A Decentralized, Scalable Read-Write Solution for Blockchains

The APIS Foundation
media@theapis.xyz

October 5, 2020

Abstract

We propose APIS, a middleware protocol for the functioning of a decentralized read-write protocol, allowing for the mainstream growth of a fully decentralized finance and decentralized web architecture. Current data index and querying solutions are either centrally architected, reliant on a party who at all times is subject to numerous performance and regulatory risks, or logically decentralized, thus requiring an in-depth knowledge of the public blockchain protocol layer and obscuring the majority of global developers from interacting with public blockchain infrastructure. APIS is architecturally decentralized, in that the providers of the endpointed data can be any actor joins the APIS network, but logically centralized, in that client-side developers will be able to reference common, frequented API formatting structures, namely RESTful and GraphQL endpoints, to query datasets generated by public blockchains.

Tables of Contents

1 Background

1.1 Background

Public blockchain architectures have allowed for the next iteration of the internet: the decentralization of digital applications. Decentralized, digital applications are currently categorized in two subsequent groups: decentralized finance and the decentralized web.

Decentralized finance has achieved significantly more adoption than decentralized web, for a variety of factors, although the most significant is the difference in computational overhead required to create a contemporary user experience: the computation of money or money-like instruments is primarily dependent on floating-point arithmetic, computationally light to conduct in contrast to the algorithms required for the par-functioning of web applications reliant on 'news feeds', 'matching algorithms', and 'product recommendations'. Consequently, decentralized finance can achieve private scalability at the same throughput as international payment networks, such as SWIFT [1], today and promises to achieve private scalability on par with modern domestic payment networks, such as VISA, in the near future. Decentralized finance will continue to subsume additional financial applications beyond payments, namely investing, trading, lending, and savings. These four use cases have begun to show meaningful growth as decentralized finance continue to scale.

The decentralized web is further from mainstream adoption: we approximate that popular decentralized web protocols today are three to five years behind their decentralized finance counterparts in terms of privacy and performance. While decentralized finance can leverage zero-knowledge-proofs to achieve private scalability, and the rate of zero-knowledge proof improvement has been exponential [2], the decentralized web is dependent on technology still further in its infancy, namely secure multi-party-computation (sMPC) and fully homomorphic encryption (FHE) [3]. As these cryptography technologies continue to scale in the same way that zero-knowledge proofs have over the last five years, the decentralized web will be able to reach the same adoption as decentralized finance.

However, so long as there exists any degree of centralization of decentralized financial and web products, the entire system is at risk. API's Law states that: "A product is only as decentralized as its most centralized component." We present APIS in order to realize a vision of decentralized finance and web, without any platform risk that would be subjugated to local regulations, specifically those in regimes that currently control the global regime and so are incentivized to mitigate disruption. The most centralized component of the decentralized finance and decentralized web stacks are the data index and querying layer, a mission-critical component without which decentralized applications would not be able to achieve mainstream adoption. Thus, APIS allows for decentralized finance and web products to permeate the world while maintaining the security properties of a fully decentralized web architecture.

1.2 Introduction

Current public blockchains (such as Ethereum and Filecoin) were launched as maximally decentralized, but quickly became re-centralized around elements that solved user pain points. The most prominent centralization success has been centralized exchanges, although the exponential adoption of decentralized exchanges renders that this centralized feature of public blockchain systems (centralized exchanges) will soon become decentralized [4].

Thus, after exchanges, we turn towards what will now become the largest, most mission-critical, yet still centralized feature of the public blockchain design space: the *database layer*, herein also referred to as the *query layer*. The query layer is mission-critical for all decentralized applications.

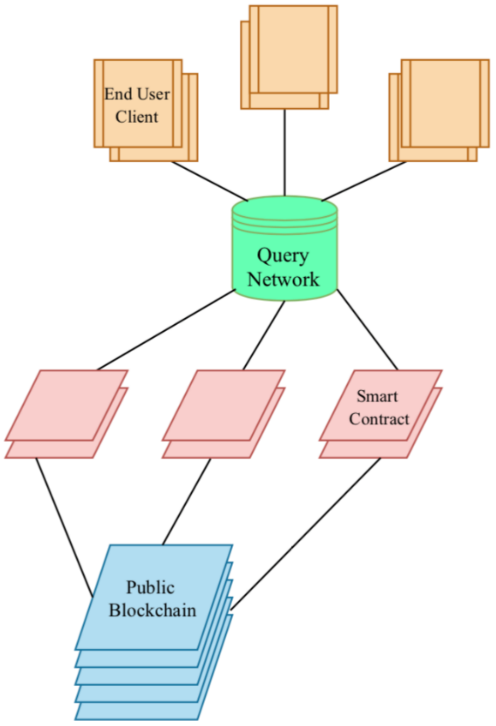


Figure 1.2.1: Current Decentralized Application Architecture.

End-user client applications typically interact with a public blockchain through the following process: a transaction is signed by the user's private key [5] via a client-side light client [6] and subsequently propagated by a full node, whose uptime is promised by the application itself or a third-party service provider, to additional full nodes, including validators of the public blockchain, who then compute the result of that transaction and package both the transaction and its results into their ensuing block, should the fee paid to the validator incentivize them to choose that transaction over additional transactions. The process of sending a transaction is herein referred to as writing to the blockchain. The most significant barrier of writing to the blockchain is the optimization of the transaction fee paid to the network [7], as variant fees for transaction inclusion exist for frequently used blockchains, which manifests itself in either failed transactions [8] or over-paid transactions [9], both of which perpetuate the poor user experiences that public blockchains are known for.

Across all public blockchain virtual machines of mainstream usage, writing to the blockchain generates events and logs [10] that exist as a result of transactions and the interactions with smart contracts; events and logs serve the primary purpose of the generation of return values for decentralized application user interfaces. However, the management of events and logs and their resultant states emitted from smart contracts is a time-intensive, computationally heavy process, for which the overwhelming a majority of developers do not possess resources to execute. Smart contract and decentralized application developers are best suited optimizing for the secure execution of their contracts, as well as finding product-market fit for those products. The market for specialized index and query services has grown at an exponential pace [11], but the market leaders are centralized service providers, whose operators and shareholders are subject to the regulation of the states in which they operate. These index and query service leaders are incentivized to form monopolies to extract excess fees and profits, ultimately passing the cost to users and developers at large. Decentralized applications reliant on centralized service providers are subject to gatekeeper and platform risk that can create an unfavorable developer experience. For any consumer application, all it requires is one poor user experience for a user to never trust a product again [12]. All of a developer's work can be destroyed overnight.

APIS provides a protocol for the decentralization of index and query services for reading data from and writing data to the blockchain, which will include gas optimization indices to help developers and users write to the blockchain as efficiently as possible, thus ensuring a fully decentralized architecture for the decentralized web, its builders, and its users.

2 Architecture

APIS achieves its decentralized query and index architecture through the utilization of a publish-subscribe message propagation protocol [13], on which is overlayed an incentive-driven, decentralized organization enabled by the Ethereum public blockchain and a customized layer-two system whose characteristics exhibit that of an optimistic rollup [14].

2.1 Message Propagation Protocol

The APIS Network is composed of two actors, both of which are open and configurable by any party: APIS Gateways and APIS Nodes. Queries requested by decentralized applications and their developers are responded to by APIS Gateways, while APIS Nodes index and manage databases pertaining to one or a group of smart contracts, such that APIS Nodes can respond to APIS Gateway requests with an average response time of 400ms ($\epsilon(t) < 400$ms), under the condition that the requested endpoint from the APIS Gateway is already maintained by an APIS Node.

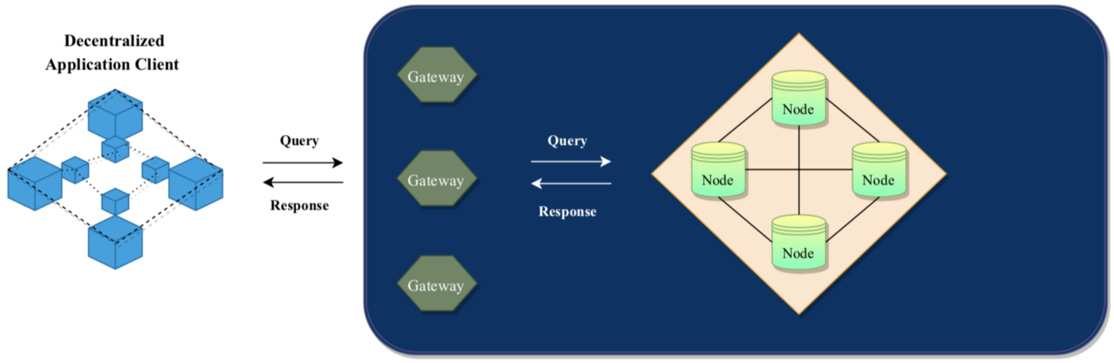


Figure 2.1.1: APIS Network Message Propagation Protocol.

Communication between Gateways and Nodes is maintained through a brokerless, libp2p publish-subscribe messaging protocol, wherein Nodes subscribe to messages from a Gateway, which they can receive directly from the Gateway or from additional Nodes who also have also subscribed to that Gateway's messaging propagation, although Nodes should assume counterparty Nodes are adversarial at all times, per any well-designed crypto-economic system.

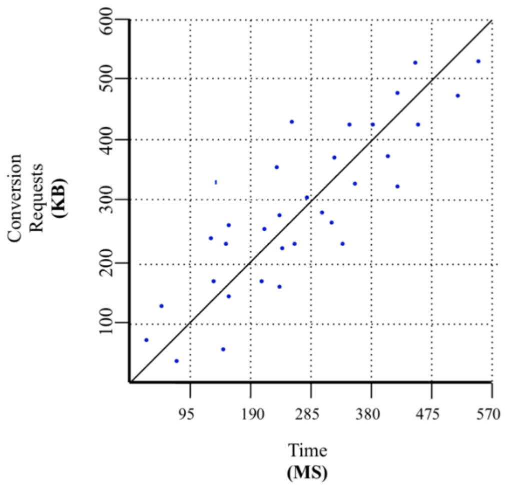
Gateways maintains a registry of the subscribing Nodes from which they request endpoints based on the <ID> of the endpoint, as Nodes will only index a select group of IDs. IDs are representational hashes of the datasets supported by the

Network; only the hash of the ID is maintained on-chain in the Node and Gateway State Tries, described further in *2.6 Optimistic Rollup Contract*. It is up to each Gateway to determine how to filter messages from Nodes for each endpoint response, although we recommend a first-in, first-out approach, wherein subsequent messages, should they differ from the first sent message, can provide the Gateway with evidence warranting of initiating a challenge, described further in *2.5 Dispute Resolution Contract Factory*.

2.2 Message Formatting Overview

Queries are sent by client-side applications directly to Gateways and returned by Gateways in industry standard GraphQL and REST formatting, thus allowing for the adoption of APIS by a global developer pool and consequently expanding the developer market of public blockchains. GraphQL and RESTful differentiate themselves in how they manage and return queries: GraphQL provides a single endpoint for the requested data, while REST provides multiple endpoints, which renders REST less efficient but also more customizable. REST has achieved more significant mainstream adoption than GraphQL due to its first-mover advantage and consequent well-known integrations; however, GraphQL can tailor its query responses to exactly what the query is requesting, thus ensuring that there is no overfetching of data and that the application only receives the data that it needs with one, unilateral mode of communication. We believe that GraphQL will ultimately overtake REST, but to encourage adoption by as many engineers as possible, APIS Gateways will support both API formats for the foreseeable future.

To maintain optimal communication between APIS Nodes and Gateways, APIS Nodes will only support RESTful requests at launch, as RESTful requests can be packaged by APIS Gateways into GraphQL format, through the APIS RG Converter, a proprietary but open-sourced technology that builds on the well-known stitching [15] and prefixing [16] techniques implemented in past conversation schemes. Thus, Gateways allows for the APIS network to support both types of queries by clients, while mitigating the operational intensity required to run an APIS Node. The APIS RG Converter can transform RESTful endpoint into a GraphQL endpoint (and vice-versa) in under 100ms ($\epsilon < 100$ms), with an approximate success time of 95ms.



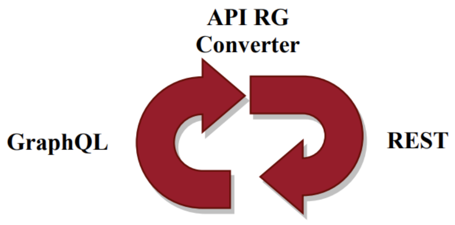


Figure 2.2.1: APIS RG Conversion Time across internal practical tests.

95ms, when added to the 400ms roundtrip estimation of message communication, renders a query to be returned in 500ms or less ($\epsilon < 500$ms). The performance of the network will scale as the network scales, as Gateways and Nodes alike will compete with return queries under the fastest possible conditions, such as to win third-party client developer business. This is a long-term economic benefit of a protocol over a centralized aggregrator, who can afford to innovate at a slower rate due to monopolistic market practices. However, in the interim, it is estimated that sub-500ms return times are adequate for decentralized finance and web user experiences.

2.3 APIS Core Contracts

APIS Gateways and Nodes act under rational economic incentives, enlisting their services for fees in an environment assumed to be perpetually adversarial. The state of the APIS Network is maintained by the APIS smart contract and layer-two rollup architecture, deployed on the Ethereum public blockchain.

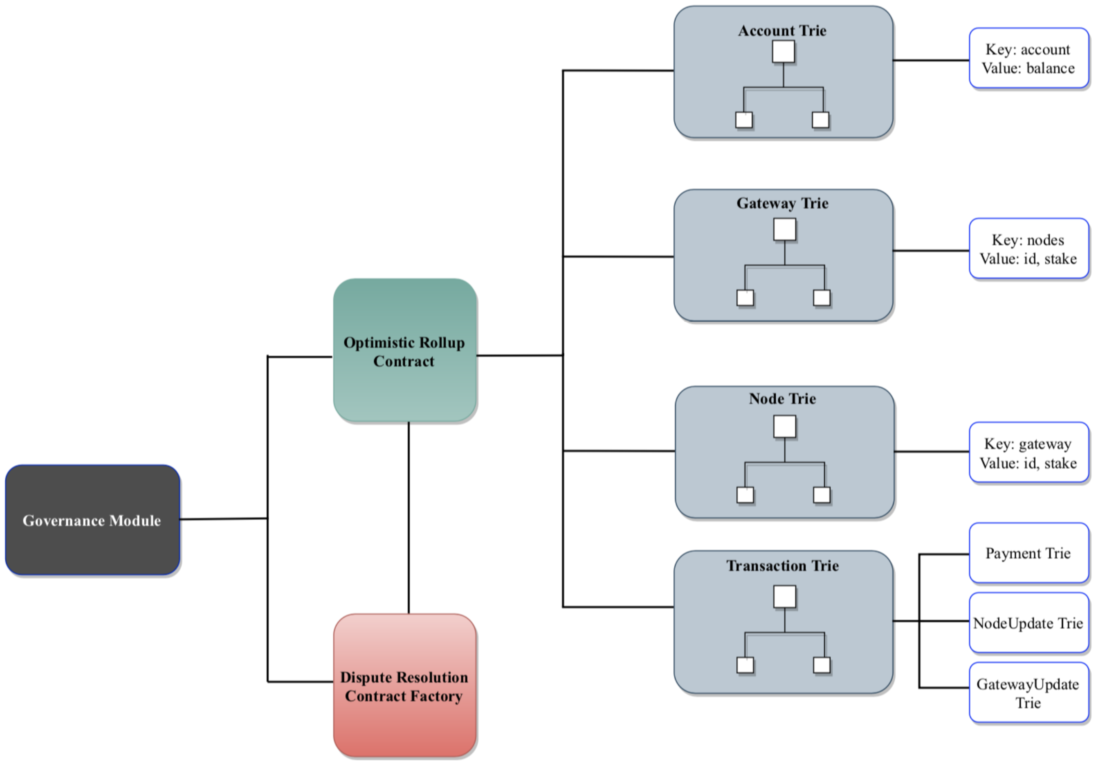


Figure 2.3.1: APIS Network Ethereum Layer-One and Layer-Two Architecture.

The three primary contracts deployed on the Ethereum public blockchain are the governance contract, the dispute resolution factory contract, and the optimistic rollup contract. The governance contract allows for the configuration and upgrade of the rest of the architecture; the dispute resolution contract factory ensures that APIS Gateways and Nodes are incentivizes to only complete accurate responses; the optimistic rollup contract allows the architecture to scale significantly beyond the throughput of the current Ethereum public blockchain. Throughout the entire design, the concept of superfluidity is emphasized with regards to the API token, such that the holder of the token can participate in governance, dispute resolution, and optimistic rollup validation simultaneously, maximizing the potential revenue generation to API token holders.

2.4 Governance Contract (GC)

APIS is upgraded and configured by the decentralized group of API token holders, thus driving the Governance Contract (GC) to be the most important object in the design. The GC is a customized fork from the open-sourced Compound Governance Contract, which has proven to be the most comprehensive, well-audited governance module in the Ethereum ecosystem, as exhibited by its redeployment by other notable layer-two protocols, such as yearn.finance [17]. The purpose of the GC is to allow API token holders to upgrade the APIS Dispute Resolution Factory Contract and the APIS Optimistic Rollup Contract, as well as to implement additional smart contracts into the APIS protocol. API token holders can vote directly or delegate their votes to trusted counterparties; all votes require that the tokens remain locked in the GC for time = 1.504 days (10,000 Ethereum mainnet blocks) after the quorum is reached and the vote is finalized, to disincentivize malicious parties from purchasing API tokens to pass APIS Improvement Proposals (APiiPs) that negatively affect the APIS Network. The quorum required to approve an APiiP will be 25% of the network, although we hope that future APiiPs will vote to raise the quorum requirement as the project becomes increasingly decentralized through the API Liquidity Mining Program, as illustrated further in *3.3 Community Ownership*.

2.5 Dispute Resolution Contract (DRC) Factory

The APIS Dispute Resolution Contract (DRC) Factory was initially designed as a function within the Governance Contract, but was later separated into its own contract to ensure complete modularity, allowing for upgrades to the DRC when a less subjective resolution mechanism is able to be deployed, which should be the case as zero-knowledge-proofs continue to scale. The purpose of the DRC Factory is to allow API holders to vote specifically on events that have happened off-chain, namely whether a Node or Gateway provided a fraudulent or misrepresented endpoint, either to another Gateway or end-user client. API tokens staked in the GC can be used to stake in the DRC (and in the *Optimistic*

Rollup Contract, as described in 2.6) simultaneously, although requiring the signing of an additional transaction on the Ethereum mainnet.

The DRC Factory utilizes customized, stake-driven voting functions, namely a <challenge> function and a <defend> function through which an API token holder signals their vote with the staking of their API tokens. The Factory merges these functions with simple opcodes, namely CREATE2, to allow anyone to deploy a Dispute Resolution Sub-contract (DRS), whose address and state is stored in the DRC Factory as a key-value mapping; a DRS can either have one of two states in the DRC Factory contract: <True> or <False>. In light of due process, all DRS are defaulted as <False>.

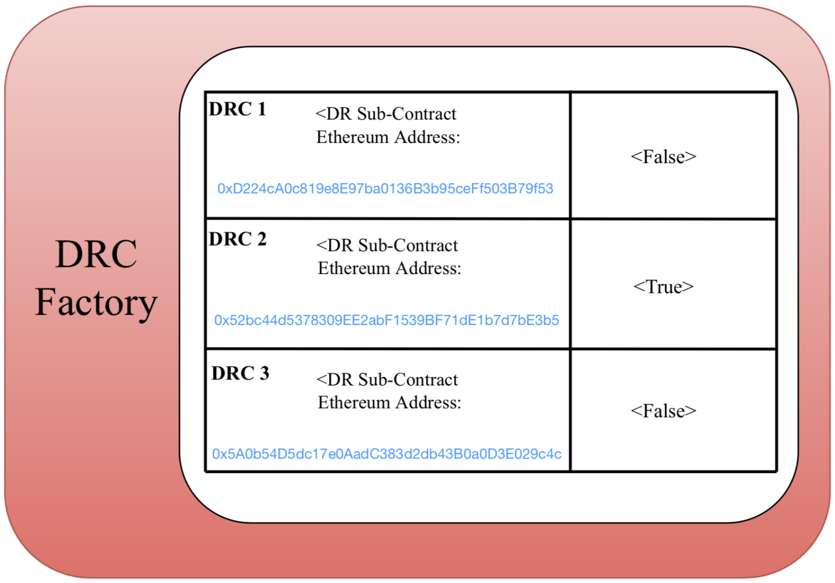

DRC Factory
DRC 1
<DR Sub-Contract Ethereum Address:
0xD224cA0c819e8E97ba0136B3b95ceFf503B79f53
<False>
DRC 2
<DR Sub-Contract Ethereum Address:
0x52bc44d5378309EE2abF1539BF71dE1b7d7bE3b5
<True>
DRC 3
<DR Sub-Contract Ethereum Address:
0x5A0b54D5dc17e0AadC383d2db43B0a0D3E029c4c
<False>

Figure 2.5.1: DRC Factory Simplified Architecture.

All DRSs have 7.52 days (50,000 Ethereum mainnet blocks) to resolve itself, after which, if still unresolved, the entire system forks into two separate universes, in a design very similar to Augur's stale-state solution [18]. At that point, API token holders must decide which fork of the network to recognize as canonical, based on what they believe the proper Dispute Resolution Outcome should be.

A DRS's only purpose is to resolve whether a transaction from the Optimistic Rollup Contract (ORC) as either <True> or <False>. To conduct a knowledgeable vote, API holders must be aware of the endpoint that was sent either from the Gateway to the User or the Node to the Gateway. Thus, every Payment Transaction in the Optimistic Rollup Contract has the structure: <Client Account Address, Client Account Signature, Server Account Address, Server Account Signature, Transaction Amount, Endpoint Hash, Nonce>, where the Client is the Payer (either an End-user Client or Gateway) and the Server is the Payee (either a Gateway or Node).

```
{
   struct PaymentTransaction {
       address client;
       signature client;
       address server;
       signature server;
       uint amount;
       string[] endpoint;
       uint nonce;
      }
   }
}
```

Figure 2.5.2: Customized Payment Transaction Structure

The Endpoint Hash provides the canonical truth of what the delivered endpoint, such that voters of a DRS can simply check the result of their Endpoint Hash of the requested endpoint against the Endpoint Hash in the transaction that has been challenged. If the voter's Endpoint Hash matches that of the transaction, the voter will vote that the transaction was <True> and consequently that DRS is <False>; if the voter's Endpoint Hash does not match that of the transaction, the voter will vote that the transaction was <False> and consequently that the DRS is <True>.

In order to generate the Endpoint Hash, the voter must have access to the Endpoint that was sent to the requester. To achieve this, all APIS Nodes and Gateways default to store transmitted Endpoints for 7.52 days (50,000 Ethereum mainnet blocks), after which Nodes and Gateways can prune the full Endpoint data from their storage. Full Endpoint data (the Endpoint Hash represents this data) consists of: <ID>, a hash representation of the dataset from which the API was called; <DataType>, with two possible returns of GraphQL or REST; <MessageInputs>, the specific variables that were requested in the API call; and <MessageOutputs>, the specific responses to the variables that were requested, with both <MessageInputs> and <MessageOuputs> strored in equal size arrays, such that they can be counter-checked. Because of the blockchain's inability to objectively verify whether a <MessageOutput> was the correct response to the <MessageInput>, *dispute resolution is a subjective process*, but with clear on-chain references to use as canons. If a DRS is settled as <True>, the server party (APIS Node or Gateway) that had been paid for the API request will have their API stake slashed, at a ratio of:

$$k * \sum_{i=1}^{n} x(\frac{z}{y}),$$

where x is their API tokens at stake, y is the time that their API stake has been locked in the *Optimistic Rollup Contract (see 2.6 below)*, z is the number of offences that have been generated since that stake was deposited (starts to 1, increases by 1 every false offense, which will be recognizable both in the Node and Gateway State Tries in the ORC), and k = 0.326. Payments are not rolled back in the case of proven fraud by a Node or Gateway; instead, the slashed stake is distributed at the ratio of 10-90 between the client that was frauded and the group of parties that voted correctly, proportionate to the size of their vote. It requires a minimum stake of 1,000 API tokens to act as a Node or Gateway, although we believe that the most used Nodes and Gateways will have staked a significantly higher amount of API, as the more API at stake, the more trustworthy they will be, the more usage they will obtain.

If a challenge itself is fraudulent, and not the API request, the challenger will have their stake slashed: it requires a stake of 500 API tokens to initiate a challenge, to mitigate spam. While this may incur friction for people wishing to challenge, obtaining 500 API tokens is feasible due to its exchange liquidity, and the upside from a correct challenge is designed to exceed the gas costs by at least ten-fold. If a challenger is wrong, their stake is distributed at a ratio of 10/90 to the party who enacted the correct API request and the group of parties that voted correctly, proportionate to the size of their vote.

In order to challenge a challenge, a group of parties must stake at least one-and-a-half times the amount of funds currently staked on the other side of the vote, within 18.1 hours of the last challenge (5,000 Ethereum mainnet blocks). If the DRS is not resolved after 7.52 days (50,000 Ethereum mainnet blocks), the system forks, with individual users deciding which fork to support. The goal of forking is to encourage faster resolution, as it is likely the market will converge upon one of the forks at a least a 90-10 ratio, as is common with most public blockchain protocol forks.

2.6 Optimistic Rollup Contract

The Optimistic Rollup Contract (ORC) holds the Node State, the Gateway State, the Transaction State, and the Account State. The Account State is most frequent across all account-based optimistic rollups, what will continue to be the status quo in transaction execution for both mainchains and rollups, despite the recent popularity in UTXO-based optimistic rollups [19]. All states are represented on the Ethereum mainchain as hashes of their corresponding Patricia Merkle Trie [20]. The Node and Gateway Tries are of depth 14, allowing for $2^{14}(1.6 * 10^{4})$ Nodes and Gateways, with the ability for the depth size to be increased in the future via an APiiP. The Account State is of depth 22, allowing $2^{22}\left(4.2 * 10^{6}\right)$ accounts (which sums to the Users, Gateways, and Nodes), with the ability for depth size to be increased in the future via an APiiP. The Transaction State Trie is of depth 40, allowing for $2^{40}\left(1.1 * 10^{12}\right)$ transactions, with transactions that have been processed and not challenged either in the DRC Factor or the ORC for 50,000 blocks (7.52 days) pruned and removed from the Transaction Trie. Transactions pruned from Trie can be maintained by Nodes and Gateways for longer if desired, although there is no reason to hold them via the incentives of the APIS protocol.

The Transaction Trie has three sub-Tries, of which referenced prior has only been the Payment Trie. In addition to the *Payment Trie*, a *Node Update Trie* and *Gateway Update Trie* have been implemented to allow for the customized updating of the Node State and Gateway State Tries. Both Node and Gateway Transactions contains identical structures, with the primary purpose of allowing Nodes and Gateways to increase or decrease their API at stake, which, to be valid, must already have been deposited into the ORC via a standard smart contract deposit transaction on the Ethereum mainnet: <Node/Gateway Address, Node/Gateway Signature, Node/Gateway Staked Deposit (negative integers are allowed, with a check that the Account Stake as listed in the Node/Gateway State Trie is greater than or equal to the withdrawal amount), ID hash (the hash of all IDs currently supported by that Node or Gateway), Nonce>.

```
{
   struct NodeUpdateTransaction {
       address node;
       signature node;
       int change;
       string identity;
       uint nonce;
      }
   }
}
```

Figure 2.6.1: Customized Node State Update Transaction Field.

The ORC also contains the Optimism Fraud Proof’s standard public functions, which allow for fraudulent state updates to be challenged and resolved without requiring interaction other than the initiated challenge itself [21]. It should be noted for emphasis that DRC and ORC fraud proofs are distinct processes (The DCR is highly interactive, due to the subjectivity over the truthfulness of the items being disputed, whereas the ORC will become highly objective overtime, due to the usage of the Ethereum mainchain as a data storage layer.). ORC Fraud Proofs rely on data availability, namely that all state updates have corresponding transactions readily available on the Ethereum mainnet, primarily stored as call data [22].

To ensure data availability while maintain scalability, a pruned-version of transaction information that is included in the Transaction Trie is stored as call data on the Ethereum mainchain: <TrieID (a reference to one of the three transaction tries), TransactionHash (a hash representation of the transaction)>, where the Trie ID is 4 bytes and the Transaction Hash is 32 bytes, such that each transaction only uses 36 bytes of call data. A standard Ethereum transaction is 247 bytes [23] and requires 21,000 gas because of its use of the Ethereum State Trie; APIS's optimistic rollup architecture has effectively batched transactions at a ratio of 125:1, thus allowing for 1,875 transactions per second, or 1,875 API calls per second, compared to the Ethereum mainnet's current throughput of 15 transactions per second.

If an ORC state update is found to be fraudulent, the validator's bonded APX tokens are slashed and distributed to the party that found and report the fraudulent block transition. Any party can submit an APIS ORC block to the blockchain so long as they have bonded 10, 000 API tokens bonded in the ORC contract. Like the DRC Factory, tokens staked in the APIS GC can be staked in the ORC as a validator, per the system's design requirement of superfluidity.

If our optimistic rollup design cannot scale to meet demand, we will transition to a zero-knowledge rollup design, leveraging either a modified version of zkSync or Validium, depending on scaling requirements required by user demand [24].

3 Applications

The Applications of the APIS network are twofold: to create better end-user experiences and to create better developer experiences. For end-users, APIS allows developers to seamless integrate features with guaranteed 100% uptime, censorship resistance, and near-zero platform risk. While the uptime and censorship resistance features of a decentralized network has been discussed thoroughly by the public blockchain community, we believe that platform risk remains under-discussed. Developers who utilize the APIS network for their decentralized applications will never be upcharged by a monopolistic, profit-seeking company, as protocols create markets, not monopolies [25]. The platform risk view can be further substantiated by Metamask's recent change in open-source licensing, leaving developers to question whether a Metamask integration will require fees in the future [26]. APIS already supports all queries and indexes of Ethereum through the aggregation of APIS's centralized competitors, and shortly thereafter will support IPFS and Filecoin due to its growing popularity. Additionally, any additional blockchains can be added to the APIS index and query protocol, should an ID for that blockchain be created by an APIS Gateway or Node.

3.1 IDs

Ethereum and IPFS both contain terabytes of data that could be indexed and queried, with their datasets growing daily. Thus, APIS introduces the concept of IDs, such that only relevant datasets to the APIS's clients are indexed by APIS Gateways and Nodes. IDs are standardized references to specific Ethereum smart contracts or IPFS hashes that developers wish to index. APIS Gateways and Nodes are incentivized to index all IDs that developers and users find relevant; thus, APIS already supports all notable decentralized finance IDs of notable volume, such that datasets from any automated market maker, decentralized exchange, on-chain lender, and on-chain aggregator can be queried.

IDs are to be requested and added through off-chain proposals on the APIS Governance Module, separate from the on-chain APIS Governance Contract. Once an ID is requested, Gateways and Nodes signal their support by conducting a Gateway or Node transaction in the ORC, adding the ID to their registry. The transaction acts of proof of their support of the ID, which allows for Gateways and Nodes to search for each other to offer more complete datasets, without needing to know each other prior. This allows for the network to become more decentralized, as Nodes and Gateway can employ a standard internal, private reputation system of their connections based on uptime and API tokens at stake, with initial communication driven by a desire to include additional IDs into their service offering. Additionally, the Dispute Resolution Contract (DRC) protocol references IDs in each dispute, such that API token holders can be fully knowledgeable of the matter of dispute.

3.2 API Token

As referenced prior, the API token is designed to be superfluid, utilized for on-chain governance, dispute resolution staking, and optimistic rollup validating simultaneously. Thus, those that own or earn API tokens maintain a voice over the entire APIS protocol, earning fees for their actions. Because a majority of tokens today are unmined, API tokens will be distributed via a usage mining program derived the usage of the protocol itself, namely through fees paid, as money is the most objective measure usage.

APIS users tend to be technically competent developers, who then can vote on APiiPs, resolve disputes, and deploy their own APIS Nodes or Gateways. It is imperative that all APIS actors who could harm the network have API at stake, due to the well-researched nothing-at-stake problem set [27].

The API token has pause function now to prevent hacking attack, as the system and community growth, the token governance shall be managed by multi sign address voted by community,which will include team member, Community and Anyone elected, Snapshot APIS DAO will allow users to insight The APIs to propose and vote for own agreed opinion, in which APIs token is the ballot ticket to vote

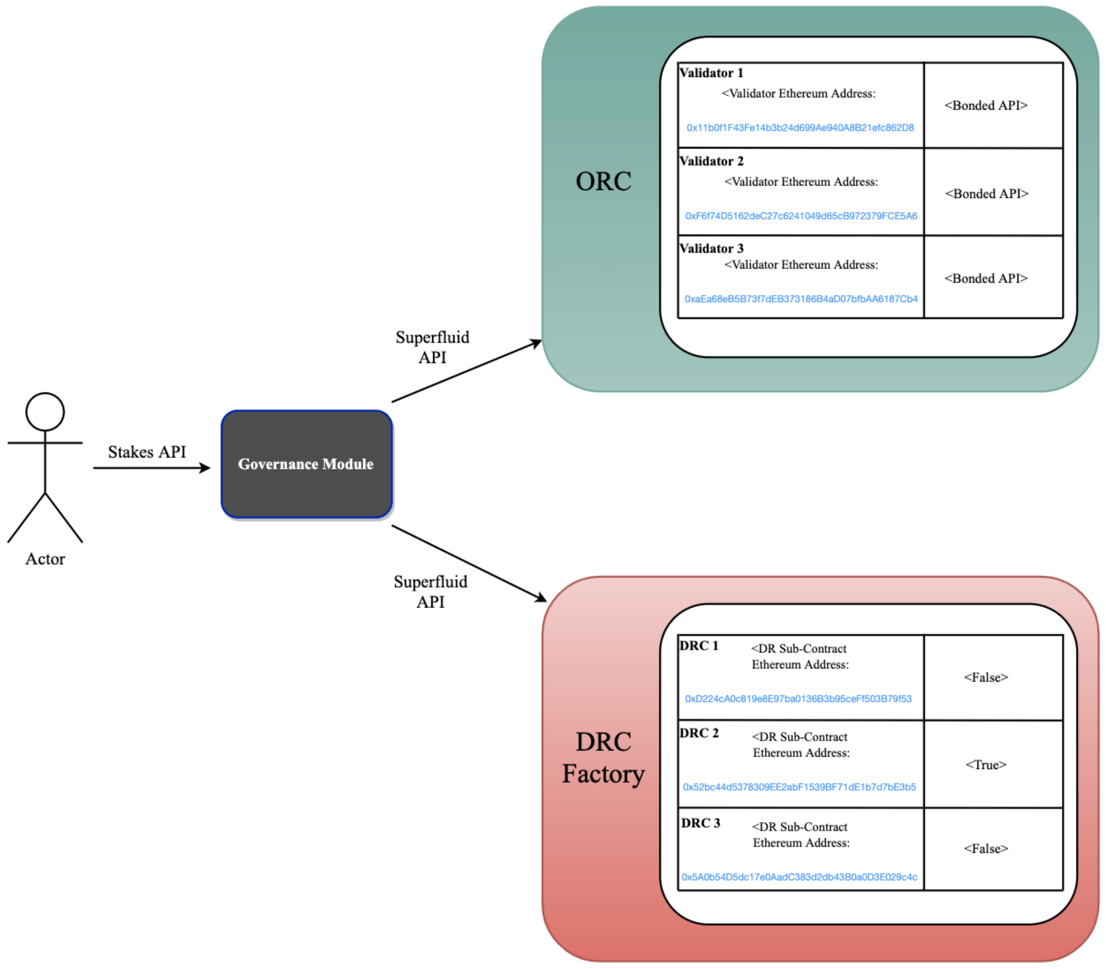

Actor
Stakes API
Governance Module
Superfluid API
Superfluid API
ORC
Validator 1
<Validator Ethereum Address:
0x11b0f1F43Fe14b3b24d699Ae940A8B21efc862D8
<Bonded API>
Validator 2
<Validator Ethereum Address:
0xF6f74D5162deC27c6241049d65cB972379FCE5A6
<Bonded API>
Validator 3
<Validator Ethereum Address:
0xaEa68eB5B73f7dEB373186B4aD07bfbAA6187Cb4
<Bonded API>
DRC Factory
DRC 1
<DR Sub-Contract Ethereum Address:
0xD224cA0c819e8E97ba0136B3b95ceFf503B79f53
<False>
DRC 2
<DR Sub-Contract Ethereum Address:
0x52bc44d5378309EE2abF1539BF71dE1b7d7bE3b5
<True>
DRC 3
<DR Sub-Contract Ethereum Address:
0x5A0b54D5dc17e0AadC383d2db43B0a0D3E029c4c
<False>

Figure 3.2.1: Superfluid API Composition.

3.3 Token Distribution, Community Ownership

In the ORC State Trie, the Liquidity Mining Account (LMA) will be publicly noted. The LMA will distribute API tokens once per week, similar to Synthetix's model. The Account is currently controlled via a nine-person multi-sig, overseen by the APSs Council, although the Council will eventually be removed and replace it with the APIS Governance Contract, such that API token holders can vote each week whether the supply is increased or not. This allows API holders to prevent future inflation, should they believe it is no longer needed, as market-blind liquidity mining, as most projects have implemented [28], has proven to be less optimal than market-aware liquidity mining, as exemplified by yearn.finance [29].

As referenced prior, API Liquidity Mining is achieved through usage, not the liquidity of the APIS platform itself. While many protocols have opted to incentivize the trading and lending of their token, the APIS network is best suited incentivizing API queries and API calls, rewarding those who use the protocol for its intended purpose, paying and receiving fees for the service that the network provides (demand side and supply side). Third-party client developers and APIS Nodes & Gateways will receive a monthly payout according the following function:

$$T * \sum(_{i=1}^{n} \frac{F}{TF)}$$

where F is fees paid by each third-party developer to each Node or Gateway, TF is the total amount of fees paid on the network, and T is the amount of tokens allocated to the program by the network for that month. We have considered taking the sum of the square root of fees paid to disincentive wash-trading and instead to encourage the usage by third-party developers of a wide variety of Gateways and Nodes, incentivizing maximal decentralization of the network. However, we will leave this decision to a community vote. T is currently a variable number of API per month, following an S-Curve that aligns with a two-year adoption cycle, and will be released online shortly, but may be altered based on the APIS community's vote at large once the governance module passes a proposal removing the power from the Council and placing it to the APIS community via an APiiP.

4 Discussion

In this paper, we have presented a scalable, decentralized protocol for the indexing and querying of datasets inherent to public blockchains. The protocol is based on knowledge of REST and GraphQL message formatting and optimization, consequently ensuring that developers can obtain data from

Ethereum, Filecoin, and all other relevant blockchains with uptime guarantees, hyper-competitive market pricing, and in formats that mainstream developers have vast experience integrating with already. The protocol is composed of two groups: Gateways and Nodes, although a Gateway can be a Node and vice-versa. The purpose of the dual infrastructure is to allow developers to select receive endpoints as quickly as possible, while ensuring that both the providers of endpoints are incentivized to act in the best interest of the APIS Network. The APIS Network is community governed, with governance primarily focused on the Dispute Resolution Contract, which has been influenced by the Augur project's approach to subjective oracle design, and the Optimistic Rollup Contract, which has been influenced by the work of Optimism. Lastly, to encourage maximal community ownership and corresponding decentralization, users of APIS will receive API tokens proportionate to their usage, with a system designed to mitigate wash-trading and encouraging further decentralization of Nodes and Gateways. The remaining of the paper is dedicated to an analysis and overview of REST and GraphQL APIs, namely how each standard functions and the tradeoffs therein.

5 Appendix: Analysis of REST versus GraphQL

5.1 REST History and Analysis

Representational State Transfer (REST) [30] is a software architectural style that defines much of how information is currently transferred between previous unconnected, distinct web applications. REST APIs now compose approximately 71% of the global API market share [31], foot-holding as the foundation upon which many of the world's most popular web services are serviced to consumers by developers.

Any web application that deploys a REST architecture for API interaction, and in doing so become known as RESTful, must comply with five guiding constraints. These constraints restrict the ways that a server can process and respond to client query requests, but, in doing so, provide websites a number of desirable properties with regards to efficiency, scalability, and simplicity. The constraints are as follows:

- **Client-Server Architecture:** The basic principle of client-server architecture is that there exists a clear separation between the data queried and the user-interface that the data provider typically utilizes to represent that data. To provide a concrete example, a Facebook API should not explicitly mirror Facebook's own front-end interface. Thus, third-party developers who query a Facebook API should not be forced to interact with, for example, a Facebook 'profile page' endpoint but rather information about a 'user,' which the third-party developer can then utilize in their web application's distinct user-interface.
- **Statelessness:** No data that has been queried should be stored on the server in-between requests, such that servicing any additional query possesses a marginal cost of zero to the data provider. As such, every request made by a client developer must contain all the information necessary for the server to service that request.
- **Cacheability:** Clients are able to cache responses. Caching means 'store for later,' which renders that the client can avoid asking a server for information if the client has already asked for it earlier. For example, if I request someone's Facebook profile through the Facebook API, I can *cache* that information so that I don't need to request it the next time I need it.
- **Layered Architecture:** A client should not ordinarily be able to tell whether the client is connected directly to the provider's end-server [32] or an intermediary server that exists between the end-server and the client. For example, if server-side web application utilizes a load balancer, a technology that allows developers to distribute requests across multiple servers such that no single server becomes inundated, the client requesting data from that web application should still continue fully performant communication with the layer of the server architecture providing the response, not requiring any additional code by the client or server architecture.
- **Uniform Interface:** A uniform interface is, per third-party developer experiences, the most significant element of REST, distinguishing it from competing architectural styles. A uniform interface decouples a web

application's interface from its implementation, meaning that regardless what a web application does, developers can interact with its API in the same way. Thus, developers can interact with all of their provided REST APIs in exactly the same way, even if the information they are retrieving is entirely distinct. REST achieves this highly desirable malleability due to four constraints:

- **Identification of Resources:** The REST style differs from preceding APIs because it is centered around *resources* as opposed to methods or procedures. These resources can effectively take any form (be it a static picture or a feed of real-time stock prices), but they usually represent entities from the business domain (customers, orders, prices, products, etc). Regardless of the data that the resource provides, the resource must be uniquely identifiable via a Uniform Resource Identifier (URI); notably, the resource's identifier must be stable even if the resource itself is changed. For example, if Spotify were to update the album art for a given song, a develop would still to find it in the same query via their API; nothing would change for the developer reliant on the API.
- **Manipulation of Resources through Representation:** A client not have direct access to a server-side database through the server's API. If a client wishes to manipulate information through the server's API, such as to change a user's profile picture, the client must grab a copy of that resource, manipulate it, send the updated representation to the server, and ask the server to update its underlying resource. The server can then decide if this is an appropriate change to make, thereby protecting the companies underlying data-store from malicious edits.
- **Self-Descriptive Messages:** The Self-Descriptive Message constraint requires a message (be it a request or a response) to include enough information for the receiver to understand it in isolation. We'll cover what this means in greater detail when describing how REST APIs work in practice below.
- **Hypermedia as the Engine of Application State:** Any RESTful application should be entirely navigable through links. Navigable

links has allowed end-user clients to navigate to any part of a website from a single 'base' URI, such as facebook.com, and, for developers requesting APIs, it means the developer doesn't need to look at specified API documentation to purvey how to obtain a desired resource.
    - **Code on Demand:** A server can extend the functionality of a client at runtime by sending code to the client that the client should execute (like JavaApplets or JavaScript). This property is similar to the cacheability of REST, highlighting the architecture's high performance.

5.2 REST APIs in Practice:

If a client-side developer wishes to interact with an API, the developer need to place a *request* that contains a certain amount of information:

- **The Endpoint:** The endpoint is the URL at which the developer is making a request, and will define the kinds of information that the develop can receive. The structure of an endpoint is as follows:
    - An endpoint always begins with a *root*, which typically represents the high-level URL that the developer is interacting with. For example, if a developer is interacting with Facebook's API (Facebook.com, not Facebook's subsidiary companies' APIs), the request will always start with https://graph.facebook.com.
    - From there, the client-side developer must specify the *path* by which to retrieve the desired information. As a metaphor, one can think of this sub-section of the endpoint as an automated answer machine, where the developer presses numbers to navigate through a menu to the intended destination. As an example, if a developer wanted to see who all of someone's Facebook friends, that path would be /user-id/friends. It's important to note that this looks exactly the same if the developer were utilizing Spotify's API to obtain the same information, going through links instead. To see who a user's friends are on Facebook via Spotify, the developer

would still need to click to the user's profile (user-id), and then click again to see the user's friends.

- **The Method:** The method defines the type of request that the client-side developer sends to the server, which are usually one of five types, and allow for all the basic CRUD (create, remove, update, delete) operations:
    - **GET:** The developer gets the resource from the server, i.e. get a user's Facebook friends.
    - **POST:** The developer creates a new resource on the server, i.e. share a photo to a user's Facebook profile.
    - **PUT:** The developer overwrites a resource on the server, i.e. adding changing a user's profile picture on Facebook.
    - **PATCH:** The developer updates parts of an existing resource in the server, i.e. changing just a user's description in the user's profile page. PUT and PATCH are fairly similar and choosing which one to use is case specific and a frequent topic of debate [33].
    - **DELETE:** The developer removes a resource from the server, i.e. removing a Facebook post.
- **The Headers:** Headers provide information to the client-side party about how information can be handled by the client-side application, namely the kinds of access an individual is allowed. For example, updating a user's Facebook profile through Facebook's API is allowed only by that user him or herself, thus rendering that the user of the client-side application must proves an access-token that proves the server has given them permission to make changes to that user's profile; that access-token would be included as a header, such that the client-side application can provide the desired actions to their end-user. An example list of headers can be found here [34].
- **The Data (or Body):** The Data (or Body) provides information that the client wishes to revert to the server and is only used to perform a create, update, or delete action. For example, if a user wants to change his or her Facebook profile picture on a client-application that utilizes Facebook's API, the message reverted to the Facebook server must include the image that needs to be uploaded as the replacement.

To put this all together, we construct an example query and response, shown below, illustrating a simple yet precise interaction with a REST API:

- **Request: api.example.com/GET/users/10**
- **Response:**

```
{
  “user”: {
        “id” : 10,
        “name” : “Satoshi Nakamoto”,
        “nickname” : “Seb”,
        “height” : 180,
        “Image_url” : “/images/10.jpg”
    }

}
```

Figure 5.2.1: REST API Response.

5.3 Strengths and Weaknesses of REST:

Now that we have illustrated what a REST API is, and how developers can interact with REST APIs, it is important to discuss the characteristics are that have made REST so successful, and what shortcomings are threatening its sustainability going forward.

- **Strengths:**
    - **Flexible across languages and frameworks:** One obvious reason why the REST API has been so successful is that it works across languages and frameworks. No matter how developers build their applications, a REST API will work for them.
    - **High interpretability:** The fact that REST APIs are URL-based, where each URL gives access to a certain amount of information, renders REST APIs incredibly simple to interpret. All a third-party developer requires is a URL, an understanding of what information they will find there, and an understanding of how they can interact with that information.

    - **Server-side Logic:** For a company building a REST API, its URL driven nature is extremely simple to work with. The server-side developers simply define a URL where they want certain information to be visible, write logic for how to handle each type of request, and the endpoint is created. However, as a company's API scales in usage, figuring out the optimal way to structure endpoints and the information they provide increases in complexity, a relative weakness of this architecture, transitioning to our next section.
- **Weaknesses:**
    - **Multiple Round Trip (Latency):** The client will often need to execute multiple trips to the server to fetch all of the information the client requires. Each endpoint specifies a fixed amount of information, and in many cases that information is only a subset of what a client requires to populate the client’s page. As exhibited in *Figure 5.2.1* above, what if the client wished to display the users image as well as their nickname in the client’s application? The client send a GET request for the users information, thus obtaining the location of the users image from the response, and then send another GET request for the image URL. Every time the client executes an extra request, another trip needs to be made back and forth to the server, which requires additional time and thus weakens user experience.
    - **Verbose:** When a client makes a request via a REST API, the developer will get back all of the information stored there, even if the developer doesn’t need all of it. For example, if a client is only interested in the name and age of a user on Facebook, when the client queries Facebook’s API, the client will get this information, but will also receive all of the other information associated with that user endpoint, such as the user’s profile picture, city of residence, email address, etc. This leads to the transfer of excess information, which causes the client to filter which parts of the response the client wishes to utilize, increasing the time to provide the end-user experience. Though developers have established numerous methods to solve the problem of REST excess, the excess

still always adds significant overhead for client-side developers, as the client-side developers need to make a lot of edge cases for certain parameters being passed through.

- **Security with Multiple Endpoints:** Given that each endpoint is static and contains only a small set of information, the number of endpoints scales almost linearly with the amount of information servers wish to expose. Thus, popular web applications tend to build out a significant number of endpoints. This is a nightmare for data security, as companies have hundreds of different access points to their server-side logic, and bad-actors will search for vulnerabilities across each one of them.
- **Documentation:** One of the biggest drawbacks of REST APIs is that they are schema-less, which means client-side developers have no idea what data structure will be returned when querying an API. There is no way to understand from the endpoint itself what the client will be receiving, and as such the back-end server-side developers must maintain robust documentation to give their users an idea of what to expect. This can be a nightmare to do, especially if the server-side developers attempt to document their API retroactively, and can also lead to slow ramp times for new developers coming in. This is far from an impossible problem to solve, but still a headache to contend with when working with REST.
- **Updates:** In practice, a significant number of endpoints are designed on the go, with the frontend server-side team reaching out to the backend server-side team with their data requirements, consequently building an endpoint for them. This introduces significant overhead in the software development process. Every design iteration on the frontend that involves a change in the displayed data needs to go through a process where the backend team is directly involved, which can render APIs brittle and error prone. APIs that are frequently changing are hard to maintain and clients will have a hard time procuring the desired data. When fields are removed from certain API responses without the client

being aware of it (the client was simply not alerted and is still running against an older API version), there's a high probability that the client's program will crash at runtime due to missing data.

Thus, although REST APIs pose significant positives, and are by far the most universal type of API in the world today, there exist significant drawbacks to implementing it at scale. Whilst these problems were less common 10-20 years ago, as APIs were less ubiquitous and the amount of data being transferred was smaller, REST's problems are becoming increasingly serious for modern developers.

GraphQL, an open-source project developed by Facebook, deals with a lot of these issues whilst still maintaining the strengths that made REST APIs as popular as they have become. After taking a further look, we hope to convince you that GraphQL is in fact the future of API development.

5.4 GraphQL History and Analysis

In the early 2010s, Facebook struggled to build out their mobile app [35]. Facebook was traditionally strong at building web-experiences, and, because of this, decided to display web views on mobile phones, instead of building a native experience tailored to this smaller, less powerful device. This caused serious issues for users, as mobile phones couldn't keep up with the bandwidth and network requirements to display these large web components.

The main problem was Facebook's *News Feed* implementation on mobile, as its web views not only required the retrieval of a post, but also significant additional information that needed to be constantly updated, namely comments and likes. Additionally, all of these Facebook posts were nested, interconnected, and recursive, rendering Facebook unable to handle the complexity with their existing APIs, due to a number of the weaknesses we mentioned above about REST, namely roundtrips and verbose responses. Thus, Nick Schrock, Alex Langenfield, Dan Schafer, and Lee Byron, four engineers working on the mobile team at Facebook in 2015, developed GraphQL.

GraphQL is an open-source language for client-side applications to query databases. GraphQL only exposes one endpoint, from which a client can retrieve all of the desired information. A GraphQL request is essentially a string that, when sent to a server, returns JSON back to the client. These responses will take the same shape as the query, so clients know the shape of the data that's returned from running the query, rendering it significantly easier for client-side developers to write queries, should client-side developers be cognizant of the sorts of data their application requires.

In GraphQL, it is up to the server to grab relevant information and return it to the client, whereas in REST the client needs to make multiple trips to return the desired information. Pushing the burden of data retrieval to the server in this way allows the client to retrieve many resources from the server in a single request. GraphQL is strongly-typed [36]: it allows API users to know exactly what data is available and the formatting of its existence. This renders it significantly easier for clients to figure out how to build their queries and for developers to maintain the API on the backend.

5.5 GraphQL as a 'Fetching Tree'

While the name GraphQL may imply that this API structure optimized for graph databases, this is not actually the case. GraphQL allows server-side developers to model the resources and processes provided by their servers as a Domain Specific Language. Thus, server-side developers can create a querying language that maps specifically to the structure of their database, and force client-side developers to conform to their naming conventions and structure in order to formulate their queries, yet simultaneously allowing client-side developers to execute more efficient queries.

The best way to understand how a GraphQL query is processed is by abstracting away what queries generally look like. Typically, client-side applications are designed in the form of discrete pages, which are seeded with some tiny bit of data, and then perform a cascade of fetches to get the data needed to provide a unique user experience to the end-user. As an example, let's assume that, on a client-sid web application, a profile page is 'seeded' with a user-id, and from that information the web application can reach out to multiple endpoints across the backend server architecture to grab the information necessary to populate the rest of the page. The fundamental insight in the development of GraphQL is that, in most cases, this contingent data fetching forms a tree that is more or less fixed for a given page. Data from early responses contain the keys for subsequent requests (such the address of my profile image in the REST example provided earlier), and the linkages between these requests are usually straightforward. As such, if the client can factor all of these disparate fetches into one spot, encode them as one large fetching tree, and send that fetching tree to the server, the client can receive all of the user's data in request, eliminating the multiple roundtrips that are often needed with REST APIs, and consequently saving significant bandwidth and latency. Now that we have established this abstraction of a query as a fetching tree, let us examine how the fetching tree defined within GraphQL.

5.6 Anatomy of a GraphQL Request

A GraphQL request always starts with at least one 'root' API operation and ensues with some finite number of follow-ups. These follow-ups serve as queries, meaning they retrieve data without changing the server in any meaningful way. GraphQL models all API operations as fields; these fields are split into two types:

- **Scalars:** Scalars represent the individual pieces of data that the server will eventually deliver to the client. These are the leaves of the request tree, and the data stored in these leaves can be arbitrarily complex.
- **Objects:** Objects are a collection of fields and serve as a junction in the tree. Objects do not return any data, but route the client to the appropriate scalars that the client is interested in.

The entire model for a given GraphQL API is known as its schema, and, in contrast to a REST API, it is strongly-typed. Every schema possesses a route query type, whose fields serve as the API's entry points. An example of this is shown below.

```
type Query {
        user(id: ID): User
        image(url: Url): Image
        # … a whole bunch more root fields
    }
```

```
 type User {
    name: String
    nickname: String
    image: Image
}

 type Image {
     url: String
     width: Int
     height: Int
}
```

Figure 5.6.1: GraphQL Schema.

A GraphQL query begins by mentioning at least one of the fields of the root query object, and the aforementioned field can be utilized to specify any follow up queries. It is important to note that any field in the request tree can take arguments, so a request can be parametrized at all depths. An example query is shown below:

```
{
  user(id: “10”) {
      name
      nickname
      image {
        url
      }
   }
}
```

Figure 5.6.2: GraphQL Request.

Here the client is instructing the server to look up a user by the user’s slug (the ID), returning specifically the user’s their name, nickname, and image URL, executing the entire request in one trip instead of the two this request would have required using REST. GraphQL achieves this improved efficiency because of *resolvers* which are worth mentioning in additional detail.

5.7 Note on GraphQL Resolvers:

In GraphQL there is only one method of information propagation: the propagation of context through a sequence of resolvers. While this sounds complex, it simply means that when a client reaches the 'image' object in *Figure 5.6.2* above, it understands the context that I am asking for the image associated with the user that has the id "10". Every field has a resolver. For scalar fields, the resolver is responsible for returning the actual data that the client sees. For object fields, the resolver instead returns a hidden chunk of data that is forwarded along to the resolvers of the fields contained in the object; these resolvers obtain their parent object's hidden data, the global context, and any arguments, using all of these values to produce the desired return value set.

Now that we have exhibited a GraphQL query, it is important to note that the response will mirror the structure of that query, as shown below:

```
   {
“data” : {
    “user” : {
       “name” : “Satoshi Nakamoto”,
       “nickname” : “Seb”,
       “image” : {
          “url” : “image_url”
       }
    }
  }
}
```

Figure 5.7.1: GraphQL Response

5.8 Direct Comparison

Now that we understand REST GraphQL in more detail, let us compare why the two architectures and articulate why we believe GraphQL is better than REST for a majority today's use cases, and why it will become entirely dominant in years to come.

Let us model a company's API as a vending machine. With traditional REST, a client presses one button on the vending machine and gets one thing. So, the client has to press lots of buttons one at a time to get everything the developer needs. This process is slow. One way to solve this problem is to create special purpose buttons, which let the client get multiple things at once. Another way is to allow the client to press a special purpose button and get a lot of information at once from the vending machine, but the client can't perfectly control the information it receives, which can lead to you transferring more information than is needed in some cases, increasing bandwidth and latency. GraphQL solves the aforementioned inefficacies in the vending machine: vending machine that allows for the client to press exactly one smart button that gives the client everything the client wants in one go.

GraphQL solves these issues by having a single 'smart' endpoint rather than having many 'dumb' endpoints. The key benefit is that smart endpoints are able to take complex queries and shape the data output into whatever the client requires. Essentially, the GraphQL layer exists between the client and data sources. Its job is to receive client requests and fetch the necessary data based on the client's requirements. The GraphQL approach to querying addresses a wide range of large-scale app development problems.

As you can imagine, GraphQL addresses most of the main weaknesses of REST that were mentioned previously:

- **Multiple Round Trip (Latency):** Now that the client can push complex queries to the server, and have it return all of the client's desired information in one go, round-trips are an item of the past.
- **Verbose:** With GraphQL the client only gets back the information the client requested, no more irrelevant information.
- **Security with Multiple Endpoints:** With GraphQL, the server only has one endpoint to protect, rendering a significantly easier task for back-end security efforts.

- **Documentation:** Given that GraphQL is strongly typed, it's a lot easier to automate the creation of documentation; there are a number of services that will do this for back-end developers. There are also great dashboards that back-end developers can use to experiment with different queries and see how the queries will be responded to by the back-end server.
- **Updates:** Additional fields can easily be added to the server, i.e. adding new product features or deprecating older features, without affecting existing clients. In this way, GraphQL brings about a backward-compatible process that eliminates the need for incrementing version numbers, simplifying versioning in general.

This is in fact one of the most valuable aspects of GraphQL - it doesn't lose any of the functionality that made REST so great, whilst improving on a lot of its weaknesses. It is a rare example of a net additive to a widely adopted system, why it has gained popularity so quickly.

It is also fairly easy to convert from an existing REST API to a GraphQL API, and given the low switching cost and explosion of managed tools to help maintain a GraphQL system, the choice to migrate is only going to get easier.

All this being said, GraphQL is far from a perfect system. In fact, similarly to Kubernetes [37], the reason it provides so many distinct benefits is because it is a significantly more complex system than REST.

5.9 Weaknesses of GraphQL

GraphQL has been optimized for modern systems, companies that are embracing future-facing architectures. The world is moving towards microservices, which means that they are breaking their large applications into independent components that communicate with one another via APIs [38]. As we have discussed, REST struggles to scale, a deal-breaker when it comes to microservices. By the nature of this decoupled architecture, there exist significantly more API calls being made, and a lot more updates being made to data schemas in the back-end, which are actions that REST struggles with significantly.

As such, it is important to understand where GraphQL faces issues when serving a microservice architecture, as for more trivial use-cases it may be over-engineered:

- **Schema Design:**
    - In a microservice architecture, like all multi-service architectures, each service exposes a certain amount of information about the back-end system, and there are multiple services working in tandem to run your application. To embrace the powers of GraphQL, back-end developers need to stitch all of these services together under a unified schema, which can cause issues around duplication of resources, and conflicting fields between resources.
    - One solution here is Prefixing, where the back-end developer adds a prefix to each of the broadcasted microservices, such the microservices are all logically separated. This adds a layer of abstraction that actually detracts from the centralized promise of GraphQL, as each service is treated as an island, meaning it becomes impossible to concatenate resource relationships across services. The benefits are that this is very easy to implement.

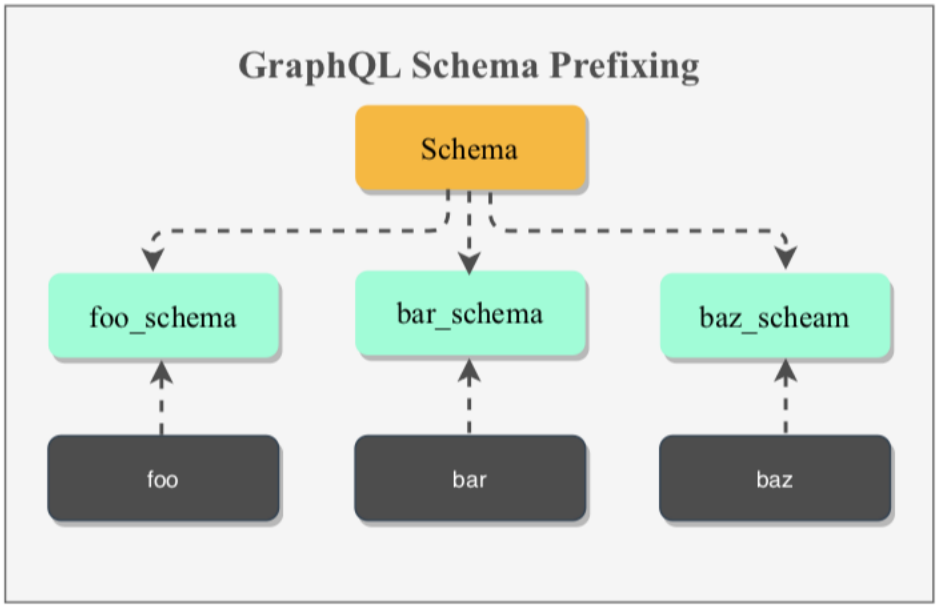


Figure 5.9.1: GraphQL Schema Prefixing.

- ○ Another solution, which is the official recommendation of GraphQL, takes the form of Bonding, where the back-end developer glues the different GraphQL Schemas from each microservice into a uniform interface. In the past, this required developers to manually stitch schemas together, which adds a serious amount of complexity, but the recent launch of Apollo Federation is a significant step in reducing the complexity of this approach [39]. As such, the community is moving towards solutions that are able to get information from multiple services whilst staying true to the GraphQL methodology.

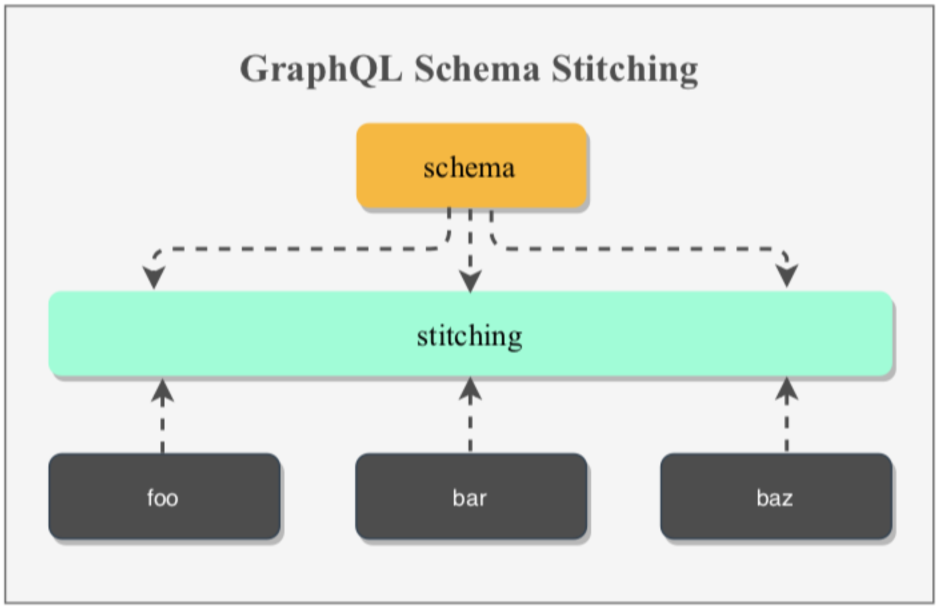


Figure 5.9.2: GraphQL Schema Stitching.

- **Certification and Authorization:**
  - ○ GraphQL renders a multi-tiered service as we've seen above. On the top level exists the GraphQL server itself, which takes the request from the client and handles the response. Beneath the top

level exists the middle tier, which handles communication from the GraphQL server to the various microservices, and the data tier from which the microservices get data to push back up to the GraphQL server.

    - As such, for authorization developers can choose to decide whether a user has permission to operate on a certain resource at the GraphQL server level, or the microservice level, and the answer to this problem culminates as a trade-off between a centralized and distributed architecture. This is actually a trickier problem than developers might anticipate. Let's assume that all the authorization is done at the GraphQL server level; this would mean that the services fully trust the call of other services, which can lead to vulnerabilities. On the other hand, if authorization is left as a service level decision, then each service needs its own logic and will need to provide different interfaces to different GraphQL requests, which adds complexity. In principle, companies need to define a hybrid approach in which secondary checks are made at the micro-service level for certain requests, but, as exhibited, this is not a simple thing to think through and set up.

- **Routing Design:**
    - When a request comes in, the GraphQL server is responsible for routing that request to the correct services from which it can get the information. If the server possesses a single backend service this is very easy, as it will have access to all the information, but if that's not the case (as is typical of large microservice deployments), then routing the requests efficiently becomes a problem.
    - GraphQL schema stitching is actually a solution here, as it will implicitly route request fragments to the appropriate microservices through the various resolvers that were updated in order to stitch the service schemas together. In this case, no manual intervention is required, unless there exists a conflict in the schema type (i.e an update has been made to the data model in the relevant microservice). There exist some nice properties of GraphQL that can help here, like the fact that developers can define the expected

time it will take for a request to execute, so the server has an idea of how to prioritize effectively, but solving this problem requires significant manual effort. Handling the routing of requests is a fundamental problem that needs to be addressed in every GraphQL system, and it is far from simple to handle a large amount of concurrent requests to various services in a reliable and efficient way.

- **Error Handling:**
    - With a typical REST query, the response given will typically describe the outcome of the request. For example, if the request succeeds, the client will receive a code 200; if the resource couldn't be found, the client will receive a code 404. In GraphQL, this isn't the case; clients will always receive a code 200 meaning the query returned successfully, and any information about errors will be wrapped in the return object. This means client-side developers must write custom logic for error handling that can be tedious and add complexity to the overall system.

Thus, despite the benefits of GraphQL, there exist significant issues that developers face when implementing it in production. However, we believe this friction is true of any new technology seeking to fully disrupt a market, and no aspect of GraphQL's design seems to be a fundamental limiter of its potential success. It is able to deal with a more complex set of use-cases because it is more complex, and, as with Kubernetes, it may take additional time before a user-friendly version of GraphQL is available to use broadly against a microservice architecture, but, once it does, it will dominate the market. It is our belief that GraphQL is perfectly positioned to capitalize on the architecture of the future, and that as the world moves to cloud and microservice architectures, GraphQL rise to dominate the API ecosystem. However, REST is still the leading solution and so must be accounted for as such.

6 Bibliography

[1] Wanseob-Lim. “Zkopru (Zk Optimistic Rollup) for Private Transactions.” *Ethereum Research*, 21 July 2020, https://ethresear.ch/t/zkopru-zk-optimistic-rollup-for-private-transactions/7717

[2] Bakst, Andrew. “Bizantine’s Law - Andrew Bakst.” *Medium*, 18 July 2020, medium.com/@apbakst/bizantines-law-c9bc93529e89.

[3] “Secure Multi-Party Computation: Theory, Practice and Applications.” *ScienceDirect*, 1 Feb. 2019, www.sciencedirect.com/science/article/pii/S0020025518308338. & “What Is Fully

Homomorphic Encryption." *Inpher*, https://www.inpher.io/technology/what-is-fully-homomorphic-encryption.

[4] Voell, Zack. "Decentralized Exchange Volumes Up 70% in June, Pass $1.5B." *CoinDesk*, 1 July 2020, https://www.coindesk.com/decentralized-exchange-volumes-up-70-in-june-pass-1-5-billion.

[5] Wikipedia contributors. "Elliptic Curve Digital Signature Algorithm." *Wikipedia*, 17 Aug. 2020, https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm.

[6] Yang, Junha. "Blockchain Light Client - CodeChain." *Medium*, 3 Feb. 2020, https://medium.com/codechain/blockchain-light-client-1171dfa1269a.

[7] Bengtsson, Ivar and Fichter, Michael. "Modeling and Optimizing Transaction Fees in a proof-of-stake cryptocurrency." 2018, http://kth.divaportal.org/smash/get/diva2:1218593/FULLTEXT01.pdf.

[8] Blocknative. "Evidence of Mempool Manipulation on Black Thursday: Hammerbots, Mempool Compression, and Spontaneous Stuck Transactions." *BlockNative*, https://blog.blocknative.com/blog/mempool-forensics.

[9] Young, Joseph. "Why A Mysterious Ethereum User Paid $2.6 Million To Send $130 Of Crypto." *Forbes*, 10 June 2020, https://www.forbes.com/sites/youngjoseph/2020/06/10/why-a-mysterious-crypto-user-paid-26-million-to-send-merely-130-in-ethereum/#485be9b9588a

[10] Pourmajidi, William and Miranskyy, Andriy. "Logchain: Blockchain-assisted Log Storage." 22 May, 2020, https://arxiv.org/pdf/1805.08868.pdf

[11] "Global Cloud Database and DBaaS Market (2020 to 2025) - Increase in the Growth of NoSQL Database Provides Opportunities - ResearchAndMarkets.Com." *Business Wire*, 17 Mar. 2020,

www.businesswire.com/news/home/20200317005638/en/Global-Cloud-Database-DBaaS-Market-2020-2025.

[12] “Usability: A Part of the User Experience.” *The Interaction Design Foundation*, 28 July 2020, https://www.interaction-design.org/literature/article/usability-a-part-of-the-user-experience.

[13] Zhao, Yuanyuan, Sturman, Daniel, and Bhola Sumeer. “Subscription Propagation in Highly-Available Publish/Subscribe Middleware.” https://www.researchgate.net/profile/Yuanyuan_Zhao17/publication/221461384_Subscription_Propagation_in_HighlyAvailable_PublishSubscribe_Middleware/links/5640600208ae34e98c4e7d50/Subscription-Propagation-in-Highly-Available-Publish-Subscribe-Middleware.pdf

[14] “Optimistic Rollups.” https://docs.ethhub.io/ethereum-roadmap/layer-2-scaling/optimistic_rollups/

[15] Giroux, Marc-AndrÉ “On GraphQL Schema Stitching & API Gateways - Marc-André Giroux.” *Medium*, 24 Oct. 2018, https://medium.com/@__xuorig__/on-graphql-schema-stitching-api-gateways-5dcb579fa90f.

[16] “Web Cryptography API.” *Wikipedia*, 7 Aug. 2020, https://en.wikipedia.org/wiki/Web_Cryptography_API

[17] Compound-Finance. “Compound-Finance/Compound-Protocol.” *GitHub*, https://github.com/compound-finance/compound-protocol/blob/master/contracts/Governance/GovernorAlpha.sol.

[18] Peterson, Jack, Krug, Joseph, Zoltu, Micah, Williams, Austin, and Alexander, Stephanie “Augur: A Decentralized Oracle and Prediction Market Platform (v2.0).” Whitepaper, https://augur.net/whitepaper.pdf

[19] “Fuel Labs.” *Medium*, https://medium.com/@fuellabs.

[20] “Patricia-Tree.” *Ethereum Wiki*, https://eth.wiki/en/fundamentals/patricia-tree.

[21] Ethereum-Optimism. “Ethereum-Optimism/Optimism-Monorepo.” *GitHub*, https://github.com/ethereum-optimism/optimism-monorepo.

[22] “When Should I Use Calldata and When Should I Use Memory?” *Ethereum Stack Exchange*, 30 Aug. 2019, https://ethereum.stackexchange.com/questions/74442/when-should-i-use-calldata-and-when-should-i-use-memory.

[23] Wood, Gavin “Ethereum: A Secure Decentralised Generalised Transaction Ledger Petersburg Version” 2020-06-08, https://ethereum.github.io/yellowpaper/paper.pdf

[24] **zkSync**: Gluchowski, Alex. “Introducing ZkSync: The Missing Link to Mass Adoption of Ethereum.” *Medium*, 18 June 2020, https://medium.com/matter-labs/introducing-zk-sync-the-missing-link-to-mass-adoption-of-ethereum-14c9cea83f58.
Validium: Kirejczyk, Market, Szlachciak, Piotr, Jelski, Krzysztof, Maretskyi, Dmytro, Wiech, Arleta, Charczuk, Joanna, and Kirejczyk, Natalia “Zero-Knowledge Blockchain Scalability.” Summer, 2020, https://ethworks.io/assets/download/zero-knowledge-blockchain-scaling-ethworks.pdf.

[25] Alex, Chris. “Balancer Thesis.” *Placeholder*, 20 July 2020, https://www.placeholder.vc/blog.

[26] The Block. “Popular Ethereum Wallet MetaMask Adopts New Software License as Firm Eyes Commercial Opportunities.” *The Block*, 21 Aug. 2020, https://www.theblockcrypto.com/linked/75751/metamask-new-software-license-commercial.

[27] Martinez, Julian. “Understanding Proof of Stake: The Nothing at Stake Theory.” *Medium*, 22 Jan. 2020, https://medium.com/coinmonks/understanding-proof-of-stake-the-nothing-at-stake-theory-1f0d71bc027.

[28] “Synthetix | Decentralised Synthetic Assets.” *Synthetix*, https://www.synthetix.io/. Accessed 22 Aug. 2020.

[29] Substreight. “YIP 30: YFI Inflation Schedule.” *Yearn.Finance*, 2 Aug. 2020, https://gov.yearn.finance/t/yip-30-yfi-inflation-schedule/1439.

[30] Fielding, Roy “Architectural Styles and the Design of Network-based Software Architectures.” UCI, 2000, https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

[31] Elements, Cloud. “The State of API Integration 2019 | Cloud Elements.” *Cloud Elements*, https://offers.cloud-elements.com/the-state-of-api-integration-2019#:%7E:text=The%20State%20of%20API%20Integration%202019%20Report&text=Our%202019%20State%20of%20API,industry%20is%20heading%20in%202019.%20Accessed%2022%20Aug.%202020.

[32] “Front End and Back End.” *Wikipedia*, 22 June 2020, https://en.wikipedia.org/wiki/Front_end_and_back_end.

[33] rapidAPI Staff “What’s the Difference between PUT vs PATCH?” *rapidAPI, 17 August 2020,* https://rapidapi.com/blog/put-vs-patch/

[34] “HTTP Headers.” *MDN Web Docs*, 27 Apr. 2020, https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers.

[35] “TechCrunch Is Now a Part of Verizon Media.” *TechCrunch*, 29 Jan. 2014, https://techcrunch.com/2014/01/29/one-app-at-a-time/.

[36] “Strongly Typed.” *Techopedia.Com,*
https://www.techopedia.com/definition/24434/strongly-typed.

[37] “Production-Grade Container Orchestration.” *Kubernetes,*
https://kubernetes.io/. Accessed

[38] “Microservices.” *Martinfowler.Com,*
https://martinfowler.com/articles/microservices.html.

[39] “Apollo Federation.” *Apollo Blog*,
https://www.apollographql.com/blog/apollo-federation-f260cf525d21/.