COMSM0010 Cloud Computing Coursework A

Lewis Bell

1 Introduction

This report details an implementation of the COMSM0010 Cloud Computing unit's Coursework A: *Horizontal Scaling for an Embarrassingly Parallel Task: Blockchain Proof-of-Work in the Cloud.* The goal of the project is to develop a cloud based system for implementing Blockchain Proof-of-Work. The task revolves around the computation of so called **Golden Nonces**. These are special numbers used in cryptography, and in particular on the blockchain to demonstrate that sufficient work has been done by the mining agents to deserve remuneration. A Golden Nonce in our context, is a decimal number that when appended in hexadecimal to a piece of data known as the *block*, and then put twice through the `SHA-256` hashing algorithm, results in a hash with a particular number of leading zeros (known as the *difficulty d*). An explanation of the alporithm can be seen in the pseudo code in Figure 1.

This task is *embarrassingly parrallelisible*, meaning that the time required to compute the answer is a linear function of the number of units you parallelise to. This means the task is well-suited to being computed in the cloud, as more worker instances can be span up easily and on demand.

This report will explore a solution developed on the Amazon AWS Cloud Platform.

```
for i in range(0, 2**32):
    first_hash = sha256(block+bytes(i))
    second_hash = sha256(first_hash)
    if leading_zeros(second_hash) >= difficulty:
        return i
return -1
```

Figure 1: Pseudo-code of the nonce discovery algorithm

2 A Minimal Implementation

At it's core, this problem requires only a few very simple components to solve sufficiently. One could imagine a completely minimal application consisting of n worker units, all running the exact same algorithm on a subset of the entire possible search space (all integers from 0 to 2^{32}), with some simple way of communicating with, starting, and stopping the workers.

In AWS, this may well look like: n `EC2`(Elastic Compute Cloud) instances, each running a custom `AMI` (Amazon Machine Image) that on machine startup executes a program to discover nonces in $\frac{1}{n}$ of the whole search space. `EC2` is an AWS service for provisioning virtual servers, each instance is a full virtualised machine in the cloud that can be accessed via SSH and can run custom software via it's `AMI`. These instances could be orchestrated and assigned work from a local script interfacing with the AWS API, and pushing and pulling results via a `SQS`(Simple Queue Service) Queue. This is illustrated in fig.2.

An execution workflow of this architecture would involve the user running a local task script. This script would:

1. Divide the total amount of work by the total number of worker nodes specified by the user (n).
2. Push n job specifications to the input queue, instructing each node exactly which subset of the space to search.
3. Start n copies of the same `EC2` instance running the same `AMI`, which would pull from the queue and begin searching for the Golden Nonce within their designated bounds.
4. Continuously poll the output queue until a message is received stating a Golden Nonce was found (or no Nonce exists)

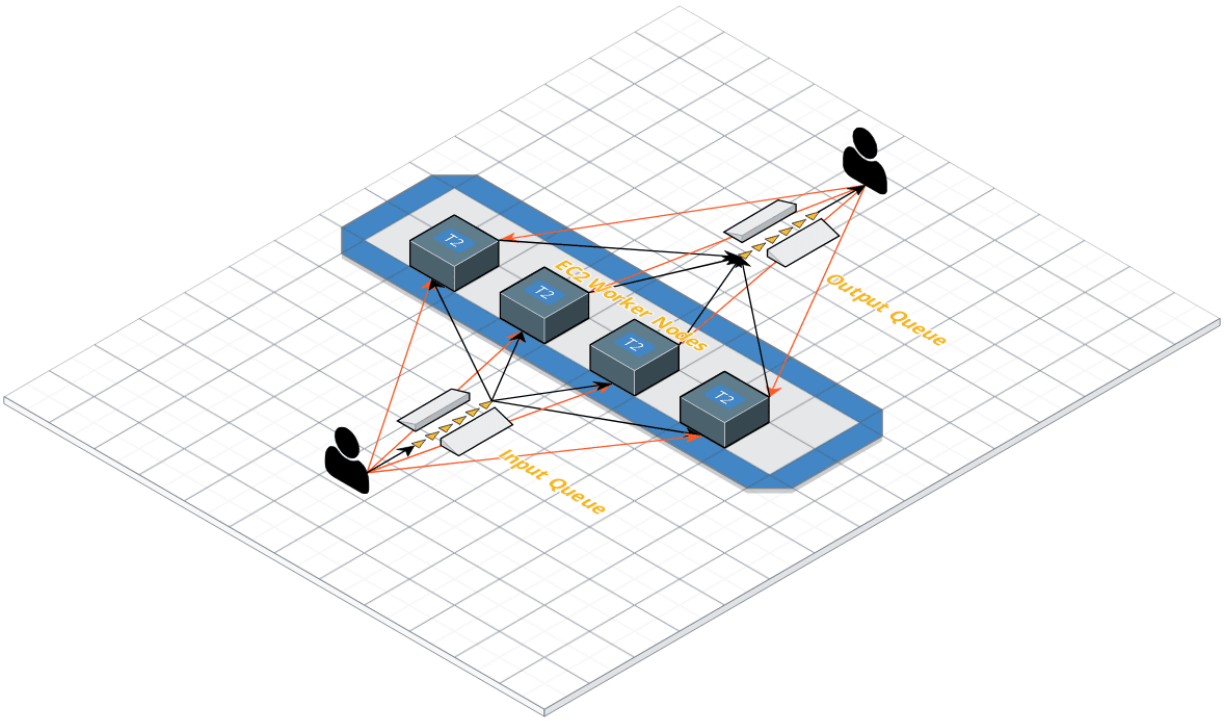


Figure 2: Minimal Cloud Architecture: Red lines = control flow, Black lines = data flow

5. Kill all running instances and purge the input and output queues.

While functional, this architecture is limited. Some downsides include:

1. The entire application is dependent on a single orchestration script, which if it crashed/the user's machine fails, would leave the infrastructure 'dangling' with no attached control process
2. Building the `AMI` involves creating an entire disk image of an operating system. Not only is this time expensive and inconvenient to do for every code change, it's completely overkill as the majority of the content will never change.
3. The application needs to manage everything about the machines running the Nonce discovery; from managing the hardware, to making sure the machines are stopped when they are finished.

3 Containerisation

Containerisation is the concept of running applications in *containers*: Minimal execution units that allow the user to package only the application code they want to run and it's associated libraries etc, without having to directly manage the operating system or underlying machine.

By containerising the Nonce Discovery application rather than creating an `AMI`, it vastly reduces the overhead and complication of changing the application code and pushing those changes quickly to production systems.

In the AWS ecosystem, Amazon provides the `ECR`(Elastic Container Registry) service. `ECR` is designed for storing user's containers and making them accessible to other AWS services, such as `ECS`(Elastic Container Service). `ECS` is a service that allows users to deploy their containers on *clusters* - groups of machines running on the `EC2` infrastructure. This means the user no longer needs to be concerned with the managing of machines and hardware, as `ECS` takes care of this itself, including stopping the instance once the container's main application has finished running.

However, in the default configuration of `ECS`, the user needs to define their cluster at setup time, including how many machines it contains, and the specification of said machines. To help simplify this even further, Amazon released the `ECS Fargate` service in 2018. Fargate simplifies the running of containers even further, allowing the

user to simply specify exactly how many copies of their container need to be ran, and how many resources should be allocated to it.

What's even simpler is starting tasks with Fargate: Once a task definition is created, containing the container desired and it's runtime environment, the task can be started on demand with just an API call from either a user or any other AWS service. Once started, the task can be monitored and stopped early from other API calls, abstracting away any need for managing the task beyond specifying it's parameters and saying go.

4 Decoupling Control and the User

In the basic architecture described in Section 2, all control of the components of the system is based around the end-user; meaning the user themselves needs to manually start each `EC2` instance, push the correct values to the queue, etc. While this is achieved in a standardised, distributable script, the user can easily change it's behaviour to perform any kind of malicious action they desire. It also requires the user to have access to all of the required AWS credentials required to interact with the SDK. All of which are not suitable to a true production cloud system with many users.

If this system were to ever be productionised, the control of the system would need to be decoupled from the user. This can be achieved by using standard interfaces like a *REST API*(Representational State Transfer Application Programming Interface). RESTful services use servers that expose certain ports to listen for incoming HTTP requests (usually with a JSON formatted payload) with some standard communication protocol, process them and perform some action corresponding to the request, then return a response back to the user. By using a REST API, any client that can send basic HTTP requests can interact with the system in a set of manageable, predefined ways. This allows encapsulation of user behaviour, preventing them from performing non-intended tasks on the infrastructure and removing the need for them to know secrets. A service limiting scheme can also be implemented at this level, keeping user activity in an accessible cost range, while also preserving the horizontal scalability characteristic to cloud services.

There are numerous ways to implement this in AWS:

- Hosting a web-server running customised application code to process requests and perform intended behaviour.
- Using a combination of `AWS API Gateway` and `AWS Lambda` to get the same functionality in a serverless fashion.

While manually hosting a web-server gives the most fine-grained control over the configuration of the system, it requires significantly more programming and maintenance effort than the second option, which is what will be adopted for the remainder of this report.

`API Gateway` is an AWS Service that abstracts away a lot of the work involved in managing a web-server within the AWS ecosystem. It allows developers to define API *endpoints* which users may send HTTP requests to, that then trigger some more interesting interactions to happen within the cloud.

The true power of `API Gateway` comes when it is used to invoke `Lambda` functions when the user makes a request to the endpoint. `Lambda` is another AWS service that allows users to define a small block of application level code (A `Lambda function`), that with absolutely minimal configuration from the user will be executed. In this sense, the developer does not have to worry about compiling the program to binary, finding and provisioning a machine to run it on, it's runtime environment, or anything else other than simply writing application level code. However `Lambda` is designed for functions that perform very simple logic - they don't have a lot of compute resources, and by default the application times out after 30 seconds. This makes Lambda unsuitable for the performing the entire Nonce Discovery task.

`Lambda` functions can be invoked by `API Gateway` endpoints, where the context of the HTTP request (timing, JSON body, etc) is passed to the function to help guide it's execution. The function can process this context and perform the appropriate behaviour, and return a response for `API Gateway` to pass back to the user.

This can be applied to the basic architecture by configuring a number of `Lambda` functions to start compute tasks, manage and monitor the queues and to end tasks when required. An instance of `API Gateway` can then be used to accept HTTP requests and then run the appropriate functions to conform to the user's desires. From this, the client the user interacts with needs to be no more complicated than a simple argument parser that sends HTTP requests. This also allows for users from more platforms than the AWS SDK currently supports, as sending HTTP requests is a standard task in almost all programming languages and frameworks.

```
resource "aws_lambda_function" "input_lambda_function" {
    function_name = "Input_Function"

    s3_bucket = "nonce-cloud-lambda-storage"
    s3_key    = "latest/deployment.zip"

    handler   = "input.request_handler"
    runtime   = "python3.6"

    timeout   = 60

    role      = "${aws_iam_role.lambda_exec.arn}"

    environment {
      variables = {
        SQS_INPUT_QUEUE_NAME= aws_sqs_queue.nonce-input-queue.name,
        ECS_CLUSTER_NAME = aws_ecs_cluster.main.name,
        NONCE_TASK_DEFINITION = aws_ecs_task_definition.app.family,
      }
    }
}
```

Figure 3: A sample of Terraform code defining a new lambda function

5 Infrastructure as Code

All of the functionality discussed thus far in this report has required significant effort from the developer in the AWS console to configure all of the appropriate infrastructure. Beyond what has been mentioned there is substantial other boilerplate work, like for example maintaining networking security between all of the components within AWS using `VPC`s (Virtual Private Cloud), or managing the permissions of components to communicate with each other using `IAM` (Identification and Authentication Management), or passing names and parameters between services (for example, the lambda functions and the worker nodes need to know the identifiers of the input and output queues in order to use them). While this is all perfectly manageable within the AWS console, if a component is lost, so is all of it's configuration, or if the developer needed to move to a new AWS account, everything would have to be redone.

These kinds of downsides are what tools like *HashiCorp Terraform* attempt to mitigate. Terraform is a system for provisioning *Infrastructure as Code* - meaning that a user can define all of their AWS components and their interactions in a text document, written in a custom scripting language, and the Terraform system will compile and validate it, then creating all of the required infrastructure in the desired cloud provider. Figure 3 shows an example of the Terraform configuration language used to define a `Lambda` function, including showing how to pass environment variables referencing properties of other infrastructure components in the terraform configuration.

Another great advantage of using tools like Terraform is the ability to version control your infrastructure. If the developer keeps all of their Terraform files in a version controlled repository, it now becomes trivial to roll back the entire cloud system to an arbitrary point in time if a new change breaks anything. The infrastructure now also becomes easy to share amongst team members and can be edited by several people at the same time. One could easily see the Terraform build tools being included in a Continuous Integration pipeline, to automatically deploy any new infrastructure changes on any new, validated changes in the version control system.

6 Final Implementation

With all of these considerations in place, the final architecture implemented for this coursework is presented.

6.1 System Architecture

The entry point of the system is the user. They have access to some client where they can specify the parameters of the execution. This client will then send HTTP Requests that adhere to the strict API Schema for interaction with the system. In this implementation, the client was defined as a Python script running the `requests` library.

The first point of contact with the AWS Ecosystem is the `API Gateway`. It exposes three endpoints to be invoked, and routes user requests to the appropriate `Lambda` function. At this level, any requests that do not conform to the

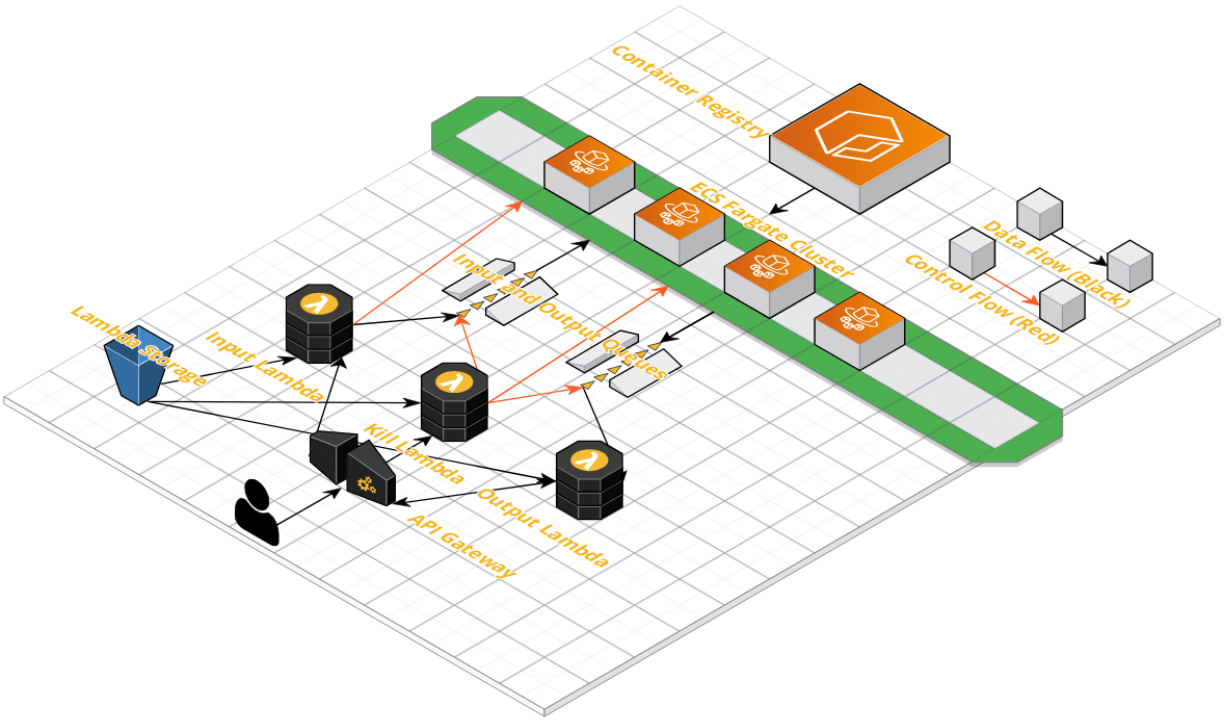


Figure 4: Final Cloud Architecture

API schema are rejected.

Following the execution flow within the system, a request to begin a new nonce search is forwarded to the Input `Lambda` function. All of the latest versions of the function code are stored in an AWS `S3` (Simple Storage Service) bucket. When invoked, this code is pulled from the bucket and executed. Once the parameters passed to the function from the endpoint are validated, if the user has not explicitly defined the number of workers they wish to execute, statistical inferences will be made to estimate the number of workers required to complete the task at a given difficulty, in a given time, with a certain confidence interval. Significant testing was performed to try and make these inferences as accurate as possible (with full testing described in section 7), this was achieved by noticing the linear relationship between the number of workers and the time required to search the entire space (i.e when no nonce can be found). This number was used to calculate the maximum possible throughput of one worker (number of candidate hashes/sec). The full scheme of calculating the required worker number is specified in subsection 6.5. The Input function then starts n copies of the Nonce discovery Fargate tasks, and pushes each of their start and end points of the search space to the input `SQS` queue.

As Fargate provisions each task to begin running on a cluster, the worker contacts Amazon `ECR` (Elastic Container Registry) to fetch the latest version of the Nonce-Discovery Docker container specified in the `ECS` task description. It then needs to 'pull' this container to each of the worker nodes and run it to begin execution. This 'pull' step takes up the majority of the time between the user sending their request and the workers actually beginning execution. As such, this start-up time was optimised as far as possible. The first thing to consider is the size of the container itself; large containers can easily be over 400MB in size and will thus take a long time to pull. The programmer can be intelligent about building their container to help with this.

Docker containers are built in 'layers', one starts with a basic container that already contains some functionality and then adds their own layer on top of this. By choosing a minimal starting layer that doesn't contain too much bloat, the container size can be vastly shrunk. However this still didn't give the performance desired for responsiveness. Currently in beta testing in the Docker ecosystem is an experimental 'squash' feature, which allows the Docker build tools to make smarter inferences about your container and essentially squash it's constituent layers together, making for less repeated data and shrinking the size of the container. This also was a significant improvement. The final optimisation made was to be even more selective of the base image used. Amazon currently maintains themselves several base Docker images, based on `Amazon Linux`. These 'official' images are cached on Amazon's servers automatically, so when used on Amazon infrastructure, especially with `ECR`, a great speed

improvement can be seen.

The Fargate workers then go about searching their subset of the entire search space for a nonce that is *of at least* difficulty d[1]. Once a result has been found, it gets pushed to the output SQS queue, and the worker halts execution.

While this has been happening, the user client will have been continuously polling the second endpoint: GET. The GET endpoint invokes the Get Lambda function which polls the output SQS queue, looking for messages, returning nothing if there's nothing to be found. Once a message does appear on the queue, the Get function returns the message to the user and invokes the Kill Lambda function. This function first stops execution of all currently running Fargate tasks linked with this run of the Nonce-Discovery service, and then purges both the input and output SQS queues so that any messages remaining in those queues do not interfere with later executions.

If at any time the user wants to end the execution early, there is a third endpoint that invokes the Kill function when called. In the implemented CLI client, this is linked to a SIGINT handler, so that the user pressing Ctrl-C will force the system to gracefully shutdown.

6.2 Logging and Monitoring

Important in any system is the ability to log progress and monitor the progress of execution. This is especially important in a system such as this where there are many independent moving parts, as it can become incredibly difficult to track the source of any issues that may arise. To assist with this AWS provides the *Cloudwatch* platform. Cloudwatch is a complete logging and monitoring solution for products inside of AWS. By configuring infrastructure components to have access to *Log Groups*, any default logging or print statements inside of user application code becomes routed and stored in Cloudwatch in near-real time. From inside the Cloudwatch console, the programmer can not only see every log statement from every run of a system component, detailing it's control and data flow, but also aggregated, higher level metrics on the overall health of the system, including for example the number of each type of response (20X, 40X, 50X) the API Gateway instance has returned in a given period of time.

6.3 Permissions + IAM

In order for any single component to be able to communicate in any way with any other component, explicit permission needs to be granted. This is managed by AWS IAM (Identity and Access Management). Every component on AWS must assume a *role* - a set of explicit permissions of what it may do. In order to maintain security within a system the role must be carefully designed to not grant a component access that it doesn't need, potentially exposing a new attack vector. The system described here contains a full set of IAM roles for all components, including custom policy documents detailing exactly what each system has permission to do.

For example; the Lambda IAM role assumed by the Input function has the following policy attached to allow the function Cloudwatch logging access:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "arn:aws:logs:*:*:*",
            "Effect": "Allow"
        }
    ]
}
```

6.4 Terraform

As detailed in section 5, this system was completely developed and provisioned with Terraform, meaning that starting from a blank AWS account, the entire system can be deployed with a single command.

[1]Note, as difficulties greater than d can still satisfy the problem, lower difficulties are exponentially easier to solve

6.5 Indirect Worker Specification

When the user requests the start of an execution, they have option to not directly specify the number of workers they want to run the computation across; instead specifying the maximum time they're willing to wait and a confidence probability of the worker returning an answer in this time.

In order to make accurate predictions on the number of workers required and avoid overspending on extra workers, or not finding answers in the time required, a strong statistical model is required.

The problem can be broken down probabilistically as such:

Assuming the hash function has an approximately uniform probability distribution of each particular hash being assigned to some input, then the probability of finding a golden nonce with difficulty d is the probability of any one bit in the hash being zero (0.5) raised to the power of the number of zeros required, or 0.5^d.

In order to find the golden nonce, the system must try some number of wrong answers k. The probability of getting one wrong answer is $1-0.5^d$, so the probability of getting k wrong answers is $(1-0.5^d)^k$. In order to satisfy the demands of finding a nonce in at most time t, with confidence p, then the probability of not finding a nonce in k steps must be less than p, or $1-(1-0.5^d)^k < p$. It follows:

$$\begin{aligned} 1-(1-0.5^d)^k &< p \\ (1-0.5^d)^k &\geq 1-p \\ log((1-0.5^d)^k) &\geq log(1-p) \\ klog(1-0.5^d) &\geq log(1-p) \\ k &\geq \frac{log(1-p)}{log(1-0.5^d)} \end{aligned}$$

This equation allows a lower bound for k to be determined, the number of wrong attempts needed to try before getting a correct guess with probability p. By combining this with estimations of throughput of a worker (the number of hashes they can compute per second), a good estimate for the number of workers required can be obtained.

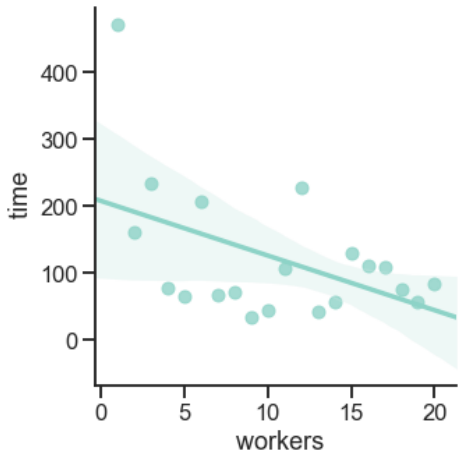
Although this statistic is theoretically sound, in reality there are many more variables that cause noise in the data that makes it less realistic of a modeling of real performance, this variability can come from uncertainties like variation in startup time due to other network traffic, or Amazon unpredictably rate limiting their services.

7 Testing, Results and Analysis

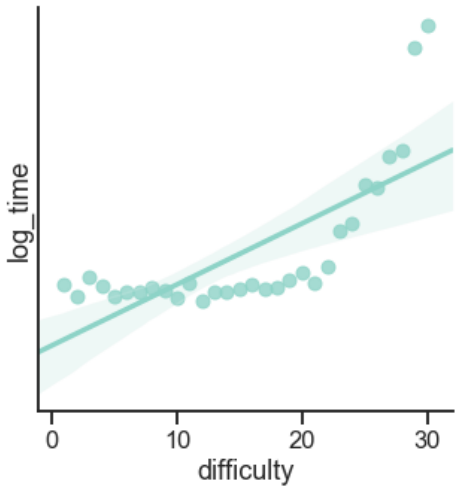
To test the system's performance, and to be able to make accurate inferences on the number of workers required to find a nonce given a difficulty in a specific amount of time, 1000 test runs were performed and timed with numbers of workers and target difficulties sampled from the uniform distribution between 1 and 30. The tests were ran on the example block "COMSM0010cloud". It's worth mentioning that it was discovered that for the given example block, there are **No golden nonces with more than 32 leading zeros**. Attempting to find a nonce with greater difficulty than this results in the workers having to search the entire input space of 2^{32} integers. This can be a good measure of throughput of each worker.

The first two graphs show expected results, time to find a nonce decreases with the number of workers working on a problem. These results were obtained by averaging the time for all runs with the same number of workers, but potentially different difficulties. While not perfectly linear, there is a moderately negative correlation between the two dimensions in the data ($r = -0.47$). More interesting is the observation that time grows exponentially relative to difficulty, with difficulty and log time being strongly correlated ($r = 0.75$), this data was also calculated by averaging all data points with the same difficulty, but with different numbers of workers.

To break this down further, the worker/time and difficulty/time graphs can be split into discrete bins for different difficulties/workers respectively. A number of different values were selected for difficulty/workers and the other variable was plotted at each of these values. This yields interesting results, it can be seen that higher difficulties (> 25) behave as we expect, the time to complete decreases linearly as the number of workers increases. However we see that for lower difficulties than this, the time does not vary very much, almost always taking under a minute to complete. A possible explanation for this is that all of the lower difficulties happen to have nonces very early in the space, so once the initial container startup time has elapsed, it is not long before one of the workers reaches the same early nonce.

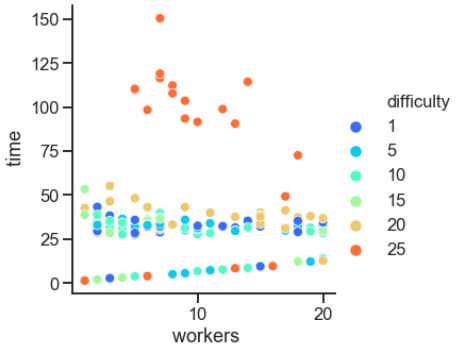


(a) Relationship between the number of workers and time taken (seconds)

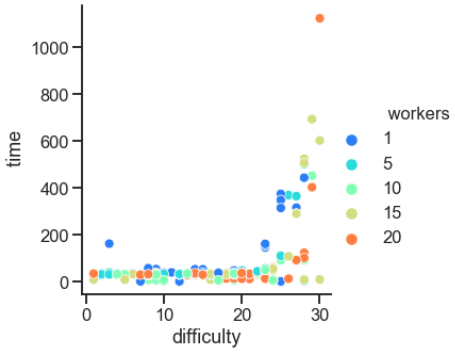


(b) Relationship between difficulty and log time

Figure 5: High level data analysis

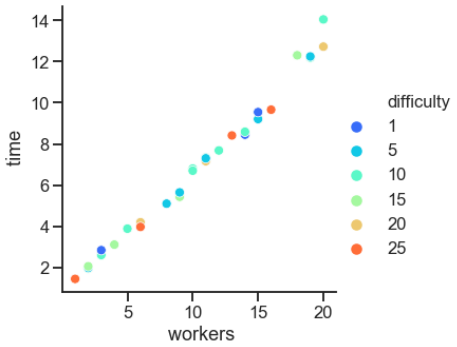


(a) Workers/time for different difficulty levels

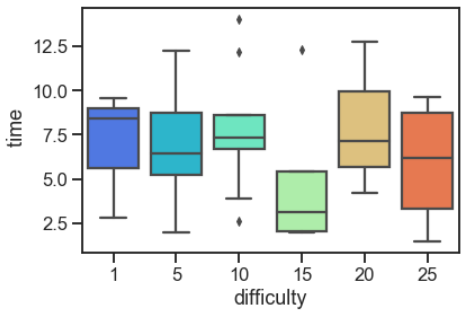


(b) Difficulty/time for different numbers of workers

Figure 6: Deeper data analysis



(a) Worker/Time anomaly



(b) Variance among runs at different difficulties

Figure 7: Data Discrepancies

You might notice an anomaly in the worker/time graph; Namely the linear line growing from $(0, 0)$. This is highlighted in Figure 7a. This is strange, as a linearly *increasing* relationship can be seen between workers and time, up to point where time gets above 20s. This is curious as the container start time is at least 30s, so seeing anything below that seems impossible; let alone realising that adding more nodes takes more time linearly to finish.

A likely explanation for this anomaly is a bug in the testing script used to gather the data. Because `SQS` uses an

Eventual Consistency model of rectifying data discrepancies between replicas of the queues (A single `SQS` queue can be in reality spread across many physical `SQS` instances) to ensure high availability as per the CAP theorem, there is some amount of time that has to pass between queue updates before they can be assumed to have propagated to all replicas [2]. As such, Amazon decided to implement a cool down between attempts to purge(clear all messages from) a queue, to make sure that the last purge has propagated before the next begins.

This is relevant to the anomaly as if there is a case where two test runs happen very close together, the second queue purge at the end of the execution may fail, resulting in some messages being potentially left in the queue until the beginning of the next test. In this case, as soon as the client begins polling the output endpoint, they will immediately fetch the ghost message from the previous run and halt execution. This explains how a result with a sub 20s time can appear. The linearly increasing relationship with the number of workers can be explained by the client blocking local execution until the HTTP request to the input endpoint has finished. The more workers the input function has to start, the longer it will take as it also has to make an HTTP request to AWS internal `ECS` APIs to start each task. The higher the number of workers, the longer the input endpoint request will take, so the longer the program waits before polling the output endpoint. Hence the linearly increasing relationship.

Figure 7b illustrates the large amount of variance among multiple runs at the same worker/difficulty. This can be explained by a) Fargate using a range of different physical machines for different task executions, giving no guarantees about the performance of each machine, and b) Amazon themselves throttling instances that are using too much CPU power using a system called `CPU Credits`, where each instance accrues a certain number of credits per hour, which they can 'spend' on unrestricted CPU power, with rate limiting being used over this amount.

8 Conclusions

This paper has illustrated a cloud-based system for performing the Golden Nonce discovery portion of the Blockchain Proof of Work process. It shows significant merit over the most simple cloud-based system by utilising techniques such as containerisation, serverless computing to decouple control from the user, and the use of Terraform in provisioning Infrastructure as Code to gain benefits of versioning and distribution of infrastructure.

The system performs well, having many optimisations in place to decrease execution time and is able to make well informed statistical estimations at appropriate numbers of worker instances needed to complete a task within certain time restraints. Significant amount of testing (over 1 million SQS messages sent, exceeding AWS Free tier limits) on the system have shown that it is not only reliable and performant, but has highlighted areas in which it could improve, like in the levels of variance in execution time across similar runs of the system, or in errors such as the overlapping SQS purges amounting to some nonsensical data.

[2] While it's impossible to guarantee that it will always reach all replicas, Amazon make some probabilistic assumption that it will reach enough replicas within 60 seconds