ReFees: a subscription-based model to hedge Gas Fees on the Ethereum Blockchain

Pau Autrand Caballero
EPFL
Lausanne, Switzerland

Amin Debabeche
EPFL
Lausanne, Switzerland

Lucas Giordano
EPFL
Lausanne, Switzerland

Augustin Kapps
EPFL
Lausanne, Switzerland

***Abstract*—Large gas prices and volatility are the cause of multiple problems on the Ethereum Mainnet. Several solutions such as Layer 2 or Gas Token were proposed over the last few years to deal with them. However, on the Mainnet, traffic keeps increasing and gas fees problems are far from being solved. To this end, we propose, *ReFees*, a scheme to harness the speculative power of gas fee volatility to a party, while at the same time offering insurance guarantees to another. This article is divided into two main sections. As a first step, we introduce the gas fee problem, investigate existing and future solutions to finally demonstrate the usefulness of our system. Secondly, we develop the mechanisms behind *ReFees*, as a proof of concept.**

***Index Terms*—Ethereum, Gas Price, Gas Mechanism, Subscription, Blockchain,**

I. Identified problem

A. Introduction to gas fees

Ethereum[1] supports smart contracts, implying that some nodes, commonly called validators, of the network have to run the solidity code of these contracts when required. Since solidity is a Turing complete language[2], it remains impossible to say if a given piece of code will terminate and how much resources it will use. It refers to the halting problem which is NP-hard[3]. To avoid malicious people or errors to crash the network with an infinite loop, the users of the contract provide a certain amount of gas x for each execution. When validators execute smart contracts, they decompose the code into basic low-level operations. Each of these operations (o) is also associated with a fixed amount of Gas ($Gas(o)$). Then, for each executed operation the validator subtracts $Gas(o)$ from x, as a reward for his work. If x goes to 0, then the execution terminates, in this case, the operations of the contract are reverted but the Gas is not refunded to the user. Because gas has a cost in ETH, to run an infinite loop of operations an infinite amount of money would be needed. Every operation has a determined gas price associated with low-level instructions, for example, 24000 Gas units are necessary for the refund given for a self-destructing account[4]. A gas unit is unitless but can be converted in Gwei a sub-unit of ETH, this conversion rate is driven by market bid and ask interactions. A specific user associate a gas price he is willing to pay to execute his operations and this is from this pool of offers that validators select which user operations are chosen to be validated. To maximise their profit, validators will surely select to validate operations with the highest gas prices. This is the reason for the gas price market spikes in times of high demand: users are willing to pay more to get their operations validated faster[5;6]

B. Gas prices and volume

The number of transactions that can be included in each new block uploaded to the Ethereum blockchain is limited. Therefore, as stated before, miners are incentivised to accept transactions at greater gas fees due to supply and demand for the network's processing capacity. As of November 2021, the daily amount of gas used per day is roughly 100 billion, and it increases every day, see Figure 1.

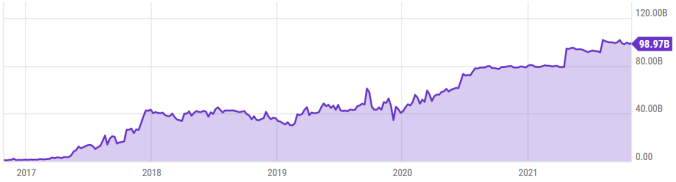


Fig. 1. Chart representing 5 years of historical data on Gas volume.[7]

This market movement creates volatility in the gas price, as seen in Figure 2, one year of historical data on gas prices shows a clear pattern of price shocks as seen in Figure 3. One also sees an increasing Gas volume needed, together with exceeding transactions requests when compared to the actual Ethereum transactions limitation[8]. Therefore, this high demand coupled with the discussed limitation deters possible investors from using DeFi (Decentralised Finance) protocols[9].

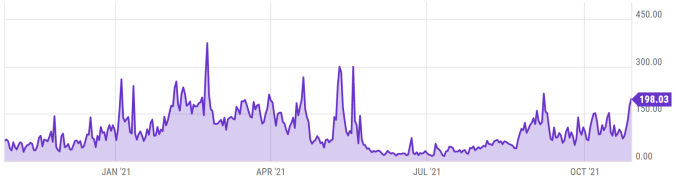


Fig. 2. Chart representing 1 year of historical data on Gas prices.[7]

C. Projects built on Ethereum

Distributed ledgers technology is truly revolutionising, together with Ethereum framework to easily create smart contracts on its blockchain, it draws attention of many developers. Currently the number of DeFi projects built on the Ethereum blockchain, referred to as Layer 1, is 220 projects. It is

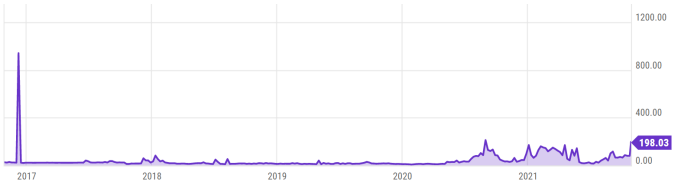


Fig. 3. Chart representing 5 years of historical data on Gas prices.[7]

important to notice that the Gas consumption is not uniformly distributed among all the projects[10]. For instance, *Uniswap*[11] is one of the projects that consume the most, with an average consumption of 6% of all the Layer 1 Gas. As a matter of fact, *Uniswap* is only surpassed by *OpenSea*[12], with a 14%.

All in all, Gas fees are becoming an increasing problem that could prevent users from using the Ethereum network[13].

II. Existing solutions

A. $GAS Token

$GAS token is an existing solution to hedge Gas price volatility, using the following simple concept: mint gas when its price is low and redeem it when the price is high[14]. Such a solution is possible thanks to the Ethereum Gas refund mechanism. A missing explanation to the previous section is that particular operations on the blockchain lead to a gas refund. Whenever storage is reduced by a miner, this latter is rewarded with a gas refund, the number of gas is lower than the gas necessary to store data on the block. Nevertheless, if the refund is triggered at time of high gas price, a return on a initial payment may achieved. Practically, writing data in a new block costs 20'000 gas and then freeing this data refunds 10'000 gas. A rough approximation of this strategy's return is as follow:

$$return = -20000 \cdot gas_{low} + (15000 - 5000) \cdot gas_{high}$$

Gas_{low} is the gas price at low volatility, while gas_{high} is the price of gas at a higher market gas price.

One could create its own smart contract to achieve such a mechanism or buy tokens on exchange platforms that perform similar methodologies, examples of tokens are CHI, GST1 and GST2. The first approach is accessible to experienced users, whereas a $GAS token has to be bought, its price is set by the market and not only by the intrinsic cost of gas[15]. Moreover, an inspection on Etherscan transactions (cf. Fig. 4) reveals that some $GAS tokens are illiquid, which is not convenient from the user's perspective. Indeed, users may want to easily and readily sell or exchange its asset without substantial loss in its value.

Txn Hash	Method ⓘ	Age
0xff19476e423cedb8477…	Swap	1 day 21 hrs ago
0xff19476e423cedb8477…	Swap	1 day 21 hrs ago
0x02e7916e013cf55d4cf…	Transfer	1 day 22 hrs ago
0x8ab59ae8c2e163694a…	Swap	7 days 13 hrs ago
0x8ab59ae8c2e163694a…	Swap	7 days 13 hrs ago

(a) GST2

Txn Hash	Method ⓘ	Age
0xd226c11bffb7c8c4f562…	0x7c025200	28 days 2 hrs ago
0xd226c11bffb7c8c4f562…	0x7c025200	28 days 2 hrs ago
0xcf1aa67822bfe7b8b06…	Transfer	157 days 17 hrs ago
0x28ed255ad0bc1a8660…	Transfer	170 days 1 hr ago

(b) GST1

Txn Hash	Method ⓘ	Age
0xf5e2e3042bd595479f4…	0xc18a84bc	2 mins ago
0xff5cbec488b2abf1e9b8…	0xc18a84bc	2 mins ago
0xb090d4b78e67cd203f…	0xc18a84bc	2 mins ago
0x0f1a9c4d7566f241ba1…	0xc18a84bc	6 mins ago

(c) CHI token (most liquid)

Fig. 4. Examples of illiquidity for $GAS tokens

B. Layer 2

The term Layer 2 refers to a series of blockchain solutions, built on a different network running on top of Layer 1, through smart contracts. They are designed to increase the scalability of Layer 1 while still being able to benefit, to some extent, from the security and the existing architecture of the Mainnet. The main idea for these protocols is to perform transactions off the Mainnet, said off-chain, to achieve a faster confirmation time, lower fees, higher throughput and traffic reduction on the Mainnet[16;17].

For the reader's reference, a detailed list of the different solutions is available in appendices A and B. These appendices aim to systematise the panel of Layer 2 protocols that emerged in the past decade. To this end, a comparison of the main solutions in terms of structure, synchronisation with Layer 1, scalability power and security is provided.

The main drawback[18] of Layer 2 solutions is that the system does not live on Layer 1, it cannot interact with Layer 1 instantaneously and this involves exorbitant costs.

A Lot of interesting projects (OpenSea, Uniswap, Aave ...) are only present in a few Layer 2 solutions. For example, Aave is only available on Polygon, and Uniswap on the main network and Optimism. This makes it impossible to create an operation using both smart contracts and this lack of combination between DeFi protocols on Layer 2 is another drawback.

Liquidity also suffers from the presence of multiple layers. Indeed, the liquidity is split across layers and does not flow easily from one to another.

According to the scalability trilemma seeFig. 5), systems cannot achieve all of the following aspects: scalability, security

and decentralisation. It means that if a solution performs well regarding scalability and security, it will have decentralisation shortfalls. Lastly, as long as a decent amount of people keeps using the Ethereum main network, gas fees volatility will persist, regardless of second layers.[19]

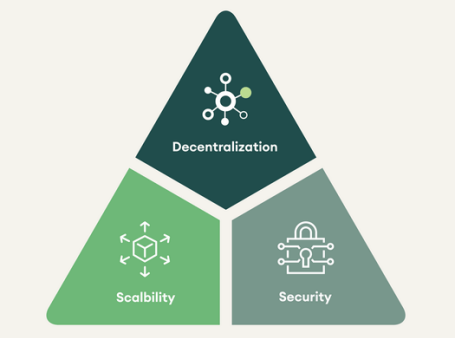


Fig. 5. Blockchain scalability trilemma[20]

Ultimately, the necessity of another tool bringing novelties not captured by the existing panel of Layer 2 solutions is shown by a proof of need, which conducts in the creation of *ReFees* protocol.

III. Future solutions

Even though the following solutions are not fully implemented for the moment, it is still important to mention them to see if *ReFees* system would still be useful even with these new solutions. They will mostly solve the network scalability and congestion problems.[21]

A. Sharding

Sharding is a solution proposed to solve the Ethereum scalability problem that could make the network less busy and thus reduce gas fees. Sharding is inspired by database Shard architecture and intends to change the network structure. Using all nodes of the network to validate every operation might not be essential. Sharding proposes to divide the network into multiple sections called "shards". Each shard processes then its own set of addresses and transactions and has its independent state. But this method implies a lot of challenges, especially on the Ethereum blockchain. Splitting the network into shards implies splitting the consensus mechanism in a balanced manner, but using proof of work: it is difficult to assign a miner to a particular shard since the miner can always compute the hashes of whichever block he wants. This is not the case on a proof of work consense since there are validators that can already lock some funds on a particular shard[22]. Sharding is already implemented on other blockchains such as Elrond.

B. Proof of stake consensus

The Eth2 upgrades also intend to solve the scalability problem by changing the consensus mechanism from proof of work to proof of stake. This change will result in a more efficient network, using less energy and it will also provide stronger support to implement the sharding mechanism described above. However, the change has already been studied for a while but there is not any official date of deployment. The introduction of this new consensus mechanism will certainly reduce the gas price by increasing the throughput of the network but these fees will still exist[23]. Even by using proof of stake one still needs to run the code of smart contracts.

IV. *ReFees*

ReFees intend to design a subscription system to tackle some of the problems induced by the volatility of gas fees on the Ethereum network. *ReFees* is decentralised and involves a subscription contract signed by the following main actors:

1) A **client** for whom gas fees volatility is perceived as a valid reason to reduce its trading volume. This latter is willing to overpay for a system that provides him with reliable gas fees management. *ReFees* allows him to pay a regular and fixed amount of ETH to a provider to get his gas fees reimbursed and therefore hedge his risk.
2) A **provider** who is assumed to have a good understanding of the blockchain infrastructure, markets and gas fees. Through the system, he locks funds in a pool used for future reimbursement of gas fees from the client. The provider speculates with the expectation that the gas fee will decrease in value over time, generating yield. This yield is a bet on the fact that the amount generated by the regular payments made by the client is higher than the actual client gas fees consumption during the lifetime of the smart contract.

Decentralisation, on top of anonymity guarantees for both providers and clients motivate the choice of blockchain as the proper support of this system.

An important consideration for the development of *ReFees* is the user-friendliness of the client intervention on the system. *ReFees* aims to make it as easy and automated as possible for the client to get his fees reimbursed.

A *ReFees* subscription contract between a provider and a client is defined in the terms of the variables shown in table I. This contract operates by phases of f blocks, defining a notion of time t for the system. The dynamics of the system are described in detail in the following sections. They are also summarised in figure 6.

A. Reserve Pool

At the core of the system, there is a reserve pool from which ETH [1] can be withdrawn to reimburse gas fees spent on the Mainnet by the client in his external transactions. The pool is funded by two sources:

1) At the contract creation time ($t = 0$), the provider needs to lock an initial reserve r to provide default liquidity to the system.
2) At the beginning of each phase ($t = f \cdot k \; \forall_{k=0,1,\dots,M/f}$), the client needs to pay a fee to be able to use the system. Without loss of generality, to ease the comparison with

[1]As defined in sectionIV-F, the pool can also hold $REF, the native tokens

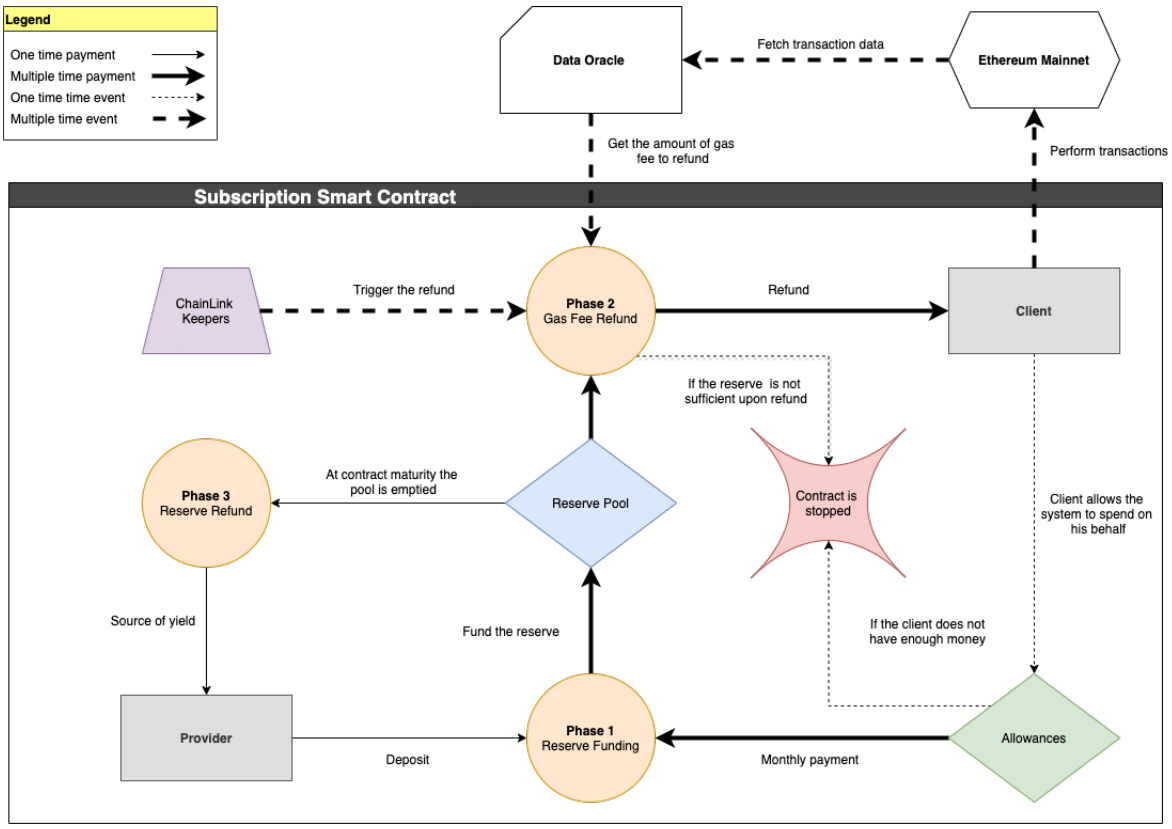


Fig. 6. System dynamics

TABLE I
SMART CONTRACT VARIABLES

Name	Ref.	***Description***	Unit
Maturity	M	The time horizon of subscription. M needs to be a multiple of f	# blocks.
Payment frequency	f	frequency of the payments that the client has to do	# blocks.
Payment amount	x	the payment amount that shall be paid by the client, at frequency f.	ETH.
Initial reserve	r	the amount of ETH that the provider has to lock in the contract. This reserve is used to reimburse the client's transaction gas fees and internal gas fees.	ETH.
Client's usage address	u_{add}	address registered by the client. The transaction performed by this address will be reimbursed.	ETH addr.
Client reimbursement address	r_{add}	Address provided by the client where to send the reimbursement of transaction gas fees generated by u_{add}.	ETH addr.
Max operations	n_{ops}	Dictionary that maps every kind of operation ("transfer", "approve", "swap"...) to an integer: the maximum number corresponding operations that can be refunded every f blocks.	dictionary
Mean factor	α	Upper bound on the reimbursement. The system only reimburses up to α times the current average (over k blocks) of the corresponding operation's gas fees.	float

standard off-chain subscription systems, f is set to a monthly frequency in the remaining sections.

The reserve pool also serves the purpose of financing the necessary gas fees to run the code of the system. These gas fees are referred to as "internal" in the following sections.

Finally, at maturity ($t = M$), the pool is emptied and the residual value is refunded to the provider. It marks the completion of the smart contract.

B. Client monthly payments

After signing the subscription smart contract, the client is subject to pay a monthly fee to finance the system until the contract maturity. One could argue that the system could be simplified to force the client to make one single bulk payment at signature times. However, it would induce several limitations on the client-side:

1) The client may not have the necessary amount of ETH for the bulk payment at signature times. Spreading the bulk payment over multiple periods provides flexibility to the client.

2) The client may want to exit his position before maturity to discharge his liabilities to the system. For instance, such a situation could occur if the client believes he made a bad deal upon signing the contract or if he thinks he will not be able to cope with the future payments. In this case, the client wishes to extract as much ETH as soon as possible out of the system. However, the bulk payment will only be partially refunded when he finds a buyer for the existing contract. The issue is limited if spread payments are used. Indeed, at time $t < M$ only $x \cdot \lceil t/f \rceil$ ETH have entered the system. In section IV-E, we demonstrate how such an exit can be implemented.

Furthermore, the *ReFees* subscription contract also makes it possible for users to create such a bulk payment scheme by setting $F = M$. In that sense, we provide a broader action space to the users of the system.

In terms of implementation, we could consider various ways for *ReFees* to collect the client payments:

1) Manual payments: the client needs to manually make a transaction of x to the subscription contract before the beginning of each new phase. If no payment is made, the execution of the smart contract is stopped. If this option is rather simple to implement and could act as a trigger event for the refund system (cf. section IV-C), it suffers from important drawbacks:
   a) Not user-friendly: an automated collection of payments would require fewer interventions on the client-side.
   b) Less incentive for the provider: the client can stop paying and leave the system which does not benefit the provider.
2) Private payment pool: an alternative to the manual payment is to ask the client to lock $x \cdot f$, the full amount he needs to pay throughout the contract lifetime, upon signature into a private pool (not used to repay gas fees). This pool will automatically trigger a payment to the main *ReFees* reserve pool at the beginning of each phase. Compared to the first approach, the pool plays the role of insurance both for the system and the provider. We mention that this implementation has the same drawbacks as the bulk approach defined at the top of this section. However, since the pool is private to the client, its content cannot be used to pay internal gas fees. Therefore, it reduces the amount that can be lost if the client wants to exit at some point.
3) Allowances: finally, we can make use of the ERC-20 allowances interface to make the client give his approval for *ReFees* to spend some ETH on his behalf. The allowance would be initially set to $x \cdot f$ and *ReFees* will collect x at the beginning of each phase. Allowances are simple to use and easy to implement. However, given all scam tokens that have risen in the past few years, the Ethereum community requires security guarantees to trust *ReFees* code[24]. Hence, auditing costs are expected to be higher for this type of implementation.

Given the user-friendliness and flexibility constraints stated in this section, the allowances choice is a better fit for *ReFees*.

C. gas fees refund

In *ReFees*, the client pays a fixed monthly fee x that allows him to get refunded the gas fees spent for his transactions on the Ethereum Mainnet. To avoid abusive behaviour on the client-side, we need to restrict the refund power in the following two aspects:

1) The maximum number of monthly transactions for which *ReFees* will refund the associated gas fees. For instance, if set to 100, it only allows for 100 refunded gas fee transactions. If this restriction did not apply, the customer could always generate a large number of transactions to empty the reserve pool, resulting in a large loss for the provider. No provider would use the system in this situation. We generalise further this notion to any operation (n_{ops}) that can be performed (e.g. "transfer", "approve", "swap"...) in the Mainnet to better capture the differences in gas fees of those operations. n_{ops} is therefore defined as a dictionary mapping events to integers.
2) An upper bound of the reimbursement per operation called the mean factor α. Indeed, upon performing a transaction the client can choose how much gas to use. The more he uses, the faster the operation validation is. Without this restriction, a client would always choose very high gas fees with negatively impacts the provider incentives to use the system. Therefore, the smart contract will only reimburse gas fees up to α times the current average (over k blocks) of the corresponding operation's gas fees. For example, if $\alpha = 1.25$ and that the average gas price for a swap operation is currently $50[Gwei]$ then the client can use up to $50 \cdot 1.25 = 62.5\,[Gwei]$ for a swap.

C.1 Fetching client transactions data

To reimburse the client's gas fee, the contract has to be aware of all his transactions. Thus, a transaction tracking mechanism for the address u_{add} needs to be implemented. Since our system will not be hosted in Layer 1 (cf. section IV-D), the project will operate on a different level than the transactions to be fetched. A data oracle[25] is the solution to this problem.

Two options are available:

1) Centralised: a server using the Etherscan API[26]. Transactions of client u_{add} are accessible at the web address "https://etherscan.io/address/u_{add}". Etherscan offers multiple plans for real-time APIs.
2) Decentralised: a framework such as *Graph*[27].

At this early step of the system, the centralised solution was chosen, mostly to simplify the development process.

To improve security, we also need to decentralise the way off-chain data is collected. To this end, *Chainlink Off-Chain Reporting*[28] will be used to build a network of nodes calling the API and aggregating the data on-chain.

C.2 The refund procedure

To facilitate the use of the system and increase user experience for the client, *ReFees* will fetch the client transactions, and decide when a refund applies. In other words, the client does not have to register transactions manually to the system. The provider can also benefit from this because it forbids the user to only register transactions with high gas fees.

For each transaction made by the client, there are three main components of the refund procedure:

1) Perform: the client performs the transaction on the Mainnet without interacting with *ReFees*.
2) Fetch data: the oracle defined in section IV-C1 will fetch the data information and extract the number of gas fees paid.
3) Refund gas fees: using this information, a *refund* function can be triggered in the smart contract receiving as input the gas fees amount from the data oracle.

As a side note, if upon receiving a refund request the reserve pool is empty or not sufficient to perform the full refund, the contract terminates and both the provider and client are released from their obligations.

In terms of implementation, this automatic mechanism is not trivial. Indeed, the system needs events to be able to trigger the chain of actions resulting in the actual refund for the client. If the user was required to manually register the transactions to the system, this registration event could simply be used as a trigger. However, it is not an option in *ReFees* as previously discussed.

A simple way to overcome this issue would be to extend the data oracle server to periodically fetches data in an automated way. Each update event can be used to trigger the smart contract refund function. However, this implementation severely reduces the decentralisation and security benefits of the system. A loop needs to be run on the server which represents a single point of failure.

Recently, *Chainlink* launched the *Keepers*[29] project, which exactly serves this automation purpose in a decentralised and reliable way. *Chainlink Keepers* makes a decentralised network of nodes available that are rewarded to execute jobs based on predefined conditions (e.g. time).

Making Keeper-compatible contracts[30] simply requires implementing two functions:

1) *performUpkeep*: a call to the *ReFees refund* method.
2) *checkUpkeep*: condition checker. For *ReFees* it acts as a scheduler and multiple options can be considered:
    a) New transaction event: each time *checkUpkeep* is called, the data oracle is called as well. If the latter returns new gas fees to be refunded for some users, then the condition status is set to true. In other words, with this method gas fees are refunded soon after the transaction is confirmed in the Mainnet which provides flexibility for the client. However, due to the complexity of the implementation and the frequency of checks, higher internal gas fees consumption is to be expected compared to point 2b.
    b) End of month event: in this case, *checkUpKeep* simply check that the current block number corresponds to the end of the monthly period. It is equivalent to make one bulk payment to refund all gas fees spent in the client's last month utilisation. It is cheap in terms of internal gas fees consumption but lacks flexibility since the client needs to wait until the end of the period to get the refund.

Eventually, *ReFees* will support both types of *checkUpkeep* functions to provide higher flexibility to the users of the system.

D. Host chains

Implementing the project on the Ethereum Layer 1 itself is not a good solution because using the contract might also involve high gas fees upon running. Indeed, the system achieves nothing if the process of reimbursing a single transaction costs more gas fees than the transaction itself.

Scaling *ReFees* to Layer 2 allows the contract to operate within the layer, achieving lower fees while still ensuring communication and compatibility with the first layer to track the client's transaction.

In terms of blockchain infrastructure, *ReFees* will first be hosted on *Polygon* to leverage the power of Layer 2 and *Chainlink Keepers*. In the future, if more networks are supported by the *Chainlink Keepers* project, we may consider hosting *ReFees* on several other Layer 2 solutions to increase availability for users.

E. ReFees Tokens and Smart Contracts

In this section, we aim to demonstrate how *ReFees* can be modelled as a set of smart contracts, operating between the provider, the client and the oracle. For reasons that will be explained in the following paragraphs, we chose to model *ReFees* using the following building blocks:

1) A native token $REF which also acts as a governance token. It follows the ERC-20 standard.
2) The *ReFees* subscription smart contract containing the reserve pool funding methods as well as the gas fees refund method.
3) A pair of NFT Tokens, representing both contracts ownership for the client and the provider. It follows the ERC-721 standard.
4) *ReFees* Factory: a general purpose collection type of smart contract managing the list of existing contracts.
5) A Chainlink smart contract to access the data oracle.
6) A Chainlink keepers smart contract to trigger refunds.

In practice, at contract creation time, both provider and client addresses are set to the creator's address. Then, the creator can trade/give the ownership of one side of the contract on a NFT marketplace. For example, if the creator wants to be the provider of the subscription, he will trade the client-side token. The buyer of the token then becomes the client.

The contract starts whenever the two addresses (client and provider) are different. During the subscription lifetime,

we provide flexibility by allowing any party to trade its ownership token and exit the system. When ownership is traded, the contract simply updates the concerned address. Ownership could be traded as NFTs on different marketplaces (ex: https://opensea.io/). In that case, a user can claim one ownership using a "claim" method of the contract. This method checks if the claimer processes the corresponding NFT and updates one of the addresses if yes. The NFTs are minted by the subscription contract itself, ensuring that they can not be counterfeited (one can verify the emitter identity of the given NFT)

Having the possibility to trade ownership allows the actors to easily get rid of theirs position (assuming liquidity is high). Imagine the scenario where a provider locks some funds in a subscription contract and some days later realises that the Gas price will go up. This provider can then sell his ownership to someone else and avoid losses. Trading ownerships also allows the client to speculate on the gas price going up. Imagine that a client starts a contract with a long maturity when the gas price is low. Then, if the gas prices suddenly rise, the client can sell his ownership to another client for a higher price than the initial and make some profit.

To conclude, the system will support the following list of requirements:

1) The subscription contract can be created by either the provider or the client.
2) The party responsible for the creation can trade ownership on the contract via a NFT marketplace.
3) In the NFT marketplace, contract prices are set by the levels of the demand and the offer.
4) The creation phase amounts to fill in the variables highlighted in table I.
5) Any party can decide to resell his ownership token on the NFT marketplace. If the token is bought, the new owner shall respect the contractual obligations defined at contract creation time.

More information on the tokenomics of the system can be found in section IV-F.

F. Tokenomics

ReFees system is ruled by the $REF token, a native and governance token. This token acts as voting power and can be used to trade contracts.

Holding some of that native tokens involves multiple benefits:

1) The system's Decentralised Authoritative Organisation (DAO) relies on the native token $REF. Having $REF tokens allows the owner to vote on the future of the project.
2) Whenever a contract terminates, a small part of the final reserve pool is evenly distributed to the token holders. This reward is paid in ETH or $REF, depending on the token used by the contract.
3) As explained in section IV-E, the ownership of a contract (client or provider side) is tradable, and every time such a transaction happens, a small part of the established price is redistributed to token holders.

ReFees's subscription contract will allow users to agree on the currency used by the system. Users will be able to choose between ETH and the *ReFees* native token ($REF). Specifically, all transactions made to the pool and refunded gas fees will occur in the specified token. To attract people on using the $REF token, a premium is distributed to both parties of a subscription contract, in $REF, that terminates. This premium shall be computed such that it is impossible to benefit from massively creating short-term contracts. In that case, since the unit of gas fees is Gwei (ETH), we need to incorporate a swapping mechanism between ETH and $REF. A possible implementation would be to use Uniswap with some default base rate. The actual details of such a mechanism are left for further study.

F.1 The ReFees solidity project

A simple **alpha** version was developed in *Solidity* on the following GitHub repository (cf. appendix C for the 3 main contracts), for the sole purpose of demonstrating the validity of the *ReFees* project. Further tests of the code will be completed. For simplicity, the following assumptions were made:

1) All transactions are made in $REF.
2) The gas fees themselves are returned by the data oracle in $REF instead of ETH.

Furthermore, *OpenZeppelin* contracts [31] templates were used for the ERC-20 and ERC-721 tokens defined in IV-E.

F.2 The ReFees decentralised application

To increase user experience, the *ReFees* system will be powered up by a decentralised application. Its purpose is threefold:

1) A platform to create and deploy smart contracts on one of the supporting Layer 2 solutions. Two creation modes will be offered:
   a) Basic: a simple approach with predefined smart contract variables for non-experienced users
   b) Professional: this mode leverages the full power and flexibility of the system by allowing any values of the smart contract variables to be chosen.
2) A NFT marketplace to trade the contract.
3) A personalised view for a user to display relevant information of his owned contracts. Indicating for examples the current yield for the provider, amount refunded for the client and any interesting analytic.

V. Applications

To study the system behaviour in practice, we used real-world block data to simulate the operations and returns of a subscription contract. The source code of the simulation script is available in the following GitHub repository. We assume that blocks are added approximately every 20 seconds to the Ethereum blockchain. Thus, there are roughly $b = \frac{24*3'600}{20} = 4'320$ blocks created every day, and a month represent $129'000$ blocks.

TABLE II
SMART CONTRACT VARIABLES

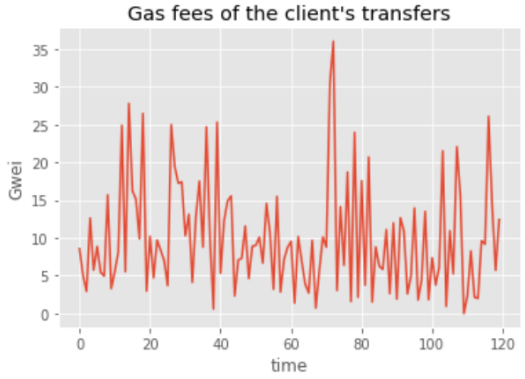
Name	Ref.	Value
Maturity	M	$60 \cdot b$ ($\sim 60\,days$)
Payment frequency	f	$10 \cdot b$ ($\sim 10\,days$)
Payment amount	x	4'200'000 Gwei
Initial reserve	r	30'000'000 Gwei
Max operations	n_{ops}	{"transfer": 20}
Mean factor	α	1

Using this information, we initialise a contract with the values highlighted in Table II.

For simplicity reasons, it is assumed that the client uses all the operations possible, 20 transfers every f blocks. We also assume that he uses the exact average gas price ($\alpha = 1$) for each of his operations. Moreover, the amount of gas used by a simple transfer can slightly vary because of how the Ethereum Virtual Machine works but, in our simulation, we use a constant value for this number, namely: $n_{gas} = 21'000$.

The parameters being set, we sample 20 every f block from real Ethereum data from block 4'000'000 to block $4'000'000 + M$ available in an external database[32]. Figure 7 contains a plot of the sampled data. Then, using this series of gas prices,

Fig. 7. Mean gas price in time



we simulate the behaviour of the contract and report the returns in Table III. We observe that the provider makes a profit using the system.

TABLE III
ReFees RETURNS WITHOUT OUTLIER.

Gas used by the client	0.02509 ETH
Total payments of the client	0.0252 ETH
Client return	0.02509-0.0252 = -0.00011 ETH
provider return	0.0252-0.02509 = 0.00011 ETH

Furthermore, we repeat this simulation on the same data after including an outlier, the gas price multiple times higher than average. This outlier also comes from real-world data. It simulates the fact that the client operates when the network is busy. We obtain the gas prices shown in figure 8. Similarly, realised returns for both client and providers are reported in Table IV.

Fig. 8. Predicted real gas price in time

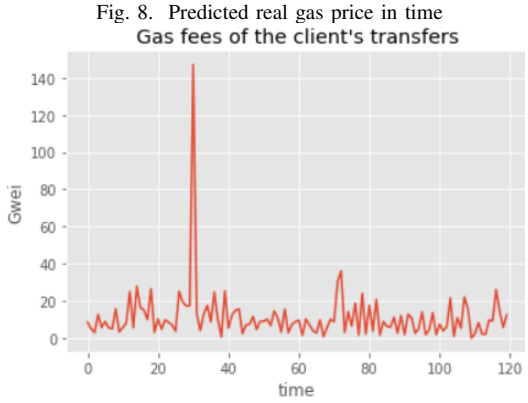


TABLE IV
ReFees RETURNS WITH AN OUTLIER.

Gas used by the client	0.02797 ETH
Total payments of the client	0.0252 ETH
Client return	0.02797-0.0252 = 0.00277 ETH
Provider return	0.02509-0.0252 = -0.00277 ETH

When analysing these numbers, it appears that the system is only beneficial for one of the two actors: there is a winner and a loser. However, this example was the simplest possible. It will be shown in section VI-E that it is possible to combine the system with $GAS tokens to make it profitable for both actors.

VI. MISCELLANEA AND CONCERNS

In this section, we address some of the challenges and future work for the system.

A. Generalisation through any subscription model

The main concern that is addressed with such smart contract is to offer a shared platform to allow any user to hedge their risk to gas volatility. This instantaneous access of liquidity to pay for gas fees, grant the possibility of protecting oneself against financial loss or diminished computing power on the Mainnet. With its price valuation depending on complex relationships between underlying fees, market volatility of *ReFees* contracts and recurrent payment options, it can be roughly similar to a traditional option.

The platform gives users high flexibility, the upper limit of contract prospecting is undefined. Transaction fee covered, time limit, recurrent payment and the other parameters are to be totally chosen by both parties, allowing for a tailor-maid gas hedging solution. There are still natural exceptions to this versatility, no negative fees or time limit would be allowed for example. The authors intend no implication of the free market but dawn on potential bid-ask gap and fancy parameters settings. From the game theory perspective, such margin behaviours is said to be existing but are naturally evicted from the market. The irrelevancy and scarcity of such actors would be monitored but said to be negligible enough as a first approximation[33;34].

A dynamic transaction fee plasticity would also be possible. As a matter of fact, one sees the volatility evolving during the day with minimum and maximum throughout the day. A gross conjecture stating on a daily partner of gas volatility based on empirical observations leads to assert variation in volatility between 60 and 65 percent[9]. Knowing more in detail this particular seasonal time series model, a smart contract allowing for its gas fee coverage to spread over its lifetime would avoid minute scale contract existence.

B. Internal gas fee usage

An in-depth study of the total number of transactions occurring in the creation of the smart contract or in the publication on the Mainnet within one iteration. It said that every iteration would have a similar amount of transactions[4]. However, as the smart contract would be using a Layer 2 solution, those are in a first approximation considered as negligeable.

There is a clear goal from the authors to lower down a maximum the gas consumption needed for the client and provider. Besides the pure code optimisation, the smart contract will be hosted on various Layer 2 solution platforms (Optimism, Arbitrum, Polygon, ...). This previous regard will permit most of the transactions to be settled down off chain. When one keeps in mind that the most important steps are to check wherever users can create a sufficient pool at the opening and register their new balances when the contract is resolved. In addition to this, it opens up the project to a maximum of potential users that are disseminated already along with those platforms.

To elaborate more in details of such platform utilisation, one can imagine hosting the recurrent payment made during the contract lifetime on a payment channel. This kind of state channel would be a fit for this utilisation as it only requires one parameter to modify the state of the channel. In the meantime, to ensure the overall security of the data associated with the current balance of each pool would be preserved on a side-chain with periodic synchronisation on the Mainnet.

C. External gas fee volatility induced

It has to be highlighted that the previous described smart contract is not and shall not be considered as a Layer 2 solution. As explained earlier such consideration would mean for the project to be a viable Layer 1 scalability increase actor[19]. Nevertheless as seen previously, the utilisation of the smart contract would engage the user in the consumption of gas. This utilisation will further induce the actual gas valuation market and potentially increase its volatility, even if the smart contract is optimised to use low gas. In reality, the project is not meant to reduce gas consumption but to simply offer either a speculation tool to the provider or a hedging tool to the client. Because one intraday cryptos trader would like to avoid its activity depending on gas market.

As mentioned in the following section, it is not a unilateral advantages relationship between the client and provider, as it may greatly induce Layer 2 operation arbitrage. One could use a mixture of solutions to optimise gas consumption of its underlying asset. The authors are currently working on a generative tool to allow the provider to undergo this optimisation process. This particular point stands as a future service allowed through the platform.

D. Resources consumption paradox

This project would also lead to an implicit usage reduction and an increase of the resources. Indeed, one of the forecasted practice attributed to providers is their retention or greediness on executing gas costing transactions depending on their profit and loss state at the time of usage. This can be summed in two particular situations:

The first one is the increase in user gas consumption even at a high volatility peaks. In fact, having your gas fee hedged during high gas price will allow you to undergo any usual process unaffected by the current gas price, as there is no need of protecting your capital.

In contrast, in times of low gas prices, clients may be found to be the "loser of the game" and will retain their operations until a positive returns on their hedging is foreseen. This second scenario is less prone to be a probable behaviour from users, as their need to ensure their transactions may be more important.

From such scheme speculations, the external gas fee induction would be strengthened by gas hedged clients.

E. Combination of ReFees with $GAS Token

In the future, *ReFees* could be combined with $GAS tokens. As shown in the example of section V, the system benefits only one of the users, either the client or the provider, but never both of them. Moreover, the yield of the provider directly depends on the presence of outliers in the gas price. Here is where $GAS tokens intervene.

Gas tokens would be a way for the provider to hedge the risk induced by gas volatility cleverly. An additional feature to our system would allow the contract to store, mint tokens, gas when its price is low and release, burn tokens, it when the gas price is high in order to obtain a rebate on the client's gas fees. These operations, storing and releasing, would be triggered by the provider. This way, if the provider has some knowledge about how gas prices behave, he could increase his profit. On the other hand, no action is required to be performed by the client. The provider hedges the risk for the client.

In practice, one can use the functionalities proposed by the GST2 contract (cf. source code) which contains minting and burning methods. Further investigation on the feasibility of the combination of both systems *ReFees*+Gas tokens, are foreseen. With a high confidence in the present research results.

Using the same contract as in section V, it is possible to simulate the yields produced by the combination of the system with $GAS tokens. The rebate obtained by using $GAS tokens is considered as another source of yield. More precisely, it amounts to $yield = -20000 \cdot gas_{low} + (15000 - 5000) \cdot gas_{high}$ as shown in section II. Since the process can be repeated multiple times by the provider, this yield is multiplied by the new variable $n_{hedgingComponents}$ set to 10 for this example.

TABLE V
[*ReFees* - $GAS TOKEN] RETURNS WITHOUT OUTLIER

Gas used by the client	0.02509 ETH
Total payments of the client	0.0252 ETH
Client return	0.02509-0.0252 = -0.00011 ETH
Provider return	-0.00117 ETH

Again, the first simulation runs on the data without spikes, seen in Fig. 8. Table V contains the result of the experimentation.

This experiment is then repeated on the data including the outlier, seen in 8. The returns obtained can be found in Table VI.

TABLE VI
[*ReFees* - $GAS TOKEN] RETURNS WITH AN OUTLIER

Gas used by the client	0.02797 ETH
Total payments of the client	0.0252 ETH
Client return	0.02797-0.0252 = 0.00277 ETH
Provider return	0.01074

The major difference with section V is that the second simulation ends up in a win-win state where both of the users benefit from the presence of the outlier. The first simulation benefits none of the users. However, one should keep in mind that the loss faced by the provider is now due to the non-used $GAS tokens. Therefore, if the provider had some knowledge on how to properly use $GAS tokens, and optimise the parameter $n_{hedging-components}$, he could also generate a profit in the first situation. Moreover, in this example, it is assumed that the non-used $GAS tokens are lost. But one can imagine that the provider keeps them for future contracts, which could result in a positive yield as well.

F. Ethereum 2.0

As previously stated, the Mainnet congestion decrease is the main motivation for lowering gas fees of Ethereum. The Serenity (Ethereum 2.0) upgrade would allow an increase in the amount of transactions per second and would be a start in the resolution of congestion stymie[22]. Undeniably, the proof of stake model used in Serenity reduces greatly the energy/power-intensive problem of the proof of work used in the current consensus mechanism[23]. Sharding increases the total number of transactions on the network as different validators are designated with a unique validation on disparate shards.

The high adaptability of *ReFees* parameters settings would allow hedging gas consumption even if the gas valuation gets a deep down correction of its quotation. Naturally its utilisation is to be considered higher before the final Serenity upgrade with a decreasing number of casual users throughout the stabilisation and lowering of gas costs of Ethereum network. Nevertheless, the authors are confident in such contract potential after the upgrade, but the user profile would maybe switch from casual hedger to structure or institutional actors that has a need in ensuring their recurring use of the Mainnet. Such example would be a country legislative adopting an e-voting system based on the Ethereum network, while taking in charge of the gas cost associated with every participant. The institution would surely reduce their gas cost by using a voting channel, a state channel dedicated to the voting application, together with the optimisation of transactions' execution depending on gas volatility. Furthermore, they would probably adopt a long-term hedging strategy to ensure the voting system stability over a period. Conversely, it would give enough time for the institution to transfer their voting system to new technology, in case in the future it is not reliable enough or a newcomer has greater potential.

G. Towards complete decentralisation

Similar to state channel solutions, the identity of the client and provider would have to be known throughout the whole contract period. This allows the external oracle to fetch the correct gas consumption balance on the Mainnet and to withdraw from the right pool the sufficient amount of liquidity. This flaws in privacy and decentralisation are the main areas of concern undergoing research from the authors. Although, the smart contract pools available to the user would be auto-regulated with no exterior decision made by a central, authoritative group.

The oracle from which the smart contract state would depend are to be directly connected to the Ethereum Blockchain data querying solution such as Etherscan. An investigation on the previously proposed solution shows that it is decentralised and does not do any external action except questioning the latest block on its transactions. This means that, from this particular aspect, a decentralisation of *ReFees* is ensured.

VII. CONCLUSION

ReFees is a highly flexible subscription model that allows a client to hedge Ethereum gas price volatility in a simple way and a provider to speculate on the Gas prices fluctuation in time. When the gas price is low, *ReFees*'s contract can be considered as insurance against the rise of this price. Moreover, when cleverly combined with other existing solutions, the system can end up in a win-win situation where both the client and provider respectively save and earn money. Anyone can create a subscription contract and trade its ownerships, which implies a competitive market. Holding some $REF tokens allows one to have voting power on the future of the system and also to earn some dividends produced by ownership transactions and contract completion. The future improvements are including better decentralisation, optimising internal gas fee usage, and a combination with gas tokens.

APPENDIX A
EXISTING LAYER 2 SOLUTIONS [35;36;37]

A. Sidechains

Ethereum network is one of the most used blockchains on the market. Nevertheless, many blockchains exist and many more are to come. With every blockchain come blockchain-specific limitations, one may want to reduce its exposure to limitations by diversifying its blockchain networks. As highlighted previously, a daily Mainnet interactions DeFi protocol unveils high operating costs to the end-user. Then it could be favourable to hold low transactions on a cheaper and less secure blockchain while the account balance and high transactions are saved on the Ethereum blockchain. This other blockchain would be called in this case a Sidechain to Ethereum blockchain [16;38].

As the name indicates, Sidechains are independent Ethereum-compatible blockchain that runs alongside to Ethereum Mainnet. Sidechains can employ different ledger parameters, security mechanisms and consensus models to process transactions. Their frameworks are chosen carefully to efficiently exercise activities in relation to the main networks it operates in. A Sidechain can be designed to operate integrally with any blockchain. In other words, any form of data from the blockchain is cross-chain transferable. Furthermore, this paralleling between siblings chains permits a multi-layer communication channels with delocalisation of the assets [39;40].

As described above, having this EVM compatible chain allows increasing the scalability of the Layer 1. The Sidechain smart contracts will manage infrastructure and distributed Sidechains statuses, while the Mainnet smart contracts manage validation proof and account states. One has to note that many Sidechains can be connected to the Mainnet.

Any decentralised application that is constrained by low security exposure but high transaction throughput would optimise its operation network by accumulating Sidechains.

It is connected to a Mainnet by a two-way bridge [41], which consists of five simple steps :

- Mainnet funds are locked;
- Funds are unlocked on the Sidechain;
- Transaction occurs with those funds;
- When finished funds on Sidechain are burnt;
- When burnt signal is received, Mainnet funds are unlocked.

This makes each Sidechain self-contained, thus making it impossible for a waterfall attack to occur.

Can it be really considered as a Layer 2 solution since it does not rely on the security of the main chain? This can be considered as the main drawback compared to its main chain. Nevertheless, if one ensure the same consensus models as the Mainnet, with similar security then the whole system could be considered as stable. Previous examples of fraud committed by a quorum validators have been seen. Showing that the decentralisation brought by the Sidechains could be harmful [42].

To guarantee the two-way bridge connection, users can be prevented from withdrawing their funds. This liquidity fastening is a must to ensure the cross-chain protocol to run. However, in terms of consumer security, it is considered at risk.

On the contrary, this permits a pre-synthetic Sharding. For deeper explanation please refer to Ref. [19], but in a few words it does improve operational speed and increase the capacity of the network by extending the Mainnet with Sidechains, the load data on the data is thus distributed [43].

Those siblings could also permit a pre-use of more *sustainable* and innovative consensus models, thus earlier shifting the environmental impact within the Mainnet consensus.

For Ethereum specific Sidechains, they are based on the Ethereum Virtual Machine means that the compatibility is ensured [21].

The Ethereum scalability improvement is mainly based on Sidechains for the moment. The main utilisation for Sidechains is application-specific transactions such as NFT art, DAO voting and stable transactions, including micro-transactions [44].

Sidechains that could operate with multiple blockchains at the same time are currently under development. The main principle would be to allow for a multiple-way bridge. This siblings interoperability would allow for faster and cheaper transactions, as the network load would be spread across every chain.

B. State channels

All applications do not need similar security standards. Imagine a DeFi service needs to make micro-payments, having those latter all recorded on the Mainnet would be gas expensive. Instead, the service could incentivise the user to lock a sufficient amount of funds in a pool that will later serve to pay the user service utilisation. This utilisation could be recorded off-chain and the reported final value recorded on a smart contract on the Ethereum blockchain that will latter transfer the necessary funds to the service holder.

To better understand the state channels Layer 2 solution, the definition of each word in the solution names help greatly.

A State is computationally defined as the condition of the variable linked to. While a channel is a communication vector employed between users interested in the mentioned variable. From those definitions, one understands state channels as a place for transactions, interactions and records that would later be published into the Mainnet. The channel is only accessible by authorised parties and should be a safe place [45].

A smart contract between two or more parties is settling on the blockchain. This contract is dynamically interactive with the two users input. For example, receiving input x and y will change the smart contract to 0. While if the inputs were x and z, the state will change to 1. For the state smart contract to work, users would have to lock ETH into the smart contract to act as liquidity pool. The blockchain state has multi-signature lock to ensure the security of capitals, the complete consensus has to be reached [46].

Only the users invited in the channel have access to those off-chain data and the interactions are instantaneous. Considering

this last point, every message in the channel is digitally signed by the user that authorised it. No cheating and early leaving would affect the user pool. The second user can publish the state channel of the latest signed results.

As you are not recording those off-chain data on the Mainnet, interactions have low fees, except for the final state. This last fee is, however, negligible in comparison to the computation that it would have been needed. The Mainnet is less congested with data[47].

It is also possible for the entering users to set up end-time conditions, this gives higher flexibility to the users.

The limitation of channels makes it impossible for the users to send funds off-chain to non-participants. For simple recurring payments, the amount of capital would be known beforehand but in the case of more complex state transactions, this amount could become very large. Not talking about margin protection like smart contracts.

The current state of the channel should be allowed to be published on the blockchain at any time. Meaning that at least one user has to publish this final state to protect each party's interests until the channel is totally closed. It has to be noted that the off-chain transaction is meaningful as long as the Mainnet is working.

This solution gets rid of the initial goal of the blockchain to act as a fully private solution. The multi-signature procedure allows each user to know the other channel user identity.

State channels are underemployed compared to their potential. Indeed, the judging smart contract set-up at the very beginning is the promise for each party to be partially treated. Meaning that this prior setting is of high importance, but may be a heavy step for potential users.

This latter judge mechanism must be flawless, give interesting properties and the initial constraints must be known. Such application with well-known constraints would be low-level service, payment channels, direct contracts and unsecured transactions (i.e. the score of a soccer game).

Actual research are on the standardisation of a judge contract and user-friendly mechanism.

C. Rollups

Rollups are yet another type of scaling solution. The core idea is to execute transactions outside Layer 1 and store the resulting transaction data as proof back on Layer 1. Correct execution on Layer 2 is ensured by a smart contract in Layer 1 using the transaction data. Since computations are moved off-chain (e.g. on Layer 2), transactions can be processed at a much higher rate. On Layer 1, they are committed in batches using data compression mechanisms to reduce both cost and network effects on Ethereum. Hence, Rollups can be viewed as a combination between state channels and side chains since they can create general-purpose applications through the EVM compatibility while still fully making use of the security mechanisms deployed on Ethereum. To incentivise users to execute and verify transactions correctly, Rollup protocols require staking a bond as collateral. If a fraudulent action is detected, the system slashes the bonds of the users involved.

We can further distinguish between different types of Rollups when it comes to proofs and security models.

C.1 Optimistic Rollups

Optimistic Rollups optimistically assume that transactions are valid by default. In other words, the system doesn't check the transactions. If they indeed are valid, no additional work needs to be done. Otherwise, users can report invalid transactions to the system which then enters a dispute resolution mode. In this state, the Rollup will execute a fraud-proof where the suspicious transactions are verified on the main Ethereum chain. If the transaction was indeed invalid, the state is reverted and the party that submitted it is punished : their staked bond is slashed. It is also worth mentioning that users who report transactions also need to stake a bond to avoid unnecessary fraud-proof computation. For the system to work correctly, the dispute resolution mode should be in an exceptional state.

C.2 Zero-Knowledge (ZK) Rollups

ZK-Rollups leverage a cryptographic instrument called SNARK (a type of zero-knowledge proof) to produce validity proofs. More specifically, to emit a valid transaction in the system, the user has to provide a validity proof. In other words, invalid transactions are rejected right away. It also allows the smart contract of the ZK-Rollup to maintain the state of transactions on Layer 2 only using these proofs instead of the whole transaction data, reducing both storage requirement and latency.

C.3 Validium

As a final note, we briefly describe *Validium*. It is extremely similar to a ZK-Rollup, with the exception that ZK-Rollup data availability is on-chain, but *Validium* data availability is off-chain[48]. *Validium* can achieve far higher throughput as a result of this, and its operator can deny any user the opportunity to move their funds without ZK-Rollup's data availability guarantees.

D. Plasma

Plasma is a Layer 2 scaling solution focused on Ethereum's growth[49]. After state channels, it is expected to be the second fully deployed scaling solution on the Ethereum Mainnet[50]. It is a platform that enables the establishment of child blockchains that rely on the Ethereum main chain for trust and arbitration. Child chains in Plasma can be developed to fulfil the needs of unique use cases, including those that are currently not possible on Ethereum. Plasma is better suited to decentralised apps that demand users to pay high transaction fees. However, the Plasma protocol is still in its early stages of development. The testing revealed a high throughput of up to 5,000 transactions per second, according to Ethereum specialists who participated. As a result, increasing the number of projects on the Ethereum Platform will not affect network congestion or transaction delays.

APPENDIX B
EXAMPLES OF PROTOCOLS

A. Optimism and Arbitrum

In terms of Optimistic Rollups, *Optimism*[51] and *Arbitrum*[52] are considered to be the most popular options which already started their deployment on the Mainnet. If they share most of their functionalities, several differences are worth mentioning :

- To tackle the long withdrawal problem, Optimism decided to partner up with MakerDao. Several tokens such as DAI can benefit from the collaboration.
- Arbitrum developed an algorithm to narrow down the scope of disputes so that only a few transactions need to be reevaluated on Layer 1 when a fraud is reported.
- Transaction ordering is handled differently in both systems. Arbirtrum uses a centralised sequencer while Optimism uses third parties for a given amount of time via an auction mechanism.

B. ZkSync and Aztec

ZkSync[53] and Aztec are two Layer 2 networks that provide Ethereum scaling solutions and are already active on the Ethereum Mainnet. To guarantee both privacy and scalability, they both use the Zero-Knowledge Rollup described in previous sections. Furthermore, while ZkSync seeks to be a generic protocol that could increase Ethereum's transaction throughput, Aztec prioritises privacy while retaining some scalability[49]. The Rollup, according to the researchers, can handle 300 transactions per second while also allowing protected ERC-20 token transactions and private interactions with decentralised finance protocols. As part of a pooling contract, users would be able to trade on Uniswap and other exchanges.

C. Polygon - Matic

Polygon (with the ticker MATIC)[42] is a multi-chained system, a framework and a protocol all in one. It connects Ethereum-compatible blockchain networks, and as with other protocols we've talked about, it was created to address the present Ethereum network's scalability difficulties[54]. Polygon uses side chains added to the primary platform cost-effectively and efficiently. Polygon's multi-chain network enables blockchain networks to communicate with one another outside of Ethereum's core chain while maintaining Ethereum's liquidity, security, and interoperability.

The Polygon ecosystem's core resource is MATIC, Polygon's token. In addition to being an asset, it is primarily utilised for staking tokens (proof-of-stake algorithm) to protect the Polygon network. The MATIC token has a maximum supply of 10 billion units, of which over 67% are now in use. With a price of $1.4 per token and a market capitalisation of more than $9 billion, it is currently among the top 25 cryptocurrencies in the world.

D. StarkEx

StarkEx[55] is a Validium, which as mentioned in previous sections is a Layer-2 scaling solution that enforces the validity of all transactions using zero-knowledge proofs while keeping data availability off-chain. This prevents the Validium's funds from being stolen because any movement of value from a user's account must be authorised by that user. For applications such as DeFi and gaming, StarkEx uses STARK technology to power scalable self-custodial transactions (trading and payments). StarkEx allows an application to scale up quickly and reduce transaction expenses. Moreover, StarkEx has a Verifier Contract Upgrade mechanism in place that allows the operator to immediately add a new item to the chain of verified contracts. You can't remove user signature checks, for example, because this would invalidate the old reasoning. Rather, it enables the inclusion of extra constraints[56].

E. Hybrid Solutions

Hybrid solutions incorporate the greatest features of different Layer 2 technologies while also allowing for customisable trade-offs. One example of this is the Celer Network, which is utilised for off-chain scaling, cost-effective and quick dApp creation, blockchain-based games, and liquidity[17].

APPENDIX C
SOLIDITY CODE

A. ReFees Factory

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.2;

import "./tokens/RefeesToken.sol";
import "./tokens/RefeesClient.sol";
import "./tokens/RefeesProvider.sol";
import "./helpers/RefeesEvents.sol";
import "./helpers/Scheduler.sol";
import "./RefeesSubscription.sol";

contract RefeesFactory is RefeesEvents {

    // system tokens
    RefeesToken immutable refT;
    RefeesClient immutable refC;
    RefeesProvider immutable refP;
    // Array of subscriptions and schedulers
    RefeesSubscription[] public subscriptions;
    Scheduler[] public schedulers;

    constructor() {
        // initilise system tokens
        refT = new RefeesToken();
        refC = new RefeesClient();
        refP = new RefeesProvider();
    }

    // creates new subscription and store it an
        array
    function createSubscription(
        uint256 _maturity,
        uint256 _frequency,
        uint256 _paymentAmount,
        uint256 _initialReserve
    ) external {
        RefeesSubscription refS = new
            RefeesSubscription(
            _maturity,
```

```
            _frequency,
            _paymentAmount,
            _initialReserve,
            refT,
            refC,
            refP
        );
        // create scheduler
        scheduler = new Scheduler(_frequency,refS);
        refS.setScheduler(address(scheduler));
        // store the subscription in the
            subscriptions array and similarly for
            the scheduler
        subscriptions.push(refS);
        schedulers.push(scheduler);
        emit SubscriptionCreated(block.timestamp,
            address(refS), address(scheduler));
    }
}
```

B. ReFees Subscription

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.2;

import './tokens/RefeesToken.sol';
import './tokens/RefeesClient.sol';
import './tokens/RefeesProvider.sol';
import './helpers/RefeesPool.sol';
import './helpers/RefeesOracle.sol';
import './helpers/RefeesEvents.sol';
import "@openzeppelin/contracts/utils/Counters.sol";

contract RefeesSubscription is RefeesEvents{
    using Counters for Counters.Counter;

    // contract variables
    uint256 immutable public maturity;
    uint256 immutable public frequency;
    uint256 immutable public paymentAmount;
    uint256 immutable public
        totalExpectedPaymentAmount;
    uint256 immutable public initialReserve;
    // NFT Tokens
    RefeesClient immutable refC;
    RefeesProvider immutable refP;
    // native Token
    RefeesToken immutable refT;
    // tokenId associated the the NFT
    uint256 public immutable clientId;
    uint256 public immutable providerId;
    // data oracle
    RefeesOracle immutable oracle = new RefeesOracle
        ();
    // mutable variables (state of contract)
    bool private wasStarted = false;
    bool private started = false;
    Counters.Counter private paymentCounter;
    RefeesPool private immutable pool;
    // scheduler
    address private schedulerAddr;

    constructor(uint256 _maturity, uint256
        _frequency, uint256 _paymentAmount,uint256
        _initialReserve, RefeesToken _refT,
        RefeesClient _refC, RefeesProvider _refP) {
        require(_maturity % _frequency == 0); //
            requirement of Refees
        // contract variables
        maturity  = _maturity;
        frequency = _frequency;
        paymentAmount  = _paymentAmount;
        initialReserve = _initialReserve;
        totalExpectedPaymentAmount = _frequency *
            _paymentAmount;
        // system tokens
        refT = _refT;
        refC = _refC;
        refP = _refP;
        // mint provider and client NFTs
        clientId   = refC.safeMint(msg.sender);
        providerId = refP.safeMint(msg.sender);
        // state variables
        pool    = new RefeesPool(_initialReserve);
        // initial deposit by the provider
        refT.transfer(msg.sender, initialReserve);
        pool.deposit(initialReserve);

    }

    function setScheduler(address _schedulerAddr)
        public returns (bool) {
        require(!started); // can only set scheduler
            before the starting the subscription
        // scheduler address
        schedulerAddr = _schedulerAddr;
        return true;
    }

    function trigger() external {
        // only scheduler is allowed to trigger the
            refund
        require(msg.sender == schedulerAddr);
        if (paymentCounter.current() == 0) {
            startSubscription();
        }
        else {
        if (paymentCounter.current() < maturity /
            frequency) {
            clientPayment();
            paymentCounter.increment();
        }
        else { // arrive at maturity
            stopSubscription();
        }
        // call the data oracle to compute the
            amount of gas to be refunded
        uint256 gasFees = oracle.getGasFee(refC.
            ownerOf(clientId));
        refund(gasFees);
        emit Refund(block.timestamp, refC.ownerOf(
            clientId), gasFees);
        }

    }

    function startSubscription() private returns (
        bool) {
        require(!started && !wasStarted);
        require(refC.ownerOf(clientId) != refP.
            ownerOf(providerId)); // requirement of
            Refees
        require(refT.allowance(refC.ownerOf(clientId
            ), address(this)) >=
            totalExpectedPaymentAmount);
        started = true;
        wasStarted = true;
        return true;
    }

    function stopSubscription() private returns (
        bool) {
        require(started && wasStarted);
        started = false;
        address addrProvider   = refP.ownerOf(
            providerId);
        uint256 residualAmount = pool.empty();
```

```
        return refT.transfer(addrProvider,
            residualAmount);
    }

    function clientPayment() private returns (bool)
        {
        address addrClient = refC.ownerOf(clientId);
        pool.deposit(paymentAmount);
        bool state = refT.transfer(addrClient,
            paymentAmount);
        emit ClientPayment(block.timestamp,
            addrClient, paymentAmount);
        return state;
    }

    function refund(uint256 gasAmount) public
        returns (bool) {
        require(started);
        // get client address
        address addrClient = refC.ownerOf(clientId);
        pool.withdraw(gasAmount);
        return refT.transfer(addrClient, gasAmount);
    }
}
```

C. ReFees Scheduler

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.2;

// KeeperCompatible.sol imports the functions from
    both ./KeeperBase.sol and
// ./interfaces/KeeperCompatibleInterface.sol
import "@chainlink/contracts/src/v0.7/
    KeeperCompatible.sol";
import '../RefeesSubscription.sol';

contract Scheduler is KeeperCompatibleInterface {

    // Use an interval in seconds and a timestamp to
        slow execution of Upkeep
    uint256 public immutable frequency;
    RefeesSubscription immutable subscription;
    uint256 public lastTimeStamp;

    constructor(uint256 _frequency,
        RefeesSubscription _subscription) {
      frequency = _frequency;
      subscription = _subscription;
      lastTimeStamp = block.timestamp;
    }

    function checkUpkeep(bytes calldata) external
        override returns (bool upkeepNeeded, bytes
        memory ) {
        upkeepNeeded = (block.timestamp -
            lastTimeStamp) > frequency;
        return (upkeepNeeded, 0);
        // We don't use the checkData in this
            example. The checkData is defined when
            the Upkeep was registered.
    }

    function performUpkeep(bytes calldata) external
        override {
        lastTimeStamp = block.timestamp;
        // trigger refund
        subscription.trigger();
    }
}
```

References

[1] "Ethereum Whitepaper." [Online]. Available: https://ethereum.org

[2] "Turing completeness," Nov. 2021, page Version ID: 1054201708. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Turing_completeness&oldid=1054201708

[3] L. Soler, "Introduction: The Solidity of Scientific Achievements: Structure of the Problem, Difficulties, Philosophical Implications," in *Characterizing the Robustness of Science: After the Practice Turn in Philosophy of Science*, ser. Boston Studies in the Philosophy of Science, L. Soler, E. Trizio, T. Nickles, and W. Wimsatt, Eds. Dordrecht: Springer Netherlands, 2012, pp. 1–60. [Online]. Available: https://doi.org/10.1007/978-94-007-2759-5_1

[4] D. G. Wood, "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER," p. 41.

[5] S. Werner, P. Pritz, and D. Perez, *Step on the Gas? A Better Approach for Recommending the Ethereum Gas Price*, Mar. 2020.

[6] S. Bouraga, "An Evaluation of Gas Consumption Prediction on Ethereum based on Transaction History Summarization," Sep. 2020.

[7] "Ethereum Average Gas Price." [Online]. Available: https://ycharts.com/indicators/ethereum_average_gas_price

[8] C. Li, S. Nie, Y. Cao, Y. Yu, and Z. Hu, "Dynamic Gas Estimation of Loops Using Machine Learning," Nov. 2020, pp. 428–441.

[9] D. Carl and C. Ewerhart, "Ethereum Gas Price Statistics," *SSRN Electronic Journal*, 2020. [Online]. Available: https://www.ssrn.com/abstract=3754217

[10] F. Schär, "Decentralized Finance: On Blockchain- and Smart Contract-Based Financial Markets." [Online]. Available: https://research.stlouisfed.org/publications/review/2021/02/05/decentralized-finance-on-blockchain-and-smart-contract-based-financial-markets

[11] H. Adams, D. Robinson, and N. Zinsmeister, "Uniswap v2 Core," p. 10.

[12] "OpenSea Developer Documentation." [Online]. Available: https://docs.opensea.io/

[13] C. R. Harvey, A. Ramachandran, and J. Santoro, "DeFi and the Future of Finance," Social Science Research Network, Rochester, NY, SSRN Scholarly Paper ID 3711777, Apr. 2021. [Online]. Available: https://papers.ssrn.com/abstract=3711777

[14] "GasToken.io - Cheaper Ethereum Transactions, Today." [Online]. Available: https://gastoken.io/

[15] A. Zarir, G. Oliva, Z. Jiang, and A. E. Hassan, "Developing Cost-Effective Blockchain-Powered Applications: A Case Study of the Gas Usage of Smart Contract Transactions in the Ethereum Blockchain Platform," *ACM Transactions on Software Engineering and Methodology*, Oct. 2020.

[16] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais, "SoK: Layer-Two Blockchain Protocols," in *Financial Cryptography and Data Security*, ser. Lecture Notes in Computer Science, J. Bonneau and N. Heninger, Eds. Cham: Springer International Publishing, 2020, pp. 201–226.

[17] "Layer 2 Rollups." [Online]. Available: https://ethereum.org

[18] 0xjim, "Layer 2 Won't Save Ethereum," Nov. 2021. [Online]. Available: https://medium.com/coinmonks/layer-2-wont-save-ethereum-a52aa2bd719b

[19] A. Hafid, A. S. Hafid, and M. Samih, "Scaling Blockchains: A Comprehensive Survey," *IEEE Access*, vol. 8, pp. 125 244–125 262, 2020, conference Name: IEEE Access.

[20] "The Blockchain Trilemma." [Online]. Available: https://www.seba.swiss/research/the-blockchain-trilemma

[21] "Comparing Layer-2 Ethereum Scaling Solutions," Mar. 2021. [Online]. Available: https://academy.ivanontech.com/blog/comparing-layer-2-ethereum-scaling-solutions

[22] D. Sel, K. Zhang, and H.-a. Jacobsen, "Towards Solving the Data Availability Problem for Sharded Ethereum," Dec. 2018, pp. 25–30.

[23] C. Nguyen, H. Dinh Thai, D. Nguyen, D. Niyato, H. Nguyen, and E. Dutkiewicz, "Proof-of-Stake Consensus Mechanisms for Future Blockchain Networks: Fundamentals, Applications and Opportunities," *IEEE Access*, vol. PP, pp. 1–1, Jun. 2019.

[24] D. Liebau and P. Schueffel, "Crypto-Currencies and ICOs: Are They Scams? An Empirical Study," *SSRN Electronic Journal*, Jan. 2019.

[25] Ethereum Foundation, "Decentralized Oracles: Reliably Triggering Smart Contracts using Decentralized Computation and TEEs," 2018. [Online]. Available: https://www.youtube.com/watch?v=yHh-j_NdRZw

[26] "Etherscan APIs." [Online]. Available: https://etherscan.io/apis

[27] "The Graph." [Online]. Available: https://thegraph.com

[28] "Chainlink Achieves Major Scalability Upgrade With Launch of OCR," Feb. 2021. [Online]. Available: https://blog.chain.link/off-chain-reporting-live-on-mainnet/

[29] "Introduction to Chainlink Keepers | Chainlink Documentation." [Online]. Available: https://docs.chain.link/docs/chainlink-keepers/introduction/

[30] "Making Keeper-compatible Contracts | Chainlink Documentation." [Online]. Available: https://docs.chain.link/docs/chainlink-keepers/compatible-contracts/

[31] "Contracts - OpenZeppelin Docs." [Online]. Available: https://docs.openzeppelin.com/contracts/4.x

[32] "Ethereum Block data." [Online]. Available: https://kaggle.com/muhammedabdulazeem/ethereum-block-data

[33] T. Akbar, "Research Paper- Economics and Game theory," *Paper*, Jan. 2017.

[34] Z. Liu, N. C. Luong, W. Wang, D. Niyato, P. Wang, Y.-C. Liang, and D. I. Kim, "A Survey on Applications of Game Theory in Blockchain," *arXiv:1902.10865 [cs]*, Mar. 2019, arXiv: 1902.10865. [Online]. Available: http://arxiv.org/abs/1902.10865

[35] "Ethereum - Scaling." [Online]. Available: https://ethereum.org

[36] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais, "SoK: Layer-Two Blockchain Protocols," Tech. Rep. 360, 2019. [Online]. Available: http://eprint.iacr.org/2019/360

[37] M. Jourenko, M. Larangeira, K. Kurazumi, and K. Tanaka, *SoK: A Taxonomy for Layer-2 Scalability Related Protocols for Cryptocurrencies*, Apr. 2019.

[38] "Sidechains." [Online]. Available: https://ethereum.org

[39] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille, "Enabling Blockchain Innovations with Pegged Sidechains," p. 25.

[40] "Layer 2 Rollups." [Online]. Available: https://ethereum.org

[41] "Welcome." [Online]. Available: https://docs.tokenbridge.net/

[42] "Polygon | Ethereum's Internet of Blockchains." [Online]. Available: https://polygon.technology/

[43] "Difference Between SideChains and State Channels," Dec. 2019. [Online]. Available: https://simpleaswater.com/difference-between-sidechains-and-state-channels/

[44] "What is Polygon Network and MATIC?" Mar. 2021. [Online]. Available: https://academy.ivanontech.com/blog/what-is-polygon-network-and-matic

[45] "State Channel." [Online]. Available: https://decryptionary.com/dictionary/state-channel/

[46] "What are multi-signature transactions?" [Online]. Available: https://bitcoin.stackexchange.com/questions/3718/what-are-multi-signature-transactions

[47] L. Moore, "Rollup Rollup! Top Layer 2 Compared — Arbitrum vs Optimism vs Polygon," Nov. 2021. [Online]. Available: https://medium.com/general_knowledge/rollup-rollup-top-layer-2-compared-arbitrum-vs-optimism-vs-polygon-4a469389faef

[48] A. Gluchowski, "zkRollup vs. Validium," Feb. 2021. [Online]. Available: https://blog.matter-labs.io/zkrollup-vs-validium-starkex-5614e38bc263

[49] "Aztec launches private smart contracts as Ethereum rollup." [Online]. Available: https://cointelegraph.com/news/aztec-launches-private-smart-contracts-as-ethereum-rollup

[50] T. K. Sharma, "A Complete Guide To Ethereum Plasma," Jun. 2020. [Online]. Available: https://www.blockchain-council.org/blockchain/a-complete-guide-to-ethereum-plasma/

[51] "Optimism - Ethereum at Lower Costs & Lightning Speed." [Online]. Available: https://www.optimism.io/

[52] "Arbitrum Bridge." [Online]. Available: https://bridge.arbitrum.io/

[53] https://matter labs.io, "zkSync — Rely on math, not

validators." [Online]. Available: https://zksync.io
[54] "Polygon (MATIC): A blockchain with Indian founders and a great potential," Oct. 2021. [Online]. Available: https://www.thenewsminute.com/article/polygon-matic-blockchain-indian-founders-and-great-potential-156293
[55] "StarkEx." [Online]. Available: https://docs.starkware.co/starkex-v3/
[56] "Introduction." [Online]. Available: https://docs.starkware.co/starkex-v3/