Imperial College London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

From Niche to Mass Adoption: Taking Native Staking Mainstream

Author:
Demetris K. Kyriacou

Supervisors:
Prof. William J. Knottenbelt
Dr. Jacob George

Submitted in partial fulfillment of the requirements for the MSc degree in Type of Degree of Imperial College London

September 2023

Abstract

Native staking is the process of using an amount of native token as a collateral in the Proof of Stake mechanism of a blockchain and it offers several benefits for the stakeholder and the blockchain.

Therefore, it would be fair to suggest that ensuring accessibility to every user, regardless of their wealth or knowledge, is favourable for the whole blockchain network. However, the existing staking pools charge fees that make it infeasible to stake or unstake small amounts.

The objective of this individual project is to solve this problem through proposing a protocol that makes staking accessible to everyone.

Over the course of this project, we suggest and study multiple ideas to mitigate the gas fees associated with native staking. These ideas are moulded into design approaches for gas efficient staking vaults.

Furthermore, a full-stack novel staking protocol is designed, developed, deployed, tested, evaluated and compared to existing staking protocols that handle assets worth hundreds of millions of pounds.

We view this project as a complete journey. A journey that starts with the need for solving a real-world problem, goes through all the computer science, mathematical and finance aspects of designing and building, concluding with a real-world solution.

Acknowledgments

First, I would like to dedicate this work in the loving memory of my godmother Fiona who left too early before we had had enough cups of tea.

I would like to thank my first supervisor, Professor Knottenbelt for his zeal and assistance, not only throughout this project, but throughout my whole experience at Imperial College London.

Additionally I would like to thank my second supervisor, Dr Jacob George for his mentoring, his perpetual support and for giving me the opportunity to become a part of the Fieldlabs team.

I would also like to thank the whole Fieldlabs team (Pietro, Theo, Julia, Tiff, Tobe, Alex, Jeremy, Yash, Issy, Yann) for making me feel welcome in their office and helping me throughout this project by giving me advice and ideas.

Lastly, I would like to thank my godfather Alan, my friends and my family who stood by me during the ups and downs of this year.

May the Seagulls fly high.

Contents

Chapter 1

Introduction

1.1 Motivation

Blockchain technology took the world by storm since it was first proposed in 2008 by Satoshi Nakamoto[1]. By leveraging the economic crisis and people's mistrust of central authority systems, it transformed an idea into a trillion-pound industry within a span of 15 years.

Furthermore, the introduction of the Proof of Stake (PoS) consensus mechanism, as an alternative to the energy inefficient Proof of Work (PoW), unlocked even more opportunities for blockchain systems. By adopting PoS, blockchains became significantly more eco-friendly, more safe and more scalable [2].

At the same time, PoS incentivised every user to be a part of the consensus mechanism in a mutually beneficial way for both the users and the blockchain. By staking native coins, stakeholders can receive an almost risk-free passive income with an annual percentage yield (APY). At the same time, by increasing the total staked amount, the blockchain becomes more decentralised and more resistant to majority attacks since accumulating a considerable percentage of the total staked amount becomes increasingly difficult for a malicious user.

In theory, blockchains are meant to be decentralised and accessible to everyone following the principles set out by Nakamoto. In practice, staking benefits only holders of sufficient funds. After all, PoS works as long as a user does not behave dishonestly as they have a lot to lose by doing so since malicious users are penalised on their staked funds. For this reason, some services providers designed solutions based on staking pools. These combine many users' funds into a single pool thereby mutualising transaction fees, making staking accessible to retail cryptocurrency holders [3, 4].

Ostensibly, many staking pools already exist and offer staking services to everyone without a minimum limit on the deposit amount. Upon closer examination, there are hidden gas fees that make it infeasible to stake small amounts. In fact, the gas fees are so high that if a user wishes to stake 100 pounds, for the major chains, they would have to wait years until the accrued rewards would be enough just to cover the staking and unstaking fees.

What is proposed in this project is a protocol that mitigates the gas fees associated with staking and make it genuinely accessible to retail users. We are also designing and deploying a prototype as a proof of concept.

1.2 Objectives

First of all, we are planning to research existing staking pools and discuss their advantages and pitfalls. Through this we demonstrate quantitatively the lack of options for retail staking and obtain ideas that could potentially be combined to solve the problem. Additionally, we aim to present the thought process and all the intermediate solutions that led up to the final solution.

The primary objective of this project is to design and implement a protocol that will make staking accessible to retail cryptocurrency holders. Our goal is not to differentiate the users in two categories of retail users and whales. On the contrary, our protocol wishes to combine the potential and the capabilities of all users in a way that will benefit everyone.

Apart from the main goal, the project has a series of secondary objectives. We intend to promote staking as a means to improve blockchain's robustness and ability to handle transactions. Finally, our aspiration is to promote staking as a passive source of income which is also:

- mainly trustless (code replaces the need for a trusted centralised authority)
- non-custodial (participants maintain control of their funds)
- risk-free (modulo slashing and smart contract risk)

All in all, we hope that this project will take some steps towards the decentralisation originally envisioned by Satoshi Nakamoto.

1.3 Contribution

The initial contribution of this project is **identifying the problem of high gas fees prices that make native staking infeasible for small amounts**. To formalise the problem, we conduct a **gas fees comparison** among some of the most popular stakers.

The project suggests some **core ideas that can be used to reduce gas fees**. These ideas could be adopted by existing staking vaults without the need of radically changing their code.

Taking it a step further, we designed **four (4) different approaches for new staking vaults** that could potentially reduce the gas fees for native staking in many different Blockchains.

Furthermore we selected one of the four approaches and designed a **complete protocol** based on it. The design contains a description for the functions, variables and data structures it consists of. We also provide the methodology for determining the values of parameters of the protocol.

Finally, we developed the **full-stack prototype of a staking vault** based on the complete protocol as a proof of concept. The prototype achieves its goal and proves that, under conditions, it is possible to achieve lower gas fees than the state of the art stakers that stake hundreds of millions of MATIC.

1.4 Legal, Social and Professional Requirements

The data used for this project were retrieved from the Ethereum mainnet. As information in the Ethereum blockchain is public, there is no concern about personal data. Same principle applies for code that is deployed and verified on the blockchain.

Some of the code produced in this project was built on top of existing code of Fieldlabs. In order to have access to the code of Fieldlabs, a Non-Disclosure Agreement (NDA) was signed. Any pre-existing code that was used for this project is public and clearly marked in the git repo. The final git repo was inspected by the CTO of Fieldlabs to ensure that there will not be any legal complications or violations of the NDA.

Furthermore, even though blockchains have been used before for malevolent and criminal activities, the current project is dedicated on native staking and cannot be used for malicious purposes.

Lastly, this project could potentially be used to produce a software product. Nevertheless, there are no copyright licensing implications since the code is either developed by me or it is already public.

1.5 Report Structure

Chapter 2 - Background: The background report consists of two parts. The former part contains fundamental knowledge and concepts which are required for the comprehension of the rest of the report, such as the Proof of Stake protocol, native staking, pooled staking and gas fees.

The latter part contains a report on the existing native staking solutions. There is also an analysis of their gas fees. This analysis justifies the need for a protocol like the one proposed in this project and is used in chapter 5 to evaluate the developed prototype.

Chapter 3 - Design: The design chapter describes the sequence of procedures that transformed the initial idea to a complete and formal design of a protocol that reduces the gas fees for native staking. It includes the rationale for choosing the Polygon blockchain, the intermediary designs and the full description of the final design. It also covers the methodology for determining the values for the parameters that control the operation of the final protocol.

Chapter 4 - Implementation: The implementation chapter describes the process of giving substance to the final design by developing a prototype with a UI. It incorporates information about the smart contract, the frontend and the auto-tasks that automatise the control of the prototype.

Chapter 5 - Evaluation: The evaluation chapter contains an analysis of the gas fees of the prototype and a comparison with the gas fees of other existing stakers.

Chapter 6 - Conclusion: The final chapter contains a summary of the achievements of this project. It also includes suggestions for further improvements of the prototype as well as ideas for future projects and applications.

Chapter 2

Background

2.1 Staking

2.1.1 Blockchain

A blockchain is an immutable distributed ledger maintained by a network of nodes. It is based on the idea that every node of the network possesses an identical copy of the ledger. The ledger consists of blocks and every block is connected to the previous one, thus creating a chain. In order to ensure that the state of the blockchain is unique and universally accepted, there is a need for a consensus mechanism [5].

2.1.2 Proof of Work(PoW)

Bitcoin, the first blockchain that was proposed by Satoshi Nakamoto [1] uses the Proof of Work (PoW) consensus mechanism. The PoW secures the integrity by making the nodes (or miners) solve a computational problem. The first miner to solve the problem is awarded the right to add a block to the blockchain in exchange for a reward. Since the problem is complex and requires significant computational power, the blockchain is safe from malicious attacks as long as the majority of the computational power of the network is controlled by honest nodes.

The idea of a blockchain was revolutionary and the PoW mechanism has proven to be robust and reliable. However, the increase of the popularity of Bitcoin revealed some significant disadvantages of PoW:

1. It demands non-trivial amounts of electrical energy which makes it expensive [6, 7, 8] and has harmful effects on the environment [9].
2. Computational power is wasted on relatively meaningless calculations.

3. Existing miners posses very powerful computational machines. This makes it extremely expensive for new miners to enter the market and can lead to de-facto centralisation [10].

4. It leads to the creation of two different types of users, the mining pools that explicitly perform mining and the rest of the users who explicitly perform transactions [11].

5. It does not scale well as the more users join the system, the more amount of computations is required [11].

6. Finality (assurance that a transaction is added to the blockchain and will not be reverted) is not explicit but probabilistic [10, 2].

7. The need for huge amounts of energy, sophisticated hardware and warehouse facilities make it less censorship resistant as these purchases can potentially be traced [12].

2.1.3 Proof of Stake (PoS)

Since the early days of Bitcoin, multiple alternative consensus mechanisms were proposed. The most popular of them is the Proof of Stake (PoS) mechanism. PoS was initially proposed by Sunny King and Scott Nadal in the PPCoin whitepaper [13] in 2012 in a hybrid mechanism that combined PoS and PoW. The main argument for the proposal was that PoW was very energy inefficient. As described by S.King and S.Nadal, proof of stake can be viewed as a proof of token ownership. The idea behind of that mechanism was to use the concept of coin age, i.e., the product of currency amount times the holding period to increase the value of unspent coins. By spending coin age, users are awarded the right to generate a new block.

This primitive version of PoS influenced various other proposals for PoS consensus mechanisms. BlackCoin [14] was the first blockchain that used a pure PoS mechanism. In the whitepaper, it is explained why coin age should be dropped. It is also suggested that it is important for the safety of a PoS mechanisms to have as many nodes online as possible.

Later innovative adaptations of pure PoS mechanisms include Nxt [15], Algorand [11] and Cardano which is based on the Ouroboros protocol [16].

The Nxt blockchain[15] is an effort to provide an agile architecture that does not depend on powerful machines. Additionally, the blockchain does not support the minting of new coins and it relies on the tokens that were issued in the genesis block. Therefore, the reward of the node that creates a new block consists solely of transaction fees. It also uses the idea of explicit finality as transactions located in a block with depth < 10 are considered safe. Lastly, it adopts the concept that the probability of a node to be selected to create a new block is proportional to the number of tokens held by the node.

In Algorand [11] blockchain, the creation of a fork is nearly impossible thus a every new block remains in the blockchain forever and all transactions in a block are considered final. The block generation procedure has two phases. First, a node is randomly selected to propose a block and secondly, a committee of nodes is randomly selected to validate the proposed block. As in Nxt, the probability of selecting a node to build a block is proportional to the node's staked assets in the system. The selection is made using algorithmic randomness which also inspired the name of the blockchain.

The Cardano blockchain, based on the Ouroboros protocol [16], uses random selection of nodes that propose blocks proportionally to their staked amount. According to the team behind Ouroboros, their innovation is that the protocol uses a coin flipping algorithm to introduce truly unbiased randomness unlike other implementations. In Cardano, time is split to slots. In each slot, a slot leader is randomly selected to add a new block to the blockchain and a slot endorser is selected to endorse the new block.

Similarly to other PoS blockchains, in Cardano it is important to have honest and punctual nodes that are always online and ready to act as slot leaders and slot endorsers. Therefore, Ouroboros protocol [16] introduces the concept of a stake delegation mechanism. The delegation mechanism leads to the creation of so called ”stake pools”. More specifically, a group of stakeholders is selected to act as delegates. The rest of the stakeholders authorise a delegate to stake their funds on their behalf. In order to have the right to stake, a delegate is required to accumulate funds that exceed a threshold. This ensures that the delegate does not gain by acting maliciously since this will lead to penalties and devaluation of the currency.

2.1.4 Proof of Stake (PoS) in Ethereum

At the time of writing, the most popular blockchain that uses a PoS consensus mechanism is Ethereum [17]. Ethereum, proposed in 2014 by Vitalik Buterin, was originally based on a PoW mechanism. However, the benefits of PoS over PoW led to what was called “The Merge” in September 2022 when the network transitioned from a PoW mechanism to a PoS mechanism. Post-merge, Ethereum became able to facilitate further scalability upgrades and its energy consumption was reduced by 99.95% [18].

Ethereum's PoS is using the Gasper protocol [19]. Gasper is a combination of the Casper finality gadget [20] and the LMD-GHOST fork choice algorithm. Finality is a state in which a block can be considered a safe and immutable part of the blockchain. Following Casper, Ethereum splits time in epochs which are then split in slots. In each slot a new block is added to the chain. The block of the first slot of each epoch is considered a checkpoint. Once the two thirds of the validators attest to a pair of checkpoints, the blocks between them are finalised.

Meanwhile, the LMD-GHOST (latest message-driven greedy heaviest observed subtree) fork choice recognises as valid the chain with the heaviest chain in regards of attestations. In case of receiving multiple messages from a validator, the latest one is the one considered accurate [19, 21].

The Gasper protocol incentivises benevolent behaviour by applying rewards and penalties [19, 21]. A validator is rewarded for proposing a valid new block or for validating a proposed block. Conversely, the amount that a stakeholder stakes acts as a collateral and if *slashing* conditions are met, the validator is penalised on their staked amount. Malevolent behaviour that leads to slashing includes: proposing more than one block in a single time slot, attesting to multiple blocks on the same time slot or contradicting previous checkpoint votes.

By applying penalties, Gasper solves the ”Nothing at stake” problem which is common in other PoS implementations. In blockchains that do not penalise dishonest behaviour, upon the creation of a fork, users can create blocks on top of every branch to maximise the probability of profit. Ethereum users are penalised for such a behaviour therefore the problem of the existence of multiple chains is quickly resolved [19, 22].

Furthermore, in order to become a validator, a node needs to stake an amount of 32 ETH. The validator’s duties are to create new blocks or attest (vote in order to validate) to newly created blocks by other validators whenever they are selected by the system. The probability of a validator to be selected to create a new block or participate in an attesting committee is proportionate to their staked amount [2].

2.1.5 Gas fees in Ethereum

As this project concentrates on designing a protocol that mitigates the gas fees in Ethereum, it is important to explain why gas fees exist and how they are calculated. Gas is a unit to measure computational labour required to execute a transaction on Ethereum. Gas fees are calculated as the amount of gas units required to execute a transaction multiplied by the gas price (i.e. cost per unit gas) [23]. The formula to calculate gas fees in ETH is:

$$\begin{aligned} \textit{total fee} &= \textit{units of gas used} \times \textit{gas price} \\ &= \textit{units of gas used} \times (\textit{base fee} + \textit{priority fee}) \end{aligned} \tag{2.1}$$

The *gas price* is the addition of two components, *base fee* and *priority fee*. The *base fee* is the minimum amount for a transaction to be considered valid. The *priority fee* is an additional tip offered to validators to increase the likeliness of the transaction be included in a block by a validator [23].

Even though just paying the *base fee* is enough for a transaction to be considered valid, the validators are free to execute and add to the blockchain any transaction they wish. This means that they would choose to include the most profitable transactions first. Therefore, the higher the *priority fee*, the faster the transaction will be included in a block. On the contrary, transactions without *priority fee* are very unlikely to be included in a block.

The *units of gas used* depend exclusively on the computation effort required to execute a transaction. The value is invariant of the gas price and ETH's dollar value. It is also invariant to the Ethereum network that the transaction is executed. This proves to be very useful as it allows comparing the gas fees of transactions executed on Ethereum's mainnet with transactions executed on testnets.

Moreover, gas fees serve multiple purposes. As already described, they are a method of payment for the node that provides the computational power needed to execute the transactions which change the state of the blockchain. They also create a virtual meaning of priority amongst the pending transactions.

More importantly though, they practically solve the halting problem in Ethereum. According to the halting problem, it is impossible, given an arbitrary piece of code, to determine whether or not it will terminate of create an infinite loop. As transactions in Ethereum are executed using the network's computational power, infinite loops could consume the network resources. If a user, intentionally or unintentionally, creates an infinite loop, the transaction will be processed until it runs out of gas. Then, it will revert but the sender will lose the fees as they will be claimed by the validator who (partially) processed it [17].

2.1.6 Proof of Stake (PoS) in Polygon

The blockchain selected for our proof of concept is Polygon, previously known as the Matic Network [24]. Rationale for this choice will be given in the following chapters. For now, it is important to explain that Polygon is not precisely a blockchain but a scaling solution that enables the existence of sidechains that are connected to a main chain like Ethereum.

Polygon's purpose is to solve scalability and user experience issues that Ethereum struggles with. As argued by the Polygon team, Polygon offers high transaction throughput, low transaction fees, faster and deterministic finality as well as security. At the same time, Polygon maintains Ethereum-compatibility [25].

To achieve these, Polygon uses an architecture in three layers. The first layer is the Ethereum layer which contains Polygon's smart contracts deployed on Ethereum mainnet. The second is the Heimdall or PoS validation layer. The third and final layer is the Bor or sidechain block producer layer.

In the Bor layer, the block producers create blocks with transactions on the sidechain. Periodically, the Heimdall nodes or validators, validate a set of blocks in the Bor layer and calculate the Merkle tree of these blocks' hashes. The Merkle root hash is stored on the Ethereum mainnet thus creating a checkpoint. Transactions included in a checkpoint are considered final [26].

2.1.7 Native Staking

The term *native staking* (or simply *staking*) refers to the process of using an amount of a blockchain's native token as a collateral in the PoS mechanism of that blockchain. By doing this, a user becomes a validator of that blockchain. A validator is required to store data and process transactions to create or validate blocks. Each PoS blockchain that supports staking has its own requirements, rules and limitations about staking but the main idea is similar in all of them.

The main benefit of native staking is that the stakeholder receives rewards which can be seen as a source of passive income, similar to putting money in a savings account [27]. In most cases, the percentage yield from the staking is affected by multiple and unpredictable factors (different for each blockchain) but usually APY varies from 5% to 20% [28], generally significantly higher than a bank's interest.

Depending on the Blockchain, the rewards can be given for block creation and/or block attestation. The funds for the rewards are collected by minting new coins and/or by collecting transaction fees. In Ethereum, the creation of a new block initiates the minting of new coins but the number of newly minted coins is not constant for every block creation [29, 30]. In Polygon, 12% of the total supply of 10 billion MATIC is reserved to be used as staking rewards [31].

At the same time, staking benefits the blockchain as it improves its security and operational resilience. This happens because the validators are discouraged from acting dishonestly as this would negatively affect the price of the cryptocurrency that they hold [27]. Furthermore, since anyone is allowed to stake, staking leads to decentralisation. The more tokens are staked, the less vulnerable the blockchain is to a majority attack. Additionally, some blockchains give stakeholders voting rights as a way to govern the blockchain in a decentralised way [32].

Nonetheless, native staking has its own risks, apart from the obvious exposure to the extremely volatile dollar values of cryptocurrencies. A stakeholder takes slashing risk as most blockchains penalise malevolent or lazy behaviour. Therefore, in cases where a stakeholder is delegating their stake, trust to the third-party that acts as a validator node is necessary. Meanwhile, staking requires trust in the code of the smart contracts that manage staking. Any software bug or mistake can have serious implications. In addition, a user that desires to withdraw staked funds is usually obliged to wait a certain waiting period. This allows the network to ensure that the user has not added any invalid blocks to the blockchain [32].

There are multiple ways for a user to stake[33, 32]:

As a validation node: This is the traditional method, it is the most profitable but it also has the most responsibilities and requirements. A validation node is required to have the hardware, the knowledge and constant connection to the internet. Any misbehaviour can lead to penalties or slashing of the node's collateral. In many cases, the validation node is required to stake more than a specific amount. The benefit of running a validation node is that the user needs to trust nobody and receives the full amount of rewards.

Using a staking-as-a-service platform: With this method the user is still required to have the minimum staking amount but does not require to own and manage the hardware to run a validation node. Instead, the user delegates their funds to a provider that runs the validation node on their behalf. This method requires trust in the third-party provider which also gets a fee for its service.

Using pooled staking: With this method, users delegate their funds to a pool that runs a validation node on their behalf. The validation node collects the rewards, keeps a fee and distributes the rest to the users. It is ideal for users that do not possess enough funds to run a validation node or use staking-as-a-service or simply do not want to stake such a large amount. Some staking services offer liquidity tokens that represent the staked funds with their accrued rewards and can be traded like any other token or used in DeFi.

Using centralised cryptocurrency exchanges: Some of the exchanges offer staking through their platforms. With this method, the users have no control of their funds and they are completely dependent on the centralised provider. This also can lead to centralisation as these providers accumulate wealth in their accounts. Using exchanges is a good solution for users with very limited knowledge that do not feel comfortable owning their own wallet.

2.1.8 Pooled Staking

This project focuses exclusively on pooled staking. The main idea of pooled staking is combining staking power of multiple users to improve the probability of being selected as validators, thus receive more rewards which are then shared. The actual validation node is run by a third-party service provider. Staking pools are designed to enable retail cryptocurrency holders to become stakeholders and make staking accessible to everyone [4].

Running a staking pool has its own operational costs. Apart from that, the owners of the pools are expected to make some profit for offering their services to delegators. The rest of the rewards accrued by the pool are shared to the delegators. Rewards are shared shared following a reward sharing scheme. In general, this scheme is not enforced by the Blockchain but is selected by the service provider and that is what makes staking pools different from each other [3].

During the following chapter we will describe and compare existing staking pools by different service providers. We will discuss their advantages and disadvantages and explain the reasons that led to the need for the solution proposed in this project.

2.2 Existing Stakers for Polygon's MATIC

2.2.1 TruStake vault by TruFin

TruStake [34] is a staking vault for MATIC pool staking on the Ethereum mainnet. It utilises the Twinstake [35] service as a validator node. TruStake offers complimentary restaking of the rewards. More specifically, the protocol uses gas from new users' transactions to claim the accrued rewards and restake them without burdening the old users with the cost of restaking.

Upon staking, the user receives an ERC-20 liquidity token called TruMatic that represents their shares in the vault. When a user wishes to withdraw, they have to wait for an 80 checkpoint waiting period. The user's TruMATIC tokens are burnt when the request is made, and the user claims the corresponding MATIC after the checkpoints have passed.

TruStake currently charges a 10% fee on the user's rewards. The user is also required to pay the gas fee for any new staking or unstaking. The gas fee for restaking is paid by the protocol (or other users that stake/unstake).

For example, if a user A wishes to stake 100k USD worth of MATIC, they should pay the gas fees for calling the `stake()` function. These gas fees also cover the cost of restaking the accrued rewards for all the users of the protocol. After a year, this amount will have accrued approximately 5k USD worth of MATIC (APY in Polygon is approx. 5%). Out of these 5k USD, the protocol keeps 500 USD as commission (10%) and the other 4.5k USD belongs to user A. Should user A desire to unstake the 104.5k USD worth of MATIC, they will have to pay the gas fees for calling the unstake() function. Meanwhile, if any other user initiates a staking or unstaking, the accrued rewards of every user are restaked. In that case, the rewards of user A after a year will be a bit more than 4.5k USD.

TruStake also offers the ability to allocate portions of the rewards to different accounts from the one that stakes the funds [36].

TruStake's current smart contract deployment can be viewed here: `https://etherscan.io/address/0xa43a7c62d56df036c187e1966c03e2799d8987ed`

2.2.2 Lido on Polygon

Lido [37] is a liquid staking provider which offers staking in multiple blockchains, including Polygon. Lido on Polygon is a DAO governed liquid staking protocol for staking MATIC on the Ethereum mainnet. Similar to TruStake, it also offers automatic restaking of the rewards.

Lido charges a 10% on the user's reward. On top of that, the user is required to pay gas fees in multiple transactions. During staking, the user needs to pay gas fees for unlocking their MATIC which allows the stMATIC contract to spend the tokens on the user's behalf. After that the user needs to pay the gas fees for calling the stake function. Similarly, when a user wishes to unstake, they must unlock their stMATIC and submit a withdrawal request by calling the unstake function. When the unbonding period ends (approximately 9 days) the user can claim their rewards. Every single one of these transactions requires gas fees paid by the user [38].

Users who stake MATIC through Lido on Polygon receive an ERC-20 liquidity token called stMATIC. This token can be traded like any other ERC-20 token. In fact, a user can trade MATIC for stMATIC, or vice versa, through a centralised exchange to skip the procedure described in the previous paragraph. However, this is not actual staking, does not help strengthen the blockchain and contributes to centralisation.

Lido's current smart contract deployment can be viewed here: `https://etherscan.io/address/0x9ee91f9f426fa633d227f7a9b000e28b9dfd8599`

2.2.3 Stader

Stader [39] is a non-custodial contract-based staking platform that offers liquid staking for Polygon and other blockchains. Stader's ERC-20 liquidity token is called MATICX and has similar functionality to the stMATIC.

Stader offers staking on both the Ethereum mainnet and the Polygon mainnet via a DEX. However, at the time this report is written, direct unstaking is only available for Ethereum mainnet staking. Depending on the network used for staking, users are required to pay gas fees using ETH or MATIC for any transactions related to staking and unstaking.

Additionally, Stader promotes decentralisation through the use of small validators that own less than 1% of the total staked amount. Stader's validators also charge commission less or equal to 5% which helps Stader maintain a relatively high APY [40].

The waiting period for withdrawal is approximately 2-3 days and they charge a 10% fee on the user's rewards. Regarding restaking, Stader's documentation does not explicitly make any references [41].

Stader's current smart contract deployment can be viewed here: `https://etherscan.io/address/0xf03a7eb46d01d9ecaa104558c732cf82f6b6b645`

2.2.4 Comparison

Feature Comparison

Table 2.1 is a comparison between the features of the stakers:

Service	Staker		
	TruStake	Lido	Stader
Commission on rewards	10%	10%	10%
Withdrawal waiting period	2-3 days	9 days	2-3 days
Liquid staking	✓	✓	✓
Automatic reward restake	✓	✓	✗
Reward allocation	✓	✗	✗
Liquidity token tradable in DEX	✗	✓	✓

Table 2.1: Feature comparison

Gas fees Comparison

Even though the smart contracts of these stakers vary a lot, the procedure of staking and unstaking is similar. More specifically, using any of these 3 stakers requires, at least, 3 separate transactions. One for depositing, one for requesting a withdrawal and one for claiming the requested amount.

As explained in section 2.1.5 - Gas fees in Ethereum, by extracting the *units of gas used* for the transactions, we can make an unbiased comparison of the gas fees between these stakers. To do that we traced the deployed smart contracts for each staker on Etherscan, retrieved a large amount of data from users' transactions with them, and made the following data analysis.

Tables 2.2, 2.3 and 2.4 present a comparison between the three stakers for the functions `deposit()`, `requestWithdraw()` and `claimWithdraw()` correspondingly.

	Average	**Median**	**Min**	**Max**	**Std**
TruStake	364,069	371,680	324,875	402,599	22,744
Lido	642,379	622,858	511,319	752,113	99,965
Stader	568,917	576,743	510,748	620,062	35,239

Table 2.2: Gas units comparison for the `deposit()` function

	Average	**Median**	**Min**	**Max**	**Std**
TruStake	436,289	430,863	405,885	524,322	33,033
Lido	1,174,935	1,181,122	885,344	1,456,518	133,575
Stader	779,693	790,863	720,614	846,189	39,874

Table 2.3: Gas units comparison for the `requestWithdraw()` function

	Average	**Median**	**Min**	**Max**	**Std**
TruStake	172,892	172,896	172,866	172,908	18
Lido	226,251	220,299	211,488	290,394	22,985
Stader	167,478	164,369	164,369	181,469	6,917

Table 2.4: Gas units comparison for the `claimWithdraw()` function

To provide a more intuitive comparison without loss of generality, we can multiply the gas units values with last month's (July 2023) average *gas price* and *ETH value in dollars*. These values are: $\overline{\text{gas price}} = 30.45\ \text{Gwei}$ and $\overline{\text{ETH value}} = 1896\ \$$.

	`deposit()`	`requestWithdraw()`	`claimWithdraw()`	**Total**
TruStake	21.02 $	25.19 $	9.98 $	**56.19 $**
Lido	37.09 $	67.83 $	13.06 $	**117.98 $**
Stader	32.85 $	45.01 $	9.67 $	**87.53 $**

Table 2.5: Average gas fees in dollars ($)

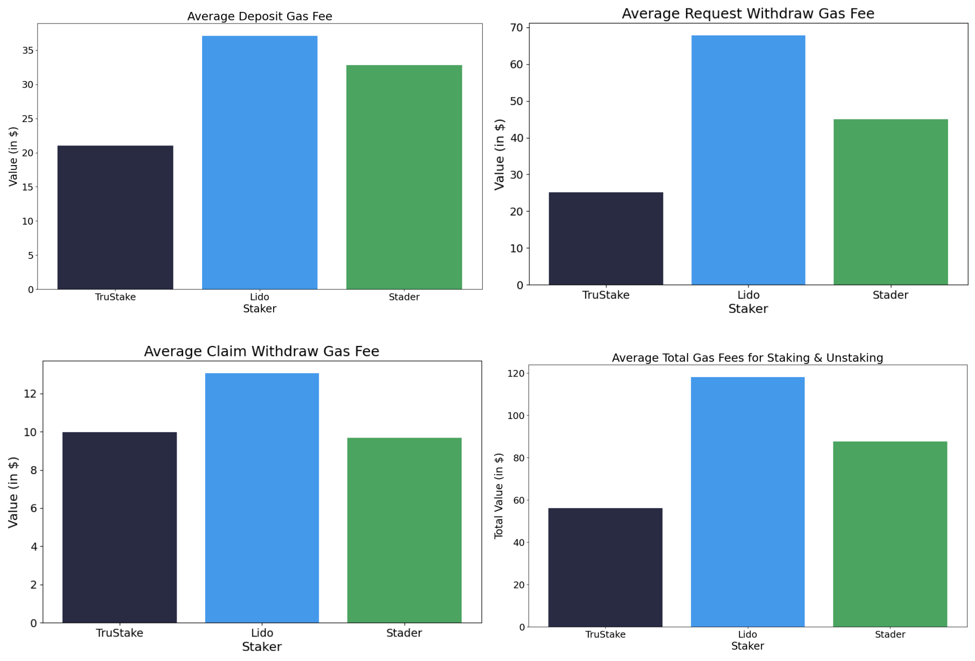


Figure 2.1: Gas fees comparison (in $)

Chapter 3

Design

3.1 Why Polygon?

As discussed above, there are multiple blockchains that could support the development of the proposed protocol of this project. Polygon was selected as it has various advantages and can be used as to build a prototype that could be adapted for other blockchains.

Polygon is compatible with the Ethereum Virtual Machine (EVM) which means that it can leverage the Ethereum ecosystem standards, tools, programming languages, etc [25]. Also, Polygon's staking management contracts are deployed on Ethereum. This means that the staking procedure is made on the Ethereum mainnet and not on the Polygon mainnet.

Staking in Polygon offers a very decent APY around 5% and allows everyone to stake MATIC tokens as a delegator and earn staking rewards [26] using one of the 100 validators. Meanwhile, if a user wishes to become a validator, they are required to have an amount larger than one of the 100th validator.

As opposed Ethereum, Polygon currently does support slashing and has a very low minimum deposit required for staking, just 1 MATIC (currently around $0.85). By comparison, Ethereum requires validators to have batches of exactly 32 ETH (around $60,000). Furthermore, until recently, Ethereum did not offer a method to unstake. Unstaking was only possible through trading staking liquidity tokens for ETH.

Additionally, Polygon does not offer automatic compound rewards restaking. Users wishing to restake their rewards need to do it manually and in most cases, the cost of doing so exceeds the benefit. Instead of viewing this as a disadvantage, we view this as an opportunity to offer an automatic restaking service without requiring extra gas fees from the user.

It should also be mentioned that unstaking in Polygon has a delay by design. Withdrawals are generally processed after a 80 checkpoint waiting period (approximately 2-3 days).

Finally, this project is designed to be compatible with the TruStake vault by TruFin which is already built for Polygon's native token.

3.2 Proposed Approaches

3.2.1 Preliminary Observations

In order to design a protocol that makes staking feasible for retail cryptocurrency holders, it is vital to reduce gas fees. To do that we need to:

- Minimise the interactions with the validation node, have as less calls to `stake()` and `unstake()` functions as possible.
- Avoid calling the `stake()` and `unstake()` functions for small amounts as the cost of calling these functions is invariant to the amount.
- Avoid loops if possible, if they cannot be avoided reduce the number of loops.
- Keep it simple and avoid complex and expensive functions or data structures.
- Minimise the changes of the blockchain state.

The approaches described in the following sections have noticeable differences between them. Even so, they share some core ideas and mechanisms which are:

- There is a vault controlled by a smart contract that accumulates funds from the users and interacts with the validator node by staking and unstaking. Its purpose is to act as an intermediary between the users and the validation node (see **Figure 3.1**). Note that funds sitting in the vault do not accrue rewards.
- Users submit deposit (stake) and withdrawal (unstake) requests to a vault. From a user's point of view, depositing is staking on the validator node and withdrawing is unstaking from the validator node. In reality they are just depositing and withdrawing funds from the vault. When necessary, the vault will interact with the validator node to stake or unstake.
- Instead of interacting with the validator node for every deposit or withdrawal request, requests of the opposite type can be netted. This is done by swapping a user's shares for another user's deposit amount without affecting the validator node.
- Users pay a fee on their rewards, approximately 10%. As we saw in the Existing Stakers section, this is common practice.

- Users do not pay the gas fees for calling `stake()` and `unstake()` functions on the validator node. Instead, they pay a small percentage fee on deposits/withdrawals which is gathered and used to cover the gas fees for the interaction with the validation node. Inevitably, the users also need to pay the gas fees for calling other functions on the smart contract but these fees will be substantially smaller.
- As Polygon does not offer automatic restaking of rewards. The protocol will use the gas fees for every call of `stake()` or `unstake()` functions, to also claim the accrued rewards.
- Whenever the `stake()` function is called, the claimed rewards will also be restaked.
- If a user's withdraw request can be netted with another user's deposit, they do not have to wait 80 checkpoints.

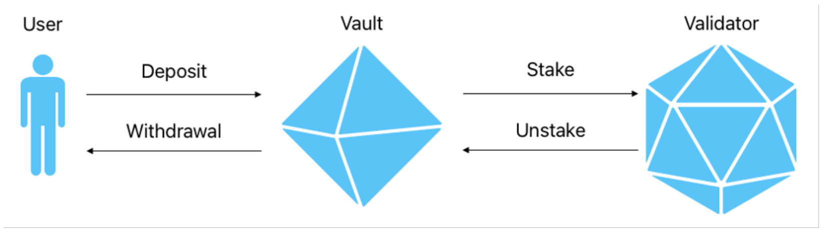


Figure 3.1: overview of the general approach

3.2.2 Approach A: Staking Rounds

Main Idea

1. There is a vault with two stacks, one for deposit requests and one for withdrawal requests. The funds from the submitted deposit requests are stored in the vault.
2. At the end of every round (eg. every month) the requests are netted. After netting, one of the stacks will be empty and the other will have some pending requests that were not netted and should be staked or unstaked. Consequently, at the end of every round, only a single validator operation needs to be performed, either stake of unstake.

Mechanics

- In order to make the protocol self-funded and independent of external fund injections, we make an initial deposit to the validation node called Stake &

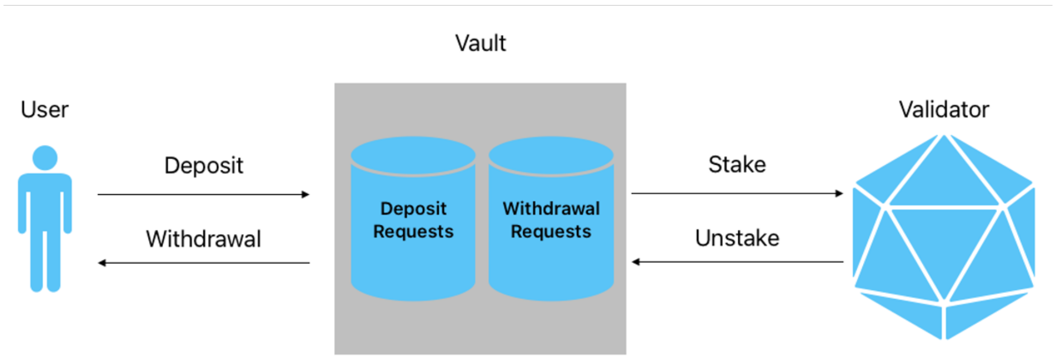


Figure 3.2: overview of approach A: Staking Rounds

Unstake Funds (SUF). SUF is calculated so that the rewards accrued by it over the period of a single round, cover the fee for staking or unstaking.

Assumptions

- To make the protocol profitable, total deposits need to generate more profit than the rewards from staking the initial capital (SUF). I.e., the protocol is profitable if: $\text{Total deposits} * \text{percentage fee on rewards } (\%) > \text{SUF}$.

*For the advantages and disadvantages of this approach see **Table 3.1**, for the profit analysis of this approach see **Appendix A.1**, for the simulation see* here.

Advantages	Disadvantages
• Simple • Self-funded (does not require any additional fund injections) • No minimum staking/unstaking amount • Does not require users to pay fee for deposit/withdrawal • Minimal number of interactions with the validator	• Not instant • Gas inefficient because of looping through the stacks and the end of every round • Requires a percentage fee on the rewards substantially higher than 10% to be profitable • Profitable only under specific conditions
Suitable for environments with extremely sparse requests	*Unsuitable for environments with even moderately frequent requests*

Table 3.1: Advantages and Disadvantages of approach A: Staking Rounds

3.2.3 Approach B: S-U level

Main Idea

1. There is a vault with two thresholds S, U with $S > U$.
2. Initially, an amount S is staked on the validator and an amount U is placed in the vault.
3. Users submit deposit (stake) and withdrawal (unstake) requests to the vault.
4. When a user submits a deposit request, their funds are transferred to the vault and the user is assigned shares in the vault.
5. When a user submits a withdrawal request, they receive funds straight from the vault and their shares are burnt.

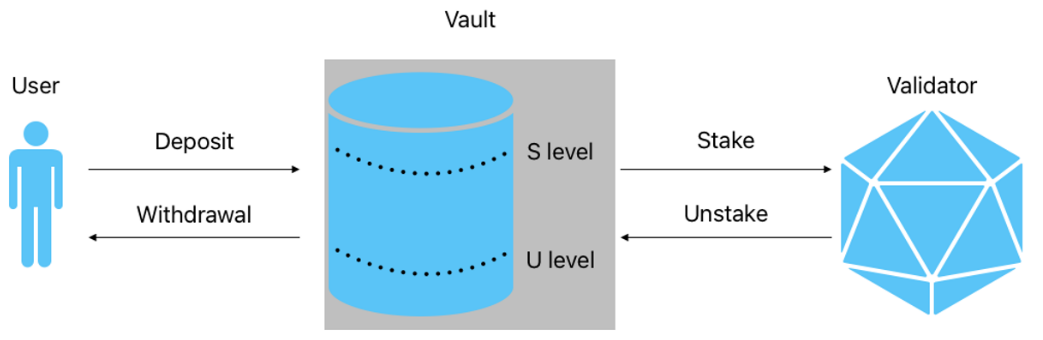


Figure 3.3: overview of approach B: S-U level

Mechanics

- When funds in vault reach the threshold S, we stake everything except amount U.
- When funds in vault are insufficient to cover the withdrawal requests, we unstake enough to cover the requests plus an amount U that stays in the vault.
- The rewards from the staked amount in the validator are shared equally to all users regardless of whether their funds are in the validator or in the vault. This means that large amounts of funds sitting in the vault, reduce the average percentage yield.

Assumptions

- To make the protocol profitable, total deposits need to generate more profit than the rewards from staking the initial capital $(S + U)$. I.e., the protocol is profitable if $\text{Total deposits} * \text{percentage fee on rewards } (\%) > S + U$.

*For the advantages and disadvantages of this approach see **Table 3.2**, for the profit analysis of this approach see **Appendix A.2**, for the simulation see* here.

Advantages	Disadvantages
• Instant, except when funds in the vault are insufficient • Self-funded • No minimum staking/unstaking amount • Gas efficient • Minimal number of interactions with the validator	• Complex • Many parameters • Non-linear, unstable and unpredictable percentage yield • A large amount of funds is constantly not staked and does not accrue rewards, this lowers the average percentage yield • Vulnerable to attacks (quick deposit-withdraw attacks) • Requires a percentage fee on the rewards substantially higher than 10% to be profitable • Profitable only under specific conditions
Suitable for environments with balanced requests of small amounts	*Unsuitable for environments with unbalanced requests or requests of large amounts*

Table 3.2: Advantages and Disadvantages of approach B: S-U level

3.2.4 Approach C: Queues with Batches

Main Idea

1. There is a vault with two queues, the D queue for deposits and the W queue for withdrawals.
2. The two queues contain batches.

3. Each queue has a current batch and cached (or complete) batches.
4. Every deposit request is added to the current batch of D queue and every withdrawal request is added to the current batch of W queue.
5. Once the total amount in current batch reaches a threshold, the batch is considered complete. If there is a complete batch of the opposite type, they are netted, else it is cached.
6. Complete batches have an expiry date such that if they are not paired with a batch of the opposite type, they are staked/unstaked along with every other complete batch.

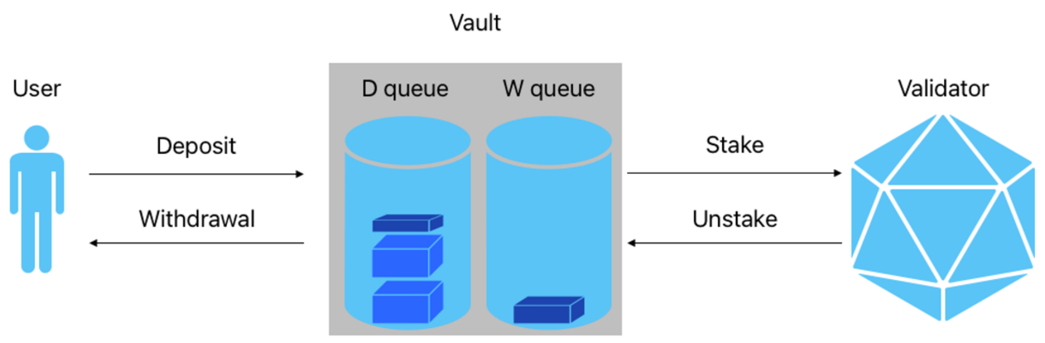


Figure 3.4: overview of approach C: Queues with batches

Mechanics

- At every point, only one of the two queues has cached batches.
- Users get preshares in the vault for pending requests in the D queue. Preshares do not accrue rewards. Once requests are processed, their preshares are swapped for shares in the validator.
- Batch size is set so that the total of the deposit/withdraw fees from the users cover the cost of staking/unstaking the batch. If the batch is netted instead of being staked/unstaked, the fees are kept as a reward for the protocol.
- It is complicated when a user wants to withdraw funds that are not yet staked and sit in the D queue. Removing deposit requests from complete batches would require rearranging the batches which would be excessively expensive. A simple solution is to prohibit a user from posting a withdrawal request while having preshares.
- To keep the cost of looping through each batch low, we need to set a minimum stake/unstake amount:

$$\text{min amount for every request \& set batch size}$$
$$\Rightarrow \text{capped number of requests per batch}$$

Assumptions

- The volume of requests exceed the batch size after a reasonable amount of time. If not, the current batches remain incomplete indefinitely.

*For the advantages and disadvantages of this approach see **Table 3.3**, for the profit analysis of this approach see **Appendix A.3**, for the simulation see* here.

Advantages	Disadvantages
• Does not require an initial capital • Self-funded • Low transaction latency in a state with balanced deposits and withdrawals • Relatively low transaction latency in a state with frequent and/or large requests of one type only • Reduced number of interactions with the validator	• Incomplete batches do not expire • User cannot withdraw while having a pending deposit request • Minimum stake/unstake amount • Transactions may be executed in parts
Suitable for environments with very frequent requests of any amount or less frequent requests of large amounts	*Unsuitable for environments where the requests of one type are very sparse and have low amounts*

Table 3.3: Advantages and Disadvantages of approach C: Queues with batches

3.2.5 Approach D: Directly Connected Queues

Main Idea

1. There is a vault with two queues, the D queue for deposits and the W queue for withdrawals.
2. The two queues contain requests.
3. Every deposit request is added to the D queue and every withdrawal request is added to the W queue.
4. Once a request is added to a queue, we check for requests on the other queue. If there is, we net them.
5. There are two types of deposit and withdraw functions.

6. Once the total summed amount in a queue reaches a threshold, we stake/unstake depending on the queue. That threshold is selected so that the total of fees from deposits/withdrawals collected until reaching the threshold are enough to cover the staking/unstaking fee.

7. Requests have an expiry date. Once the oldest request of a queue expires, we stake/unstake the whole queue.

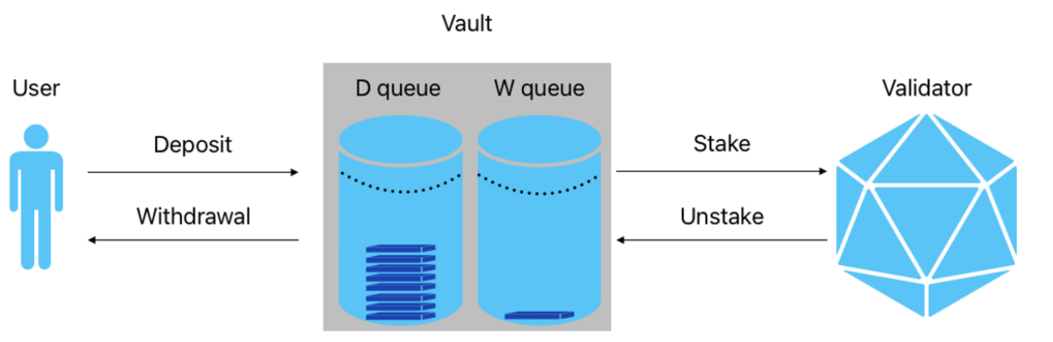


Figure 3.5: overview of approach D: Directly Connected Queues

Mechanics

- At every point, at most one of the queues has requests.
- Users get preshares in the vault for requests in the D queue. Preshares do not accrue rewards. Once requests are processed, their preshares are swapped for shares in the validator.
- If a user with pending deposit requests submits a withdrawal request, the request is netted with their pending deposit requests first (this solves the similar problem in Approach C).
- There are two types of deposit functions that a user can call:
  - `indirectDeposit()`: vault charges the user a percentage fee on their deposit amount called a deposit fee (approximately 0.1%). The user's request is then added to the D queue and is processed within its expiry period.
  - `directDeposit()`: the user pays the gas to call the `stake()` function. The user's request is processed immediately along with all the pending requests in the D queue.
- There are two types of withdraw functions that a user can call.

  - `indirectWithdraw()`: vault charges the user a percentage fee on their withdrawal amount called a withdrawal fee (approximately 0.1%). The user's request is then added to the W queue and is processed within its expiry period.
  - `directWithdraw()`: the user pays the gas to call the `unstake()` function. The user's request is processed immediately along with all the pending requests in the W queue.

- To keep the cost of looping through the queue low, we need to set a minimum stake/unstake amount:

$$\text{min amount for every request \& set queue size threshold}$$

$$\Rightarrow \text{capped number of requests in the queue}$$

Assumptions

- A request expires when for period of time:
  - there are no requests of the opposite type.
  - there are no calls of the direct function of the same type.
  - the total amount of the indirect requests of the same type does not exceed the threshold.

  When a request expires, all the requests in the same queue are processed and some of the cost of staking/unstaking, burdens the protocol. We assume that as long as this is not common, the protocol does not require any fund injections and it remains profitable.

*For the advantages and disadvantages of this approach see **Table 3.4**, for the profit analysis of this approach see **Appendix A.4**, for the simulation see* here.

3.3 Final Design

3.3.1 Idea

As described in the previous section, each one of the proposed approaches has its own advantages and disadvantages. Moreover, each one is more suitable for different environments. **The final design is based on approach D: Directly Connected Queues**. This approach appears to have the most advantages and it is also more suitable for a broad range of different environments while being relatively simple and sleek.

Advantages	Disadvantages
• Relatively simple and straightforward • Does not require an initial capital • It can be integrated to the TruStake vault • Very low transaction latency in a state with balanced deposits and withdrawals • Low transaction latency in a state with with frequent requests only of one type • Transactions are processed within a predefined time limit • User can add deposit and withdraw requests whenever they like • The parameter values like min stake/unstake amount or D/W queue threshold, are adjustable • It is suitable both retail users and whales • Reduced number of interactions with the validator	• Minimum stake/unstake amount • When a request expires, the protocol to pay a portion of the fees for staking/unstaking • Not completely self funded, it might require fund injection in the unlikely event of very long periods with scarce, low amount and unbalanced requests • Transactions may be executed in parts
Suitable for environments with at least moderately frequent requests of any type and size	*Unsuitable for environments with very sparse requests*

Table 3.4: Advantages and Disadvantages of approach D: Directly Connected Queues

Additionally, the final design exploits ideas proposed by the existing vaults presented in section 2.2. More specifically, the final protocol offers **automatic reward restake** and **liquid staking** using an ERC-20 token similar to TruMATIC. **The commission rate on the rewards is 10%** following the general norm.

Withdrawal waiting period varies for the indirect withdrawals as the protocol seeks to net them with deposit requests. For the the direct withdrawals, waiting period is 80 checkpoints (2-3 days) as enforced by Polygon network. However, should adequate deposit requests exist, the process of a **withdrawal request can be immediate** as the unstaking is omitted. This is unique compared to other stakers. Lastly, due to smart contract size limitation, the final design does not support reward allocation.

3.3.2 Functions

The final protocol's design is composed of multiple functions. The most important functions of the protocol are described in **Table 3.5**.

There are two ways to separate these functions into categories. The first way to categorise them is based on the who calls them. A function can be called by a user who wishes to interact with the vault or by the owner of the protocol to prevent single users from paying gas fees that should be covered collectively by the protocol. Finally, some functions are internal and are initiated from within the protocol by other functions.

Functions can also be divided into view and write based on whether they change the state of the blockchain. The former type of functions requires gas fees paid by the caller of the function. If a write function is called internally from a different function, gas fees are paid by the user who called the initial function.

Function	`Parameters`	`Returns`	Caller	Type	Description
`sharePrice`	-	`price`	any	view	*Returns the current share price*
`indirectDeposit`	`amount`	-	user	write	`user` *transfers* `amount` *plus deposit fee to the vault, receives pre-shares and the request is added to the dQueue*
`directDeposit`	`amount`	-	user	write	`user` *transfers* `amount` *to the vault and initiates the* `stake()` *function*
`indirectWithdraw`	`amount`	-	user	write	`user` *adds a withdrawal request to the wQueue and their shares are burnt*
`directWithdraw`	`amount`	-	user	write	`user` *adds a withdrawal request to the wQueue, their shares are burnt and* `unstake()` *is initiated*
`indirectStake`	-	-	owner	view	*when dQueue threshold is reached or the oldest deposit request expires, this function initiates the* `stake()` *function*
`indirectUnstake`	-	-	owner	view	*when wQueue threshold is reached or the oldest withdrawal request expires, this function initiates the* `unstake()` *function*
`unstakeClaim`	-	-	owner	write	`owner` *claims the funds from the validator (after the waiting period of 80 checkpoints) and distributes them to the users with withdrawal requests*

`stake`	-	-	internal	write	*funds in the vault along with any claimed rewards, are staked on the validator, every deposit request is processed, all preshares are swapped for freshly minted shares and dQueue is cleared*
`unstake`	-	-	internal	write	*an amount of funds is requested to be unstaked from the validator to cover all the withdrawal requests in the wQueue and wQueue is cleared*
`reduceDRequests`	`user address, amount`	-	internal	write	*if a* `user` *submits a withdrawal request while owning preshares, the* `amount` *is subtracted from their pending deposit requests*
`pairQueues`	-	-	internal	write	*requests from the two queues are netted until one of the queues is empty*
`expiryCheck`	-	-	internal	view	*the oldest deposit and withdrawal requests are checked whether they expired and if so, a call of the* `stake()` / `unstake()` *function is initiated*

Table 3.5: Basic functions of the final design

3.3.3 Parameters

The final protocol's design relies on multiple variables. The most important variables of the protocol are described in **Table 3.6**. These parameters can be separated into two categories, the protocol's constants that are set by the owner of the protocol and the variables whose values change frequently.

Protocol is designed in a way that enables owner to change constants' values. The values presented in **Table 3.6** show the initial values selected based on theoretical analysis which used an approximation for the gas fees for interacting with the validation node.

In the Chapter 5 - Evaluation, we retrieve enough data from the prototype to conduct a more substantiated analysis. Using the same methodology proposed for the selection of the initial values, we will adjust the values of the parameters to be more realistic.

Parameter	Type	Value	Description
`totalStaked`	variable	$\in [0, cap]$	*total amount of MATIC that are staked in the validator node*
`vaultBalance`	variable	$\in [0, cap]$	*total amount of MATIC sitting in the vault*
`unclaimedRewards`	variable	≥ 0	*amount of unclaimed rewards in MATIC*
`totalShares`	variable	≥ 0	*total amount of shares (liquidity tokens)*
`totalPreshares`	variable	≥ 0	*total amount of preshares (equal to dQueueBalance)*
`sharePrice`	variable	≥ 1	*exchange rate between MATIC and the ERC-20 liquidity token*
`dQueueBalance`	variable	$\in [0, 15000]$	*current total amount of MATIC accumulated in the dQueue from requests*
`wQueueBalance`	variable	$\in [0, 15000]$	*current total amount of MATIC from requests in the wQueue*
`latestUnbondingNonce`	variable	≥ 0	*integer value indicating the nonce of the latest unbonding nonce created from an unstaking. The first unbonding nonce is equal to 0*
`fee`	constant	10%	*protocol's commission on users' rewards*
`cap`	constant	$1,000,000$	*maximum capacity of the vault in MATIC*
`depositFee`	constant	0.1%	*percentage fee on deposit amount for indirect deposits*
`withdrawalFee`	constant	0.1%	*percentage fee on withdrawal amount for indirect withdrawals*
`minDepositAmount`	constant	100	*minimum amount in MATIC for an indirect deposit*
`minWithdrawalAmount`	constant	100	*minimum amount in MATIC for an indirect withdrawal*
`expiryPeriod`	constant	`7 days`	*time until a request expires*
`dQueueThreshold`	constant	$15,000$	*limit of the total amount of MATIC accumulated in the dQueue from requests for triggering the* `stake()` *function*
`wQueueThreshold`	constant	$15,000$	*limit of the total amount of MATIC from requests in the wQueue for triggering the* `unstake()` *function*

Table 3.6: Basic parameters of the final design

3.3.4 Variables' value analysis

Vault balance

Vault balance is the total amount of MATIC sitting in the vault. It consists of funds from: (1) pending deposit requests in the dQueue, (2) deposit fees, (3) withdrawal fees, and (4) claimed rewards:

$$vaultBalance = dQueueBalance + depositFees + withdrawalFees + claimedRewards \tag{3.1}$$

Shares - Funds equilibrium

Shares and preshares represent units reflecting the total amount of funds in the protocol. Funds are stored in three places: (1) staked in the validator, (2) rewards in the validator that are not staked, and (3) sitting in the vault. Also, as soon as a withdrawal request is submitted, the user's shares are burnt immediately but the funds remain in the protocol. Therefore, at any given time, the following equilibrium should hold:

$$\begin{split} totalShares * sharePrice + totalPreshares = totalStaked + unclaimedRewards \\ + vaultBalance - wQueueBalance \end{split} \tag{3.2}$$

Share price

Share price is the exchange rate between MATIC and the ERC-20 liquidity token (LT). I.e., the price of one LT share in MATIC. It is a very efficient method to calculate the rewards for each user without keeping track of the timestamps and the amounts for each deposit and withdrawal.

Share price is not affected by deposits and withdrawals. It only changes when rewards accrue. Therefore, it is constantly increasing with a relatively stable rate. Its initial value is 1.

Whenever a user wishes to make a deposit / withdrawal of an amount X in MATIC, the protocol calculates the number of LT shares that should be minted / burnt as:

$$X_{LT} = X_{MATIC}/sharePrice \tag{3.3}$$

As long as there are no deposits / withdrawals, the number of a user's share remains stable as rewards accrue. Therefore the user's balance in MATIC is calculated as:

$$Balance_{MATIC} = Balance_{LT} * sharePrice \tag{3.4}$$

Share Price is calculated as:

$$sharePrice = \frac{totalAmountFromShares_{MATIC}}{totalShares_{LT}}$$

$$= \frac{totalStaked + vaultBalance - dQueueBalance - wQueueBalance}{totalShares} + \frac{(1 - fee) * unclaimedRewards}{totalShares} \tag{3.5}$$

Notes:

- $dQueueBalance$ needs to be subtracted as it is included in $vaultBalance$, but the shares for this amount have not yet been minted (they are still in the form of preshares).
- $wQueueBalance$ needs to be subtracted as the shares for it have been burnt (to avoid further reward accruing), but the amount is still in the protocol.
- $unclaimedRewards$ needs to be multiplied by $1 - fee$ to subtract the amount the the protocol keeps as commission on the rewards. The amount $unclaimedRewards * fee$ can be considered as a deposit by the treasury yet to happen. As soon as these rewards are claimed, shares are minted for the treasury.
- $claimedRewards$ (included in $vaultBalance$) is not multiplied by $1 - fee$ because shares are minted for the treasury as soon as the rewards are claimed.

3.3.5 Constants' value analysis

Polygon has an implicit currency risk. Its native token is MATIC however staking is performed on Ethereum using ETH for the gas fees. As a result, the network is greatly affected by disruptions in the exchange rate between the two currencies and so is this protocol.

More specifically, when a user calls the directDeposit() function, they pay for staking in ETH. On the other hand, a user that calls the indirectDeposit() function, pays fees in MATIC and the protocol takes the currency risk of covering the staking gas fees in ETH.

As explained during the early stages of the design, $stakingFee$ and $unstakingFee$, i.e. the cost of calling the `stake()`/`unstake()` function of the validator node, are vital parameters for this protocol. These gas fees are paid in ETH and their value varies depending on the network's congestion. During the following analysis, we **consider a constant average price equal to 15 MATIC for both parameters** $\bar{stakingFee}$ **and** $\bar{unstakingFee}$.

All these factors were taken into consideration during the selection of the values for the protocol's constants. Additionally, many parameters are co-dependent creating trade-offs. In Chapter 5 - Evaluation, we will use the prototype to obtain more information about the gas fees and adjust the values of the constant parameters.

Deposit fee - dQueue threshold trade-off

In order for the protocol to be viable, it is important that the funds collected from deposit fees in MATIC are sufficient to cover the staking gas fees in ETH. If this doesn't hold, protocol will operate at a loss. To prevent this, the parameters `depositFee` and `dQueueThreshold` need to satisfy a certain relationship.

To illustrate this, consider a scenario where users exclusively use the `indirectDeposit()` function and there are no withdrawals. This means that no request is netted and no user directly deposits and pays the staking gas fees for everyone else. In that case, the protocol will initiate the `stake()` function whenever `dQueueBalance` reaches `dQueueThreshold`.

The idea behind the deposit fee is that each user pays a small percentage fee for every indirect deposit so when `dQueueThreshold` is reached, the funds collected from the deposit fees are sufficient to cover the gas fees for calling the `stake()` function.

Therefore, it is vital that the following inequality holds:

$$dQueueThreshold \times depositFee(\%) \geq stak\bar{i}ngFee \tag{3.6}$$

As it emerges from inequality 3.6, `depositFee` and `dQueueThreshold` are inversely proportional. Ideally, users would like `depositFee` to be as small as possible. On the other hand, if the `dQueueThreshold` is too large, it is harder for `dQueueBalance` to reach it within the expiry period, and the protocol will suffer a loss.

The overall aim is to keep `depositFee` relatively insignificant to the APY (approximately 5%) while keeping `dQueueThreshold` relatively small. Using the assumption that $stak\bar{i}ngFee = 15MATIC$ we concluded that a **depositFee = 0.1** with a **dQueueBalance = 15,000** are viable values for these constants.

Indirect - direct deposit fees

As long as inequality 3.6 holds, the protocol is not suffering a loss. Taking it a step further, by not setting a threshold and letting the user decide whether they want to use directDeposit or indirectDeposit, the protocol not only is it not losing but it can be profitable.

To explain this, the first step is to compare the total cost of calling the `directDeposit()` and the `indirectDeposit()` functions. Each function has an individual cost in gas fees independent of the cost of calling validator's `stake()` function. Nevertheless, these costs are very similar since the code for the two functions is nearly identical. Thus, the individual costs of each functions are omitted for this comparison. Furthermore, the actual cost of `stake()` function can be calculated by subtracting the average cost for `indirectDeposit()` function from the average cost for `directDeposit()` function.

Besides that, `directDeposit()` has the additional cost (in ETH) of the staking gas fees and `indirectDeposit()` has the additional deposit fee (in MATIC). To simplify the calculations, we convert all values to MATIC. Consequently, the cost of `directDeposit()` is constant while the cost of `indirectDeposit()` is proportional to the deposit amount:

$$Cost[directDeposit()] = staking\_Fee \tag{3.7}$$

$$Cost[indirectDeposit()] = depositFee(\%) * depositAmount \tag{3.8}$$

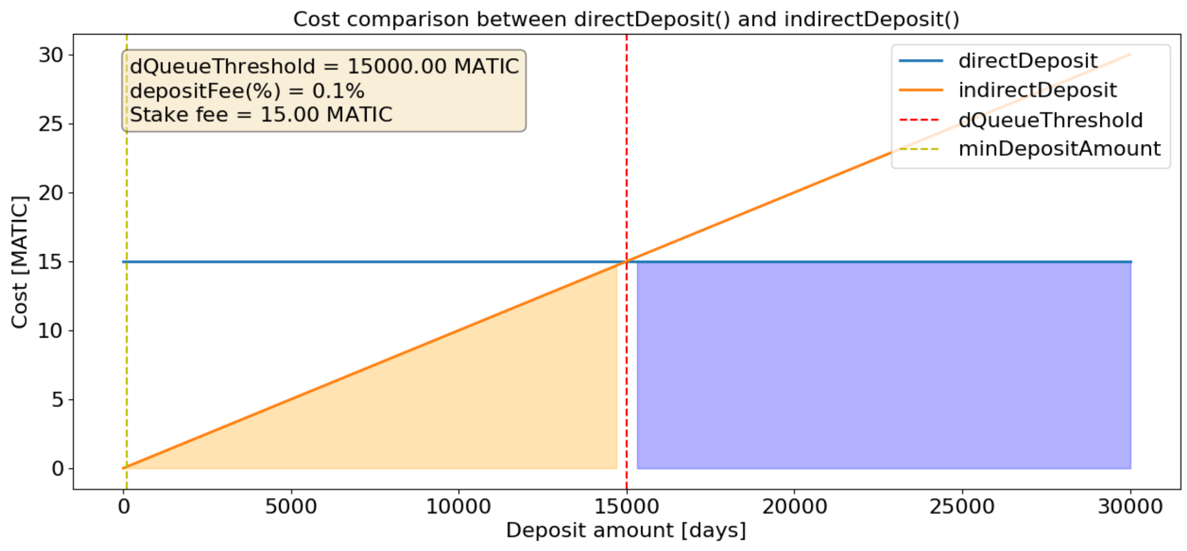


Figure 3.6: Cost comparison

It is safe to assume that a rational user would always chose the cheapest option when depositing. As shown in **Figure 3.6**, this means that for deposit amounts smaller than the `dQueueThreshold`, user would chose `indirectDeposit()` (orange region) and for larger amounts they would chose `directDeposit()` (blue region).

However, the protocol does not explicitly inform the user for the values of `dQueueThreshold` and $staking\_Fee$. It does not have a strict threshold to oblige users to use one or the other function depending on the deposit amount either. Hence, the user might not make the cheapest choice every time and when they do not, the protocol makes a profit equal to the vertical distance between the two lines of **Figure 3.6**.

Min deposit amount - dQueue threshold trade-off

Another trade-off that arises through the selection of constants' values is the one between `minDepositAmount` and `dQueueThreshold`. From the user's perspective, the minimum deposit amount for indirect deposits should be as low as possible to make staking tiny amounts possible. From the protocol's perspective, low `minDepositAmount` means large number of loops when iterating through the dQueue which increases the gas fees. The upper bound for the number of loops is:

$$\lceil NumberOfLoops_{dQueue} \rceil = dQueueThreshold / minDepositAmount \tag{3.9}$$

For **dQueueBalance = 15,000MATIC**, it was decided it is reasonable to set **minDepositAmount = 100MATIC** (currently around 68$) which lowers the upper bound of loops to 150.

Expiry period - dQueue threshold trade-off

Lastly, there is the trade-off between `expiryPeriod` and `dQueueThreshold`. On one end, users would like to have their indirect requests processed as soon as possible. On the other end, lowering the `expiryPeriod` means that there is less time for `dQueueBalance` to reach `dQueueThreshold`. This implies that the possibility of the collected deposit fees to be insufficient to cover the staking gas fees increases.

For **dQueueBalance = 15,000MATIC**, it was decided it is reasonable to set **expiryPeriod = 7 days**

Withdrawals

The same trade-offs that were explained for the deposits, exist for withdrawals too. The approach for selecting the values remains the same. The equations for withdrawals are the following:

$$wQueueThreshold \times withdrawalFee(\%) \geq unsta\bar{k}ingFee \tag{3.10}$$

$$Cost[directWithdraw()] = unsta\bar{k}ingFee \tag{3.11}$$

$$Cost[indirectWithdraw()] = withdrawalFee(\%) * withdrawalAmount \quad (3.12)$$

$$\lceil NumberOfLoops_{wQueue} \rceil = wQueueThreshold/minWithdrawalAmount \quad (3.13)$$

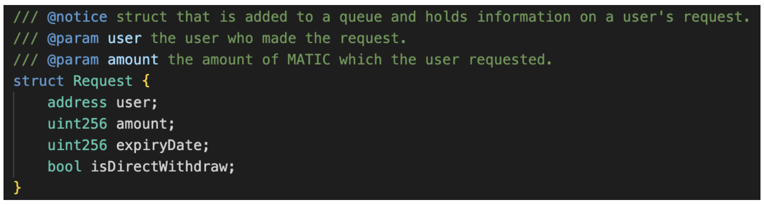
3.3.6 The queues

The D queue and the W queue are two identical FIFO data structures modified to support the functionality of the protocol. Using a queue ensures that all the requests are processed sequentially in a chronological order. Furthermore, the pending requests are always located in consecutive positions without any processed requests in between.

As Solidity does not offer a built in implementation for a FIFO data structure, these queues were implemented manually using a mapping. The initial implementation was using an array but that made it more complex and increased the gas fees.

General form Each queue has two integers `first` and `last` indicating the front and the back of the queue correspondingly. The queue contains structs of type Request (see **figure 3.7**).

Since this is a custom queue implementation using a mapping, any element can be accessed directly using its key in O(1). Furthermore, the elements that are dequeued are not automatically deleted. **Figure 3.8** shows a visualisation of a queue.



```
/// @notice struct that is added to a queue and holds information on a user's request.
/// @param user the user who made the request.
/// @param amount the amount of MATIC which the user requested.
struct Request {
    address user;
    uint256 amount;
    uint256 expiryDate;
    bool isDirectWithdraw;
}
```

Figure 3.7: Request struct

Enqueueing Whenever a new request is enqueued, it is placed on the back of the queue with a mapping key equal to `last` and `last` is increased by one (see **Figure 3.9**).

Dequeueing When a request is dequeued, it is extracted from the front of the queue, `first` is increased by one and the request is deleted (see **Figure 3.9**).

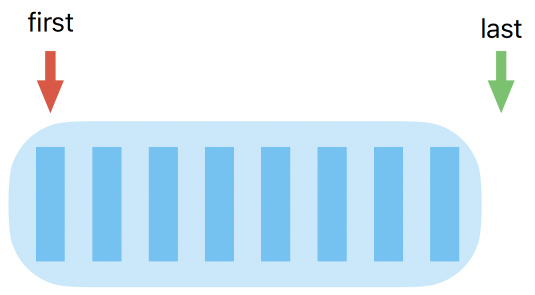


Figure 3.8: Visualisation of a queue

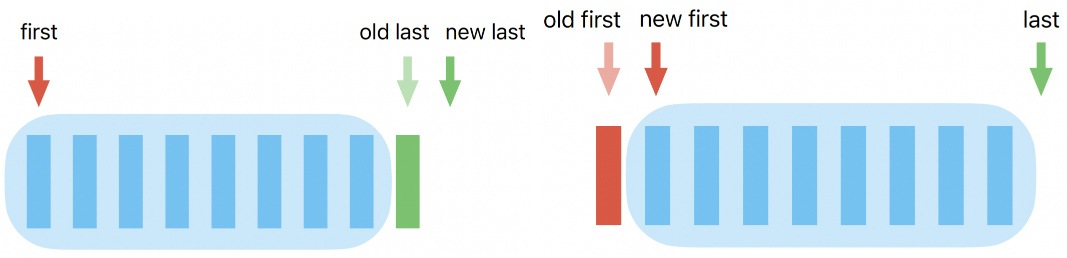


Figure 3.9: Enqueueing and Dequeueing

W queue modification As described previously, after unstaking, it takes 2-3 days to claim the amount from the validator node. However, the W queue needs to be emptied immediately as soon as `unstake()` function is called. If the requests were deleted though, it would be impossible to share the claimed funds to the users. Therefore, we came up with a clever trick to resolve this issue.

When the `unstake()` function is called, the protocol stores the current `first` and the `last` integers along with the corresponding unbonding nonce and clears the W queue by setting `first = last`. This makes the W queue virtually empty but the requests are not deleted.

After the waiting period, the funds are claimed and the previously stored `first` and the `last` values are retrieved. By looping through these requests, the funds can be shared to the users and the requests are finally deleted permanently (see **Figure 3.10**).

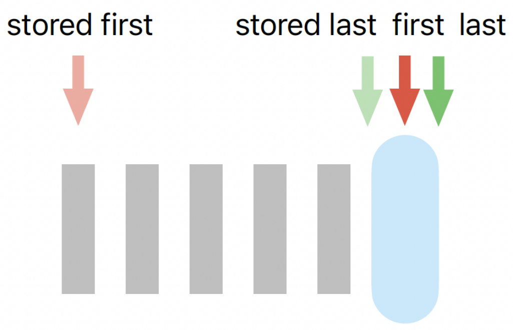


Figure 3.10: The unstaking trick on W queue

Chapter 4

Implementation

4.1 Technical Architecture

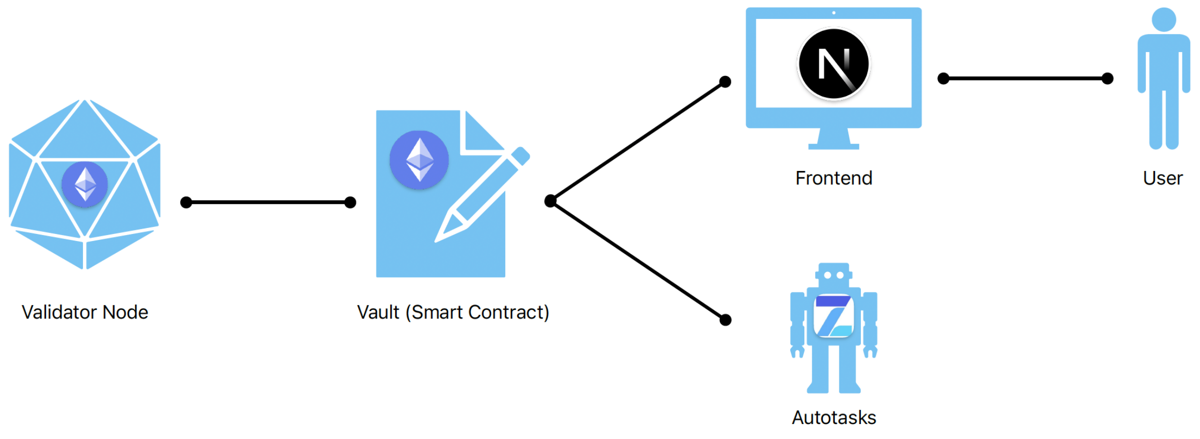


Figure 4.1: Technical Architecture

The technical architecture of the protocol's full-stack prototype for Polygon is illustrated in **Figure 4.1**. The prototype is called TruStake Picanha vault. The basic components of the prototype are:

The validator node: is the node used by protocol for staking. The vault interacts with the validator node using functions from the node's smart contract.

The vault: is a smart contract deployed on the blockchain. The Prototype's smart contract is deployed on Goerli testnet.

Frontend: is a website that supports user's interaction with the vault

Autotasks: are code scripts that monitor the vault and initiate transactions on behalf of the protocol's owner

4.2 Smart Contract

The prototype's smart contract was developed, tested and deployed using Hardhat framework [42]. We used Trufin's [34] code as basis for this prototype and built on top of it. The code of the final GitHub repo is clearly marked to indicate what is existing code of Trufin and what is new additions for this prototype.

Before writing the Smart Contract the first step was to create a mock-up in Python to test the functionality of the Prototype on a higher level. The mock-up implemented the functions and the parameters described in Section 3.3 - Final Design. This mock-up helped identify logical errors, test different parameter values and correct the equations in the analysis for the parameters. It can be found in the repo in the following url: `https://github.com/demKyr/PicanhaStaker-MScIndividualProject/tree/main/SmartContracts/python-mockup`

Following that, the smart contracts were written in Solidity. There are four (4) smart contracts in total (url: `https://github.com/demKyr/PicanhaStaker-MScIndividualProject/tree/main/SmartContracts/contracts/main`):

QueueWithMap.sol custom implementation of a queue data structure using a mapping

Types.sol implementation of structs that are used by TruStakeMATICv2.sol and QueueWithMap.sol

TruStakeMATICv2Storage.sol declaration of the variables of TruStakeMATICv2.sol

TruStakeMATICv2.sol the main smart contract of the prototype which contains the implementation of all the functions described in **section 3.3.2 - Functions** and a few other functions

The final phase before deployment was testing. As the framework used was Hardhat, the tests were written in Javascript using Mocha [43] and Ethers [44]. A complete list of the tests can be found in **Appendix B** and the code can be found here: `https://github.com/demKyr/PicanhaStaker-MScIndividualProject/tree/main/SmartContracts/test`.

After thorough testing, the prototype was ready to be deployed on the Goerli testnet. The contract can be seen on Etherscan on: `https://goerli.etherscan.io/address/0x9437eff6e8713cf1619d9507695489a6639b758d`

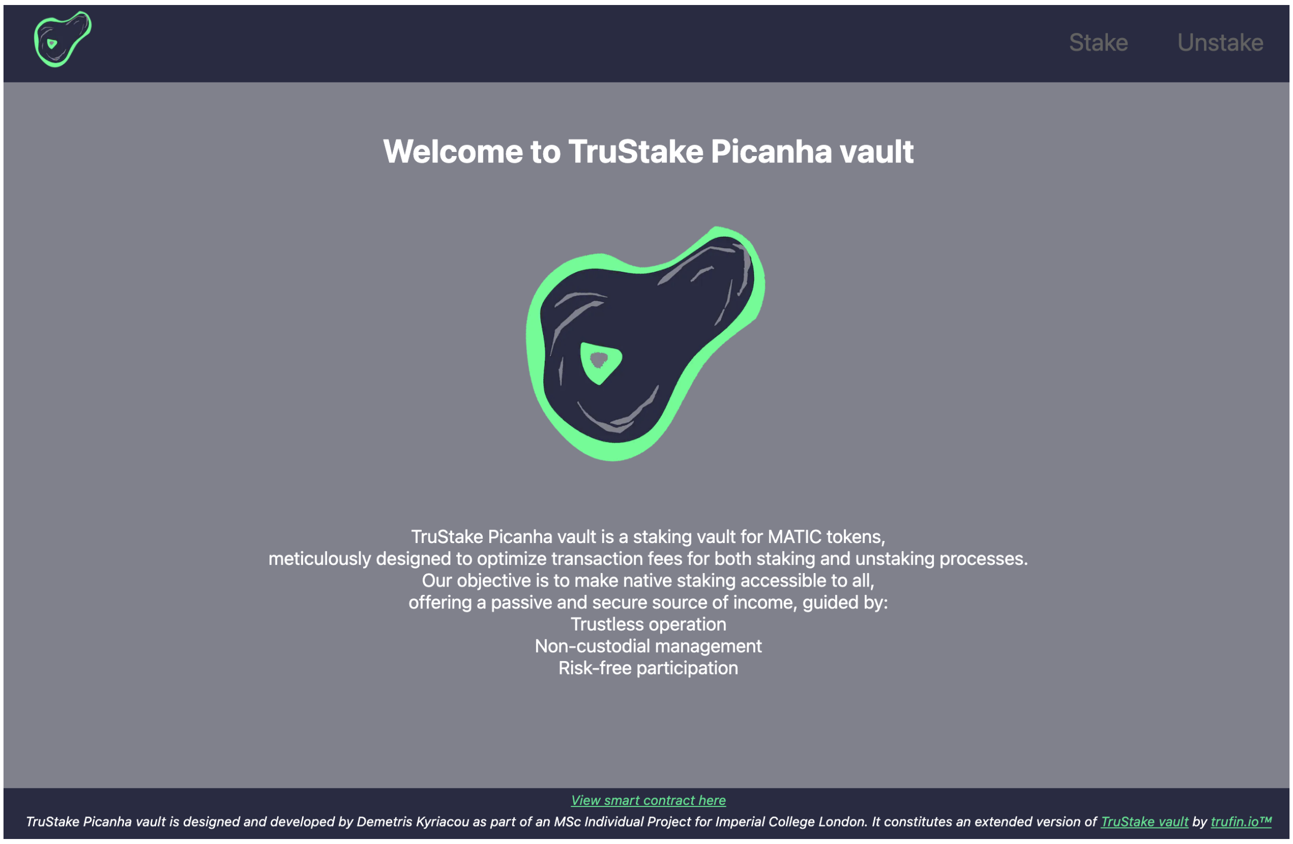
4.3 Frontend

Prototype's frontend was developed using Javascript and the NextJS framework [45] and it serves two purposes. Primarily, it makes the user's interaction with the vault easier and improves the whole user experience. The user simply connects their Metamask wallet [46] to the website and uses the UI. Besides that, it allows the owner of the protocol to monitor the smart contract and initiate transactions.

The frontend consists of the home page, the stake page, the unstake page and the admin page which (visible to the owner only). The final version of the frontend is connected to the deployed smart contract and deployed using Vercel [47] here: `https://picanha-staker.vercel.app/` and the code can be found here: `https://github.com/demKyr/PicanhaStaker-MScIndividualProject/tree/main/Frontend`

4.3.1 Home page

A screenshot of the Home Page is shown in **Figure 4.2**. This is the landing page of the frontend and it only contains some minimal information about the protocol.



Welcome to TruStake Picanha vault

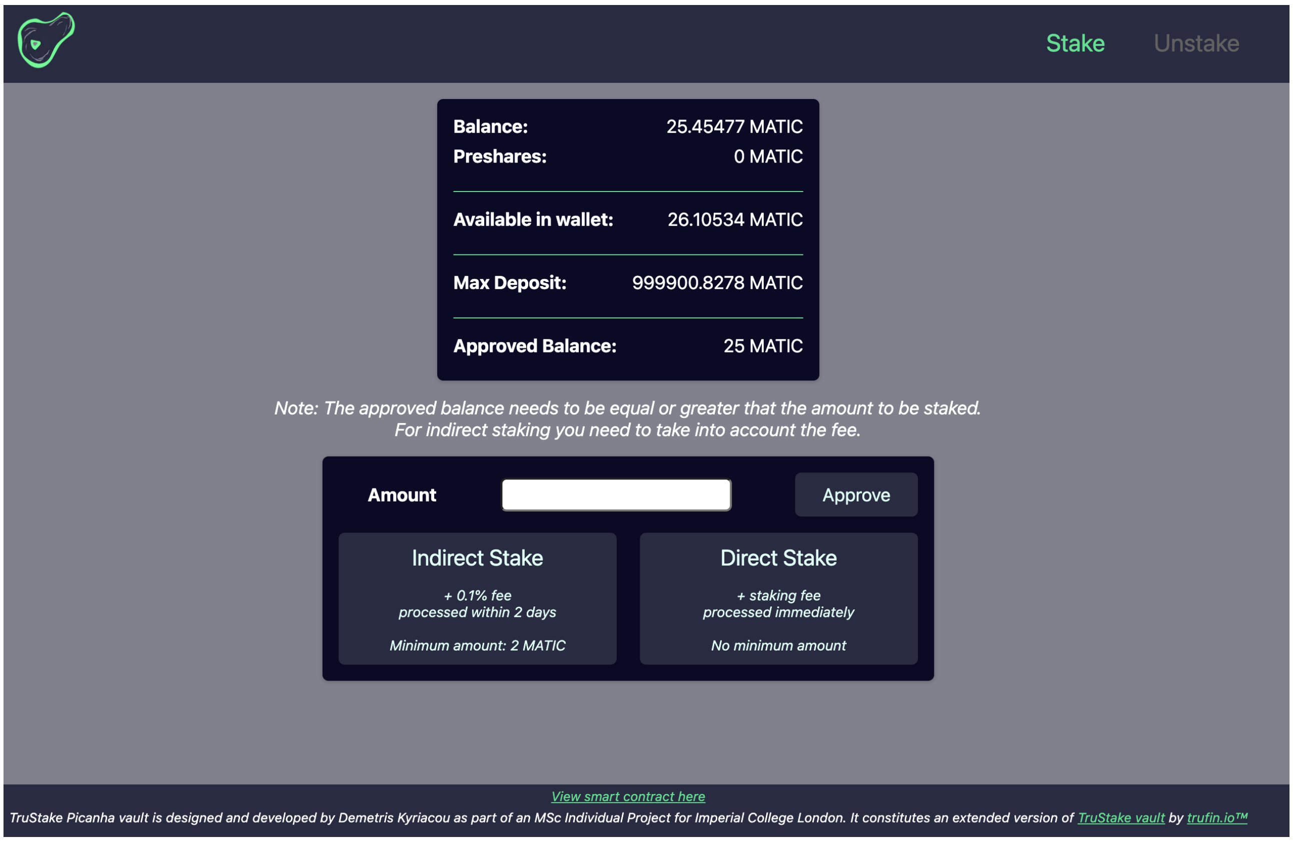
TruStake Picanha vault is a staking vault for MATIC tokens,
meticulously designed to optimize transaction fees for both staking and unstaking processes.
Our objective is to make native staking accessible to all,
offering a passive and secure source of income, guided by:
Trustless operation
Non-custodial management
Risk-free participation

View smart contract here
TruStake Picanha vault is designed and developed by Demetris Kyriacou as part of an MSc Individual Project for Imperial College London. It constitutes an extended version of TruStake vault by trufin.io™

Figure 4.2: Home page

4.3.2 Stake page

A screenshot of the Stake Page is shown in **Figure 4.3**. Through this page a user can deposit funds that will be staked on the validator. Initially, the user is required to approve the amount they wish to stake. This step is vital because, otherwise, the protocol is not authorised to access the user's MATIC. As long as the amount is approved, the user can choose whether they wish to stake indirectly or directly. Clicking the Indirect Stake button initiates a call of the `indirectDeposit()` function while the Direct Stake button initiates the `directDeposit()` function.



Balance:	25.45477 MATIC
Preshares:	0 MATIC
Available in wallet:	26.10534 MATIC
Max Deposit:	999900.8278 MATIC
Approved Balance:	25 MATIC

Note: The approved balance needs to be equal or greater that the amount to be staked.
For indirect staking you need to take into account the fee.

TruStake Picanha vault is designed and developed by Demetris Kyriacou as part of an MSc Individual Project for Imperial College London. It constitutes an extended version of TruStake vault by trufin.io™

Figure 4.3: Stake page

Additionally to staking, this page presents some important information to the user:

Balance: user's shares converted to MATIC using equation 3.4

Preshares: user's funds in pending deposit requests

Available in wallet: user's MATIC balance in their wallet

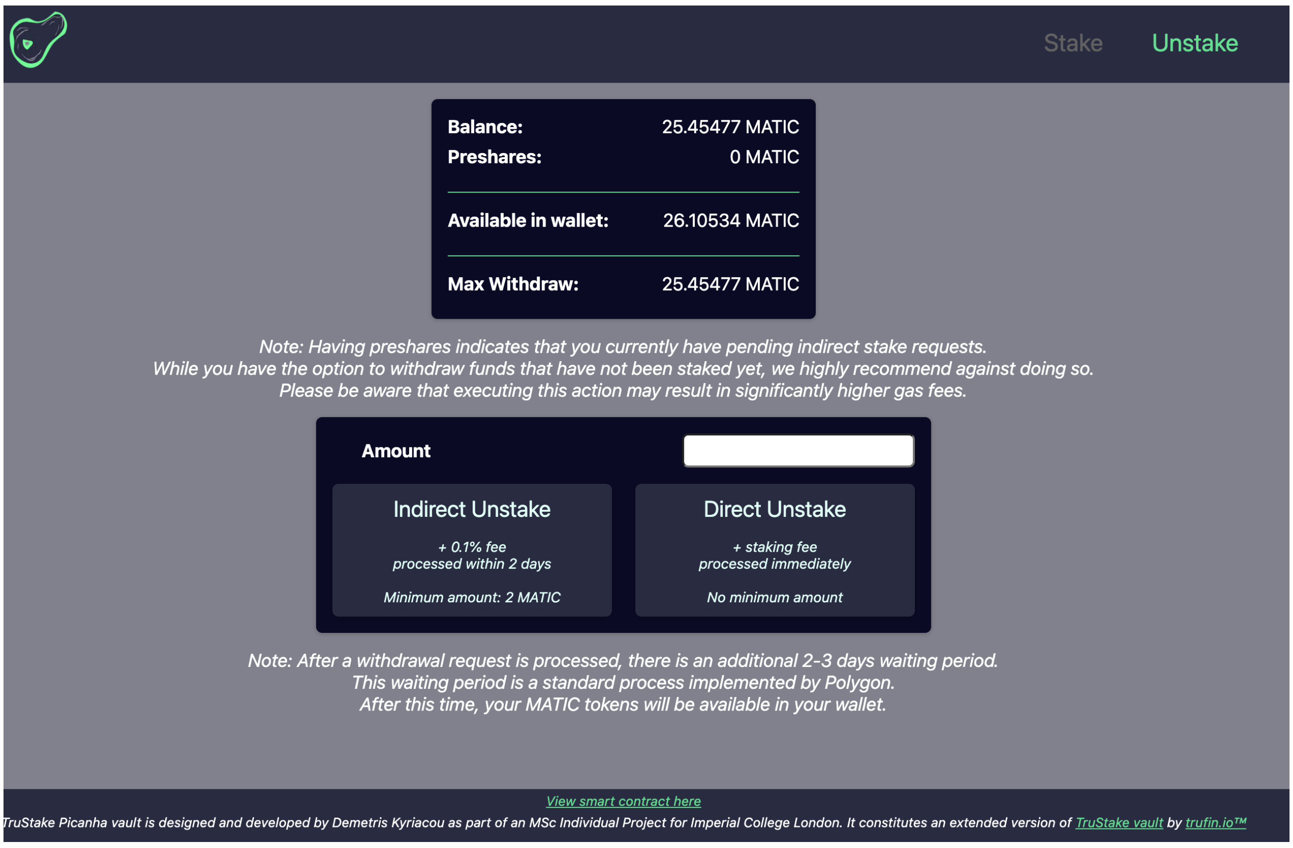
Max Deposit: vault's cap minus the total funds that are already deposited

Approved Balance: amount of MATIC in user's wallet that the user has approved to the protocol

4.3.3 Unstake page

A screenshot of the Unstake Page is shown in **Figure 4.4**. Through this page a user can request an amount to be unstaked and returned to their wallet. Similarly to the Stake Page, the user can select whether they want to indirectly or directly unstake. They can choose either option regardless of the staking method they have used before. Clicking the Indirect Unstake button initiates a call of the `indirectWithdraw()` function while the Direct Unstake button initiates the `directWithdraw()` function.

Following the same pattern as Stake Page, this page displays some information. The only new information is the `Max Withdraw` value that displays the sum of Balance and Preshares which is the maximum amount that the user can request to unstake. Finally, this page displays a warning about the waiting period and the additional costs of unstaking while having pending deposit requests.



Balance:	25.45477 MATIC
Preshares:	0 MATIC
Available in wallet:	26.10534 MATIC
Max Withdraw:	25.45477 MATIC

Note: Having preshares indicates that you currently have pending indirect stake requests.
While you have the option to withdraw funds that have not been staked yet, we highly recommend against doing so.
Please be aware that executing this action may result in significantly higher gas fees.

Indirect Unstake	Direct Unstake
+ 0.1% fee	*+ staking fee*
processed within 2 days	*processed immediately*
Minimum amount: 2 MATIC	*No minimum amount*

Note: After a withdrawal request is processed, there is an additional 2-3 days waiting period.
This waiting period is a standard process implemented by Polygon.
After this time, your MATIC tokens will be available in your wallet.

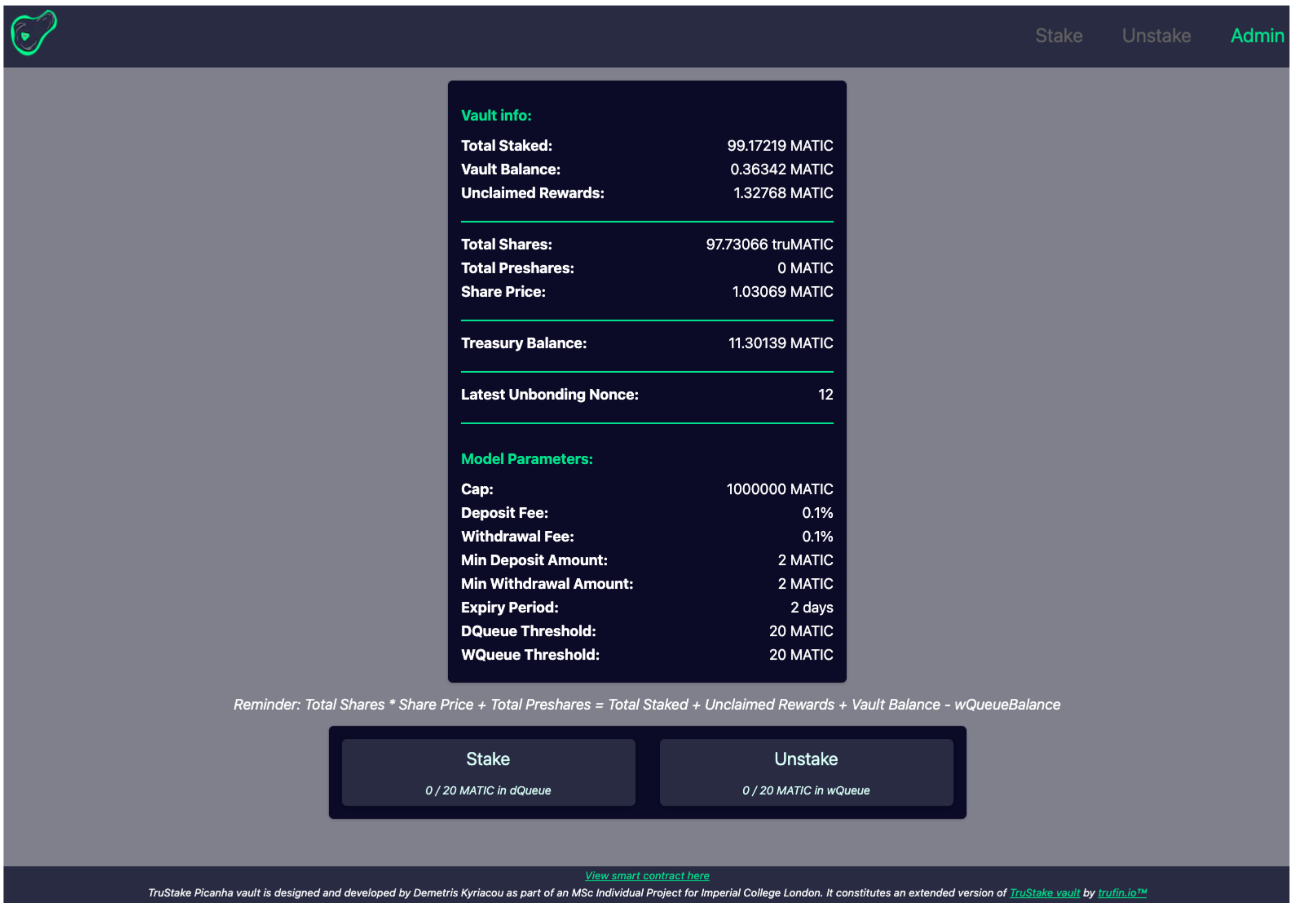
View smart contract here
TruStake Picanha vault is designed and developed by Demetris Kyriacou as part of an MSc Individual Project for Imperial College London. It constitutes an extended version of TruStake vault by trufin.io™

Figure 4.4: Unstake page

4.3.4 Admin page

A screenshot of the Admin Page is shown in **Figure 4.5**. This page is only visible to the owner of the protocol and it serves a monitoring purpose. Due to the nature of Blockchain, all these information can be viewed by everyone through a tool like Etherscanner. Nevertheless, they are presented here in a convenient and straightforward way.

The parameters monitored on this page were explained previously in table 3.6. The only new parameter is `Treasury Balance` which is the Balance of the Treasury's shares converted to MATIC. This page also allows user to manually call the `indirectStake` or the `indirectUnstake` functions.



Vault info:

Total Staked:	99.17219 MATIC
Vault Balance:	0.36342 MATIC
Unclaimed Rewards:	1.32768 MATIC
Total Shares:	97.73066 truMATIC
Total Preshares:	0 MATIC
Share Price:	1.03069 MATIC
Treasury Balance:	11.30139 MATIC
Latest Unbonding Nonce:	12

Model Parameters:

Cap:	1000000 MATIC
Deposit Fee:	0.1%
Withdrawal Fee:	0.1%
Min Deposit Amount:	2 MATIC
Min Withdrawal Amount:	2 MATIC
Expiry Period:	2 days
DQueue Threshold:	20 MATIC
WQueue Threshold:	20 MATIC

*Reminder: Total Shares * Share Price + Total Preshares = Total Staked + Unclaimed Rewards + Vault Balance - wQueueBalance*

Stake
0 / 20 MATIC in dQueue

Unstake
0 / 20 MATIC in wQueue

View smart contract here
TruStake Picanha vault is designed and developed by Demetris Kyriacou as part of an MSc Individual Project for Imperial College London. It constitutes an extended version of TruStake vault by trufin.io™

Figure 4.5: Admin page

4.4 Autotasks

Autotasks are short Javascript code scripts that automate the monitoring of the the protocol and enable it to operate autonomously, without the need of external intervention. The tool used for this is Open Zeppelin defender [48].

4.4.1 Relay

A relay is an Ethereum account with a designated purpose. It has a balance and it is used to sign transactions that are sent using an autotask [49]. The relay created for this prototype can be viewed on Etherscan in this URL: `https://goerli.etherscan.io/address/0x814fF0BEAcF489FD963D51b4B58aC86eedF6cf81`

4.4.2 Sentinels

Sentinels are used to monitor transactions of a contract [49]. For this prototype they act as event listeners. Apart from triggering the autotask, they can also be programmed to send a notification to the owner by email. The sentinels programmed for the prototype are:

StakeRequired: This sentinel is triggered when the event `StakeRequired()` is emitted

UnstakeRequired: This sentinel is triggered when the event `UnstakeRequired()` is emitted

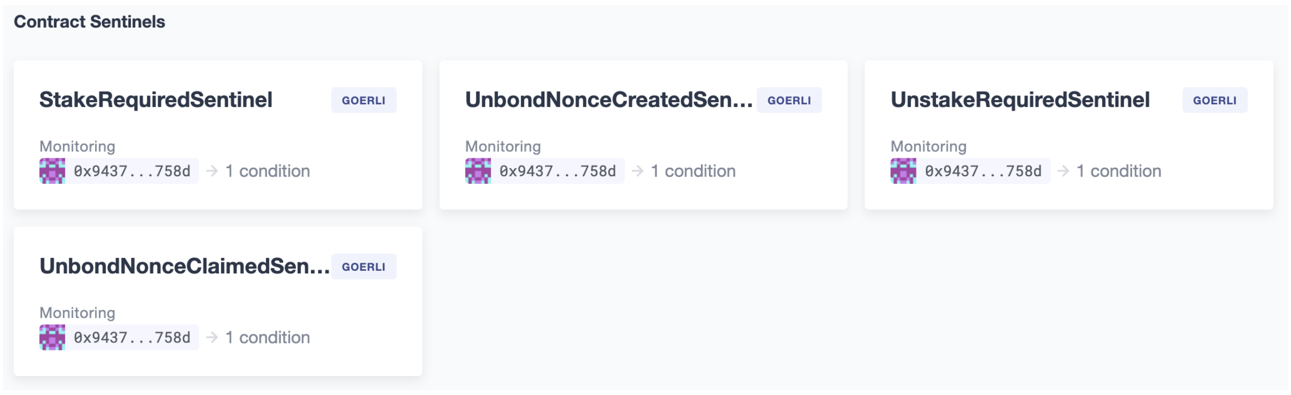
UnbondNonceCreated: This sentinel is triggered when the event `UnbondNonceCreated(unbondNonce, currentEpoch)` is emitted

UnbondNonceClaimed: This sentinel is triggered when the event `UnbondNonceClaimed(unbondNonce)` is emitted

4.4.3 Autotasks

Autotasks are code snippets that run on regular basis or when invoked by a sentinel. They can also integrate a relay to sign transactions [49]. This prototype is monitored using the following autotasks:

Stake: This autotask is invoked from the StakeRequired sentinel and it calls the `indirectStake()` function.


Contract Sentinels

StakeRequiredSentinel GOERLI
Monitoring
0x9437...758d → 1 condition

UnbondNonceCreatedSen... GOERLI
Monitoring
0x9437...758d → 1 condition

UnstakeRequiredSentinel GOERLI
Monitoring
0x9437...758d → 1 condition

UnbondNonceClaimedSen... GOERLI
Monitoring
0x9437...758d → 1 condition

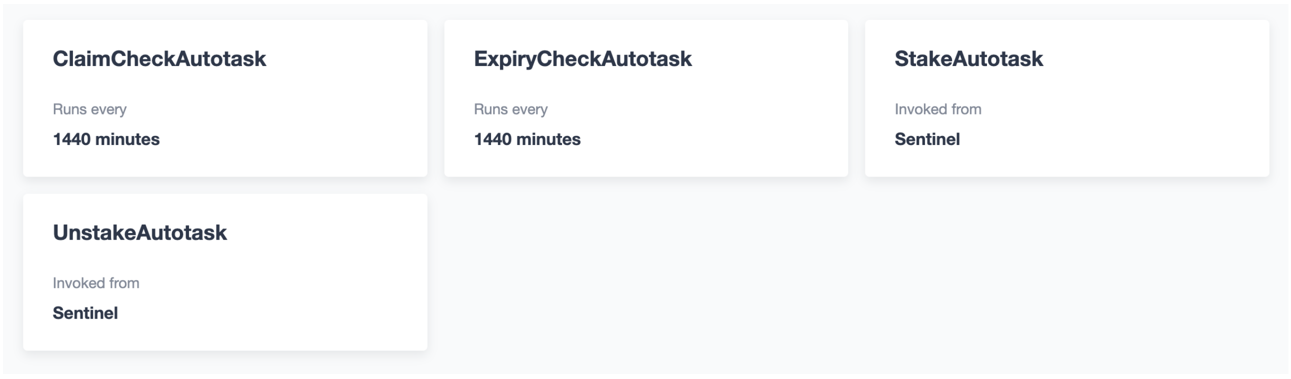
Figure 4.6: Sentinels

Unstake: This autotask is invoked from the UnstakeRequired sentinel and it calls the `indirectUnstake()` function.

ExpiryCheck: This autotask is invoked every 24 hours and it calls the `expiryCheck()` function that checks whether the oldest requests in any queue expired.

ClaimCheck: This autotask is invoked every 24 hours and it checks whether there are any unbonding nonces to be claimed. If so, it calls the `unstakeClaim()` function for all of them.

The code for these autotasks can be found here: `https://github.com/demKyr/PicanhaStaker-MScIndividualProject/tree/main/autotasks`


ClaimCheckAutotask
Runs every
1440 minutes

ExpiryCheckAutotask
Runs every
1440 minutes

StakeAutotask
Invoked from
Sentinel

UnstakeAutotask
Invoked from
Sentinel

Figure 4.7: Autotasks

Chapter 5

Evaluation

In this chapter we will evaluate the performance of the TruStake Picanha vault prototype. We will make an analysis on the gas fees of the Picanha prototype and a comparison with the gas fees of the other stakers seen in section 2.2 - Existing Stakers.

The developed prototype differs a lot from the existing stakers. A main difference is that the functions for the stake/unstake operations are not static but dynamic and dependent on various factors. For the existing stakers, the functions behind these operations are rather static and executed in similar ways every time with similar gas fees costs.

For this prototype, there are not only two different types of functions for each operation (direct and indirect), but the execution of the same function can also vary significantly depending on the existence of other pending requests. Therefore, before calculating the average gas fees for transactions in the prototype, we had to divide them into types.

5.1 Types of functions executions

5.1.1 User functions

Diagrams 5.1 - 5.4 show the different types of execution for each of the function that a user can call to stake or unstake. The following list analyses the factors that affect the gas fees for each type:

no pair (indirect functions): This type does not depend on any factors. The request is simply added to the end of the corresponding queue and gas fees for this type are relatively stable.

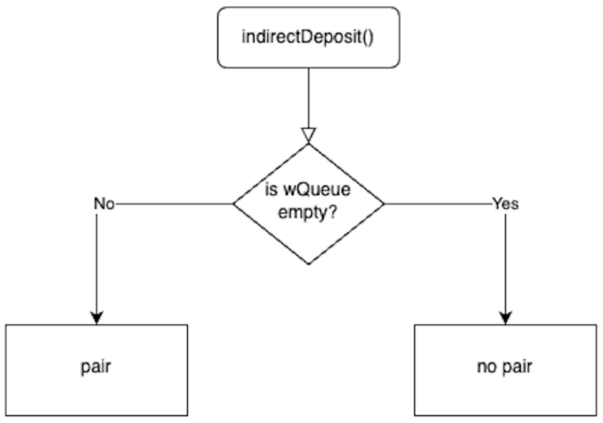


Figure 5.1: Types of executions for `indirectDeposit()` function

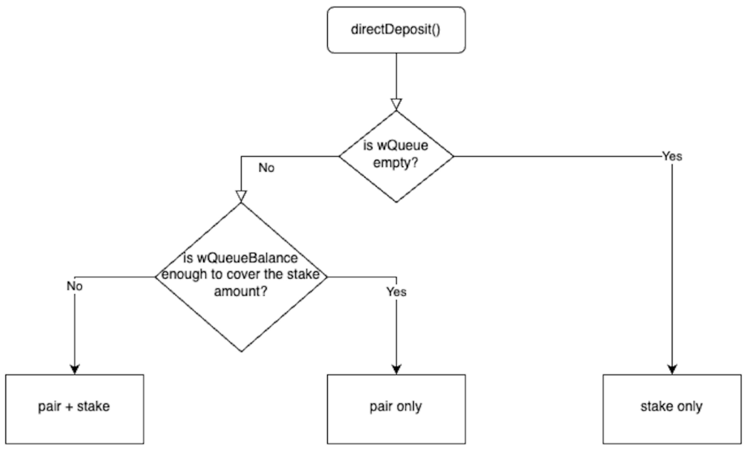


Figure 5.2: Types of executions for `directDeposit()` function

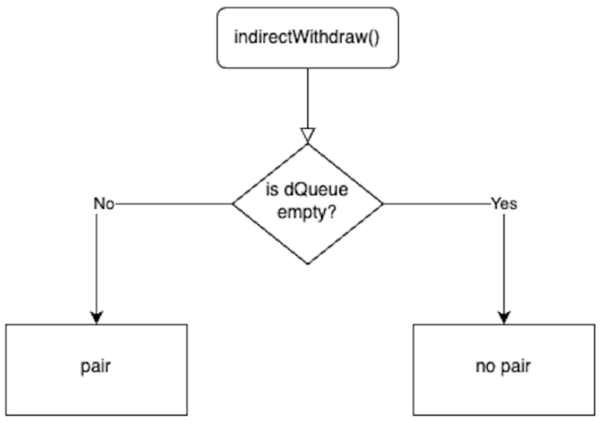


Figure 5.3: Types of executions for `indirectWithdrawal()` function

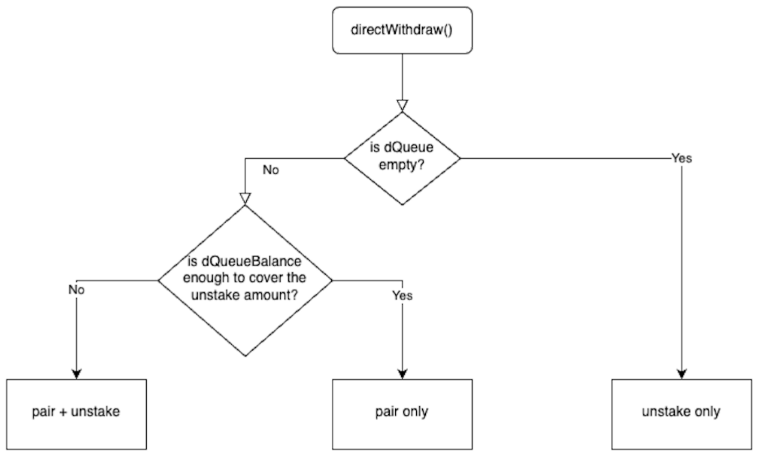


Figure 5.4: Types of executions for `directWithdrawal()` function

pair (indirect functions): This type depends on the number of requests of the opposite type that this request is paired with. Under normal circumstances, this number is small and in many cases its just equal to one. Nevertheless, the more requests that need to be paired, the more expensive the function. Also, the cost is independent of whether the request is fully netted on not.

stake only (direct deposit function): This type depends on the number of pending deposit requests. The more requests in the dQueue, the more expensive the gas fees. Under the worst circumstances, this number is equal to the number of loops calculated using equation 3.9. However, this is highly unlikely as it will only happen if all the requests have the minimum amount and the queue is almost full.

unstake only (direct withdraw function): This type does not depend on any factors. There is no loop through the pending withdrawal requests, just a call to the `unstake()` function.

pair only (direct functions): This type is identical to the pair type of the indirect functions and depends on the number of requests of the opposite type that this request is paired with.

pair + stake/unstake (direct functions): This type depends on the number of requests of the opposite type that this request is paired with. Nonetheless, it is more expensive than the pair only because `stake()`/`unstake()` function is also called.

It must be noted that for the unstake operation, the gas fees can increase significantly if the user has preshares (i.e. pending deposit requests). Even so, this is omitted from this evaluation since the users are strongly advised against it. Users that choose to perform this operation are warned about the high fees and should only perform this if they urgently wish to withdraw their funds.

5.1.2 Internal functions

For the evaluation of the prototype, we also need to take into consideration the gas fees for the functions that are called internally by the protocol and paid by the owner. These functions are always executed in the same way but the gas fees depend on other factors. More information can be found in the following list:

`indirectStake()`: The gas fees of this function depend on the number of requests in the dQueue. Under the worst circumstances, this number is equal to the number of loops calculated using equation 3.9

`indirectUnstake()`: This function does not depend on any factors.

`unstakeClaim()`: The gas fees of this function depend on the number of requests in the wQueue. Under the worst circumstances, this number is equal to the number of loops calculated using equation 3.13.

5.2 Experiments

For the evaluation of the prototype, we performed multiple transactions (txs) in the Goerli testnet using the frontend of the prototype. Fieldlabs very generously offered an abundance of GoerliETH and Test MATIC for this purpose.

For evaluation purposes, six (6) different accounts were used. The first account was the account that deployed the contract and it was used as the owner and the treasury of the smart contract. Another account was used as a whale user and the remaining four were used as retail users.

The first step was to change the constant parameter values by lowering the queue thresholds, the minimum amounts for deposit and withdrawal, and shorten the expiry period enabled more. This enabled more extensive and quick testing.

Afterwards, we conducted the simulations. Each simulation consisted of a series of transactions, thoughtfully selected to replicate every type of function execution outlined in the preceding section. The aggregate count of transactions surpassed 250, all of which were manually executed through the frontend. This process also served as a secondary testing procedure for the prototype. The full list of transactions can be found on Etherscan in this URL: `https://goerli.etherscan.io/txs?a=0x9437eff6e8713cf1619d9507695489a6639b758d&ps=100&p=1`

The average gas fees values in gas units for every type of function execution can be found in **Table 5.1**. For some functions the gas fees depend on the number of transactions (requests) they process. For those functions the average gas fees are in the form: $a \times txs + c$. The methodology used to calculate the parameters a and c is the following:

1. We calculate samples of average gas fees Y for different numbers of txs X in the form: $X_{txs} \rightarrow Y_{gas\,units}$ (like the ones seen in **Table 5.1**)

2. For every pair of samples $X_i \rightarrow Y_i$ and $X_j \rightarrow Y_j$, let $X_j > X_i$ we calculate different values for the parameter a as:

$$a_{ij} = (Y_j - Y_i)/(X_j - X_i)$$

3. We calculate a by averaging all the a_{ij} parameters

4. For every sample we calculate a different value for the parameter c as:

$$c_i = Y_i - X_i \times a$$

5. We calculate c by averaging all the c_i parameters

5.3 Observations and outcomes

5.3.1 Observations

- The prototype achieves its primary goal of reducing the gas fees but they still remain significant.

- The initial accepted upper bound of 150 loops was an overestimation. A for loop with 150 iterations can be exceedingly expensive.

- As a result of the previous point, the initial value for the deposit/withdrawal fee was an underestimation.

- The gas fees of the indirect functions were underestimated. The users still need to pay noticeable gas fees even without interacting with the validation node.

- The dQueue threshold was underestimated. A successful staking vault is very likely to have more than 15k of MATIC deposited/withdrawn weekly. Meanwhile the USD price of MATIC dropped significantly from around \$0.85 to around \$0.60 in the last three months when the first draft of this report was first released.

function	Avg gas fees (in gas units)
`indirectDeposit()`	
no pair	**281,518**
pair (1-3 txs)	**372,717**
`directDeposit()`	
stake only	
1 tx	405,287
5 txs	432,715
7 txs	506,881
10 txs	590,878
25 txs	846,712
on average	**21,615 $\times$ txs + 348,988**
pair only (1-3 txs)	**322,729**
pair + stake (1 tx)	**452,455**
`indirectWithdrawal()`	
no pair	**268,481**
pair (1-3 txs)	**369,196**
`directWithdrawal()`	
unstake only	**559,261**
pair only (1-3 txs)	**335,122**
pair + unstake (1 tx)	**601,128**
`indirectStake()`	
1 tx	306,806
4 txs	371,043
5 txs	389,921
7 txs	377,590
11 txs	511,035
25 txs	759,663
on average	**17,180 $\times$ txs + 300,920**
`indirectUnstake()`	**382,002**
`unstakeClaim()`	
1 txs	206,213
2 txs	230,749
5 txs	286,047
10 txs	384,841
on average	**20,299 $\times$ txs + 185,616**

Table 5.1: Average gas fees per type of function execution (gas units)

- The individual costs of indirect and direct deposit excluding the $stakingFee$ are not equal as predicted in paragraph 3.3.5 - Constants' value analysis. In practice, the individual cost of indirect functions is larger because it includes an additional minting of shares for the treasury from the deposit/withdrawal fee. This could have been avoided but it would require significant changes in the smart contract and redeployment. Therefore the cost of stake only `directDeposit()` minus the no pair `indirectDeposit()` is not equal to the cost of `indirectStake()` for the same number of transactions. Consequently, **Figure 3.6** is not accurate in practice and `directDeposit()` can be cheaper than `indirectDeposit()` even for amounts smaller than `dQueueThreshold`.

- The individual costs of indirect and direct withdrawal excluding $unstakingFee$ are similar as predicted in paragraph 3.3.5 - Constants' value analysis. For this reason the cost of unstake only `directWithdrawal()` minus the no pair `indirectWithdrawal()` is similar to the cost of `indirectUnstake()`
- The gas fees of `indirectstake()` and stake only `directDeposit()` depend on the number of deposit requests. This happens because for each request there is a minting of shares and burning of preshares. This cost should be covered by each user through deposit fee. The analysis in paragraph 3.3.5 uses a constant $stakingFee$. This constant is calculated by using maximum number of Loops from equation 3.9.
- The gas fees of unstakeClaim() depend on the number of withdrawal requests. This happens because for each request there is a transfer of MATIC to the user. This cost should be covered by each user through withdrawal fee.
- $stakingFee$ is equal to the gas fees of `indirectStake()` for maximum number of loops
- $unstakingFee$ is equal to the gas fees of `indirectUnstake()` plus the gas fees of `unstakeClaim()` for maximum number of loops
- when converted to MATIC using $\mathbf{gas\bar{}price = 30.45\ Gwei}$ and $\mathbf{ETH\bar{}value = 1896}$ $\$$ from paragraph 2.2.4 and the average USD value in MATIC for the same period (July 2023) $\mathbf{USD\ value = 1.3751\ MATIC}$, the $staking\bar{}Fee$ and the $unstaking\bar{}Fee$ in MATIC are:

$$staking\bar{}Fee = (17,180 \times txs + 300,920)\, gas\, units = (1.36 \times txs + 23.89)\, MATIC \tag{5.1}$$

$$unstaking\bar{}Fee = (20,299 \times txs + 567,618) gasunits = (1.61 \times txs + 45.06) MATIC \tag{5.2}$$

This means that they were initially underestimated.

5.3.2 Outcomes

- The deposit/withdrawal fee should be increased.

- The upper bound of loops should be decreased.
- The expiry period can be shortened and/or dQueue/wQueue threshold increased. However since the latter would also increase the upper bound of loops, it was decided to do the former.
- The minimum deposit/withdrawal amount could be increased slightly to lower the number of loops.

5.4 Adjustment of constant parameters

Following the observations and outcomes from paragraph 5.3, it was decided that the following parameters should maintain their values:

- `dQueueThreshold` = **15,000 MATIC**
- `wQueueThreshold` = **15,000 MATIC**

Meanwhile, the following parameters need to be adjusted:

- `expiryPeriod`
- `depositFee`
- `withdrawalFee`
- `minDepositAmount`
- `minWithdrawalAmount`

For the adjustment of these parameters (except expiry period) we need to reuse the following equations:

$$dQueueThreshold \times depositFee(\%) \geq stak\bar{i}ngFee \tag{3.6}$$

$$\lceil NumberOfLoops_{dQueue} \rceil = dQueueThreshold/minDepositAmount \tag{3.9}$$

$$wQueueThreshold \times withdrawalFee(\%) \geq unstak\bar{i}ngFee \tag{3.10}$$

$$\lceil NumberOfLoops_{wQueue} \rceil = wQueueThreshold/minWithdrawalAmount \tag{3.13}$$

$$\bar{stakingFee} = (17,180 \times txs + 300,920)\, gas units = (1.36 \times txs + 23.89)\, MATIC \quad (5.1)$$

$$\bar{unstakingFee} = (20,299 \times txs + 567,618)\ gas\ units = (1.61 \times txs + 45.06)\ MATIC \quad (5.2)$$

Since the $numberOfLoops$ is equal to the number of txs, we only need to choose a value for that parameter for determining the value of the rest of the parameters. For $numberOfLoops = 100$:

- `expiryPeriod` = **3 days**
- `depositFee` = **1.1%**
- `withdrawalFee` = **1.4%**
- `minDepositAmount` = **150 MATIC**
- `minWithdrawalAmount` = **150 MATIC**

5.5 Gas fees comparison

	Indirect staking		**Direct staking**			
TruStake	-		364,069			
Lido	-		642,379			
Stader	-		568,917			
	no pair	pair	stake only (1 tx)	stake only (10 txs)	pair only (1-3 txs)	pair + stake (1 tx)
Picanha	281,518	372,717	405,287	590,878	322,729	452,455

Table 5.2: Staking gas fees comparison (in gas units)

	Indirect unstaking		**Direct unstaking**		
TruStake	-		609,181		
Lido	-		1,401,186		
Stader	-		947,171		
	no pair	pair	unstake only	pair only (1-3 txs)	pair + unstake (1 tx)
Picanha	268,481	369,196	559,261	335,122	601,128

Table 5.3: Unstaking gas fees comparison (in gas units)

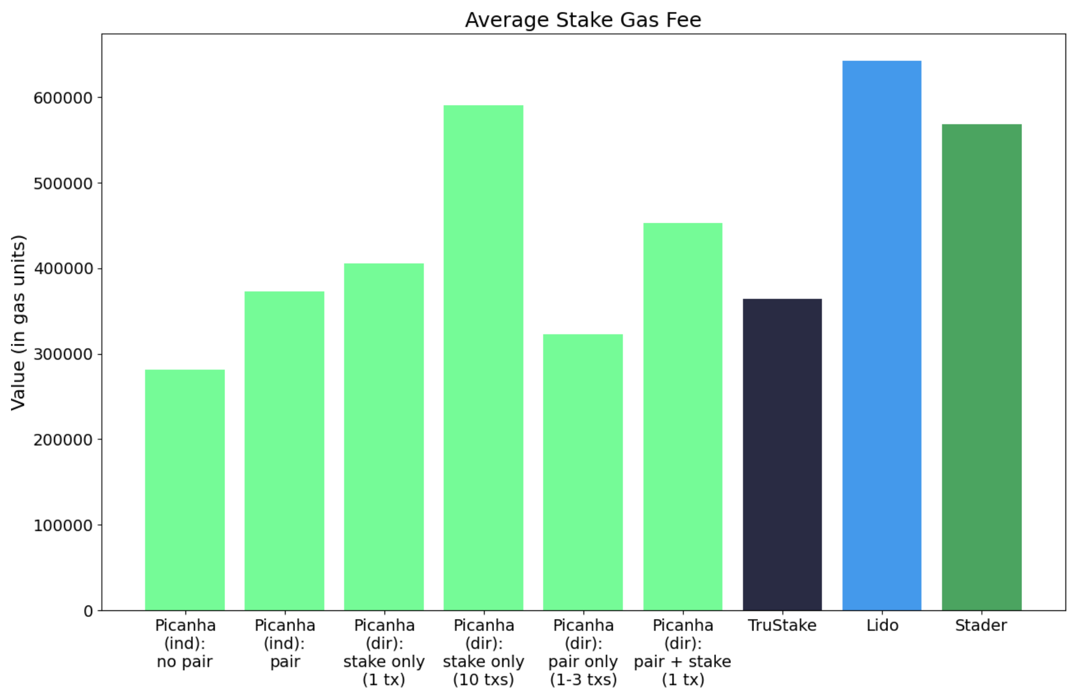


Figure 5.5: Staking gas fees comparison

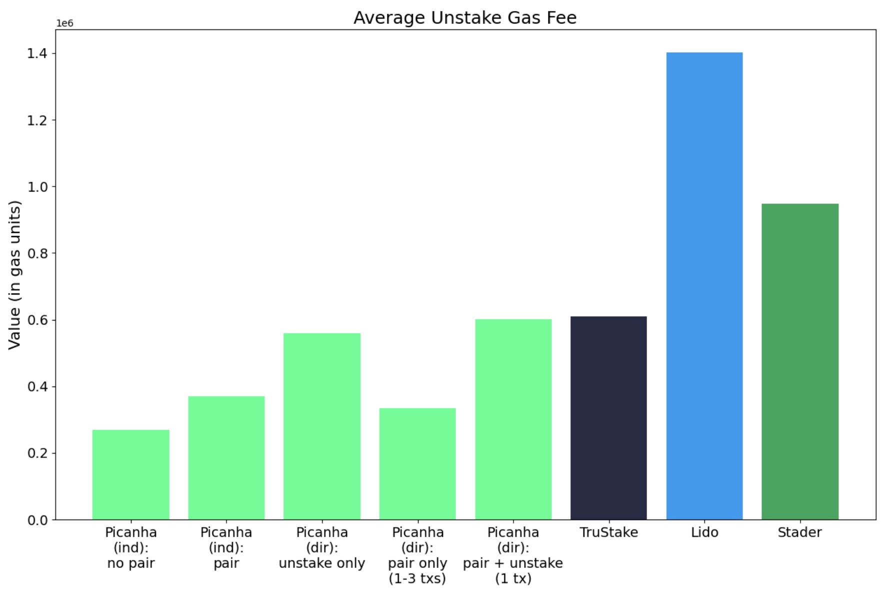


Figure 5.6: Unstaking gas fees comparison

5.6 Comparison findings

It appears that the prototype achieves its goal very successfully for unstaking and partially achieves its goal for staking. For unstaking, Picanha is always cheaper than any other staker. For staking, Picanha is generally, but not always, cheaper than the other stakers.

In order to improve the protocol's performance for the staking operation is important to split the staking and the share minting procedures in different functions. This will make the `directDeposit()` function invariant to the number of requests in the dQueue for stake only and have relatively stable gas fees like `directWithdraw()` - unstake only. A way to do this is by creating a new function that loops through the requests in dQueue and mints shares after the `stake()` function is called. Nevertheless, there is a better way that takes advantage of the fact that the expiry period is reduced.

For the second version of the Picanha, we suggest that the concept of preshares is removed completely and every user immediately receives freshly minted shares for every direct or indirect deposit. The drawbacks of this decision is that users will receive rewards for funds sitting in the vault and lower the APY for everyone. In the worst case, these funds are equal to the `dQueueThreshold` and stay unstaked for the whole `expiryPeriod` every time. However, if the `totalStaked` amount is a few millions, that amount is insignificant and the effect on the APY is infinitesimal. The benefits of removing the preshares are:

- the staking gas fees will significantly be reduced
- the gas fees of `directDeposit()` will be invariant to the number of requests in the dQueue
- the equations of paragraph 3.3.4 - Variables' value analysis will be simplified and therefore the complexity of the code of the vault will be reduced
- unstaking while having pending deposit requests will no longer be a problem

It must be noted that this is a comparison in gas units. As a result it does not include the fees for indirect staking/unstaking and it is invariant to the staking/unstaking amount.

Chapter 6

Conclusion

6.1 Summary of Achievements

Through this project we initially identified and formalised the need for a retail-friendly native staking protocol. To solve this problem we suggested some core ideas for reducing the gas fees associated with native staking. By utilising these ideas we designed four (4) different approaches for gas efficient staking vaults. Furthermore, we expanded one of the approaches and designed a complete protocol with a detailed description of how it should be build and adjusted to create a staking vault. Lastly, we developed a prototype of a fully functioning staking vault based on the designed protocol to demonstrate that the ideas suggested in this project can be implemented successfully in the real world.

We hope and believe that we made a meaningful contribution that promoted the benefits of native staking and combined the potential of whale and retail users in a way that is advantageous for everyone.

To elaborate further, this protocol **benefits the retail users** as it:

- reduces the gas fees associated with staking and unstaking
- provides a user friendly interface that makes staking simple and quick
- does not distinguish between whale and retail users and everyone has the same rights and advantages

At the same time, it has notable **benefits for the whale users** as it:

- is the only protocol that provides instant withdrawals if the request can be paired

- can potentially offer very low fees if the request can be paired
- offers complimentary fund claiming

Meanwhile, it can be a fully functioning and successful product that can be more profitable for the owner that the existing stakers due to the additionaal earnings from deposit and withdrawal fees.

6.2 Future work

The suggestions for future work can be split into two categories, changes for the prototype that can potentially turn it into a complete product and methods that the work of this project can be exploited for creating other projects or relevant applications.

Picanha improvements:

- Observation of the gas price of the network to perform the daily autotasks (ClaimCheckAutotask and ExpiryCheckAutotask) when the gas price value. This a very simple way to significantly reduce the gas fees.
- Split of the share minting and sharing procedures by removing the concept of preshares (as explained in the paragraph 5.6 - Comparison findings).
- Not mint shares for treasury immediately for every indirect deposit/withdrawal to avoid the extra transaction. Instead mint shares with every staking/unstaking when new shares are already minted from the rewards.
- Pairing a request with many requests of the opposite type can be expensive. Therefore, the number of pairings should be limited.
- Indirect requests that are fully paired should not pay deposit/withdrawal fees.
- Reevaluation of the parameters' values frequently (e.g. every month).
- Rewriting some of the code in a more gas efficient manner.

Ideas for future projects / applications:

- Use the final design or any of the other approach for staking on blockchains other than Polygon (like Ethereum or Solana).
- Make a hybrid protocol that implements more than one of the suggested approaches and adjusts to the environment (frequency and size of the requests).
- Integrate some of the core ideas suggested in this project in already existing stakers to reduce the gas fees.

6.3 Epilogue

Reaching the end of this project, I feel like a journey that started last January is coming to an end. Even though it was not always easy and smooth, I enjoyed it and will look back to these months as a happy memory.

Through this project I got a better understanding of the world of Web3 and blockchains. I fathomed how gas fees work, how dApps are built, and familiarised with the financial and mathematical aspects behind designing a DeFi protocol. It was a well-rounded experience that involved research and fullstack development and involved Computer Science, Mathematics and Finance.

Additionally, I had the chance to work in a corporate environment and be a part of a team. I met great people and gained valuable experience.

I hope this project contains ideas that will inspire people and help build great things in the future.

Bibliography

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. URL www.bitcoin.org. pages 1, 5

[2] Ethereum. Proof-of-stake (pos), . URL https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/. [Accessed:18/05/2023]. pages 1, 6, 8

[3] Lars Brünjes, Aggelos Kiayias, Elias Koutsoupias, and Aikaterini-Panagiota Stouka. Reward sharing schemes for stake pools. *CoRR*, abs/1807.11218, 2018. URL http://arxiv.org/abs/1807.11218. pages 1, 12

[4] Hans Gersbach, Akaki Mamageishvili, and Manvir Schneider. Staking pools on blockchains. 3 2022. URL http://arxiv.org/abs/2203.05838. pages 1, 11

[5] Ethereum. Consensus mechanisms, . URL https://ethereum.org/en/developers/docs/consensus-mechanisms/. [Accessed:18/05/2023]. pages 5

[6] Fahad Saleh. Blockchain without waste: Proof-of-stake. *The Review of Financial Studies*, 34:1156–1190, 3 2021. ISSN 0893-9454. doi: 10.1093/rfs/hhaa075. URL https://doi.org/10.1093/rfs/hhaa075. pages 5

[7] Eric Budish. The economic limits of bitcoin and the blockchain, 2018. URL https://EconPapers.repec.org/RePEc:nbr:nberwo:24717. pages 5

[8] Fahad Saleh. Volatility and welfare in a crypto economy. *ERN: Other Monetary Economics: Financial System Institutions (Topic)*, 2019. URL https://api.semanticscholar.org/CorpusID:158099391. pages 5

[9] Bruno Biais, Christophe Bisiere, Matthieu Bouvard, and Catherine Casamatta. The blockchain folk theorem. *The Review of Financial Studies*, 32:1662–1715, 2019. ISSN 0893-9454. pages 5

[10] Ethereum. Proof-of-work (pow), . URL https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/. [Accessed:20/05/2023]. pages 6

[11] Jing Chen and Silvio Micali. Algorand. 7 2016. URL http://arxiv.org/abs/1607.01341. pages 6, 7

[12] Vitalik Buterin. Why proof of stake, . URL https://vitalik.ca/general/2020/11/06/pos2020.html. [Accessed:23/05/2023]. pages 6

[13] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. 2012. URL https://api.semanticscholar.org/CorpusID:42319203. pages 6

[14] Pavel Vasin and Blackcoin Co. Blackcoin's proof-of-stake protocol v2, 2014. URL www.blackcoin.co. pages 6

[15] Nxt community. Nxt whitepaper, 2014. URL https://bitbucket.org/JeanLucPicard/nxt/src. pages 6

[16] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. pages 357–388. Springer, 2017. pages 6, 7

[17] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. 2014. URL https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf. pages 7, 9

[18] Ethereum. The merge, . URL https://ethereum.org/en/roadmap/merge/. [Accessed:22/05/2023]. pages 7

[19] Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X Zhang. Combining ghost and casper. 3 2020. URL http://arxiv.org/abs/2003.03052. pages 7, 8

[20] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. 10 2019. URL http://arxiv.org/abs/1710.09437. pages 7

[21] Ethereum. Gasper, . URL https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/gasper/. [Accessed:23/05/2023]. pages 8

[22] Vitalik Buterin. Proof of stake faq, . URL https://vitalik.ca/general/2017/12/31/pos_faq.html. [Accessed:23/05/2023]. pages 8

[23] Ethereum. Gas and fees, . URL https://ethereum.org/en/developers/docs/gas/. [Accessed:11/08/2023]. pages 8

[24] Jaynti Kanani, Sandeep Nailwal, and Anurag Arjun. Matic whitepaper. 2019. URL https://github.com/maticnetwork/whitepaper. pages 9

[25] Polygon. Polygon lightpaper, 2021. URL https://whitepaper.io/document/646/polygon-whitepaper. pages 9, 16

[26] Polygon. Introduction to polygon pos — polygon wiki, . URL https://wiki.polygon.technology/docs/pos/polygon-architecture. [Accessed:27/05/2023]. pages 10, 16

[27] Krisztian Sandor. Crypto staking 101: What is staking? URL `https://www.coindesk.com/learn/crypto-staking-101-what-is-staking/`. [Accessed:28/05/2023]. pages 10

[28] David Rodeck. What is staking in crypto? - forbes advisor. URL `https://www.forbes.com/advisor/in/investing/cryptocurrency/what-is-staking-in-crypto/`. [Accessed:28/05/2023]. pages 10

[29] Ethereum. Intro to ether, . URL `https://ethereum.org/en/developers/docs/intro-to-ether/`. [Accessed:01/08/2023]. pages 10

[30] Ethereum. Attestations, . URL `https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/attestations/`. [Accessed:01/08/2023]. pages 10

[31] Polygon. Matic staking — stake matic tokens for network rewards, . URL `https://polygon.technology/staking`. [Accessed:01/08/2023]. pages 10

[32] Binance Academy. What is crypto staking and how does it work? — binance academy. URL `https://academy.binance.com/en/articles/what-is-staking`. [Accessed:28/05/2023]. pages 10, 11

[33] Ethereum. Ethereum staking, . URL `https://ethereum.org/en/staking/`. [Accessed:17/05/2023]. pages 11

[34] TruFin. Trufin, . URL `https://www.trufin.io/`. [Accessed:29/05/2023]. pages 12, 38

[35] Twinstake. Twinstake – staking services built with institutions, for institutions. URL `https://twinstake.io/`. [Accessed:29/05/2023]. pages 12

[36] TruFin. Trustake (matic) staker - trufin documentation, . URL `https://trufin.gitbook.io/docs/trustake-vaults/trustake-matic-staker`. [Accessed:29/05/2023]. pages 12

[37] Lido. Lido - liquid staking for digital tokens, . URL `https://lido.fi/`. [Accessed:30/05/2023]. pages 13

[38] Lido. Overview — lido on polygon docs, . URL `https://docs.polygon.lido.fi/`. [Accessed:30/05/2023]. pages 13

[39] Stader Labs. Liquid staking - stader labs, . URL `https://www.staderlabs.com/`. [Accessed:30/05/2023]. pages 13

[40] Stader Labs. Your guide to staking matic with stader, . URL `https://blog.staderlabs.com/your-guide-to-staking-matic-with-stader-e64fcb5c1e62`. [Accessed:02/08/2023]. pages 13

[41] Stader Labs. Polygon — staderlabs docs, . URL `https://www.staderlabs.com/docs-v1/category/polygon`. [Accessed:30/05/2023]. pages 13

[42] Hardhat. Hardhat - ethereum development environment for professionals by nomic foundation. URL `https://hardhat.org/`. [Accessed:07/08/2023]. pages 38

[43] Mocha. Mocha - javascript test framework. URL `https://mochajs.org/`. [Accessed:07/08/2023]. pages 38

[44] Ethers. Ethers documentation. URL `https://docs.ethers.org/v5/`. [Accessed:07/08/2023]. pages 38

[45] Next.js. Next.js by vercel - the react framework. URL `https://nextjs.org/`. [Accessed:08/08/2023]. pages 39

[46] MetaMask. Metamask - the crypto wallet for defi, web3 dapps and nfts. URL `https://metamask.io/`. [Accessed:08/08/2023]. pages 39

[47] Vercel. Vercel. URL `https://vercel.com`. [Accessed:08/08/2023]. pages 39

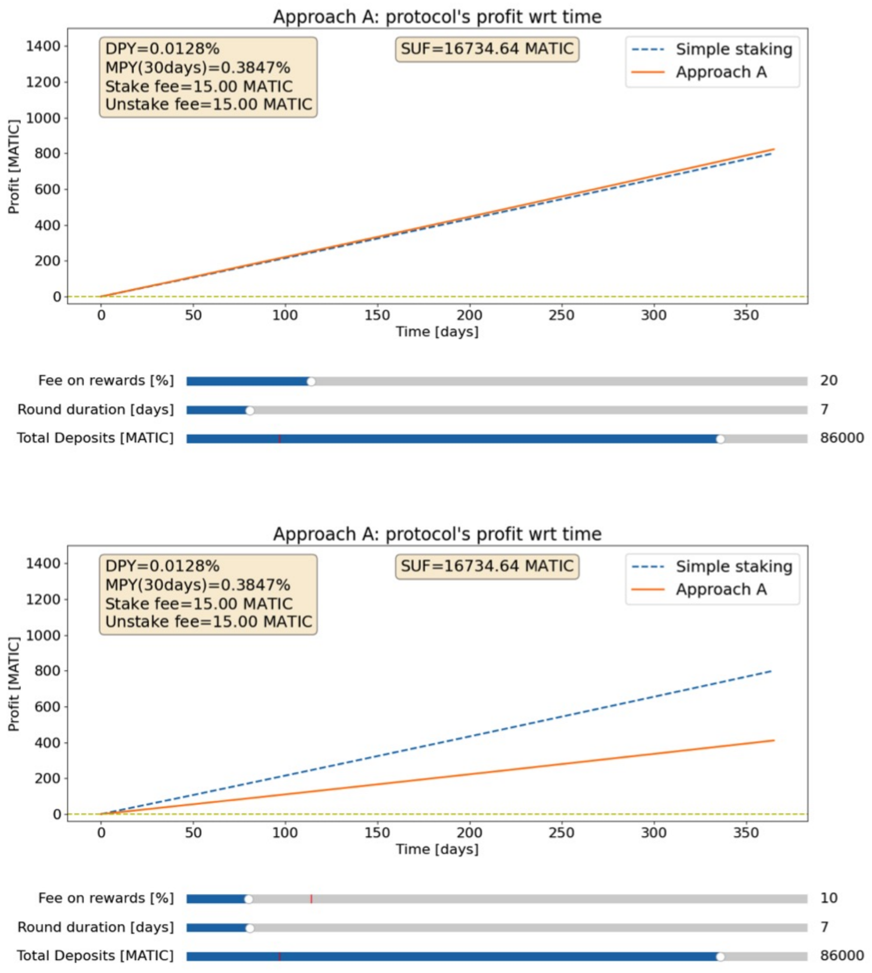
[48] Open Zeppelin Defender. Open zeppelin defender. URL `https://defender.openzeppelin.com/`. [Accessed:08/08/2023]. pages 43

[49] OpenZeppelin. Defender - openzeppelin docs. URL `https://docs.openzeppelin.com/defender/`. [Accessed:08/08/2023]. pages 43

Appendix A

Profit Analysis

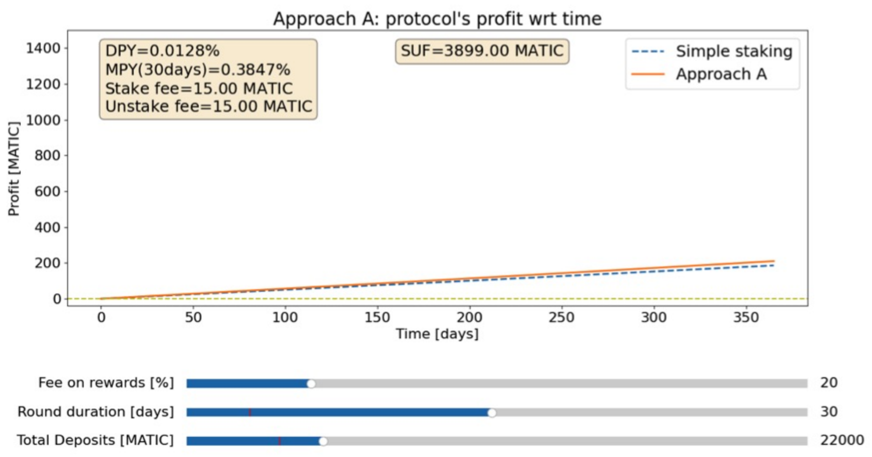
A.1 Approach A: Staking Rounds



Fee on rewards [%] 20
Round duration [days] 7
Total Deposits [MATIC] 86000

Fee on rewards [%] 10
Round duration [days] 7
Total Deposits [MATIC] 86000

Figure A.1: Profit analysis for a round period of 7 days



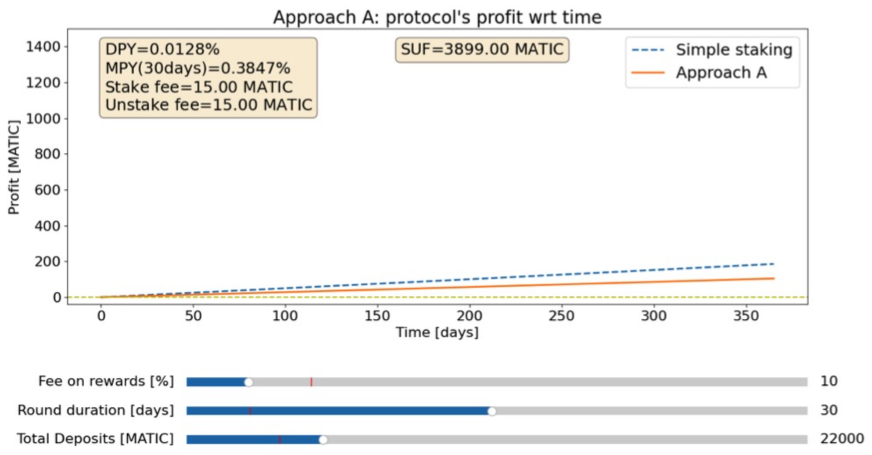


Figure A.2: Profit analysis for a round period of 30 days

A.2 Approach B: S-U level

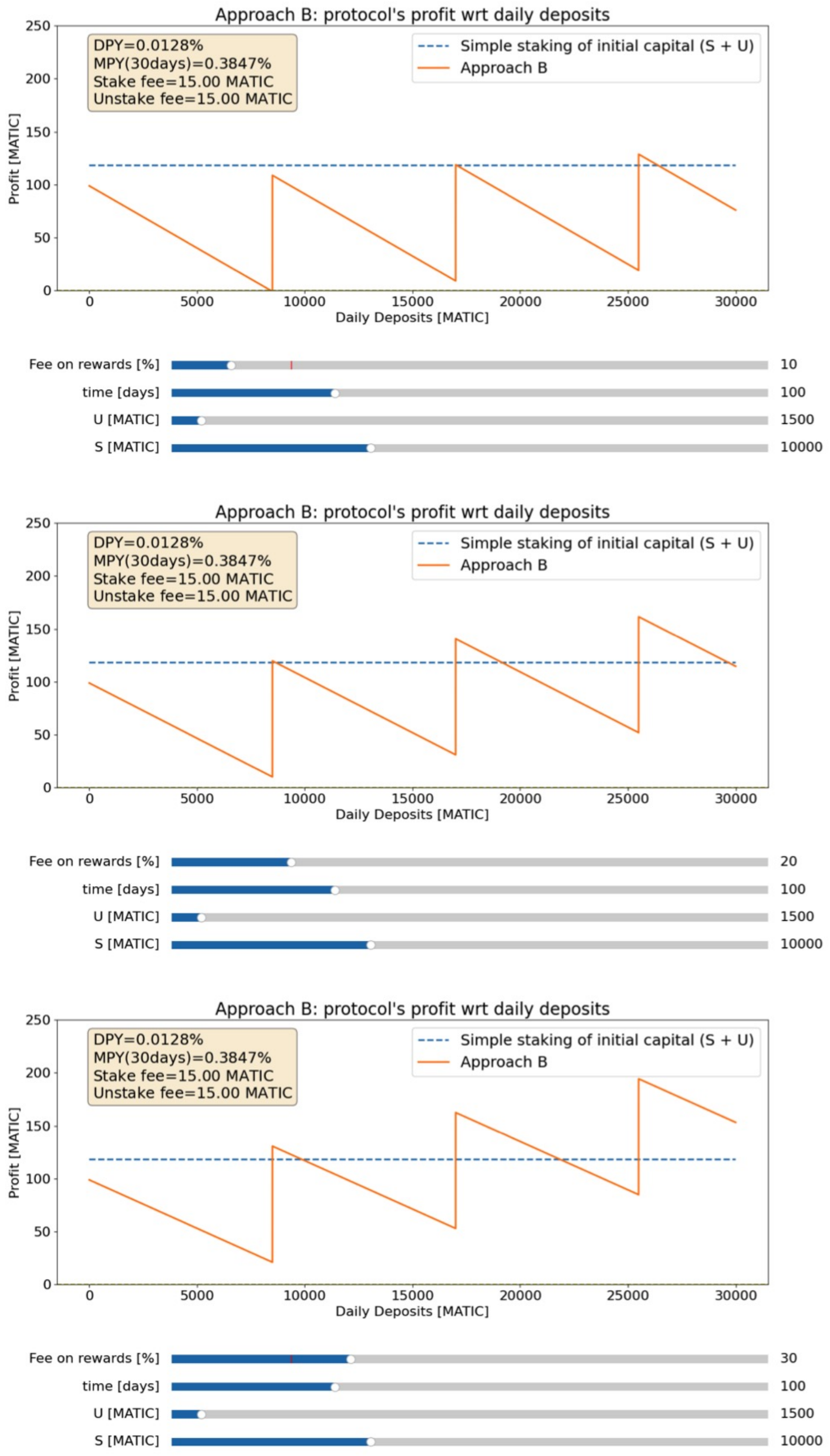


Figure A.3: Profit analysis for different percentages for fee on rewards wrt daily deposits

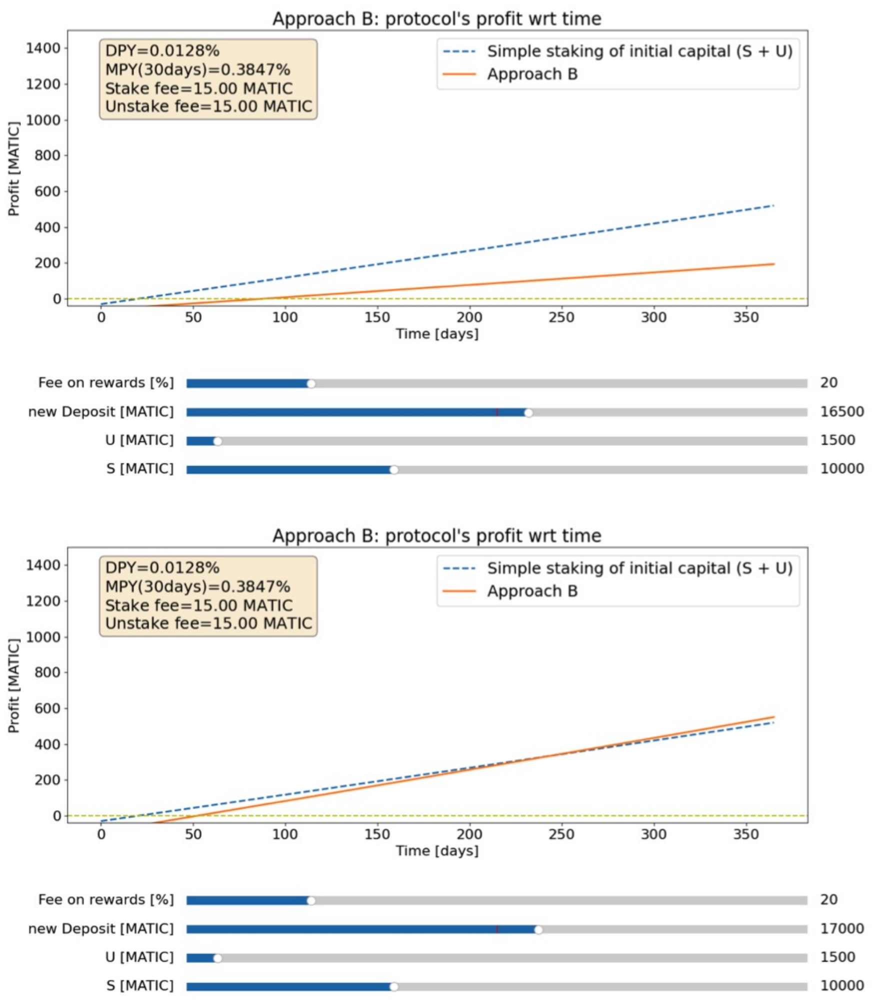


Figure A.4: Profit analysis around the staking threshold

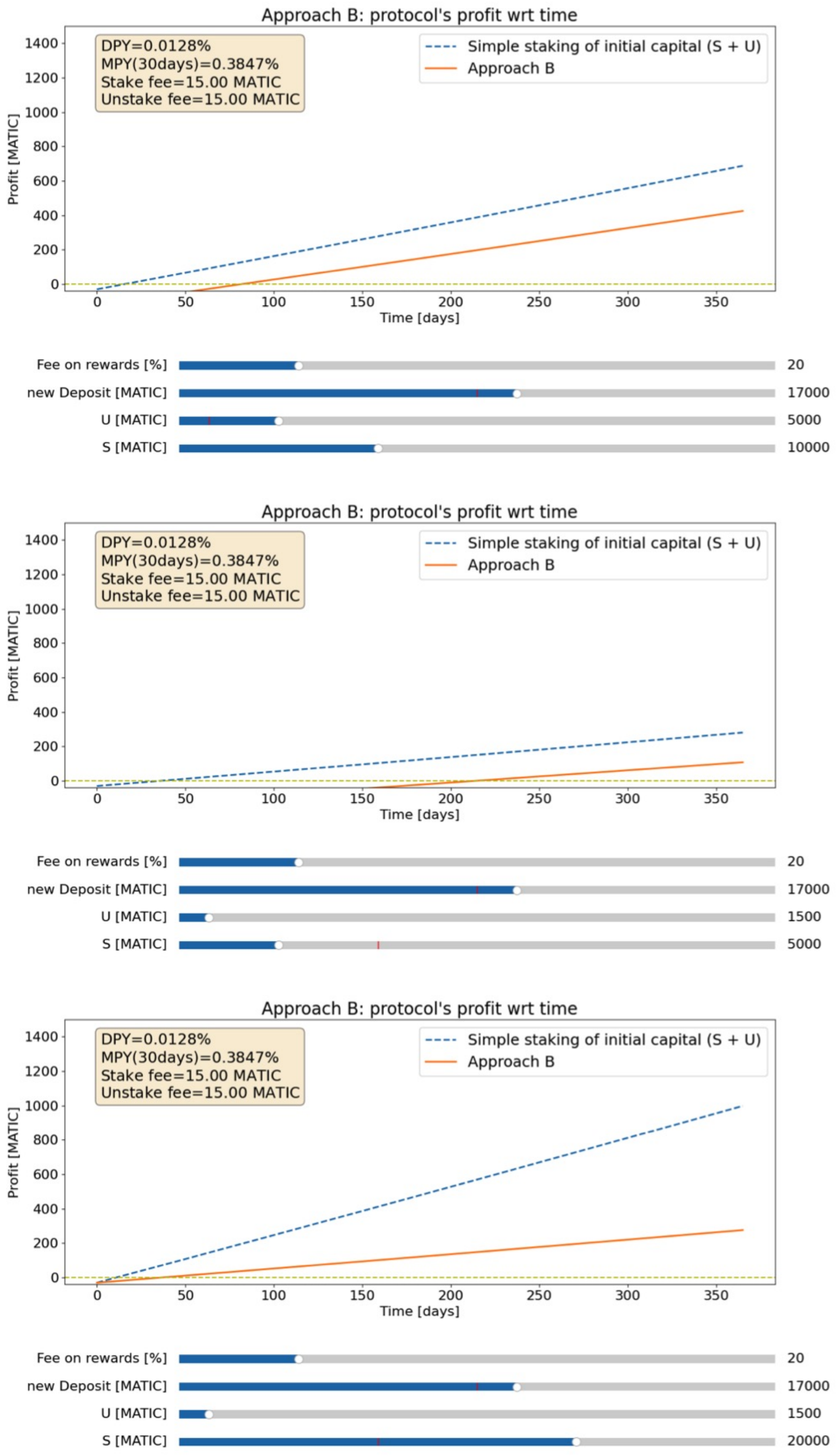


Figure A.5: Profit analysis with various U and S values

A.3 Approach C: Queues with Batches

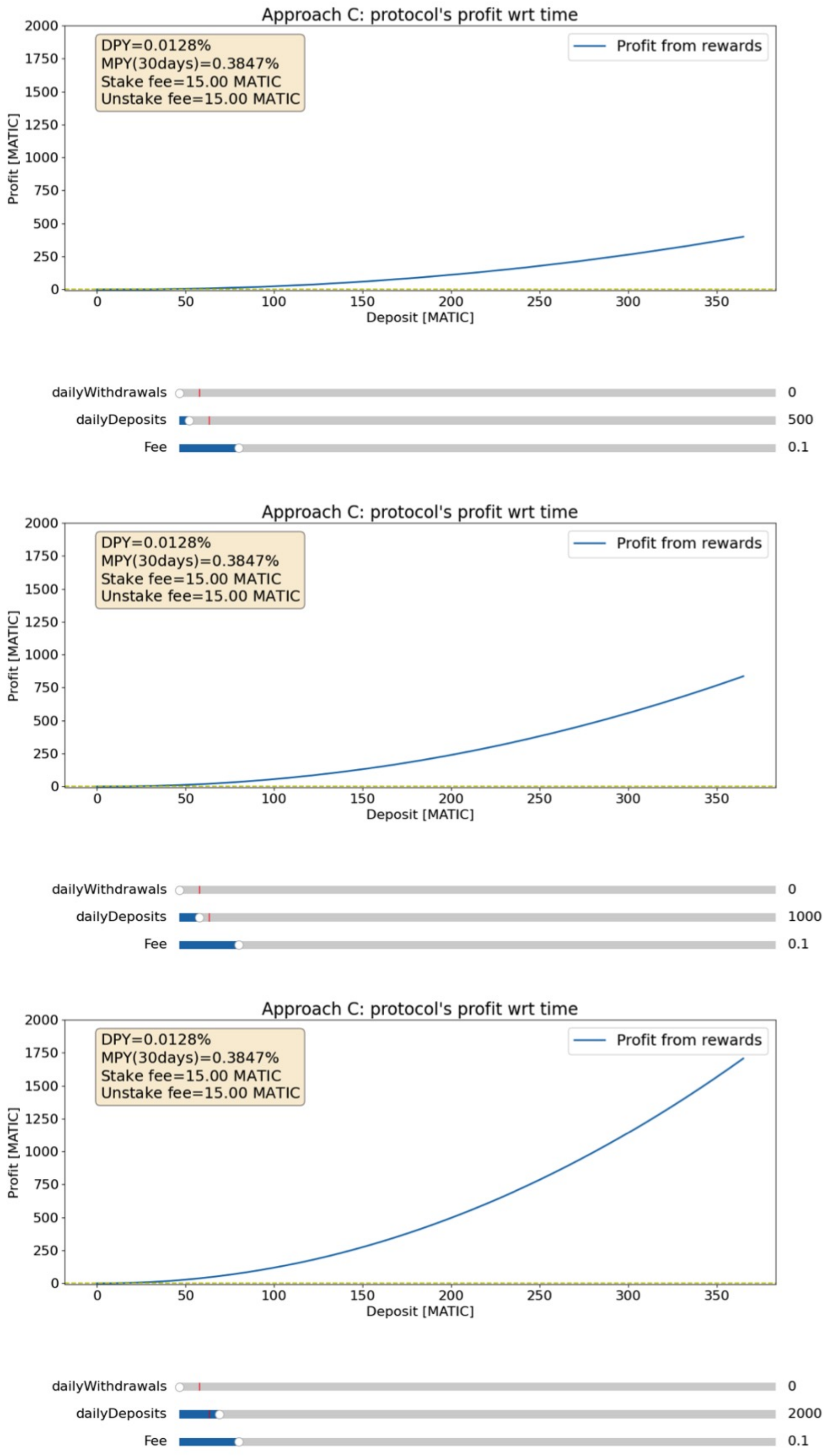


Figure A.6: Profit analysis for a single user

A.4 Approach D: Directly Connected Queues

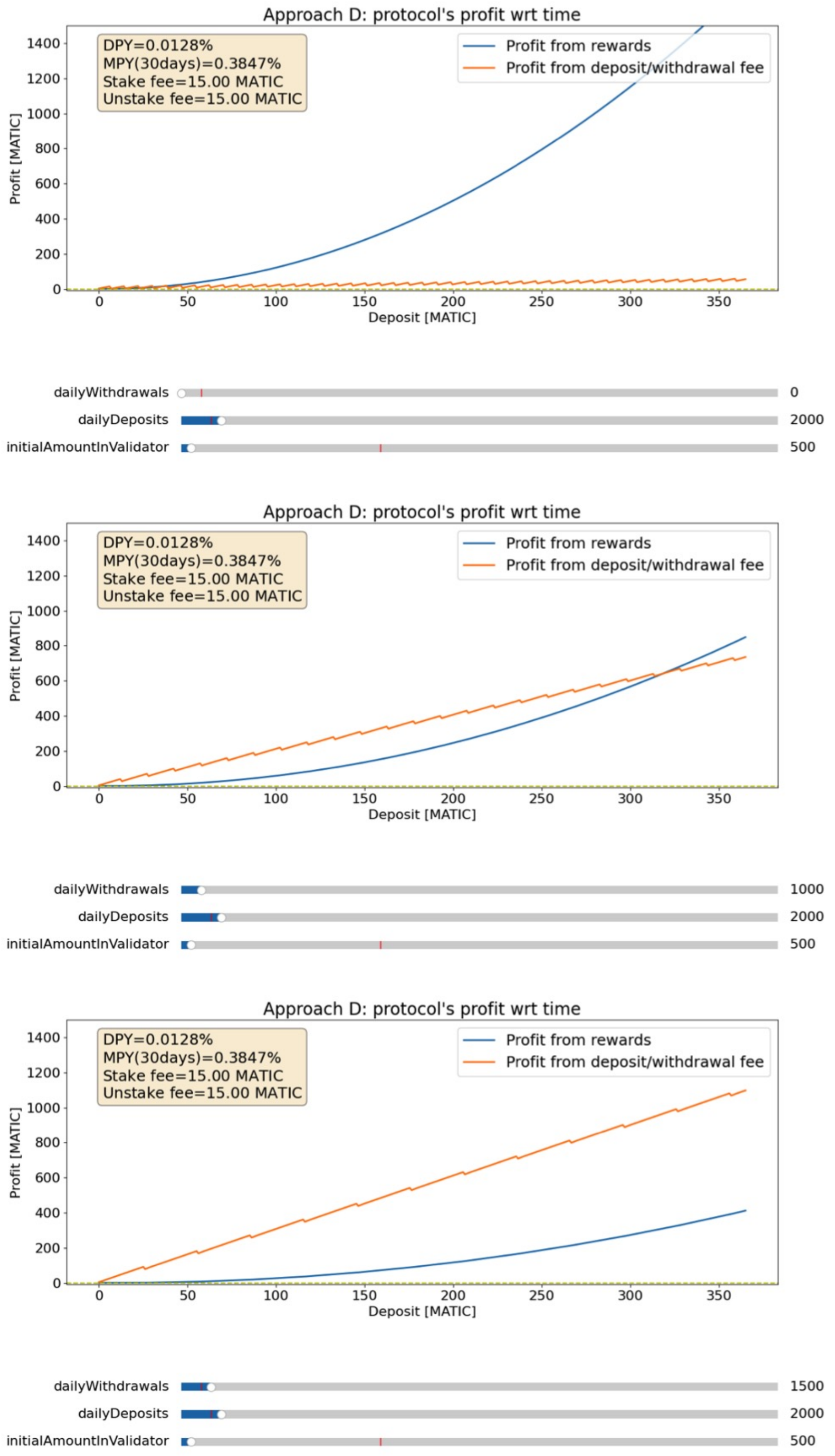
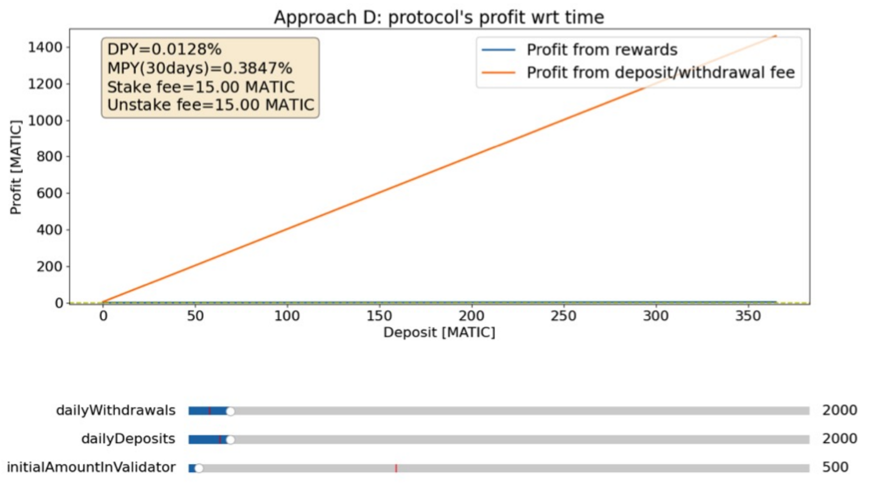


Figure A.7: Profit analysis when daily deposits $>$ daily withdrawals



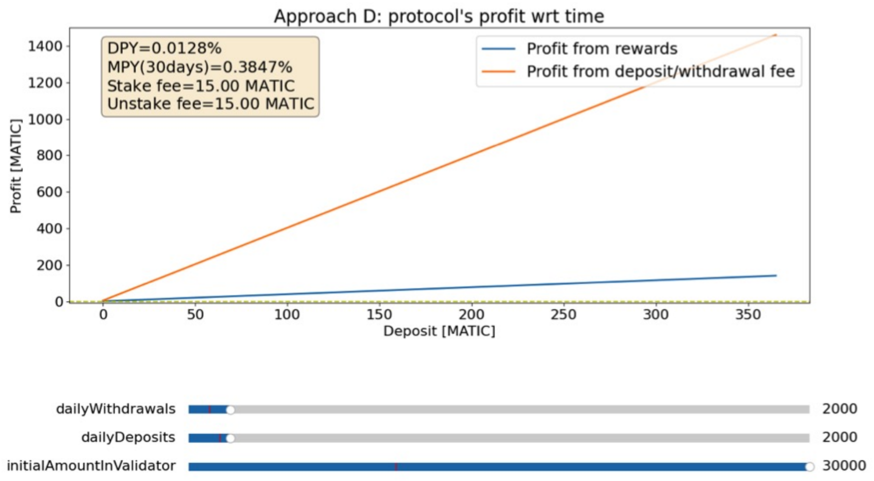
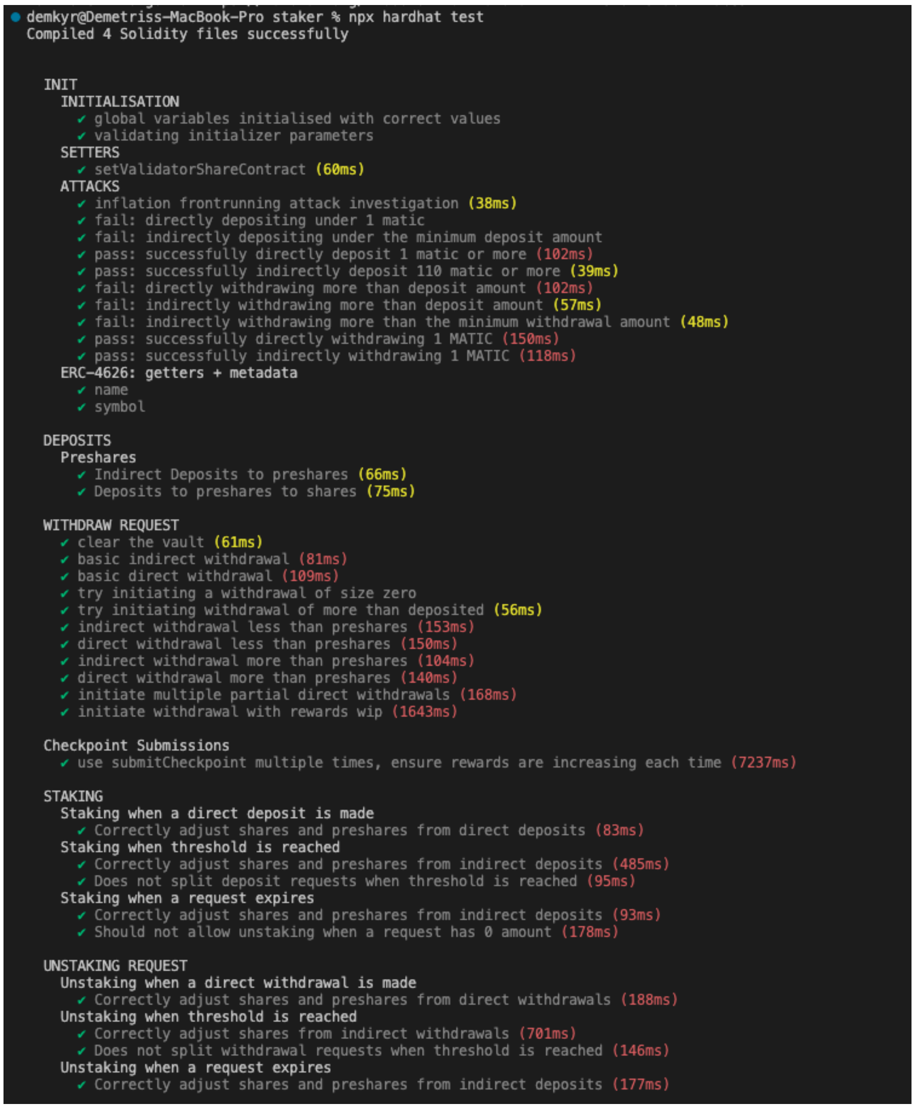


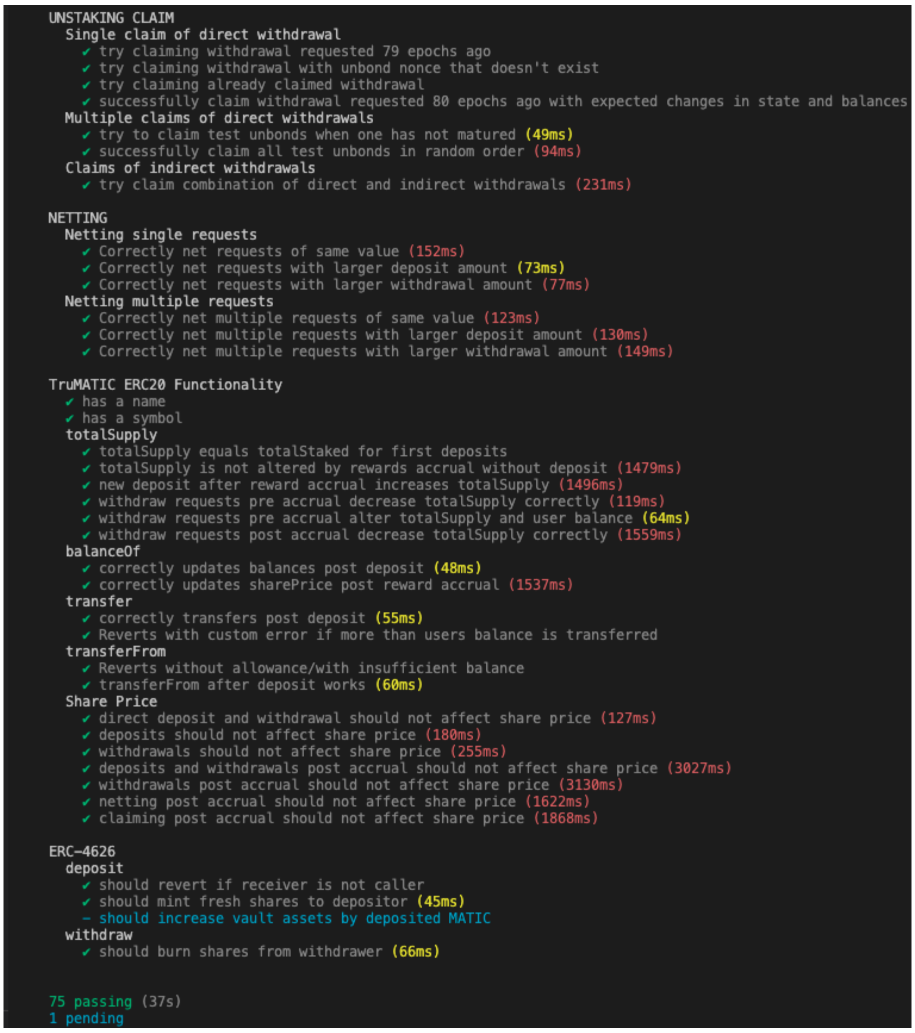
Figure A.8: Profit analysis when daily deposits = daily withdrawals

Appendix B

Testing Screenshots



```
● demkyr@Demetriss-MacBook-Pro staker % npx hardhat test
  Compiled 4 Solidity files successfully


    INIT
      INITIALISATION
        ✔ global variables initialised with correct values
        ✔ validating initializer parameters
      SETTERS
        ✔ setValidatorShareContract (60ms)
      ATTACKS
        ✔ inflation frontrunning attack investigation (38ms)
        ✔ fail: directly depositing under 1 matic
        ✔ fail: indirectly depositing under the minimum deposit amount
        ✔ pass: successfully directly deposit 1 matic or more (102ms)
        ✔ pass: successfully indirectly deposit 110 matic or more (39ms)
        ✔ fail: directly withdrawing more than deposit amount (102ms)
        ✔ fail: indirectly withdrawing more than deposit amount (57ms)
        ✔ fail: indirectly withdrawing more than the minimum withdrawal amount (48ms)
        ✔ pass: successfully directly withdrawing 1 MATIC (150ms)
        ✔ pass: successfully indirectly withdrawing 1 MATIC (118ms)
      ERC-4626: getters + metadata
        ✔ name
        ✔ symbol

    DEPOSITS
      Preshares
        ✔ Indirect Deposits to preshares (66ms)
        ✔ Deposits to preshares to shares (75ms)

    WITHDRAW REQUEST
      ✔ clear the vault (61ms)
      ✔ basic indirect withdrawal (81ms)
      ✔ basic direct withdrawal (109ms)
      ✔ try initiating a withdrawal of size zero
      ✔ try initiating withdrawal of more than deposited (56ms)
      ✔ indirect withdrawal less than preshares (153ms)
      ✔ direct withdrawal less than preshares (150ms)
      ✔ indirect withdrawal more than preshares (104ms)
      ✔ direct withdrawal more than preshares (140ms)
      ✔ initiate multiple partial direct withdrawals (168ms)
      ✔ initiate withdrawal with rewards wip (1643ms)

    Checkpoint Submissions
      ✔ use submitCheckpoint multiple times, ensure rewards are increasing each time (7237ms)

    STAKING
      Staking when a direct deposit is made
        ✔ Correctly adjust shares and preshares from direct deposits (83ms)
      Staking when threshold is reached
        ✔ Correctly adjust shares and preshares from indirect deposits (485ms)
        ✔ Does not split deposit requests when threshold is reached (95ms)
      Staking when a request expires
        ✔ Correctly adjust shares and preshares from indirect deposits (93ms)
        ✔ Should not allow unstaking when a request has 0 amount (178ms)

    UNSTAKING REQUEST
      Unstaking when a direct withdrawal is made
        ✔ Correctly adjust shares and preshares from direct withdrawals (188ms)
      Unstaking when threshold is reached
        ✔ Correctly adjust shares from indirect withdrawals (701ms)
        ✔ Does not split withdrawal requests when threshold is reached (146ms)
      Unstaking when a request expires
        ✔ Correctly adjust shares and preshares from indirect deposits (177ms)
```



```
  UNSTAKING CLAIM
    Single claim of direct withdrawal
      ✔ try claiming withdrawal requested 79 epochs ago
      ✔ try claiming withdrawal with unbond nonce that doesn't exist
      ✔ try claiming already claimed withdrawal
      ✔ successfully claim withdrawal requested 80 epochs ago with expected changes in state and balances
    Multiple claims of direct withdrawals
      ✔ try to claim test unbonds when one has not matured (49ms)
      ✔ successfully claim all test unbonds in random order (94ms)
    Claims of indirect withdrawals
      ✔ try claim combination of direct and indirect withdrawals (231ms)

  NETTING
    Netting single requests
      ✔ Correctly net requests of same value (152ms)
      ✔ Correctly net requests with larger deposit amount (73ms)
      ✔ Correctly net requests with larger withdrawal amount (77ms)
    Netting multiple requests
      ✔ Correctly net multiple requests of same value (123ms)
      ✔ Correctly net multiple requests with larger deposit amount (130ms)
      ✔ Correctly net multiple requests with larger withdrawal amount (149ms)

  TruMATIC ERC20 Functionality
    ✔ has a name
    ✔ has a symbol
    totalSupply
      ✔ totalSupply equals totalStaked for first deposits
      ✔ totalSupply is not altered by rewards accrual without deposit (1479ms)
      ✔ new deposit after reward accrual increases totalSupply (1496ms)
      ✔ withdraw requests pre accrual decrease totalSupply correctly (119ms)
      ✔ withdraw requests pre accrual alter totalSupply and user balance (64ms)
      ✔ withdraw requests post accrual decrease totalSupply correctly (1559ms)
    balanceOf
      ✔ correctly updates balances post deposit (48ms)
      ✔ correctly updates sharePrice post reward accrual (1537ms)
    transfer
      ✔ correctly transfers post deposit (55ms)
      ✔ Reverts with custom error if more than users balance is transferred
    transferFrom
      ✔ Reverts without allowance/with insufficient balance
      ✔ transferFrom after deposit works (60ms)
    Share Price
      ✔ direct deposit and withdrawal should not affect share price (127ms)
      ✔ deposits should not affect share price (180ms)
      ✔ withdrawals should not affect share price (255ms)
      ✔ deposits and withdrawals post accrual should not affect share price (3027ms)
      ✔ withdrawals post accrual should not affect share price (3130ms)
      ✔ netting post accrual should not affect share price (1622ms)
      ✔ claiming post accrual should not affect share price (1868ms)

  ERC-4626
    deposit
      ✔ should revert if receiver is not caller
      ✔ should mint fresh shares to depositor (45ms)
      - should increase vault assets by deposited MATIC
    withdraw
      ✔ should burn shares from withdrawer (66ms)


  75 passing (37s)
  1 pending
```

Figure B.1: Testing screenshots