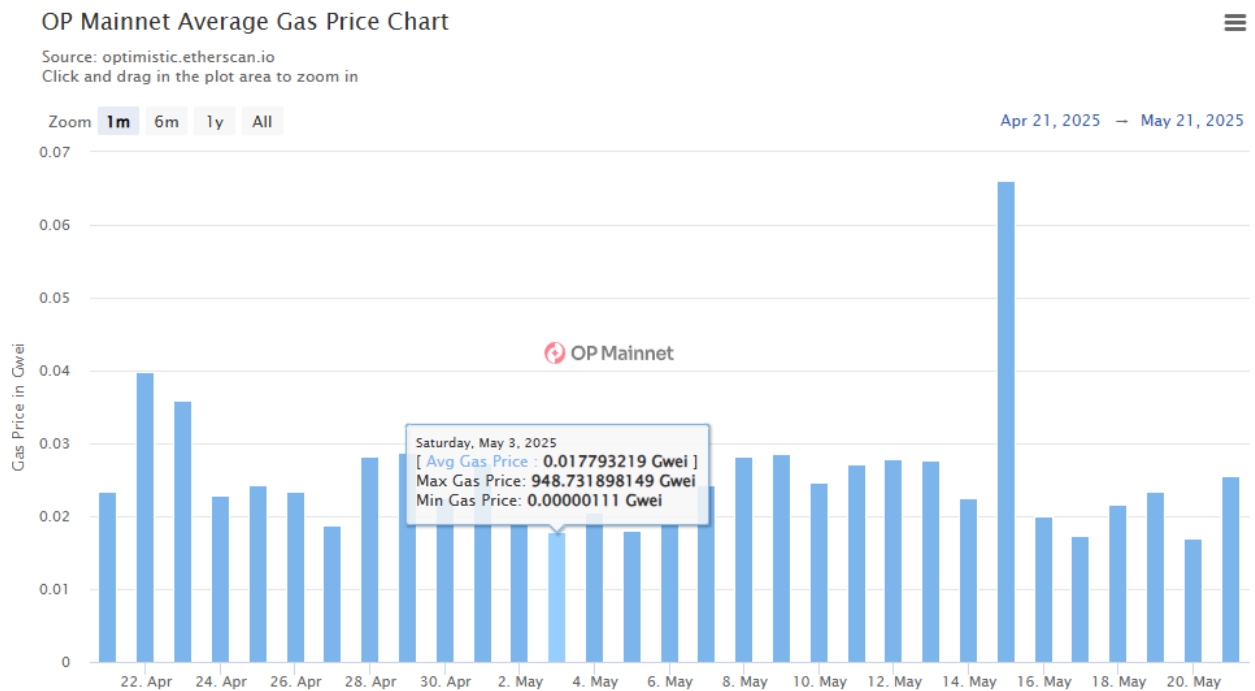


Figure 5.2: Optimism Mainnet gas price metrics [43]

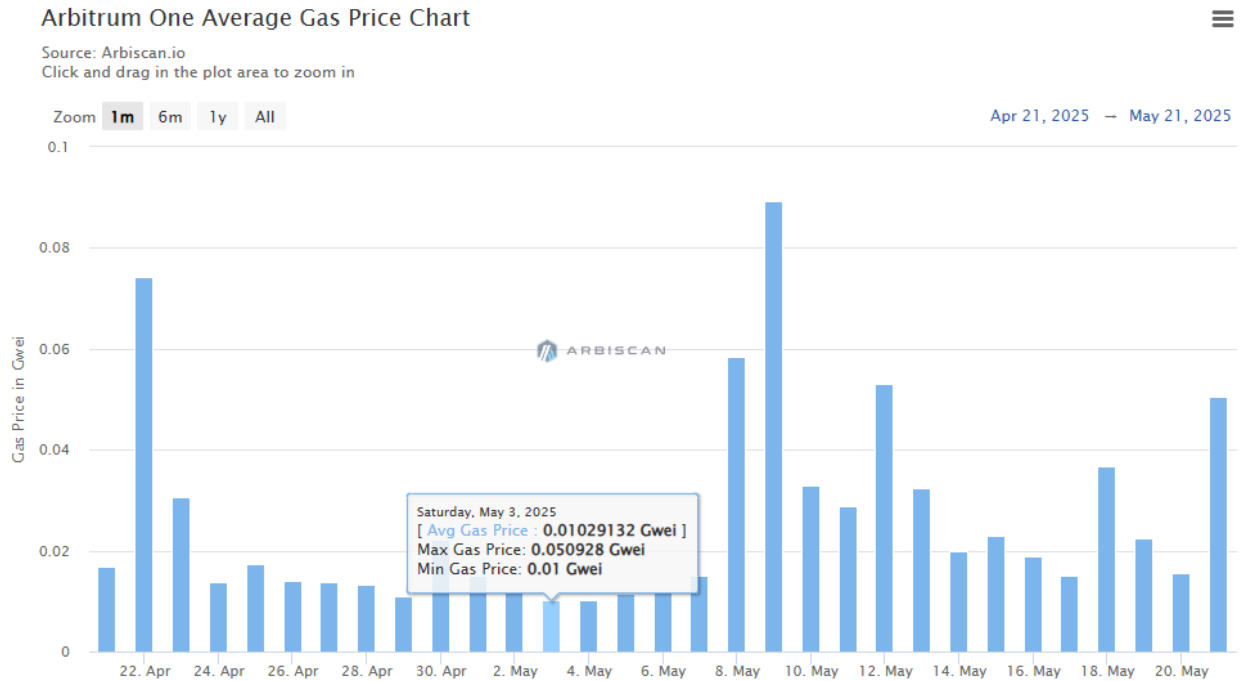


Figure 5.3: Arbitrum Mainnet gas price metrics [44]

Observe that the gas prices in **Figure 5.1**, **Figure 5.2** and **Figure 5.3** correspond to Ethereum, Optimism and Arbitrum mainnets instead of their testnets. In the experiments performed in this section, we will apply these gas prices to the gas amounts obtained in the tests made on the testnets.

In this way, the results are as precise as possible since they are not altered by temporary networks fluctuations, providing a fair reference for measuring the cost differences between Layer 1 and Layer 2 solutions. Moreover, the price of one ETH is also taken into account, with a price of 1,833.95\$ USD at date May 3, 2025 [45].

All this information is summarized in **Table 5.1**, which represents the average gas price in Gwei units. Hence, the data for the calculations is shown in **Table 5.1**.

Network	Layer	TPS	Gas Price	ETH Price (USD)
Ethereum Mainnet	L1	11.7	1.600286173 (Gwei)*	1,833.95 \$/ETH
Optimism Mainnet	L2	12.2	0.017793219 (Gwei)*	1,833.95 \$/ETH
Arbitrum Mainnet	L2	14	0.01029132 (Gwei)*	1,833.95 \$/ETH

Table 5.1: Calculations metrics and data at date May 3, 2025

* The conversion is: 1 *Gwei* = 1,000,000,000 *wei* (10^9 *wei*)

5.2 Plain transactions

This section covers the direct ETH transactions, also known as *plain transactions*, which are the simplest type of experiment, where an exchange of ether occurs between two addresses and no smart contracts are needed for now. In the following subsections we evaluate plain transactions between EOA accounts (externally owned accounts) in both Layer 1 and Layer 2 testnets and then the results obtained are interpreted.

5.2.1 Sepolia Layer 1

The Layer 1 transaction was directly executed using *Metamask*, from our main *Metamask* account to a second account with no ETH.

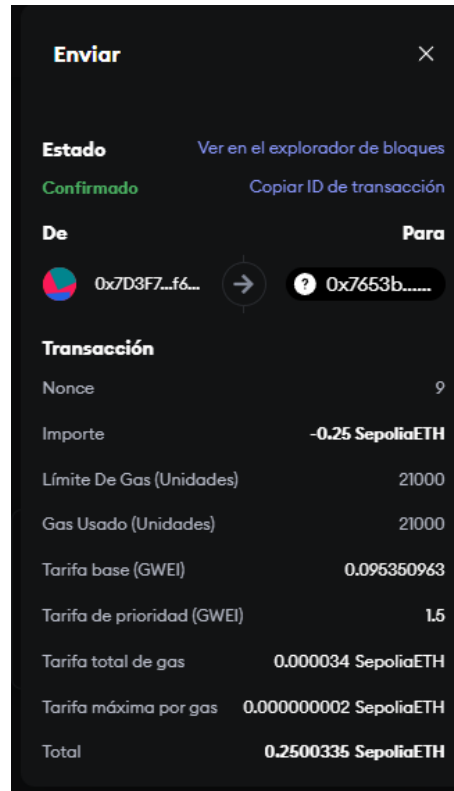


Figure 5.4: Transaction information from a Layer 1 to Layer 1 transaction through Metamask

Figure 5.4 shows that the transaction consumed the minimum amount of gas for a transaction, which is 21,000 gas units. If we access the transaction directly on *etherscan* [46], more information is displayed about the transaction in **Figure 5.5**, including a more precise amount of the final fee paid.

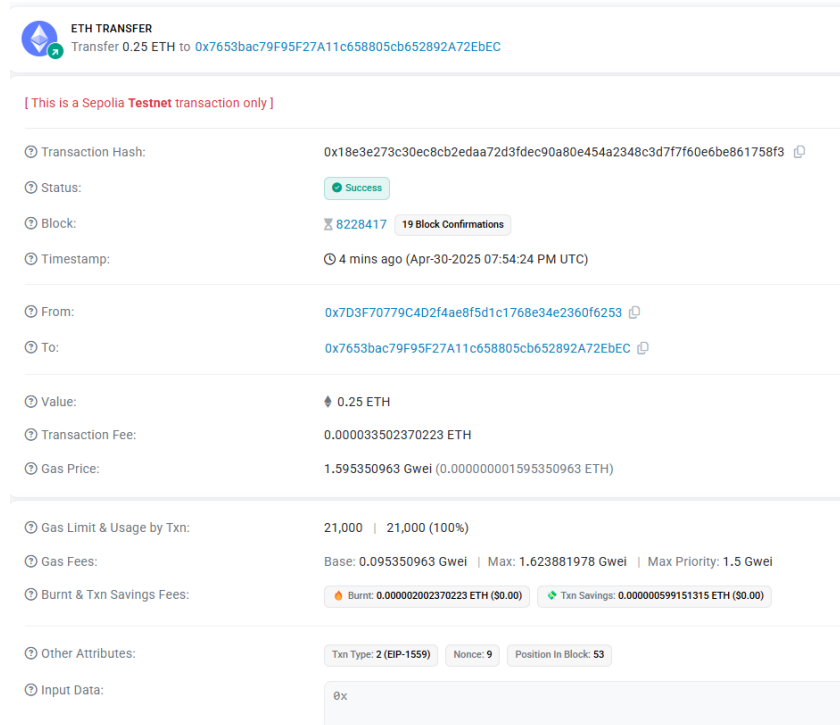


Figure 5.5: An etherscan analysis of a direct transaction on the Sepolia testnet [46]

5.2.2 Optimism testnet

In order to interact with the *Optimism* Layer 2 network, it is necessary to mint ETH from *Sepolia* using the *bridge contract*. Once the funds are received in the same address of the Layer 2, the transaction to the second address, which also exists in the *Optimism* Layer 2, is executed using *Metamask* as shown in **Figure 5.6**.

It is important to bear in mind that the current *Optimism testnet* does not allow to withdraw ETH from Layer 2 to Layer 1, as the *Optimism Mainnet* does. This is an issue that a great deal of developers have complaint about, but it has not been solved as for the date of this document. For this reason, the withdrawal of funds is not taken into account this time.

Firstly, some ETH is sent to the *bridge contract*:

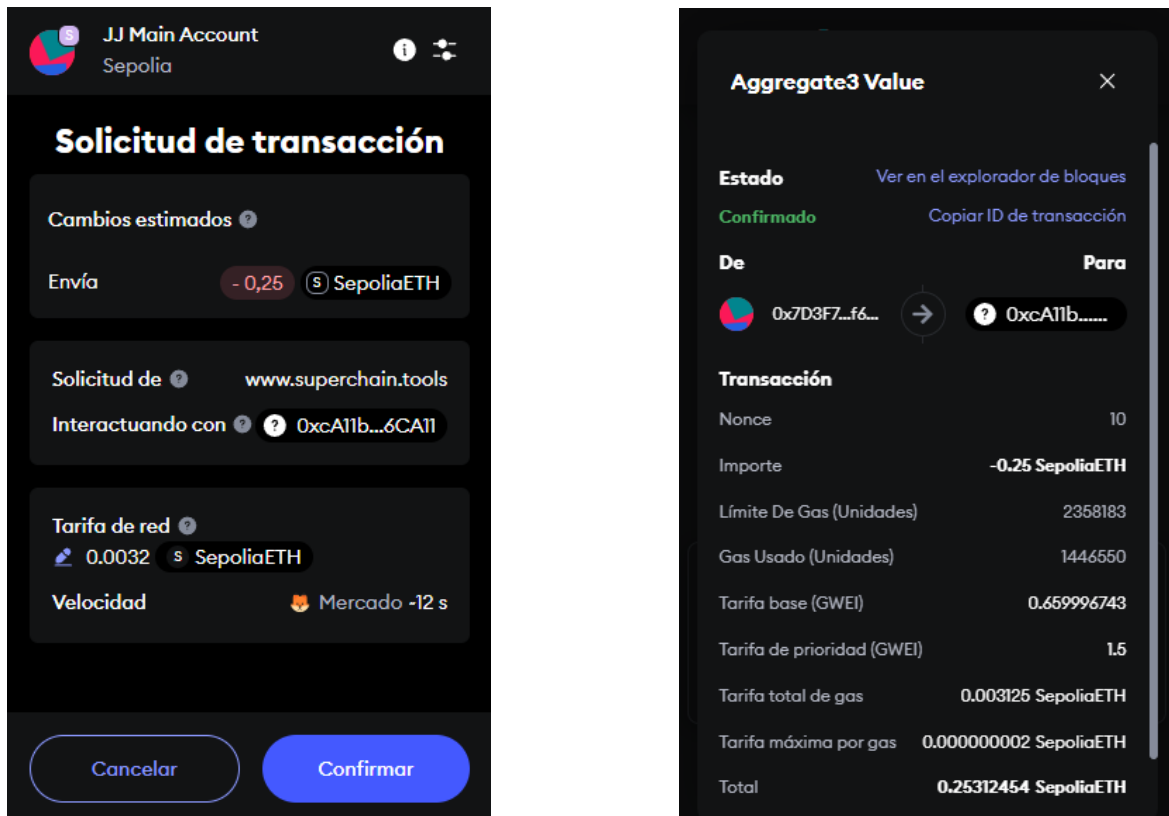


Figure 5.6: A plain transaction from Layer 1 mainnet to the Optimism bridge contract (LEFT) and transaction details from a transaction to the Optimism bridge contract (RIGHT).

Now that there is ETH in the *Optimism testnet*, the transaction is executed **Figure 5.7:**

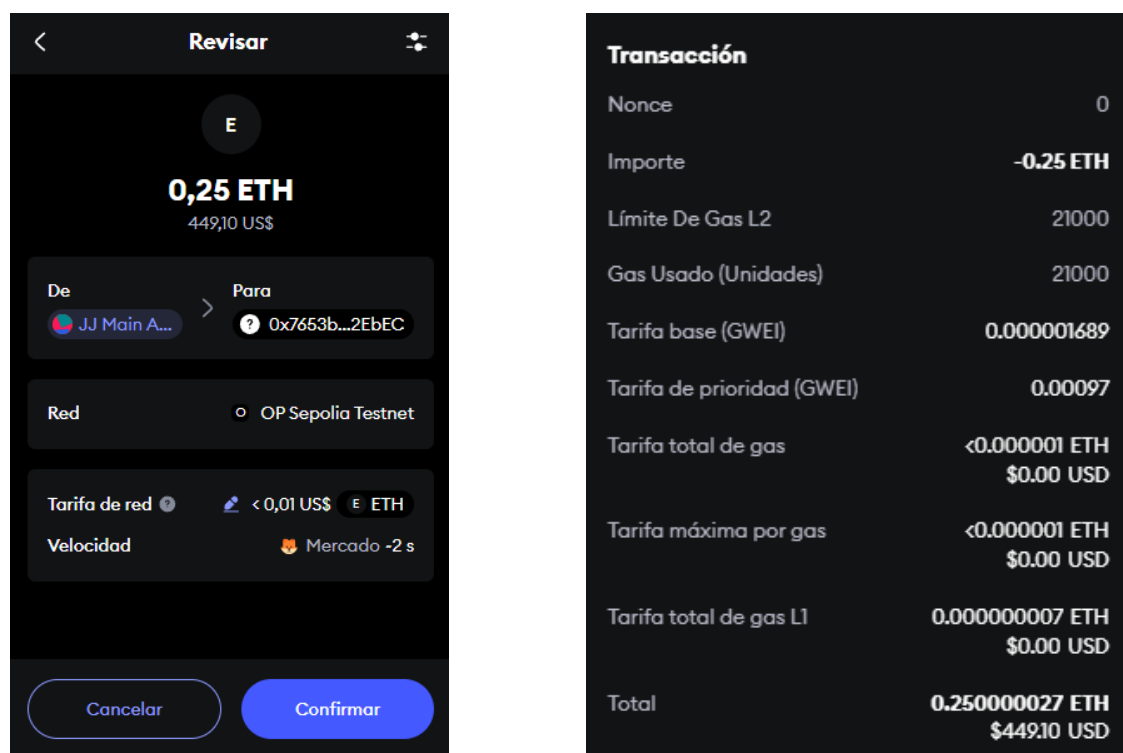


Figure 5.7: An ETH transfer to a second address in the OP Sepolia Testnet (LEFT) and details of a transaction to the Optimism bridge contract (RIGHT)

The transaction can be consulted using *Etherscan*, in **Figure 5.8**:

[This is a OP Sepolia Network Testnet transaction only]	
Transaction Hash:	0xd081312b6f1ea96406e431aaa8dce130b89132c9ae21860bc0c59beed2e8cbe8
Status:	Success
Block:	27122209 Confirmed by Sequencer
Timestamp:	51 secs ago (Apr-30-2025 09:02:38 PM +UTC)
From:	0x7D3F70779C4D2f4ae8f5d1c1768e34e2360f6253
To:	0x7653bac79f95f27A11c658805cb652892A72EBeC
Value:	0.25 ETH
Transaction Fee:	0.00000002704624851 ETH (\$0.00005)
Gas Price:	0.000971689 Gwei (0.000000000000971689 ETH)
Gas Limit & Usage by Txn:	21,000 21,000 (100%)
Gas Fees:	Base: 0.000001689 Gwei Max: 0.000975058 Gwei Max Priority: 0.00097 Gwei
L2 Fees Paid:	0.000000020405469 ETH
L1 Fees Paid:	0.00000000664077951 ETH
L1 Gas Price:	0.000000000546116729 ETH (0.546116729 Gwei)
L1 Gas Used by Txn:	1,600
L1 Fee Scalar:	0

Figure 5.8: An etherscan analysis of a direct transaction on the OP Sepolia testnet

All in all, 1,446,550 gas units were needed to get the *Sepolia ETH* from Layer 1 to the Layer 2 by means of the *bridge contract*. Furthermore, the transaction on Layer 2 also consumes 21,000 gas units.

5.2.3 Arbitrum Sepolia

The same transaction has been performed on *Arbitrum Sepolia*. Similarly to *Optimism*, the *Arbitrum Sepolia bridge contract* is also needed to interact with the *Arbitrum Sepolia testnet*. Just like before, the funds are received in the same address but of the Layer 2. In contrast to *Optimism*, the minting process does not happen immediately, the user must wait 10 minutes. **Figure 5.9** shows the details of this transaction.

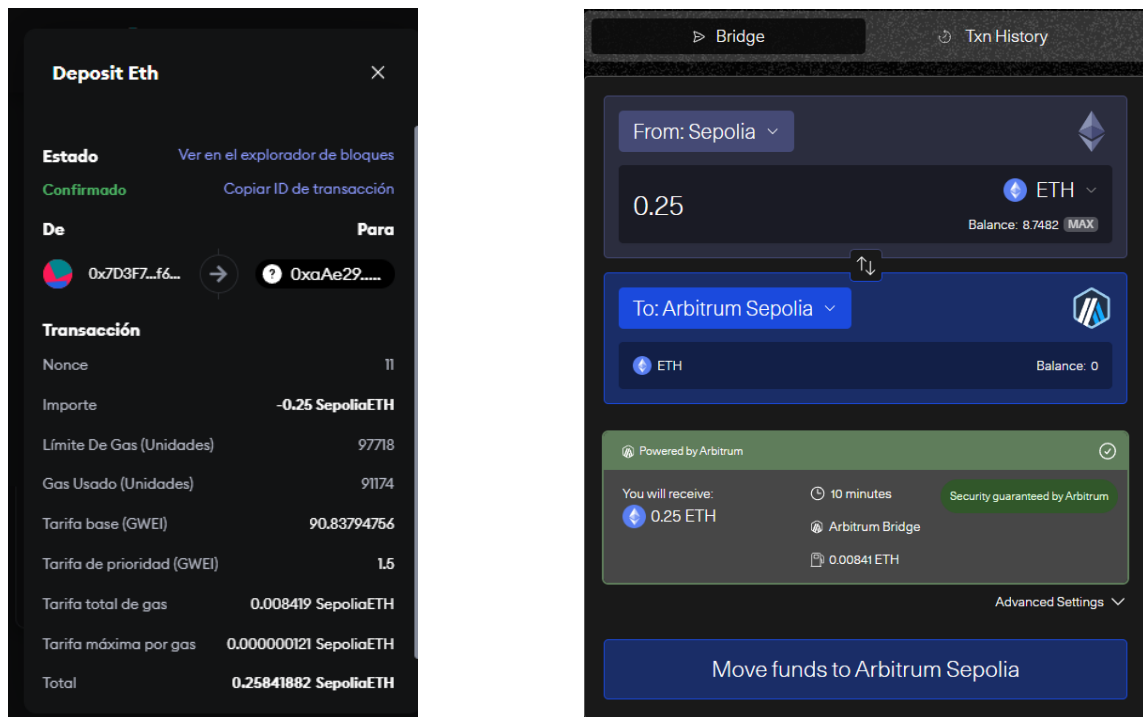


Figure 5.9: Transaction details from a transaction to the Arbitrum bridge contract (LEFT) and the Arbitrum bridge contract user interface (RIGHT)

For this case, the *bridge contract* allows the developer to perform a withdrawal of the testnet Layer 2 funds. However, it is not possible to connect the *Arbitrum Sepolia testnet* to *Metamask* thus the transaction cannot be tested on Layer 2 for this case, as shown in **Figure 6.4**. This time, 91,174 gas units were needed to get the *Sepolia ETH* from Layer 1 to Layer 2 through the *bridge contract*. Since there are no executed transactions, no comparison can be made.

To perform the withdrawal in this *bridge contract*, there is not a real *challenge period* of 7 days to be done, however, the user has to pay two fees in order to receive its funds and wait for around an hour to have the *Sepolia ETH* back which works as a simulation of the *challenge period* as shown in **Figure 5.10** and **Figure 5.11**.

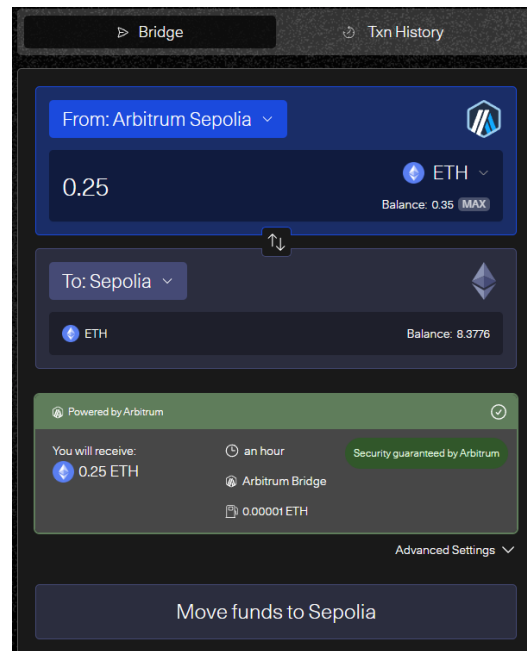
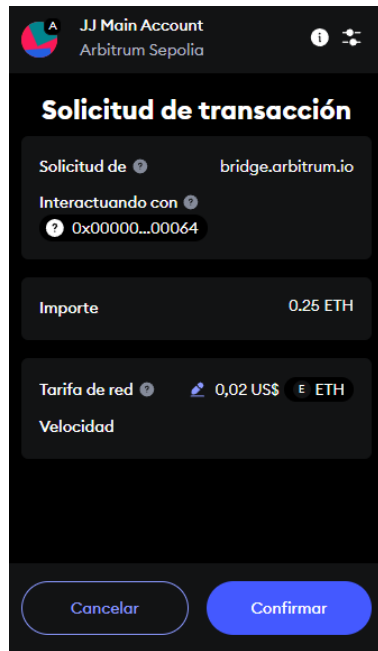


Figure 5.10: Transaction request to withdraw from Arbitrum Sepolia testnet through Metamask (LEFT) and Arbitrum bridge contract transaction from Layer 2 Arbitrum Sepolia testnet to Layer 1 Sepolia testnet (RIGHT)

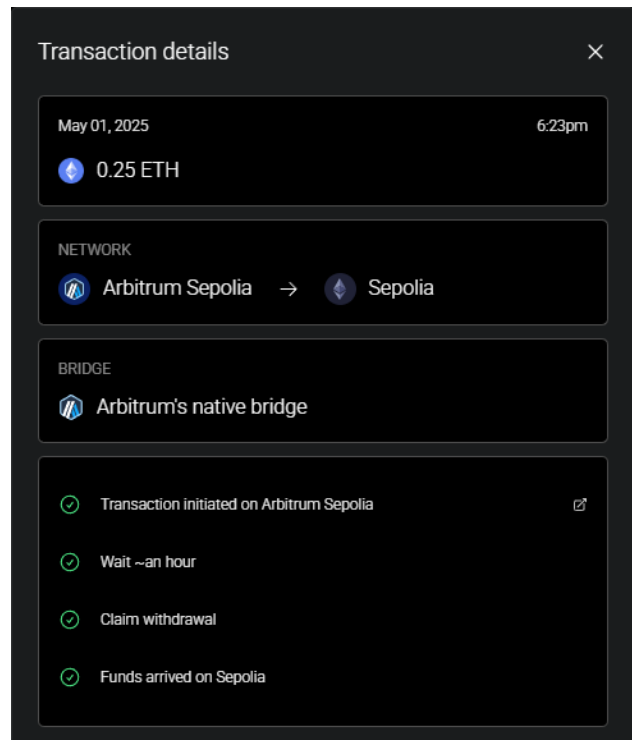


Figure 5.11: Second transaction withdraw made Layer 1 Sepolia testnet through Metamask (LEFT) and Arbitrum bridge contract confirmation of withdraw (RIGHT)

Therefore, performing the withdraw process has a negligible value from the *Arbitrum Sepolia testnet*. Moreover, an hour is needed to withdraw the funds, simulating the *challenge period*.

5.2.4 Results

Firstly, a simple transaction on Layer 1 can consume up to 21,000 gas units, which, with the record gas price on **Table 5.1** of 1.600286173 Gwei. On the other hand, the same transaction on the Layer 2 of *Optimism* has a consumption of 1,446,550 gas units, with an approximated gas price of 0.017793219 Gwei according to the data on the same table. The formula to compare the transaction in each network is the following:

$$ETH\ Cost = \frac{gas\ used \times gas\ price}{10^9}$$

For the Layer 1 the cost in ETH with the reference value results in:

$$ETH\ Cost = \frac{21,000\ gas \times 1.600286173\ Gwei}{10^9} = 0.000033606\ ETH$$

While for the *Optimism* Layer 2 the cost is:

$$ETH\ Cost = \frac{21,000\ gas \times 0.017793219\ Gwei}{10^9} = 0.00000037365\ ETH$$

As expected, executing the transaction on Layer 2 is much cheaper than executing it directly on Layer 1. However, the cost of using the *bridge contract* to interact with the Layer 2 on the minting and withdrawing processes must be taken into account. Therefore, the cost for sending ETH to the Layer 2 on *Optimism* is:

$$ETH\ Cost = \frac{1,446,550\ gas \times 1.600286173\ Gwei}{10^9} \approx 0.00231489\ ETH$$

Which represented in USD is a cost of:

$$USD\ Cost = 0.00231489\ ETH \times 1,833.95\ \$/_{ETH} \approx 4.245\ \$$$

While for Arbitrum Layer 2 is:

$$ETH\ Cost = \frac{91,174\ gas \times 1.600286173\ Gwei}{10^9} \approx 0.000146\ ETH$$

Which represented in USD is a cost of:

$$USD\ Cost = 0.000146\ ETH \times 1,833.95\ \$/_{ETH} \approx 0.268\ \$$$

As we can see, it is really expensive to transfer data from one layer to another. However, the *Arbitrum bridge contract* consumes significantly less gas units than the *Optimism bridge contract* when transferring data from the Layer 1 to the Layer 2. This is because each *bridge contract* has its own design, and in a *bridge contract* there are multiple storage slots, emitted events containing a calldata or other pieces of information depending on the design, so the *Arbitrum bridge contract* is simplified compared to the one of *Optimism*, doing minimal work on chain.

Despite the fact that very few gas units were consumed when executing the transaction on Layer 2, it is important to bear in mind the huge amount of gas units needed to move ETH between testnets. Therefore, if we compare these results to the 21,000 gas units needed to perform the transaction on the mainnet, it is not profitable to use optimistic rollups to perform only one transaction. It will be worthy when a large volume of transactions is executed, in such a way that if all transactions were done directly on the mainnet, the total amount of gas used would have exceeded the gas consumed by the *bridge contract*.

Since the *Optimism bridge contract* does not allow the user to withdraw funds from Layer 2 to Layer 1, it has not been possible to measure its cost. Therefore, despite the fact that withdrawing funds from Layer 2 to Layer 1 is much cheaper than sending funds from Layer 1 to Layer 2 as shown in the *Arbitrum Sepolia bridge contract* in **Figure**

5.10, a symmetrical cost is assumed for the withdraw operation. In this way, although in practice sending assets from Layer 2 to Layer 1 is much cheaper than sending assets from Layer 1 to Layer 2, this allows to compute estimations with an upper bound of the worst possible case.

Hence, an estimation of the total cost of the process, including sending ETH to Layer 2, executing the transaction, and transfer and publish the data from Layer 2 to Layer 1 has a total final cost of:

$$\begin{aligned} &L1 \text{ to } L2 \text{ bridge contract} + \text{Layer 2 transactions} + L2 \text{ to } L1 \text{ bridge contract} \\ &= 4.245 \$ + (\approx 0.00) + 4.245 \$ \cong 8.50 \$ \end{aligned}$$

5.3 Smart contract deployment with multiple plain transactions

In this section a more realistic experiment is carried out in order to get a precise evaluation of the efficiency of *Optimistic rollups* in a real-world context.

5.3.1 Experimental smart contract

The following smart contract in **Figure 5.12** contains the code in *Solidity* to perform a number of transactions consecutively with a predefined *amount* in each transaction to a *receiver* address. The *smart contract* firstly initializes these variables with the user input as long as the contract has enough ETH to make all transfers. Then, the *run* method can be executed to perform the transfers.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract RepeatedSender {
    address public receiver;
    uint256 public amount;
    uint256 public totalTransactions;

    constructor(address _receiver, uint256 _amount, uint256 _nTransactions) payable {
        require(msg.value >= _amount * _nTransactions, "ERROR: Incorrect ETH sent");

        receiver = _receiver;
        amount = _amount;
        totalTransactions = _nTransactions;
    }


    function run() public {  infinite gas
        for (uint256 i = 0; i < totalTransactions; i++) {
            payable(receiver).transfer(amount);
        }
    }
}
```

Figure 5.12: Smart contract for sending multiple transactions

5.3.2 Sepolia Layer 1

Firstly, the smart contract is deployed and executed on the *Sepolia testnet* to obtain the non-optimized data.

5.3.2.1 Smart contract Deployment (with a single transaction)

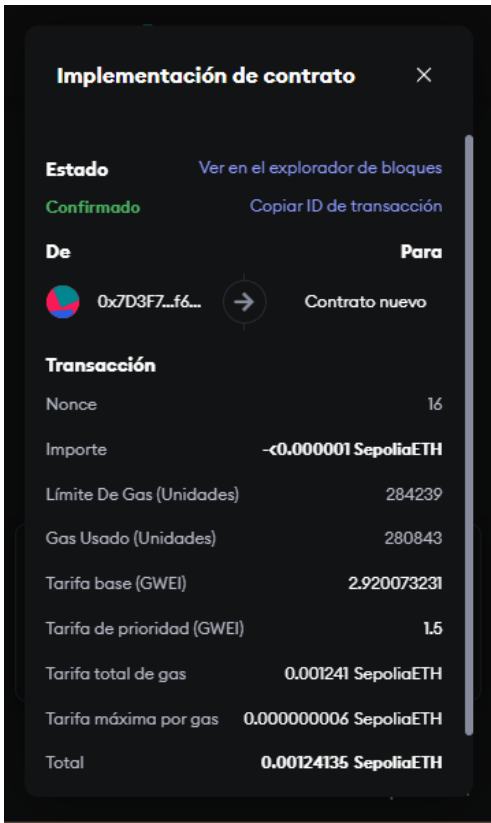


Figure 5.13: Deployment costs of a smart contract on Sepolia testnet from Metamask

According to **Figure 5.13**, deploying a smart contract on the *Sepolia testnet* has a cost of around 281,000 gas units.

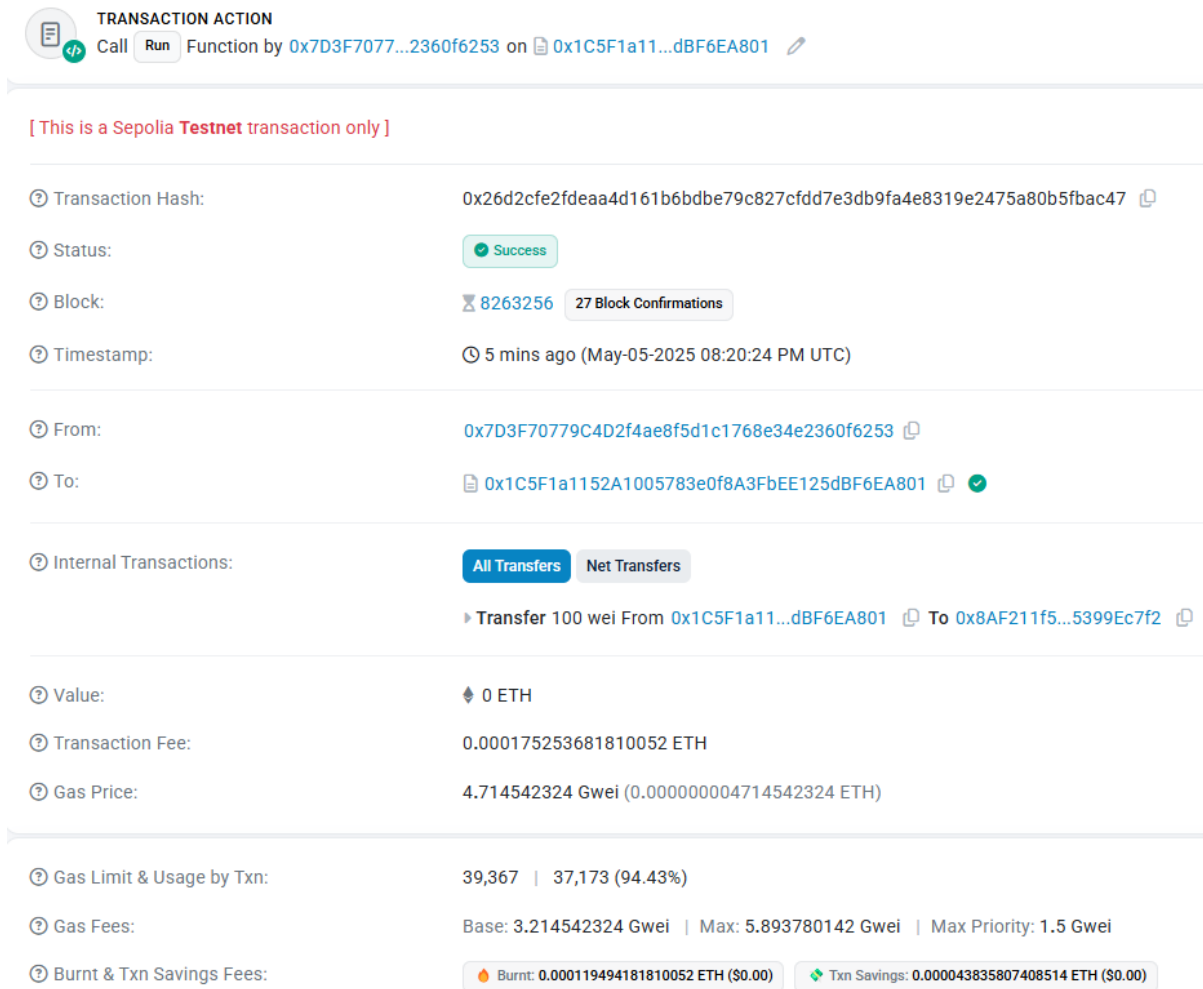


Figure 5.14: An etherscan snapshot of the execution of a single transaction smart contract on Sepolia testnet

After executing the **run()** method of the smart contract with a single transaction, around 37,000 gas units are consumed as shown in **Figure 5.14**. If we take into account the data from previous **Section 5.2**, the cost of a single transaction is 21,000 gas units, thus executing a plain transaction using a smart contract has additional costs caused by the execution of contract code.

5.3.2.2 Executing multiple plain transactions

On the other hand, to prove in future sections the efficiency of *Optimistic rollups* by using a larger volume of transactions, the same smart contract is executed with 1,000 transactions instead, in order to understand how the costs increase depending on the number of transactions.

Deploying the contract has a similar cost close to 281,000 gas units, which is the same cost as before for a single transaction. Furthermore, to execute the **run()** method of the contract, it is necessary to increase the gas *limit* so that the contract does not run out of gas while executing the transactions and aborts the process. In this case, the gas *limit* is increased to 20,000,000 gas units as shown in **Figure 5.15**.

The image shows a screenshot of an Etherscan transaction page for a Sepolia Testnet transaction. The transaction is titled "TRANSACTION ACTION" and is a "Call" to the "Run" function of a smart contract. The transaction hash is 0xc4e5bae6734c57c7561a85ee578fa0398268c1a70647776bc8c089930c818694. The status is "Success" with 31 block confirmations. The transaction was executed 6 minutes ago on May-06-2025 at 10:13:24 AM UTC. The transaction is from 0x7D3F70779C4D2f4ae8f5d1c1768e34e2360f6253 to 0x4E1a535399Cb341e863720CC7De3cca8658D1514. The transaction value is 0 ETH. The transaction fee is 0.531824918523531432 ETH. The gas price is 72.644924276 Gwei. The gas limit is 20,000,000, and the gas usage is 7,320,882 (36.6%). The gas fees are Base: 71.144924276 Gwei, Max: 97.764903565 Gwei, and Max Priority: 1.5 Gwei. The burnt and transaction savings fees are 0.520843595523531432 ETH (\$0.00) and 0.183900404217212898 ETH (\$0.00) respectively.

TRANSACTION ACTION	
Call Run Function by 0x7D3F7077...2360f6253 on 0x4E1a5353...8658D1514	
[This is a Sepolia Testnet transaction only]	
Transaction Hash:	0xc4e5bae6734c57c7561a85ee578fa0398268c1a70647776bc8c089930c818694
Status:	Success
Block:	8267340 31 Block Confirmations
Timestamp:	6 mins ago (May-06-2025 10:13:24 AM UTC)
From:	0x7D3F70779C4D2f4ae8f5d1c1768e34e2360f6253
To:	0x4E1a535399Cb341e863720CC7De3cca8658D1514
Internal Transactions:	<div>All Transfers Net Transfers</div> <div>0x4E1a5353...8658D1514 sent 0.000000000000005 ETH</div> <div>0x8AF211f5...5399Ec7f2 received 0.000000000000005 ETH</div> <div>Scroll for more</div>
Value:	0 ETH
Transaction Fee:	0.531824918523531432 ETH
Gas Price:	72.644924276 Gwei (0.000000072644924276 ETH)
Gas Limit & Usage by Txn:	20,000,000 7,320,882 (36.6%)
Gas Fees:	Base: 71.144924276 Gwei Max: 97.764903565 Gwei Max Priority: 1.5 Gwei
Burnt & Txn Savings Fees:	<div>Burnt: 0.520843595523531432 ETH (\$0.00)</div> <div>Txn Savings: 0.183900404217212898 ETH (\$0.00)</div>

Figure 5.15: An etherscan snapshot of the execution of a thousand transactions smart contract on Sepolia testnet

However, when the **run()** method of the smart contract is executed with 1,000 transactions, close to 7,321,000 gas units are consumed. If we compute the cost per transaction, we obtain:

$$\frac{7,321,000 \text{ gas units}}{1,000 \text{ transaction}} = 7,321 \text{ gas for each transaction}$$

Hence, since a single transaction through the smart contract consumed around 37,000 gas units, as the number of transactions is increased, the gas per transaction is decreased, since executing transactions in a loop share the same computations at a low-level so a lot of processes can be reused. In fact, the cost of executing the contract as a whole as far as gas cost is concerned with the reference values from **Table 5.1** is:

$$USD \text{ Cost} = \frac{7,321,000 \text{ gas} \times 1.600286173 \text{ Gwei}}{10^9} (ETH) \times 1,833.95 \text{ \$/ETH} \approx 21.50 \$$$

Moreover, according to **Figure 5.15**, the **Table 5.1** and the assumption of 1 Gwei as Max Priority [47], the transaction fee of the operation is:

$$transaction \text{ fee} = \frac{(1.600286173 \text{ Gwei} + 2 \text{ Gwei}) \times 7,320,882 \text{ gas}}{10^9} = 0.02636 \text{ ETH}$$

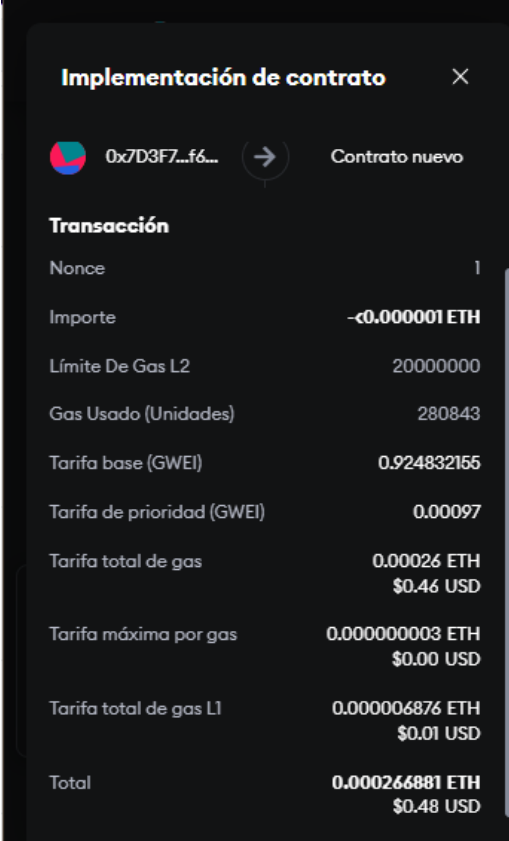
The *transaction fee* of the operation is of around 0.02636 ETH is paid so as to execute all the transaction of the contract, which is a huge amount of ETH in fees for executing plain transactions. This number translated to USD cost with the data from **Table 5.1** is around:

$$USD \text{ Cost} = 0.02636 \text{ ETH} \times 1,833.95 \text{ \$/ETH} \approx 48.34 \$$$

5.3.3 Optimism testnet

The smart contract is also deployed on the *Optimism Sepolia testnet*, which is the Layer 2 network where the *Optimistic Rollup* architecture is used to execute the transactions at a lower cost.

5.3.3.1 Smart contract Deployment (with a single transaction)



Implementación de contrato	
0x7D3F7...f6...	Contrato nuevo
Transacción	
Nonce	1
Importe	-0.000001 ETH
Límite De Gas L2	20000000
Gas Usado (Unidades)	280843
Tarifa base (GWEI)	0.924832155
Tarifa de prioridad (GWEI)	0.00097
Tarifa total de gas	0.00026 ETH \$0.46 USD
Tarifa máxima por gas	0.000000003 ETH \$0.00 USD
Tarifa total de gas L1	0.000006876 ETH \$0.01 USD
Total	0.000266881 ETH \$0.48 USD

Figure 5.16: Deployment costs of a smart contract on Optimism Sepolia from Metamask

According to **Figure 5.16**, deploying a smart contract on the *Optimism Sepolia testnet* has a cost of around 281,000 gas units, which is the same gas units as deploying the Smart Contract on the mainnet. However, considering the values of **Table 5.1**, at the end the deployment is done at a lower cost on Layer 2.

[This is a OP Sepolia Network **Testnet** transaction only]

Transaction Hash:	0x0f69266213ba868067a409c067801e8f607b430c5663fdf890f02fd3f1298ca1
Status:	Success
Block:	27363315 Confirmed by Sequencer
Timestamp:	2 mins ago (May-06-2025 10:59:30 AM +UTC)
From:	0x7D3F70779C4D2f4ae8f5d1c1768e34e2360f6253
To:	0x96571Dd0DDDA6588205237A261742841F7df5cBa
Internal Transactions:	<div>All Transfers Net Transfers</div> <div>Transfer 100 wei From 0x96571Dd0...1F7df5cBa To 0x8AF211f5...5399Ec7f2</div>
Value:	0 ETH
Transaction Fee:	0.000035816020209734 ETH (\$0.07)
Gas Price:	0.941775201 Gwei (0.000000000941775201 ETH)
Gas Limit & Usage by Txn:	20,000,000 37,173 (0.19%)
Gas Fees:	Base: 0.940805201 Gwei Max: 2.811560671 Gwei Max Priority: 0.00097 Gwei
L2 Fees Paid:	0.000035008609546773 ETH
L1 Fees Paid:	0.00000807410662961 ETH
L1 Gas Price:	0.00000066398903197 ETH (66.398903197 Gwei)
L1 Gas Used by Txn:	1,600
L1 Fee Scalar:	0

Figure 5.17: An etherscan snapshot of the execution of a single transaction smart contract on Optimism Sepolia testnet

After executing the **run()** method of the smart contract with a single transaction, around 37,000 gas units are consumed as well as shown in **Figure 5.17**, but the transaction fees are much lower than the ones paid on the mainnet, shown in **Figure 5.14**.

5.3.3.2 Executing multiple plain transactions

Deploying the smart contract on the Layer 2 has a cost of around 307,000 gas units as shown in **Figure 5.18**.

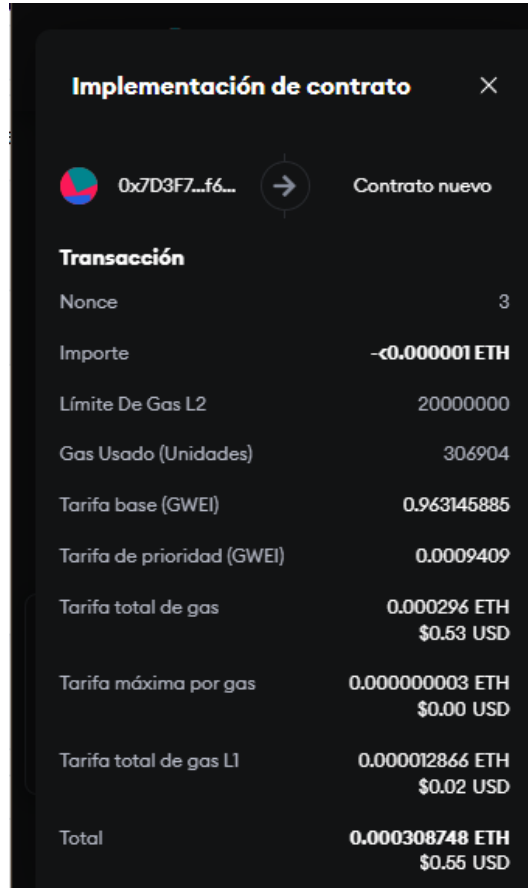


Figure 5.18: Deployment costs of a smart contract for a thousand transactions on Optimism Sepolia from Metamask

For this execution of the contract, the *gas limit* is also set to 20,000,000 gas units as shown in **Figure 5.18**. After the execution, 7,438,882 gas units are consumed, an amount slightly superior to the gas consumed on the mainnet for the same execution. As a whole, the amount consumed by each transaction has also been decreased because of the huge number of transactions executed, so a similar architecture is shared at low-level between layers in order to optimize this type of loops:

$$\frac{7,439,000 \text{ gas}}{1,000 \text{ transactions}} = 7,439 \text{ gas for each transaction}$$

Transaction Hash:	0x8bad22e5dcb92d2b0470fce83876b851bdba03a59c34d253221368b92a6255c6
Status:	Success
Block:	27364192 Confirmed by Sequencer
Timestamp:	4 mins ago (May-06-2025 11:28:44 AM +UTC)
From:	0x7D3F70779C4D2f4ae8f5d1c1768e34e2360f6253
To:	0x326EE31a83757ec496c50F0407eEB1A66C4dCf8C
Internal Transactions:	<div>All Transfers Net Transfers</div> <div>0x326EE31a...66C4dCf8C sent 0.000000000000005 ETH</div> <div>0x8AF211f5...5399Ec7f2 received 0.000000000000005 ETH</div>
Value:	0 ETH
Transaction Fee:	0.007256178196362547 ETH (\$13.43)
Gas Price:	0.975251796 Gwei (0.000000000975251796 ETH)
Gas Limit & Usage by Txn:	20,000,000 7,438,882 (37.19%)
Gas Fees:	Base: 0.974281796 Gwei Max: 2.899289836 Gwei Max Priority: 0.00097 Gwei
L2 Fees Paid:	0.007254783030732072 ETH
L1 Fees Paid:	0.000001395165630475 ETH
L1 Gas Price:	0.000000114734015657 ETH (114.734015657 Gwei)
L1 Gas Used by Txn:	1,600
L1 Fee Scalar:	0

Figure 5.19: An etherscan snapshot of the execution of a thousand transactions smart contract on Optimism Sepolia testnet

As expected, the cost of the gas units needed to execute the smart contract on Layer 2 is much lower than on the mainnet **Figure 5.19**. According to values in **Table 5.1**, the cost in USD of the gas consumed is:

$$USD\ Cost = \frac{7,439,000 \times 0.017793219\ Gwei}{10^9} (ETH) \times 1,833.95\ \$/_{ETH} \approx 0.243\ \$$$

As it is shown in the computation, executing the smart contract on Layer 2 has a negligible cost of 0.243\$ USD. However, the key difference between layers is the amount of fees paid for the execution for the same smart contract. For the Layer 2 execution, according to **Figure 5.19**, the **Table 5.1** and the same assumption as before of 1 Gwei as *Max Priority* [48], the transaction fee of the operation is:

$$transaction\ fee = \frac{(0.017793219\ Gwei + 1\ Gwei) \times 7,439,000\ gas}{10^9} = 0.007571\ ETH$$

Hence, a *transaction fee* of only 0.007571 ETH is paid for the execution of the **run()** method. This number translated to USD cost with the data from **Table 5.1** is around:

$$USD\ Cost = 0.007571\ ETH \times 1,833.95\ \$/ETH \approx 13.88\ \$$$

5.3.4 Comparative Analysis

Taking all the factors mentioned into account, it has been proved that performing the executions on Layer 2 using the *Optimistic rollups* architecture is much more efficient than doing it directly on the mainnet.

In other words, the total cost of executing the smart contract with a thousand transactions on the mainnet has a cost of around 48.38 \$ + 21.50 \$ = 69.88 \$ considering the costs of *fees* and *gas* used. Whereas for the Layer 2, the cost of *fees* and *gas* is around 13.88 \$ + 0.243\$ ≈ 14.12 \$, which is a significant lower value. Notice that the 280,000 gas units consumed in the contract deployment are not included in the computation.

Furthermore, for the Layer 2 calculations, the cost to get the ETH from Layer 1 to Layer 2 and back from Layer 2 to Layer 1 through the *bridge contract* in **Section 5.2.4** should also be taken into account. Therefore, an additional 4.245 \$ from Layer 1 to Layer 2 and 4.245 \$ from Layer 2 to Layer 1 should be added to the computation, so the final cost for the Layer 2 execution can be considered as a value close to 22.62 \$.

Observe it is assumed that the cost from Layer 2 to Layer 1 is equal to the cost from Layer 1 to Layer 2. We could not obtain a more precise result because the *Optimism*

testnet does not provide this feature. In any case, our assumption is a safe upperbound of the actual cost, as the bridge contract code is simpler in this case.

However, it is important to bear in mind that for a single transaction the gas consumed is around 37,000 gas units. Therefore, since there is a mandatory cost of around 8.50 \$ to get the ETH from one layer to the other, the use of the Layer 2 is only profitable if the final cost of the operations performed is lower than the cost on Layer 1, including the cost of using the *bridge contracts*. Notice that this threshold will change depending on the *fees*, the *gas consumed* and the price of gas in the different networks used.

Finally, the reduced cost can be represented as follows:

$$\text{Cost reduction} = \left(1 - \frac{L2 \text{ costs}}{L1 \text{ costs}}\right) \times 100 = \left(1 - \frac{22.62 \$}{69.88 \$}\right) \times 100 \approx 67.63\%$$

As a result, the costs of executing a thousand transactions using a smart contract can be reduced up to 67.63 %. These results are also summarized in the following **Table 5.2**.

Costs	Sepolia testnet (Layer 1)	Optimism testnet (Layer 2)
Fees cost	48.38 \$	13.88 \$
Gas consumed cost	21.50 \$	0.243 \$
Bridge contract cost	-	≈ 8.50 \$
Total cost	69.88 \$	22.62 \$

Table 5.2: Costs differences when executing a thousand transactions through a smart contract on Layer 1 and on Layer 2

In conclusion, the reduction in gas consumption shown in **Table 5.2** is just one of the multiple benefits provided by rollups. Furthermore, a key advantage of *optimistic rollups* is that multiple rollups can operate simultaneously, reducing transaction load on

Layer 1 since thousands of transactions are being executed off-chain at the same time, increasing the scalability and efficiency of the Ethereum environment.

5.4 Other interesting use cases

An interesting functionality that can be done with the *Optimism bridge* contract is to send an amount of *ether* from an EOA account in Layer 1, i.e. an external user, to an already deployed smart contract on Layer 2, in such a way that this contract has a *receive()* method that performs an action every time it receives assets, as shown in **Figure 5.20**.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;

contract ReceiverL2 {
    event Received(address sender, uint256 amount);

    receive() external payable {
        emit Received(msg.sender, msg.value);
    }
}
```

Figure 5.20: A snapshot of a smart contract deployed on Layer 2 with an implemented *receive()* method

To do so, a transaction from Layer 1 will be performed, using the interface of the function *depositETHTo()*, which is used to send *ether* to the *bridge* contract, previously explained in **Section 4.3**. This interface is executed through the *Remix* online tool connected to the *Metamask* account on Layer 1 as shown in **Figure 5.21**.

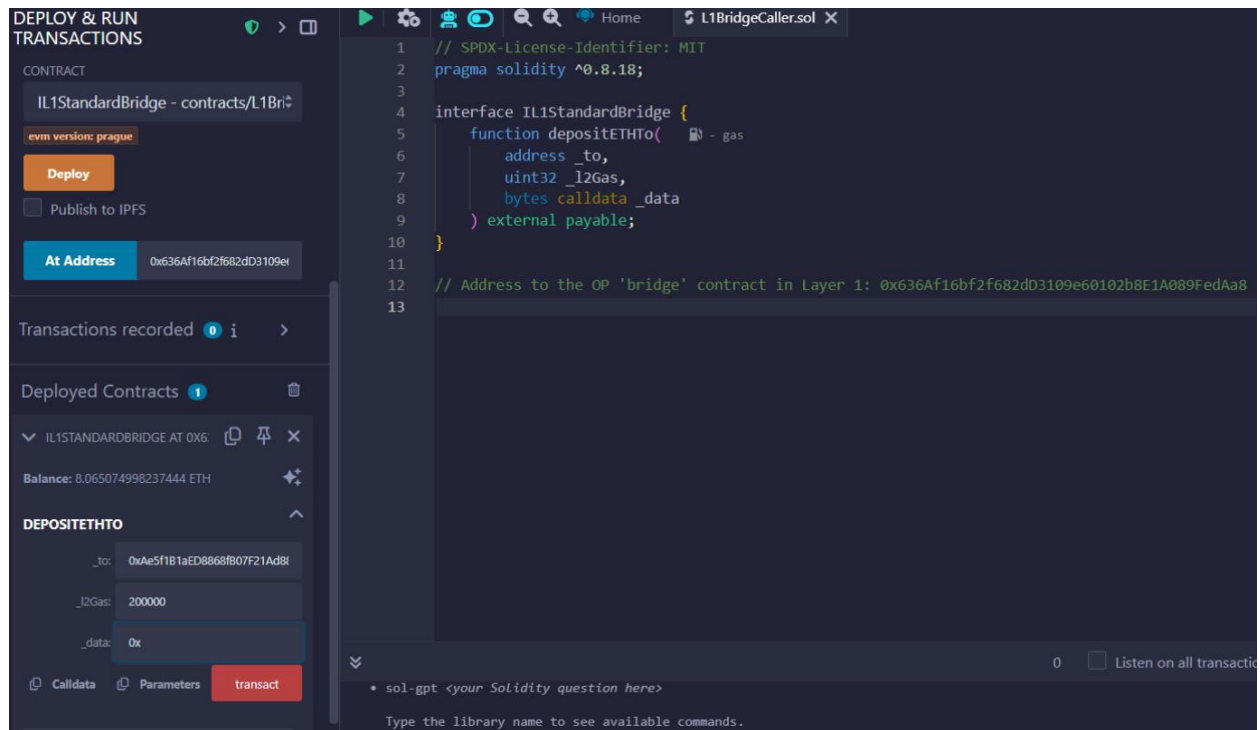


Figure 5.21: A snapshot of the Remix environment executing the `depositETHTo()` interface [49]

This feature opens up a wide range of interesting use cases for real applications. For instance, it allows the automatic activation of services deployed on Layer 2, such as subscriptions.

Another example is the use of this functionality in videogames based on the blockchain, where a smart contract represents an element of the game, such as a character or an item, that updates its state when it receives *ether* from Layer 1.

Executing these actions on Layer 2 allows deploying part of the mainnet application on Layer 2, being possible to deploy temporary or event-driven smart contracts that remain inactive until they are triggered by a transaction. This mechanism reduces costs using the Layer 2 while allowing the interaction with the application from Layer 1.

Chapter 6. Conclusions and future work

In this final section, the results collected throughout the project from the experiments and the calculations are analysed and explained in order to provide a final review of this Layer 2 solution.

Moreover, the different problems that appeared during the development process are discussed, providing details about the issues encountered throughout the project and explaining how these problems led to the final state of the project. Additionally, some useful ideas for the future and other technologies are recommended to improve the final results.

6.1 Analysis of results

The main goal of this final project was to deeply understand the *optimistic* rollups architecture, being able to describe the different components of the system and the relation between them.

The conceptual part was achieved through **Chapter 3**, and in particular the drafting of the diagram in **Figure 3.1** that represents the flow of the topology, as well as the different computations that demonstrate the efficiency of this Layer 2 solution. Some of the most relevant results include, as shown in **Section 3.4**, an increase of the transactions per second of the network, a gas reduction of 93% just with the use of blobs instead of calldata and a gas reduction of 98% with the use of *Merkle trees* to represent transactions on-chain.

Moreover, the pros and cons of the most used Layer 2 scalability solutions were discussed and compared.

At the beginning of the project, we wanted to perform a practical part with experiments to prove the efficiency of *optimistic rollups* in real-world uses cases. Despite the different problems encountered, as shown in **Section 6.2** below, we managed to do some tests in real-world testnet environments. The main result was that transactions executed on Layer 2 compared to the ones executed on the mainnet consumed similar

amounts of gas units. However, since the congestion of the Layer 2 network is reduced, the price of the gas is much lower, having a reduction of almost 68% in the final gas costs. For instance, executing a smart contract that executes a thousand consecutive transfers from peer-to-peer has a cost of 69.88\$ in the mainnet, while it has a cost of just 22.62\$ on Layer 2 with the use of *optimistic rollups*.

As a final conclusion, my blockchain and Ethereum knowledge has significantly improved throughout the project, understanding different theoretical concepts that explain how the Ethereum ecosystem works at a lower level.

6.2 Problems and issues

Although the results obtained are really meaningful and useful to explain the efficiency of *optimistic rollups*, there have been several issues in the practical part that had to be constantly modified.

6.2.1 Ganache

Initially, the first approach was to create an experimental environment with the help of *ganache*, which is a tool to start a local Ethereum blockchain to run tests while controlling how the chain operates.

The plan was to connect both *ganache* blockchains, one working as Layer 1 and the other working as Layer 2, taking into account that both blockchains cannot communicate by their own, they need to emit events with relevant information that must be caught by an external component.

At the beginning, the plan was to use libraries such as *web3.js* or *ethers.js* to build the external architecture, but as the *optimistic rollup* topology was further studied, the complexity of the protocol increased too much to start from scratch.

After building an isolated virtual machine, installing all the dependencies needed, connecting *Metamask* to the system and setting up the *truffle dashboard* to see the transaction results, I discovered it was really difficult to build a reliable experimental environment with *ganache* since the system needed a bridge contract to transfer assets

between layers, create batches and build *Merkle trees* for the *fraud proof* protocol, use blobs to store data, set a *challenge period* after users finished executing transactions, simulate congestion in the layers in order to obtain realistic results, etc. Hence, it was too difficult to be implemented this way and another solution was required.

6.2.2 OP Stack installation

Later, we found the *OP Stack* tutorial [50], that showed how to build your own local testnet.

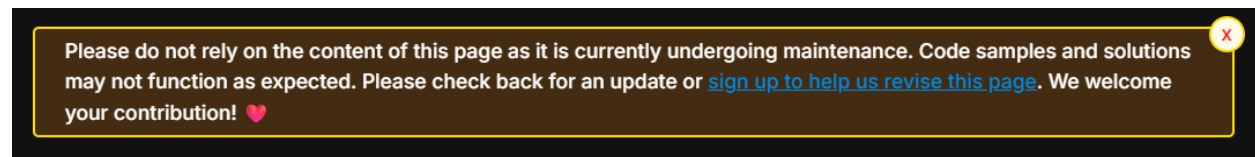


Figure 6.1: Warning message shown at the beginning of the Optimism tutorial

Despite the fact that this message in **Figure 6.1** was shown at the beginning of the tutorial, I still tried to build it since the result was exactly what I needed.

After installing all the dependencies, creating an *Alchemy* account in order to have access to a *Sepolia* node, fixing some errors in the installation as shown in **Figure 6.2**, changing the configuration of essential files such as *intent.toml*, deploying the smart contracts required on Layer 1, doing some other file configurations and doing the whole tutorial from the beginning several times, I reached a point where the *op-node* component was not working correctly and could not connect to the *Sepolia* node, not allowing me to finish the tutorial.

```
> NX Running target build for 8 projects
✖ nx run @eth-optimism/core-utils:build
> @eth-optimism/core-utils@0.13.1 build /home/tfg/optimism/packages/core-utils
> tsc -p tsconfig.json

src/common/hex-strings.ts(42,22): error TS2769: No overload matches this call.
  Overload 1 of 3, '(array: WithImplicitCoercion<ArrayLike<number>>): Buffer<ArrayBuffer>', gave the following error.
    Argument of type 'string | Buffer<ArrayBufferLike>' is not assignable to parameter of type 'WithImplicitCoercion<ArrayLike<number>>'.
      Type 'string' is not assignable to type 'WithImplicitCoercion<ArrayLike<number>>'.
  Overload 2 of 3, '(arrayBuffer: WithImplicitCoercion<ArrayBufferLike>, byteOffset?: number, length?: number): Buffer<ArrayBufferLike>', gave the following error.
    Argument of type 'string | Buffer<ArrayBufferLike>' is not assignable to parameter of type 'WithImplicitCoercion<ArrayBufferLike>'.
      Type 'string' is not assignable to type 'WithImplicitCoercion<ArrayBufferLike>'.
  Overload 3 of 3, '(string: WithImplicitCoercion<string>, encoding?: BufferEncoding): Buffer<ArrayBuffer>', gave the following error.
    Argument of type 'string | Buffer<ArrayBufferLike>' is not assignable to parameter of type 'WithImplicitCoercion<string>'.
      Type 'Buffer<ArrayBufferLike>' is not assignable to type 'WithImplicitCoercion<string>'.
        The types returned by 'valueOf()' are incompatible between these types.
          Type 'Buffer<ArrayBufferLike>' is not assignable to type '{ valueOf(): string; }'.
          Type 'Buffer<ArrayBufferLike>' is not assignable to type 'string'.
ELIFECYCLE Command failed with exit code 2.
```

```
> NX Running target build for 8 projects
→ Executing 1/7 remaining tasks...
└─ nx run @eth-optimism/contracts-bedrock:build
✖ 1/1 failed
```

Figure 6.2: Error found in file 'hex-strings.ts', which had to be modified to be solved

Apparently, the *op-node* component needed to be updated, however, I installed recent versions and still did not work. I asked in *github* discussions as shown in **Figure 6.3**, but no solutions were provided [51].

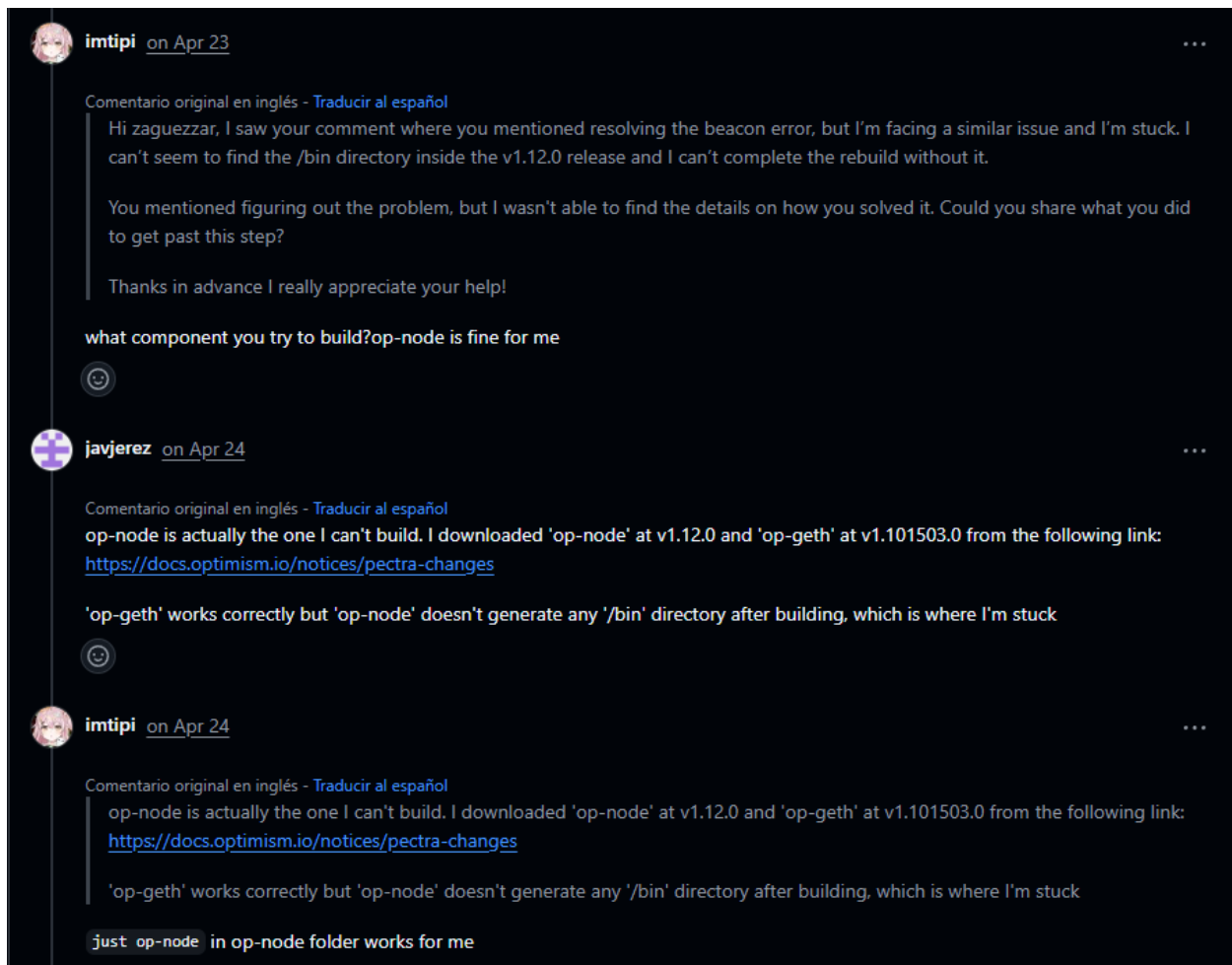


Figure 6.3: A snapshot of a Github discussion trying to solve the *op-node* issue

Therefore, after different attempts to solve the problem, we decided to do the tests directly on public testnets using Sepolia ETH obtained during the process.

6.2.3 Arbitrum Sepolia testnet

Initially, the experiments were meant to be done on two different Layer 2 blockchains in order to compare result between them, which were *Optimism Sepolia testnet* and *Arbitrum Sepolia testnet*. However, at the time to connect the Metamask account to the *Arbitrum RPC* through [52], an error occurred, and the connection could not be done, as shown in **Figure 6.4**.

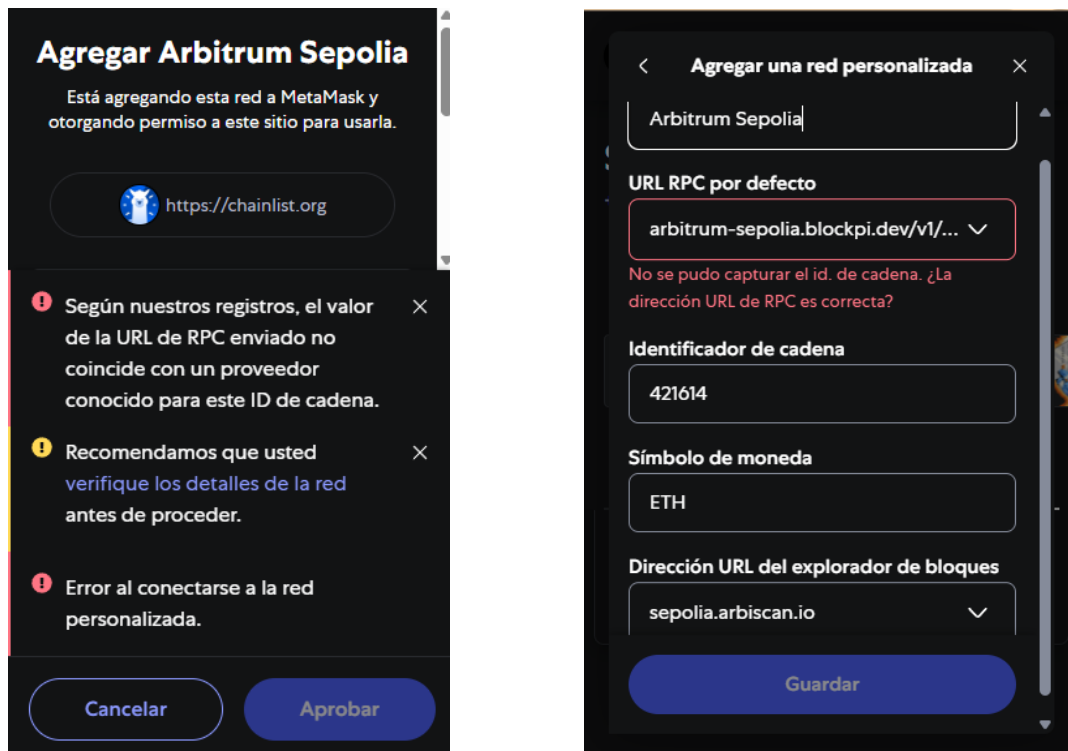


Figure 6.4: An error occurs when trying to connect Metamask to the Arbitrum Sepolia RPC

6.2.4 Final deduction

Despite all the problems encountered throughout the project, thanks to these difficulties I have had another approach to carry out my final project, coming to the conclusion that making all the experiments on official testnets has allowed me to study *optimistic rollups* in the most accurate way, although without full control over these networks.

Furthermore, I have used professional tools that current developers use to test smart contracts and decentralized applications before deploying them to the Ethereum mainnet.

For these reasons, I consider having faced and attempted to solve these challenges as an opportunity to understand the multiple difficulties you may encounter in the blockchain field, learning how to execute and analyse real-world use cases in professional environments.

6.3 Future Work

During the project development process, I made some important decisions that affected the general workflow, as well as other aspects that could be further explored through practical examples.

For instance, if I had been able to connect the *Metamask* account to the *Arbitrum Sepolia testnet*, which was a problem I was not able to solve due to an *RPC* error, I would have obtained more accurate results since the *Arbitrum Sepolia testnet* allows developers to withdraw funds from Layer 2 to Layer 1. In the project, I had to assume that the cost from withdrawing assets from Layer 2 to Layer 1 was symmetrical to sending funds from Layer 1 to Layer 2, which is still a good approximation of the cost. For this same reason, if the *Optimism Sepolia testnet* allowed developers to withdraw funds from Layer 2 to the mainnet, more accurate results could have been provided, instead of an approximation.

Furthermore, the project does not include an experiment on how to create a *Merkle root* in order to challenge the publication of an *operator* during the *challenge period*. From my point of view, it would be interesting to study how an external user can challenge publications on Layer 1.

In relation to the last experiment done in **Section 5.4**, which allows external users to send *ether* from Layer 1 directly to a smart contract deployed on Layer 2, in such a way it triggers an action written in the *receive()* method of the smart contract, it would be interesting to explore real-world use cases that could apply this functionality.

Finally, regarding ZK-rollups, we have to bear in mind that these are the rollups likely to be used in the long term. Therefore, it would be very useful to study in detail how these rollups perform by repeating the same experiments carried out with the *optimistic rollups*, and compare the outcomes based on real data in order to decide which rollups are better for specific scenarios.

BIBLIOGRAPHY

- [1] V. Buterin, 31 12 2017. [Online]. Available:
https://vitalik.eth.limo/general/2017/12/31/sharding_faq.html.
- [2] V. Buterin, 05 January 2021. [Online]. Available:
<https://vitalik.eth.limo/general/2021/01/05/rollup.html>. [Accessed 14 09 2024].
- [3] Hereda. [Online]. Available: <https://hedera.com/learning/consensus-algorithms/proof-of-stake-vs-proof-of-work>. [Accessed 20 05 2025].
- [4] geeksforgeeks, 15 04 2025. [Online]. Available:
<https://www.geeksforgeeks.org/what-is-double-spending-in-blockchain/>.
[Accessed 21 05 2025].
- [5] ethereum.org, 14 05 2025. [Online]. Available: <https://ethereum.org/en/what-is-ethereum/>. [Accessed 21 05 2025].
- [6] wackerow, 12 02 2025. [Online]. Available:
<https://ethereum.org/en/developers/docs/smart-contracts/>. [Accessed 21 05 2025].
- [7] C. Smith, 25 2 2025. [Online]. Available:
<https://ethereum.org/en/developers/docs/gas/>. [Accessed 21 05 2025].
- [8] crypto.com, 03 01 2020. [Online]. Available:
<https://crypto.com/en/university/blockchain-scalability>. [Accessed 21 05 2025].
- [9] Nervos, 22 04 2023. [Online]. Available: https://www.nervos.org/es/knowledge-base/blockchain_trilemma. [Accessed 21 05 2025].
- [10] C. Smith, 13 02 2025. [Online]. Available:
<https://ethereum.org/en/developers/docs/scaling/>. [Accessed 21 05 2025].

- [11] D. Davó, GRASIA Research Group, UCM.
- [12] Z. U. Press, 2023. [Online]. Available:
<https://www.sciencedirect.com/journal/blockchain-research-and-applications>.
[Accessed 22 05 2025].
- [13] P. G. Revision. [Online]. Available:
<https://docs.plasma.group/en/latest/src/plasma/sidechains.html>. [Accessed 22 05 2025].
- [14] ethereum.org, 14 05 2025. [Online]. Available: <https://ethereum.org/en/layer-2/learn/>. [Accessed 22 05 2025].
- [15] wackerow, 12 02 2025. [Online]. Available:
<https://ethereum.org/en/developers/docs/scaling/state-channels/>. [Accessed 22 05 2025].
- [16] G. P. S. Lydia D. Negka, 30 11 2021. [Online]. Available:
<https://ieeexplore.ieee.org/document/9627997>.
- [17] wackerow, 12 02 2025. [Online]. Available:
<https://ethereum.org/en/developers/docs/scaling/plasma/>. [Accessed 22 05 2025].
- [18] H. Q. Ashwin Ramachandran, 27 01 2020. [Online]. Available:
<https://medium.com/dragonfly-research/the-life-and-death-of-plasma-b72c6a59c5ad#>.
- [19] C. Smith, 13 02 2025. [Online]. Available:
<https://ethereum.org/en/developers/docs/scaling/optimistic-rollups/>. [Accessed 14 09 2024].

- [20] C. Smith, 25 02 2025. [Online]. Available:
<https://ethereum.org/en/developers/docs/scaling/zk-rollups/#what-are-zk-rollups>.
[Accessed 22 05 2025].
- [21] E. Brinde, 06 2024. [Online]. Available: <https://simplystaking.com/wp-content/uploads/2024/06/A-Snapshot-Comparison-of-Layer-2-Solutions-on-Ethereum.pdf>.
- [22] O. Pomerantz, "Merkle proofs for offline data integrity," 30 12 2021. [Online]. Available: <https://ethereum.org/en/developers/tutorials/merkle-proofs-for-offline-data-integrity/>. [Accessed 10 10 2024].
- [23] E. C. R. D. M. S. I. N. A. B. Vitalik Buterin, 13 04 2019. [Online]. Available:
<https://eips.ethereum.org/EIPS/eip-1559>.
- [24] D. G. Wood, "Ethereum: A secure decentralised generalised transaction ledger. Yellow Paper, Shanghai version," 04 02 2025. [Online]. Available:
<https://ethereum.github.io/yellowpaper/paper.pdf>.
- [25] Blobscan. [Online]. Available: <https://blobscan.com/> . [Accessed 12 04 2025].
- [26] ethereum.org, 12 02 2025. [Online]. Available:
<https://ethereum.org/en/developers/tutorials/merkle-proofs-for-offline-data-integrity/>. [Accessed 15 04 2025].
- [27] etherscan.io, 28 04 2025. [Online]. Available:
<https://etherscan.io/tx/0x4c5649aa1c1f71f4fb5793b119a021ba0674640d781d9984a7130beebe9e82bc>. [Accessed 28 04 2025].
- [28] V. Buterin. [Online]. Available: <https://www.eip4844.com/>. [Accessed 30 04 2025].
- [29] D. Mihal. [Online]. Available: <https://l2fees.info/>. [Accessed 30 04 2025].

- [30] O. Foundation, 7 04 2025. [Online]. Available: <https://docs.optimism.io/operators/chain-operators/architecture>. [Accessed 05 05 2025].
- [31] O. Foundation. [Online]. Available: <https://specs.optimism.io/protocol/overview.html>. [Accessed 05 05 2025].
- [32] O. Foundation. [Online]. Available: <https://github.com/ethereum-optimism/optimism/tree/develop>. [Accessed 06 05 2025].
- [33] O. Foundation. [Online]. Available: <https://github.com/ethereum-optimism/optimism/blob/develop/op-node/rollup/derive/channel.go>. [Accessed 06 05 2025].
- [34] O. Foundation. [Online]. Available: <https://github.com/ethereum-optimism/optimism/blob/develop/op-program/client/driver/driver.go>. [Accessed 06 05 2025].
- [35] O. Foundation. [Online]. Available: <https://github.com/ethereum-optimism/optimism/blob/develop/packages/contracts-bedrock/src/L1/L1StandardBridge.sol>. [Accessed 06 05 2025].
- [36] ethPandaOps. [Online]. Available: <https://light-sepolia.beaconcha.in/>. [Accessed 01 05 2025].
- [37] pk910. [Online]. Available: <https://sepolia-faucet.pk910.de/>. [Accessed 20 03 2025].
- [38] O. Foundation. [Online]. Available: <https://www.superchain.tools/bridge>. [Accessed 30 04 2025].

- [39] O. Labs. [Online]. Available: <https://bridge.arbitrum.io/?destinationChain=arbitrum-sepolia&sourceChain=sepolia>. [Accessed 01 05 2025].
- [40] MetaMask. [Online]. Available: <https://metamask.io/>. [Accessed 25 10 2024].
- [41] DefiLlama. [Online]. Available: <https://chainlist.org/chain/11155420>. [Accessed 30 04 2025].
- [42] etherscan. [Online]. Available: <https://etherscan.io/chart/gasprice>. [Accessed 03 05 2025].
- [43] etherscan. [Online]. Available: <https://optimistic.etherscan.io/chart/gasprice>. [Accessed 03 05 2025].
- [44] etherscan. [Online]. Available: <https://arbiscan.io/chart/gasprice>. [Accessed 03 05 2025].
- [45] etherscan. [Online]. Available: <https://etherscan.io/chart/etherprice>. [Accessed 03 05 2025].
- [46] etherscan. [Online]. Available: <https://sepolia.etherscan.io/>. [Accessed 30 04 2025].
- [47] M. Cutler, 26 08 2021. [Online]. Available: <https://www.blocknative.com/blog/eip-1559-fees>. [Accessed 06 05 2025].
- [48] O. Foundation, 08 04 2025. [Online]. Available: <https://docs.optimism.io/stack/transactions/fees#priority-fee>. [Accessed 07 05 2025].
- [49] Remix. [Online]. Available: <https://remix.ethereum.org/>. [Accessed 23 05 2025].

- [50] O. Foundation, 07 04 2025. [Online]. Available: <https://docs.optimism.io/operators/chain-operators/tutorials/create-l2-rollup>. [Accessed 16 02 2025].
- [51] I. GitHub, 25 03 2025. [Online]. Available: <https://github.com/ethereum-optimism/developers/discussions/775>. [Accessed 24 04 2025].
- [52] DefiLlama. [Online]. Available: <https://chainlist.org/chain/421614>. [Accessed 01 05 2025].

