LinkPool

A Trust-less Staking Network for ChainLink

ABSTRACT

ChainLink is poised to revolutionise smart contracts by providing support to communicate with external systems. This is supported with an implementation of new functionality called an *oracle* that acts as a mediator between blockchains and the outside world. For ChainLink to operate, it relies on an internet of oracles with an aggregation method, both on-chain and off-chain to create the decentralised network (1).

In this paper, we present LinkPool; an expansion of the ChainLink functionality by providing a trust-less staking mechanism for ChainLink oracles. A "LinkPool" represents the grouping of oracles within the same on-chain contracts. We provide on-chain mechanisms for allowing aggregation of oracles into a single-owner-based contract, end-user informed staking mechanisms and the fair distribution of tokens to stakers within any given node and pool. The LinkPool managed service will be a software-as-a-service platform, consisting of a highly-scalable, high availability network of off-chain oracles.

Website: https://linkpool.io

Twitter: https://twitter.com/linkpoolio

Author: Jonny Huxtable, Reviewed By: Mat Beale

TABLE OF CONTENTS

TABLE OF FIGURES

1 Introduction

Since ChainLink relies on a network of oracles to fulfil data requests that originate in a blockchain, the overall performance and usability of the network relies on the quality of the oracles. With the network relying on trust-less mechanisms, the oracles can be managed and ran by any entity whom has any incentive in doing so. The oracles within the ChainLink network are referred to as *nodes*.

For the network to succeed and provide reliable external system communication, it needs the following attributes:

- **High Uptime:** To provide reliable data input to on-chain contracts, the node network needs to provide a near-flawless uptime.
- **Quality Data Sources:** The network of oracles is only as valuable as the data it can provide, so data sources relevant to market demands need to be provided.
- **Collaboration:** To ensure the decentralised mechanism is retained, large majorities of the network need to provide the same data sources to remove any potential centralisation.

Based on the attributes above, a decentralised network of oracles could face the following issues:

- **Poor Quality of Oracles:** If the network is consisted of what we term as *hobbyist nodes*, then the up-time and quality of data will suffer. As any hobbyist node could reside on home-networks or with providers that lack any real SLA or core network accessibility, and could lead to poor network connectivity or latency.
- **Lack of Data Sources:** A hobbyist node provider may not have the desire to provide data sources which are outside of the in-built functionality within a ChainLink node, lowering the quality and total of data sources provided.
- **No Collaboration between Oracles:** For a majority of oracles on the network to support any given data source, there needs to be a mechanism of collaboration. If there is no mechanism, then there's no understanding of what data sources on the network are being provided or are valuable at any given time. In effect, this would increase the risk of centralisation within any given data source as node operators can't easily see which data sources to provide.
- **Scalability:** If there are peaks of adoption in the network, the amount of oracles provided should naturally scale to ensure there are enough oracles to meet the demand of the end-users.

LinkPool's on-chain mechanisms and service aids to mitigate the problems specified above by providing a large, enterprise-grade service that aims to push the boundaries of what the network allows both in data support and collaboration.

2 WHITEPAPER ROADMAP

In this whitepaper, we review the API economy and the profitability of running a ChainLink Oracle. We will describe and review the on-chain mechanisms LinkPool uses to provide trust-less staking and token distribution. Finally, give a high-level design of the LinkPool service, detailing how the service is architectured and how it mitigates the problem statements defined in the introduction.

All of the information in this whitepaper has been written to the current alpha specification of ChainLink. Any area is subject to change within development which could invalidate details in this whitepaper.

3 API ECONOMY

API's (Application Programming Interfaces) have quickly become new focus areas within the digital space, providing vehicles of transformation within many sectors globally. For example, an API directory called *ProgrammableWeb* currently has over 19,428 API's listed, having adding over 5,900 API's between 2014-2017 alone (2).

The increase in these API's are from a new initiative for businesses to open up their internal functions and data, potentially monetizing them. Opening up API's encourages businesses and developers to build on-top of these API's, allowing for new apps, websites and internal systems to be more collaborative and information-rich.

This initiative has led to new business models, allowing new API marketplaces to aggregate these data sources and to monetise them, encouraging more use by defining standard API specifications and easy inter-connectivity between different data sources. For example, one of these marketplaces *RapidAPI* recently announced that they're serving over 400 Billion API calls each month. The API economy has been forecasted to generate over $2.2 trillion in the next 10 years. (3).

ChainLink's aim is to create a decentralised API marketplace, allowing developers and businesses who seek to use blockchain technology to buy external data provided by a network of oracles, creating an API economy specifically tailored to the Blockchain. LinkPool is aiming to provide numerous ChainLink oracles, helping deliver its API economy while supporting its decentralised mechanisms.

4 Oracle Economy

To provide incentive for node operators to be part of the network, each node operator will be paid in LINK tokens for each data request or job that they serve (4). This incentive alone creates the economy for oracles, providing a revenue stream that will encourage more oracle providers to join the network and facilitate more data requests.

To display the sustainability and feasibility of the oracle economy within ChainLink, we have broken down the estimated potential operational cost and revenue based on a fraction of the RapidAPI marketplace usage (0.001%), which results in 4 million requests a month. We deem 0.001% of the throughput of RapidAPI a conservative estimate and would expect to see substantial factors of growth based on adoption.

All of the examples are calculated in USD. The reason for this is that we envisage the cost of data retrieval will be tied to the relative FIAT value, rather than an amount of LINK. Therefore, we speculate that the amount an oracle will generate in revenue will always be tied to the amount of requests that the network undertakes, rather than the LINK token value. Although, any historic income generated on an oracle is paid in LINK, so it will fluctuate in value based on the token price volatility.

4.1 Operational Cost

For the purpose of this example, we will not be including the operational cost of the hardware involved to run a ChainLink oracle as it is widely varied based on approach. This example just includes the cost for an oracle to retrieve public data and write it back to the Blockchain.

There will be two events when during a jobs life-cycle where an oracle needs to pay GAS, these are:

- The transfer of LINK penalties to the order matching contract (if applicable and subject to change).
- The writing of the data on-chain for data aggregation.

The GAS costs of these operations are unknown as they're yet to be released and tested. Although, for the purpose of the examples, we will use the average GAS cost of an ERC-20 token transfer increased by 30%. That results in 35,000 GAS (5).

We will also be assuming that each job that the oracle is accepting is also requiring the oracle operator to put forward a penalty payment, meaning both on-chain operational costs will always be included in the examples specified in this whitepaper.

Altogether, this results in a GAS cost of roughly 70,000. Which in the current market climate at a price of 1 Gwei, equalling in a price of $0.02744 (6).

4.2 Oracle Revenue and Profit

For an oracle to ensure that it generates profit, a node operator can define a minimum price in LINK for any data request they serve. This minimum price doesn't limit the amount of revenue they may earn, rather sets a safeguard to ensure that any individual data request is profitable for the oracle provider.

For the purpose of these examples, we will be using a minimum price of $0.05488 per request, providing a 100% margin on the data request that has been served. In addition, due to the decentralised nature of ChainLink, multiple nodes will facilitate each data request. For the purpose of this example, I will use four nodes for each request as an average.

To accurately estimate the potential revenue of each individual node, the amount of oracles on the network needs to be known. Since the network isn't yet currently available, I will refer to the 19,000 oracle providers that registered interest with the ChainLink team (7).

To now breakdown this example using the variables defined:

Total Requests per Node: (4,000,000 *4) / 19,000 = 842

Revenue per Node: 842 * 0.05488 = $46.20896

Operational Cost per Node: 842 * 0.02744 = $23.10448

Profit per Month per Node: 46.20896 - 23.10448 = $23.10448

Figure 1 - Breakdown of Revenue/Profit based on 0.001% of RapidAPI Usage

Although, there are some important points to consider in regards to the example:

- GAS costs fluctuate based on Ethereum network usage, operation of cost can suddenly rise and node operators need to take that into account in their defined minimum price.
- Due to the nature of on-chain contracts and the frequency in which they are likely to be called, we think the amounts of calls per unique user will be lower than any traditional API usage in existing systems.
- The requests are always presumed to be paid at minimum price. Any high value contracts will most likely use more nodes and pay more for the requested data. This is a critical point, as very few of the requests will be at minimum price.
- Off-chain computation is an item in the technical roadmap within the ChainLink whitepaper. Once this is implemented, it will drastically reduce operational costs within the network. (1)
- This is with the assumption that the 19,000 node count will all be within the same reputation provider on the ChainLink network.

5 LinkPool Revenue

With the practical example specified in section 4, we can take the result to then calculate an example revenue generated to the owners, contributors and stakers. In the BETA registration phase of LinkPool, interested parties specified an amount of LINK tokens they would be looking to stake into the platform. By knowing the amount of tokens are of interest in being staked on-to the platform, we can estimate how many nodes we will be operating, allowing us to then calculate expected revenue amounts.

During the 75 day window registrations were open, we generated staking interest of:
13,969,252.36 LINK

Figure 2 - LinkPool Generated Staking Interest

For the purpose of this example, we will be rounding that up to 14 million LINK tokens.

To calculate how many nodes we will be running, we need to understand the staking limit of each node which the LinkPool service runs. The optimum amount of LINK before diminishing returns on a single node is still largely unknown and will most likely change during operation of the network on-go live. For the purpose of this example, we will set the staking limit of each node to 20,000 LINK.

To now calculate the amount of nodes we will be running based on the 14 million tokens, simply divide the token amount by the staking limit:

Number of Nodes: 14,000,000 / 20,000 = 700

Figure 3 - Total Number of Oracles based on Staking Interest

To now calculate potential earnings of the LinkPool platform, with the share of 75/25 to stakers/owners and using the example in section 4:

LinkPool Service Profit: 700 * 23.10448 = $16,173.136
Distributed to Stakers: 16173.136 * 0.75 = $12,129.852
Distributed to Owners: 16173.136 * 0.25 = $4,043.284

Figure 4 - LinkPool Service Profit

6 Solution Architecture

From this section onwards, the on-chain mechanisms for staking and the high-level-design of the LinkPool service is going to be broken down, defining the current approach and items that are on our technical road-map to be completed after the crowd-sale.

6.1 On-Chain Contracts

For the LinkPool platform to work as a trust-less staking mechanism for the ChainLink network, on-chain contracts have been designed to ensure ownership, fair token distribution and node management. To provide a table of the contracts built to support the LinkPool platform:

Contract Name	Functionality
LinkPool	Entry point for node-creation within the contract suite, getters for the information residing within PoolStorage instances for each node; providing abstraction for any changes within storage instances for node data.
Node	Main area of functionality for any given node, provides: stake limit adjustment, withdrawal, token distribution, node status management, makers percentage adjustments and penalty amount tracking.
NodeFactory	Provides the logic for creating new node instances within the contract suite.
NodeStaking	Entry-point for stakers entering the LinkPool platform. Adds stakes to any given node.
PoolOwners	Contract the makers fees get transferred to. Provides token distribution mechanism for the owners of LinkPool.
PoolStorage	Getter, setters and deletion methods for all needed data types within the contract suite.
PoolStorageFactory	Contract to create and store all the PoolStorage addresses for the pool and nodes.

Figure 5 - LinkPool Contracts and Functionality

The main focus of the contract design was to allow each node to work independently rather than acting as one single entity within the ChainLink network. In result, there is a far higher potential income per each node as requests aren't shared divided within the pool.

In the scope of this white-paper, each contract won't be explained in detail of each method due to it all being subject to change during development. Although, the process of contract upgrades and the dependency tree will be; in addition to the areas of focus and improvement within the current suite of contracts. All of the LinkPool contracts will be made open-source prior to launch, and any questions sent to the team around the structure and design of them will be answered.

6.1.1 Dependencies

To show the dependency tree of each contract within the platform:

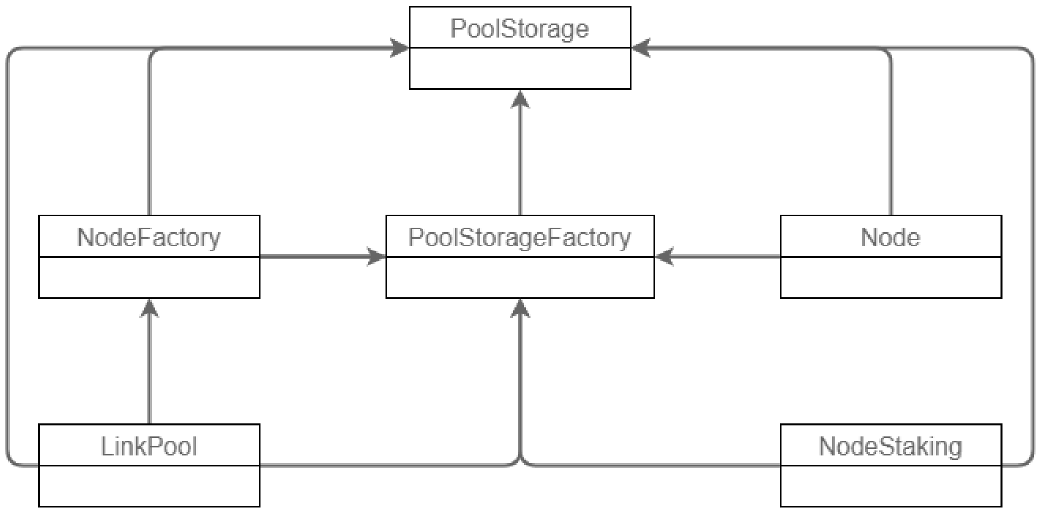


Figure 6 - LinkPool Contract Dependency Tree

Due to the nature of Solidity, contract interfaces are used for each dependency. This allows for only the method signatures to be imported into any contract which needs it as a dependency, rather than importing the full contract including method bodies. This is due to a variety of reasons:

- A non-interface contract import creates a new instance of the contract being imported.
- Due to each import being a new instance, it then drastically increases GAS usage, blocking deployment of the larger contracts which have multiple dependencies.
- Redundant imports that wouldn't use the imported method bodies, rather only using method signatures that refer to a contract address outside of the imported contract.

6.1.2 Upgrades

The ability to upgrade the LinkPool contracts without re-mapping any of the data inside the contracts to the new instances is a critical requirement. Due to the immutability of Blockchains in general, this adds extra effort into correctly designing and implementing as you can't upgrade/edit an already deployed contract. Without taking this into consideration, it would make upgrading of the contracts prone to high-severity bugs and would incur a significantly high cost.

If you refer to Figure 6, it shows every contract has a dependency on both the *PoolStorageFactory* and the *PoolStorage* instances. This is due to all the data being set/retrieved/deleted is completely abstracted away from the contracts which store the logic that manage that data. This abstraction allows for a clean upgrade of the logic which manages the LinkPool platform. When an upgrade of the contract instances that contract business logic is required, this can be done in isolation without effecting/re-mapping any of the data already existing for any node or the pool itself.

To demonstrate the flow of this action:

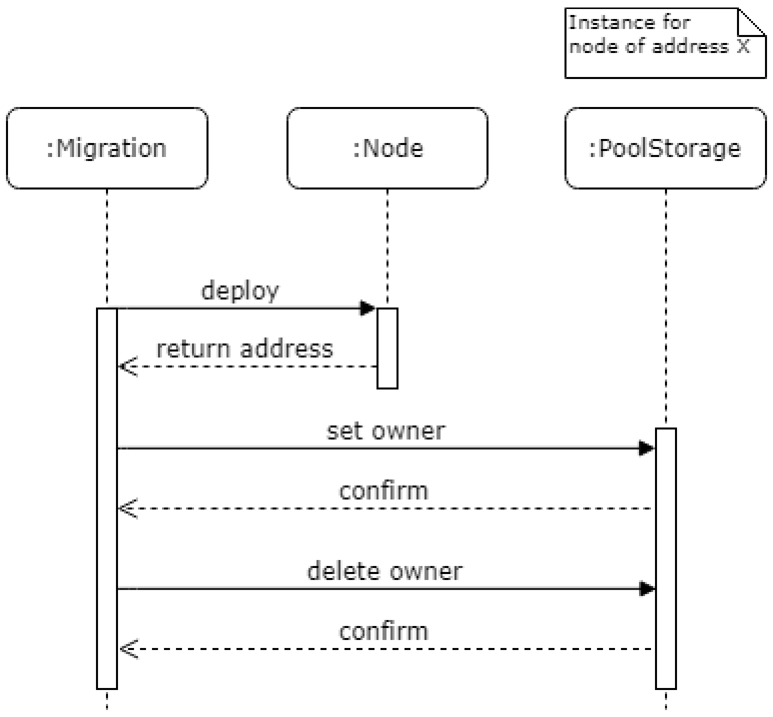


Figure 7 - Node Contract Upgrade Flow

From referring to the above figure, there are only two actions being done on a contract upgrade. This is the deployment on the new contract, then providing ownership of the *PoolStorage* instance to the new contract just deployed. This results in the new version of the node contract having permission to set/delete the data in the existing *PoolStorage* instance for the node address of X, without touching any of the existing data on the deployment of the new contract.

6.1.2.1 Upgrade Design Issues

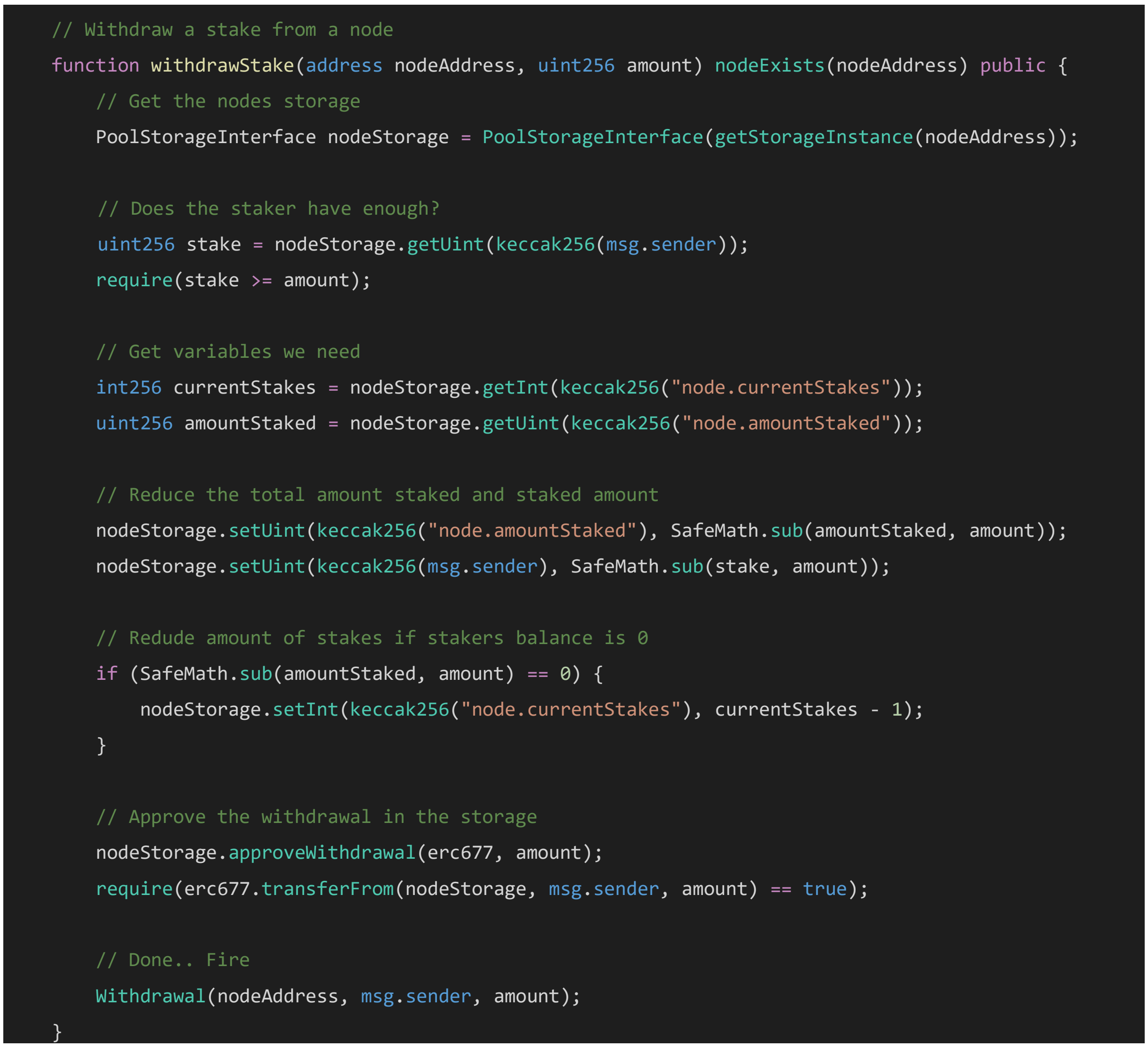
Even though a lot of design decisions have been made around the ability to upgrade contracts without affecting the data, and that has been achieved, there are some issues within the contract suite which are yet to be addressed and are mandatory prior to launch:

- Contract addresses are passed into constructors for each contract which needs the address of a contract instance. This breaks the upgrade approach, as even though a new version of the contract would work within its own scope, there's still variables within other contracts which store the address of the old contract instance. They would have to be re-deployed for the upgrade process to work, increasing cost.
- *PoolStorageFactory* contains hard-coded addresses of all the PoolStorage address instance for the pool and each node address. This contract is designed to be un-upgradeable and limits the potential scope of upgrade. For example, if new contracts are designed in the contract and are then needed to be stored within *PoolStorageFactory*, that contract has to stay unchanged which then blocks that upgrade.

With the above points being rectified, it allows full upgradability in the contracts, being able to deploy new instances of each contract individually, without then having to re-deploy existing instances of contracts with variable changes.

6.1.2.2 *Example: Withdrawing a Stake*

Below is an example method from the *Node* contract, which allows a staker to withdraw any amount of their token balance:



```
// Withdraw a stake from a node
function withdrawStake(address nodeAddress, uint256 amount) nodeExists(nodeAddress) public {
    // Get the nodes storage
    PoolStorageInterface nodeStorage = PoolStorageInterface(getStorageInstance(nodeAddress));

    // Does the staker have enough?
    uint256 stake = nodeStorage.getUint(keccak256(msg.sender));
    require(stake >= amount);

    // Get variables we need
    int256 currentStakes = nodeStorage.getInt(keccak256("node.currentStakes"));
    uint256 amountStaked = nodeStorage.getUint(keccak256("node.amountStaked"));

    // Reduce the total amount staked and staked amount
    nodeStorage.setUint(keccak256("node.amountStaked"), SafeMath.sub(amountStaked, amount));
    nodeStorage.setUint(keccak256(msg.sender), SafeMath.sub(stake, amount));

    // Redude amount of stakes if stakers balance is 0
    if (SafeMath.sub(amountStaked, amount) == 0) {
        nodeStorage.setInt(keccak256("node.currentStakes"), currentStakes - 1);
    }

    // Approve the withdrawal in the storage
    nodeStorage.approveWithdrawal(erc677, amount);
    require(erc677.transferFrom(nodeStorage, msg.sender, amount) == true);

    // Done.. Fire
    Withdrawal(nodeAddress, msg.sender, amount);
}
```

Figure 8 - Withdraw Stake Contract Method

To run through the example code block, the method has two parameters *nodeAddress* and *amount* specified. The *nodeAddress* is the Ethereum wallet address of the LinkPool service node, and *amount* is the amount of LINK tokens in wei that is being withdrawn. The *nodeExists* modifier verifies that the node address that has been passed in is an existing node by ensuring that it has an instance of *PoolStorageInterface* associated to that wallet address.

To start on the method body, *getStorageInstance* is an internal method call to get the *PoolStorage* instance through the *PoolStorageFactory* contract. The *PoolStorageFactory* contract has a mapping array which contains the addresses of the node wallets with each instance of *PoolStorage*, when this is called, it simply returns the address of the *PoolStorage* instance. Once it has the storage instance, it can firstly verify that the amount of LINK that the staker has on the node is greater than or equal to the amount they're withdrawing. If it passes that assertion, then it can fetch all the required variables from that storage instance, including the current amount of active stakes on the node and the total amount staked on the node.

After the method has received the data it needs for that node, it then modifies the total amount staked on the node, subtracting the amount being withdrawn from the total. It then also subtracts the amount the staker is withdrawing from their own balance. Once updated, the contract then checks if they've fully withdrawn from the node and if so, reduced the current amount of stakers on the node by one.

After the internal management of data of the node has been updated, a request is sent to the *NodeStorage* instance for that node to approve the withdrawal amount it has just been received. The *approveWithdrawal* method only allows the owners of that storage instance to approve any withdrawals, with consists of: *Node, NodeStaking* and *NodeFactory*.

Once the approval is completed, the *transferFrom* ERC20 token method is called from the ERC677 token address, transferring the LINK tokens from the *NodeStorage* instance, to the stakers wallet. Depending on the result of that transfer, then the Solidity event *Withdrawal* is fired with the information of the withdrawal just processed.

These notations and storage patterns are common throughout the LinkPool suite as this provides the upgradable contract design as specified in this whitepaper.

6.1.3 Trust-less Solution

Security and the integrity of LinkPool is paramount, and with the on-chain contracts now being trust-less, it allows us to safely secure the LINK tokens within a contract that retains ownership to the address of the staker rather than transferring ownership to the node.

6.1.3.1 ChainLink Integration

To realise this solution, the contracts we've developed have to integrate with the ChainLink contracts that aggregate each node. After discussions with ChainLink, we're pre-emptively designing our solution to be able integrate cleanly upon its release. Although, it is possible that the design decisions were taking may be effected by any changes upon further development of the on-chain contracts for ChainLink.

Firstly, we will be creating *PoolStorage* instances in a way that they will act as a proxy for each node we have on the network, storing each nodes token balance. Due to each *PoolStorage* instance being an individual contract, they each have an Ethereum address, allowing each node address to be mapped to the address of the *PoolStorage* instance. Due to this mapping, we can successfully trace each nodes staked tokens through each stage of the lifecycle of a job, fully retaining ownership.

With that in mind, we can now use the address of the *PoolStorage* instance as the address of the node within the ChainLink network. So once we start adding our nodes to reputation providers within ChainLink, our on-chain contracts represent our nodes (like a proxy). Our nodes will then subsequently monitor whether they've received any jobs by monitoring its respective *PoolStorage* address, fulfilling any job that it gets selected for.

6.1.3.2 Token Security

As explained above, the tokens themselves are stored in the *PoolStorage* instance of each node. When a staker enters or adds more tokens to their stake, the tokens get transferred directly to this contract with data entries stating which addresses have staked with each amount. So even though the tokens are technically stored in one address for each node, the contract holds the exact amount each staker address holds of that staked amount.

Due to penalties needing to be transferred for the accepting of jobs that require them, tokens will need to be transferred from this contract to the relevant destination address. This creates an issue, as there still needs to be a mechanism for the LinkPool service to be able to initiate a transfer of these tokens. If that mechanism wasn't correctly designed by us, it means that the owners of the contract (LinkPool), would still be able to transfer tokens to any address it desires, breaking the trust-less solution.

We have mitigated this issue by designing the contract to only transfer LINK tokens to the ChainLink network contracts that require penalty payments, creating a whitelist approach. As an extra security step, we will be looking at enforcing contract detection within the address whitelist. For example, if the address being added to the whitelist isn't a contract that resides within ChainLink, it won't allow it to be added to the white-list. That results in a full trust-less solution, as token ownership is retained and the owners can't maliciously transfer any tokens out of the pool.

Due to the public transparency of on-chain contracts, the whitelist will be publicly retrievable, allowing any concerned party to audit the addresses that are allowed transfer.

6.1.4 Token Distribution

The token distribution within LinkPool and the Owners contracts distribute tokens to the stakers and owners by proportionate distribution. This means that token distribution is performed by calculating the percentage of the staker or owner within the contract and then distributing the right amount of tokens based on the operational profit and the percentage. The calculation of this is as follows:

S = Staked Amount
T = Total Staked on a Node
O = Operational Profit
D = Token Distribution

$$D = S/T * O$$

Although, due to Solidity and the uint256 value being the lowest common demonstrator, there's no concept on decimals within the wei unit. This concept blocks easily implementing percentages, as there's no easy way to calculate the percentage of the token amount within a pool. To alleviate this issue, we use a mechanism which is similar to parts per notation, moving a decimal point to correctly be able to calculate percentage of units (8).

Unfortunately, these mechanisms increase gas costs within the contract along with needing to loop through the stakers to correctly calculate the token distribution amounts at any given time. The mechanism to distribute tokens can be triggered by anyone, not just the owners. As LinkPool, we will be running scheduled tasks which distribute tokens, but it's also possible to the stakers themselves to trigger this action.

To calculate the operational profit, we need to always set the rate of operational cost in the contract based on our mark-up as explained in section 4.1. For example, if we set the cost of a data request at double the gas price of facilitating, then the contract will have a 50% operational cost stored inside. The contract would then deduct that amount on token distribution, leaving the rest to be divided up between the owners and the stakers.

Due to the operational cost being calculated by the minimum price, any request which is executed over the minimum price will mean operational profit will reside in the operational cost as any requests over the minimum price results in a higher mark-up. Any token profit generated from the operational cost will be transferred to the owner's contract to be distributed fairly between the contributors and owners.

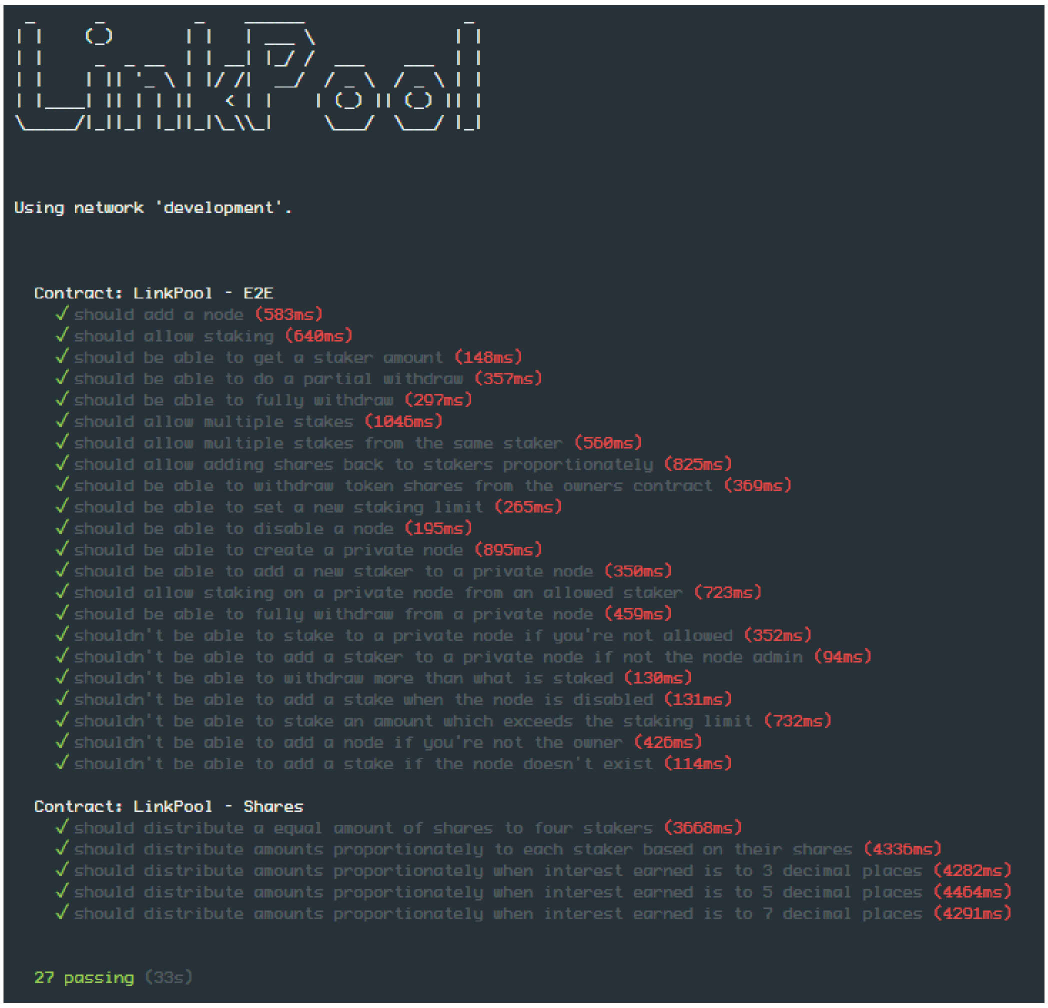
To stay competitive in the market, we are planning to be able to update this on the fly between nodes and the contract, always making sure the two are sync'd. This will allow us to dynamically update our minimum price in the market allowing us to stay competitive and securing the number of requests we could receive.

6.2 Contract Test Suites

6.2.1 LinkPool

The contract suite is built in the Truffle framework, enabling the ability to easily implement integration tests against the LinkPool contract suite. For LinkPool, we've created a suite of end-to-end tests covering all major "happy-paths" of the contract suite. It also includes negative testing scenarios. There's also the more unit-test like distribution shares tests that focuses on the exact precision of token share distribution, going up to 7 decimal places all with different proportionate ownership amounts.

The current progress on the LinkPool test suite as of writing is shown below (refer to the test case names for supported functionality):

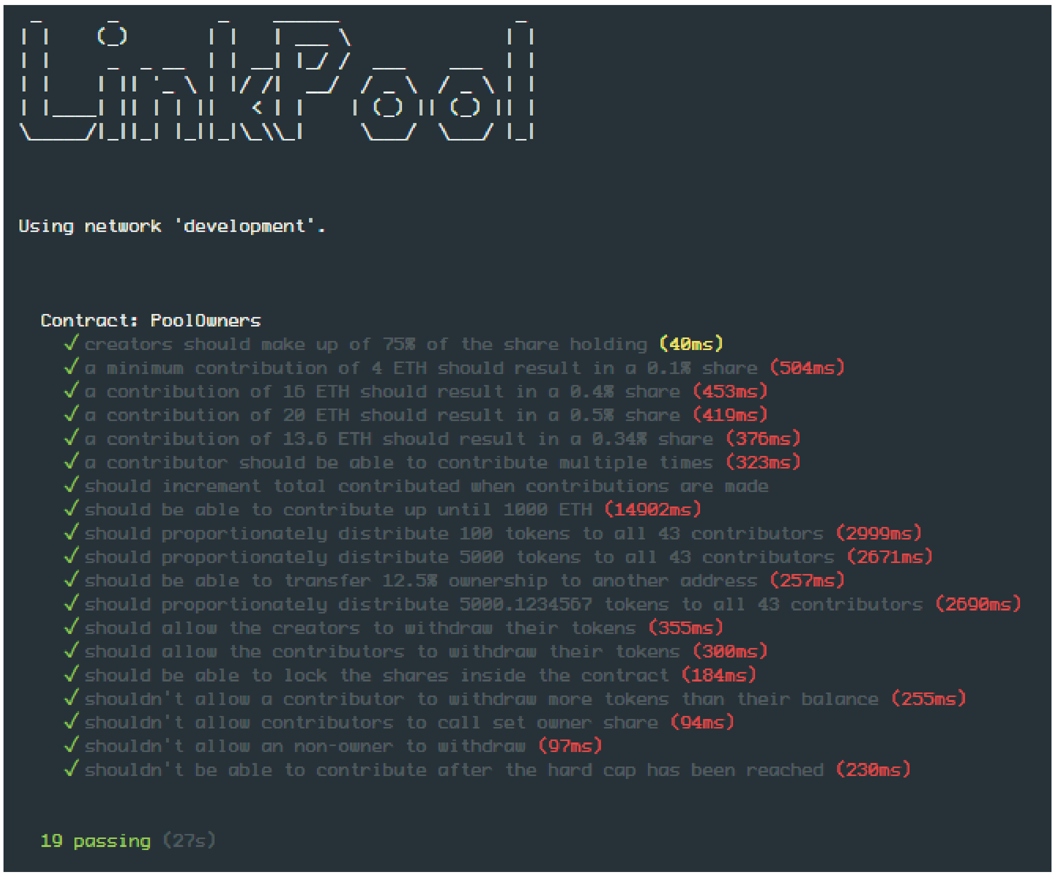


```
 _      _       _    _____            _ 
| |    (_)     | |  |  __ \          | |
| |     _ _ __ | | _| |__) |__   ___ | |
| |    | | '_ \| |/ /  ___/ _ \ / _ \| |
| |____| | | | |   <| |  | (_) | (_) | |
|______|_|_| |_|_|\_\_|   \___/ \___/|_|



Using network 'development'.



  Contract: LinkPool - E2E
    ✓ should add a node (583ms)
    ✓ should allow staking (640ms)
    ✓ should be able to get a staker amount (148ms)
    ✓ should be able to do a partial withdraw (357ms)
    ✓ should be able to fully withdraw (297ms)
    ✓ should allow multiple stakes (1046ms)
    ✓ should allow multiple stakes from the same staker (560ms)
    ✓ should allow adding shares back to stakers proportionately (825ms)
    ✓ should be able to withdraw token shares from the owners contract (369ms)
    ✓ should be able to set a new staking limit (265ms)
    ✓ should be able to disable a node (195ms)
    ✓ should be able to create a private node (895ms)
    ✓ should be able to add a new staker to a private node (350ms)
    ✓ should allow staking on a private node from an allowed staker (723ms)
    ✓ should be able to fully withdraw from a private node (459ms)
    ✓ shouldn't be able to stake to a private node if you're not allowed (352ms)
    ✓ shouldn't be able to add a staker to a private node if not the node admin (94ms)
    ✓ shouldn't be able to withdraw more than what is staked (130ms)
    ✓ shouldn't be able to add a stake when the node is disabled (131ms)
    ✓ shouldn't be able to stake an amount which exceeds the staking limit (732ms)
    ✓ shouldn't be able to add a node if you're not the owner (426ms)
    ✓ shouldn't be able to add a stake if the node doesn't exist (114ms)

  Contract: LinkPool - Shares
    ✓ should distribute a equal amount of shares to four stakers (3668ms)
    ✓ should distribute amounts proportionately to each staker based on their shares (4336ms)
    ✓ should distribute amounts proportionately when interest earned is to 3 decimal places (4282ms)
    ✓ should distribute amounts proportionately when interest earned is to 5 decimal places (4464ms)
    ✓ should distribute amounts proportionately when interest earned is to 7 decimal places (4291ms)


  27 passing (33s)
```

Figure 9 - Current LinkPool Test Suite

6.2.2 Pool Owners

Due to the share sale for LinkPool, the *PoolOwners* contract instance will be deployed separately to the main LinkPool contract suite. With it being deployed prior to the LinkPool contracts, it resides in its own repository and has its own test suite. The test suite for *PoolOwners* covers all life-cycles of the share sale including contribution of ETH and setting the percentage share; contribution up until the hard-cap being hit; distribution of tokens and the withdrawing of tokens between owners and contributors.



```
 _      _       _    _____            _
| |    (_)     | |  |  __ \          | |
| |     _ _ __ | | _| |__) |__   ___ | |
| |    | | '_ \| |/ /  ___/ _ \ / _ \| |
| |____| | | | |   <| |  | (_) | (_) | |
\______/|_||_| |_||_|\_\\_|   \___/ \___/|_|


Using network 'development'.



  Contract: PoolOwners
    ✓ creators should make up of 75% of the share holding (40ms)
    ✓ a minimum contribution of 4 ETH should result in a 0.1% share (504ms)
    ✓ a contribution of 16 ETH should result in a 0.4% share (453ms)
    ✓ a contribution of 20 ETH should result in a 0.5% share (419ms)
    ✓ a contribution of 13.6 ETH should result in a 0.34% share (376ms)
    ✓ a contributor should be able to contribute multiple times (323ms)
    ✓ should increment total contributed when contributions are made
    ✓ should be able to contribute up until 1000 ETH (14902ms)
    ✓ should proportionately distribute 100 tokens to all 43 contributors (2999ms)
    ✓ should proportionately distribute 5000 tokens to all 43 contributors (2671ms)
    ✓ should be able to transfer 12.5% ownership to another address (257ms)
    ✓ should proportionately distribute 5000.1234567 tokens to all 43 contributors (2690ms)
    ✓ should allow the creators to withdraw their tokens (355ms)
    ✓ should allow the contributors to withdraw their tokens (300ms)
    ✓ should be able to lock the shares inside the contract (184ms)
    ✓ shouldn't allow a contributor to withdraw more tokens than their balance (255ms)
    ✓ shouldn't allow contributors to call set owner share (94ms)
    ✓ shouldn't allow an non-owner to withdraw (97ms)
    ✓ shouldn't be able to contribute after the hard cap has been reached (230ms)


  19 passing (27s)
```

Figure 10 - LinkPool Owners Test Suite

6.3 INFRASTRUCTURE

Our infrastructure is what is powering the LinkPool service: providing a high-availability, highly-scalable network of ChainLink nodes that is future-proofed and will be able to stand the test of time. All the decisions being made around our network have security, scalability and reliability at the core. We're proud to be using the latest and greatest in dev-ops technologies to power this network, leading the way in-terms of node operation and support.

The scope of this section is to detail the decisions we've made which provide security, scalability and reliability and why we feel that what we will offer will pave the way.

6.3.1 Scalability

Due to running on Amazon's AWS cloud platform, there is a mass of tools and different approaches we can take to designing LinkPool, all providing similar results. The decision we made to easily scale the LinkPool nodes is by using Amazon Elastic Container Service. ECS is a managed container service, it provides a highly configurable and flexible way to run Docker containers within AWS (9).

ECS consists of three main layers: clusters, services, tasks. A cluster is made up of one or many services and each service consists of one or many instances of a given task. To put this into practise into LinkPool, we have a *Nodes* cluster, which then contains the services of *Node-A, Node-B* with each one of those services containing one or multiple node instances.

Although, using ECS by default has some issues. Due to the containerisation, there is no concept of stateful storage beyond the mounting points of the EC2 hosts the containers run on. This would mean that if a container is upgraded and in result is moved to a new EC2 instance, the previous state of the node including its database and wallet store would be lost.

To make sure our nodes are stateful, we've implemented a custom *Elastic File Storage* wrapper upon container start-up. By using EFS, we can have the benefits of having unlimited high-performance NFS storage which employs encryption yet is highly available across all regions and availability zones. The way our wrapper provides a state is by detecting the ECS service name of the current container by querying AWS API's. It then mounts the EFS volume for that ECS service within the Docker containers, using that volume mount as the data storage location for the node within the service.

With this solution, creating a node within LinkPool is as easy as sending one API call to the AWS API's, creating a new service which will then start-up the Docker containers; automatically adding itself to the LinkPool contracts upon boot based on whether the ECS service name is newly created.

6.3.2 Reliability

Due to ECS being a managed service, the Docker containers are consistently monitored and any down-time is managed automatically by the service. For example, if for any reason a Docker container within the ECS service went down, it would be recognised instantly and then recovered by the ECS service. This creates complete autonomous management and recovery of our containers. Even if the host itself went down in-which a container runs on, ECS would also recognise this and then transfer that container to either an empty host, or create a new host to recover that container if there is none free.

ECS will also manage this across availability zones. For example, in LinkPool we are using all three availability zones within us-east-1 and if anyone of those availability zones were hit with downtime, ECS would manage that by moving the failed containers over availability zones automatically.

By using this solution, we can be confident that our service will always available even in the event of any unexpected crashes within containers. Although due to the block time of Ethereum, we are not prioritising running two instances of each ChainLink node within a single LinkPool node as it would provide no benefit without off-chain computation. Our reasoning for this is because of a high chance of corrupting the data of any given node, as there is currently no real way of implementing full replication and clustering of Bolt databases as it's a key-value store. There's also no concept of running a master and slave node, meaning both nodes would have to be available at any given time, duplicating data fulfilment requests, using the same database and resulting in general confliction.

With off-chain computation not taking the Ethereum block time into account, any second of node downtime including the process of container recovery may incur in penalties. With that in mind, upon the build-up to off-chain computation, being able to run two of the same nodes will be of upmost priority. This isn't a concern with on-chain computation, as any node will have recovered from any downtime by time of the next block.

6.3.3 Security

One of the benefits of AWS is the ease of security implementation. By having easily defined security groups, we can completely manage and audit the communications between all services within the network, ensuring only the services which need to communicate actually can.

The LinkPool network is only accessible in AWS via a VPN. The VPN forces 2FA authentication for all users and is restricted by IP of only to the owners of LinkPool. There is no publically accessible service within the LinkPool network other than the public webservers (linkpool.io) and there is no inter-communication between public and private servers. Every LinkPool node is not accessible through public internet, including our parity nodes. SSH access is restricted to only the VPN server, and there is no SSH access between each instance on the network. For example, if a SSH session was granted on to a ChainLink node, there is then no ability to also SSH onto other nodes or a parity instance. All instances on the network implement SSH key access only, and these keys are stored only on encrypted hard-disks that are password protected.

In addition to the steps we're taking, it's also worth noting that the ChainLink network will not have any visibility of IP addresses of the nodes in the network. The nodes themselves are always tied by their Ethereum address as defined in section 6.1.3.1. Even if an IP of a LinkPool node was somehow discovered, any port scanning of that IP would yield no results. There is no external facing communication to that or any of the servers within the network.

6.4 Technical Roadmap

We've got a long list of features we want to implement before and after the ChainLink network releases. These features are to improve collaboration, real-time monitoring and automation within the platform by creating services which aren't business critical but add a lot of value.

6.4.1 Define AWS in Terraform

Even though the current implementation uses Amazon ECS and is easily scalable, there's still room for improvement within the defined implementation in section 6.3.1, rectified by defining our infrastructure in code within Terraform. For those unfamiliar with Terraform, it provides a descriptive language to define infrastructure in code; allowing you to then automatically plan and deploy your infrastructure changes (10).

By using Terraform, it allows us to easy create, modify and destroy AWS infrastructure by using a series of *terraform* command-line utilities with the current state of the network being stored persistently in S3. To compliment Terraform, we will utilise Jenkins (Continuous Integration Platform) to automate the deployment of infrastructure.

6.4.2 Contract Refactor

As stated in section 6.1.2.1, there are some improvement points in regards to upgrading of the contracts within the suite. We will be looking to mitigate this issue by changing the way the addresses of the contracts in each contract is stored. To ensure that we don't need to deploy a new version of each contract on change, we will store all the addresses of contracts within *PoolStorageFactory* and its own *PoolStorage* instance. If contract addresses are retrieved from this instance on each method call, it provides a single point of maintenance for the storage of contract addresses. In practise, it means any single contract deployed can be replaced individually without disrupting the function of the network.

When a migration script is ran, it will deploy the new instance of which ever contract is being upgraded. Once then deployed, it will retrieve the *PoolStorage* instance of *PoolStorageFactory* and for example, set the *contract.Node* variable to the new address of the *Node* contract. The migration will then add the new address as the owner to each *PoolStorage* of the nodes and finally remove the old address as owner. Allowing only one contract to be deployed on upgrade.

6.4.3 Real-Time Monitoring

We believe that the more data we extract out of LinkPool, the better. This is why we're planning to develop a real-time monitoring solution blending together Grafana and Prometheus. By using these two technologies, we can centrally monitor any metrics we extract in centrally managed dashboards. Some of the metrics we're planning to include are:

- Hardware Usage (CPU, RAM, Disk, Network)
- Tokens generated per each node in real-time
- Amount of tokens locked within penalties
- Assignments currently in progress
- Assignments completed
- Reputation Statistics
- Breakdown of Assignment Types and API Usage

By including all of these metrics in real-time, we will be able have an extensively detailed and constant understanding of the performance of the platform. We will also be investigating the viability of making these dashboards public, as we feel it is crucial to both the contributors and stakers to be able to watch the platform in real-time. Whether it is made public is based on security considerations.

6.4.4 Continuous Integration

The automation of the LinkPool platform is absolutely vital in a couple of aspects. For example, due to functionality like penalty transfer, assignment bidding and transferring earnings on to nodes all being manual processes (at this stage), we need to automate this so the service is self-sufficient.

The technology we'll be using for this is Jenkins which is an open-source continuous integration platform. We will be automating the jobs we create within Jenkins by using their Groovy DSL language in pipelines stored within GIT. This includes:

- Creating new nodes once there is none available
- Ensuring enough ETH is on each node/server to suffice any transactions
- Managing the transfer and deposit of penalty payments into contracts
- Withdrawal of operational income from any completed jobs
- Distribution of tokens to stakers (if needed)
- Infrastructure deployments and management

The automation of the platform will always reside within the boundaries of the VPN and not be made public.

7 REFERENCES

1. **Ellis, Steve, Jules, Ari and Nazarov, Sergey.** Whitepaper. *SmartContract.* [Online] 4 September 2017. https://link.smartcontract.com/whitepaper.

2. **ProgrammableWeb.** API Directory. *ProgrammableWeb.* [Online] 30 March 2018. https://www.programmableweb.com/apis/directory.

3. **Lunden, Ingrid.** RapidAPI, an API marketplace that processes 400B API calls each month, raises $9M led by A16Z. *TechCrunch.* [Online] 13 March 2018. https://techcrunch.com/2018/03/13/rapidapi-an-api-marketplace-that-processes-half-a-billion-api-calls-each-month-raises-9m-led-by-a16z/.

4. **Ellis, Steve.** Job Pipeline. *Chainlink Wiki.* [Online] 11 March 2018. https://github.com/smartcontractkit/chainlink/wiki/Job-Pipeline.

5. **Etherscan.** Chainlink Token. *Etherscan.* [Online] 30 March 2018. https://etherscan.io/token/0x514910771af9ca656af840dff83e8264ecf986ca.

6. **ETH Gas Station.** Tx Calculator. *ETH Gas Station.* [Online] 30 March 2018. https://ethgasstation.info/calculatorTxV.php.

7. **Nazarov, Sergey.** ChainLink, an Overview and Our Focus. *Medium.* [Online] 20 December 2017. https://medium.com/chainlink/chainlink-an-overview-and-our-focus-14f03335b803.

8. **Hitchens, Rob.** Division in Ethereum Solidity. *Stackoverflow.* [Online] 3 March 17. https://stackoverflow.com/questions/42738640/division-in-ethereum-solidity.

9. **Amazon.** What is Amazon Elastic Container Service? *AWS Documenation.* [Online] https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html.

10. **Hashicorp.** Introduction to Terraform. *Terraform.* [Online] https://www.terraform.io/intro/index.html.