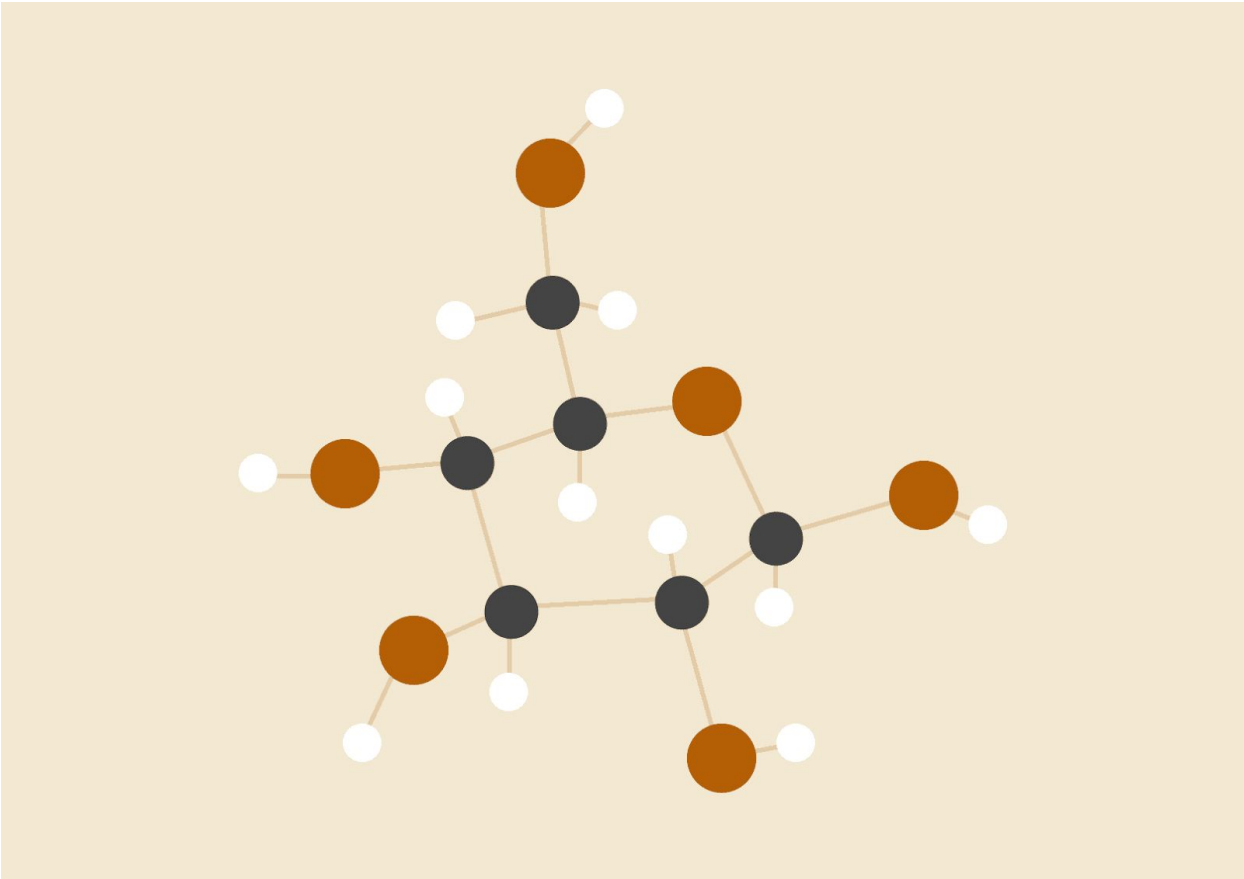
CLOUD COMPUTING PROJECT

INFECTIOUS DISEASE SIMULATION USING AWS



AKASH MALHOTRA

ANANT GUPTA

EMIR NURMATBEKOV

HAROON RASHID

NITHISH SANKARANARAYANAN

UPC, BARCELONA

Table of Contents

INTRODUCTION

ARCHITECTURE

METHODOLOGY
- Agile
- 12 Factor

COMPONENTS
- CDN
- React App
- Cognito
- API Gateway:
- Lambda
- EMR
- Parameter Store
- Cloud Watch
- Kibana
- Simulation App

DevOps
- CI / CD
- Terraform

Time Allocation And Learnings:
- Sprint 1:
  - Akash
  - Anant
  - Emir
  - Haroon
  - Nithish
- Sprint 2:
  - Akash
  - Anant
  - Emir
  - Haroon
  - Nithish
- Sprint 3:
  - Akash

- Anant
- Emir
- Haroon
- Nithish

Sprint 4:

- Akash
- Anant
- Emir
- Haroon
- Nithish

Sprint 5:

- Akash
- Anant
- Emir
- Haroon
- Nithish

INTRODUCTION

The team set out to build a SaaS for running infectious disease spread simulation. This product would be made available to governments and think tanks to help making policy decisions around lockdowns and contact tracking apps. There is a need for this service to be cheap and elastic. We leverage the cloud for this to scale out and in as needed. The project in essence takes in parameters from the user and runs a simulation job in scaled out fashion while being transparent to the user.

The application is exposed as a web application accessible over the internet and hence requires minimal configuration from the users running our simulation, the application created is responsive and accessible from any device which can run a JS enabled browser.

ARCHITECTURE

The architecture from the ground up is built to be highly available and scalable, we rely heavily on the cloud services specifically AWS for this; moreover we have an elastic architecture that can scale out and in as needed thereby saving costs. Below we present insights into how our initial architecture looked like and how we had to adapt based on feedback from the product owner in our review meeting.

Our architecture from initial conception to the MVP has not changed much, the authentication layer and the front layer remain as is, but we had to react to feedback we had received during our initial sprint meetings that the product owner wanted a scale out processing of per simulation, our proposed architecture scaled out for many simulations but each simulation was to still run in a single node. After careful consideration we decided to make our simulation app as an Apache Spark application so that we could focus on the simulation logic and not deal with the creating a distributed processing framework. This is reflected in our new architecture by the use of AWS EMR which is the compute platform for simulations. Note that since we had to pivot to a scale out model per simulation we had to drop the requirement of being fault tolerant for the MVP; although in incremental releases this feature will be added. This for us was an acceptable compromise as we do achieve a certain degree of fault tolerance by virtue of using Spark, as Spark auto retries failed jobs and the number of retries is a configurable property but this is not ideal as some failures could be the result of bad configuration from the user or an error in the business logic and hence the simulation would have to

be retried on a new version of the app. This is where some of the niceties like dead letter queue offered by a queuing system like SQS would play in.

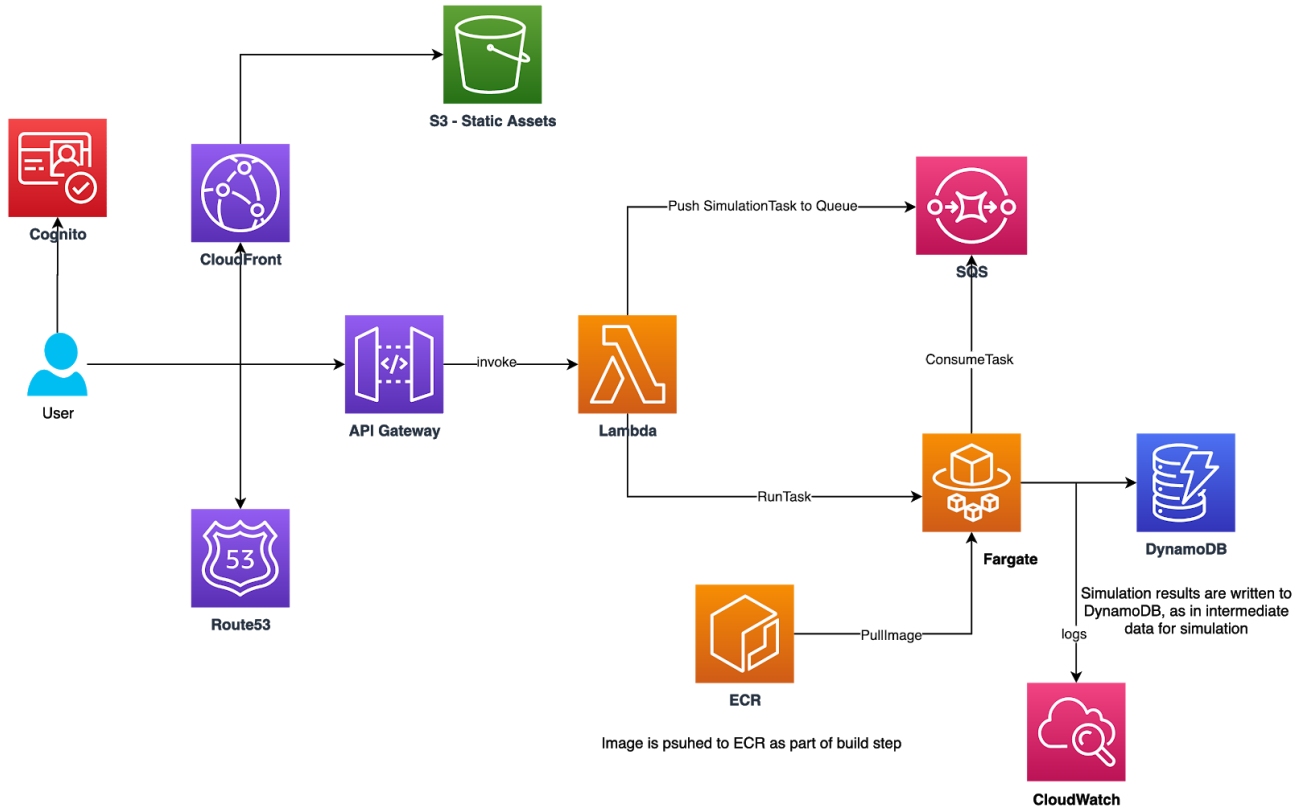


Figure 1: Initial Architecture

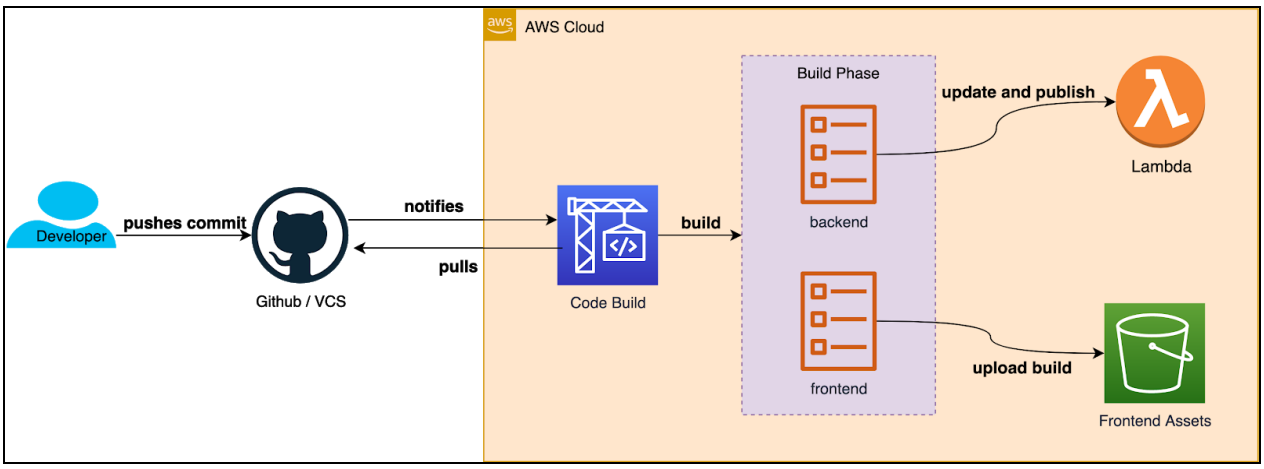


Figure 2: CI / CD Flow

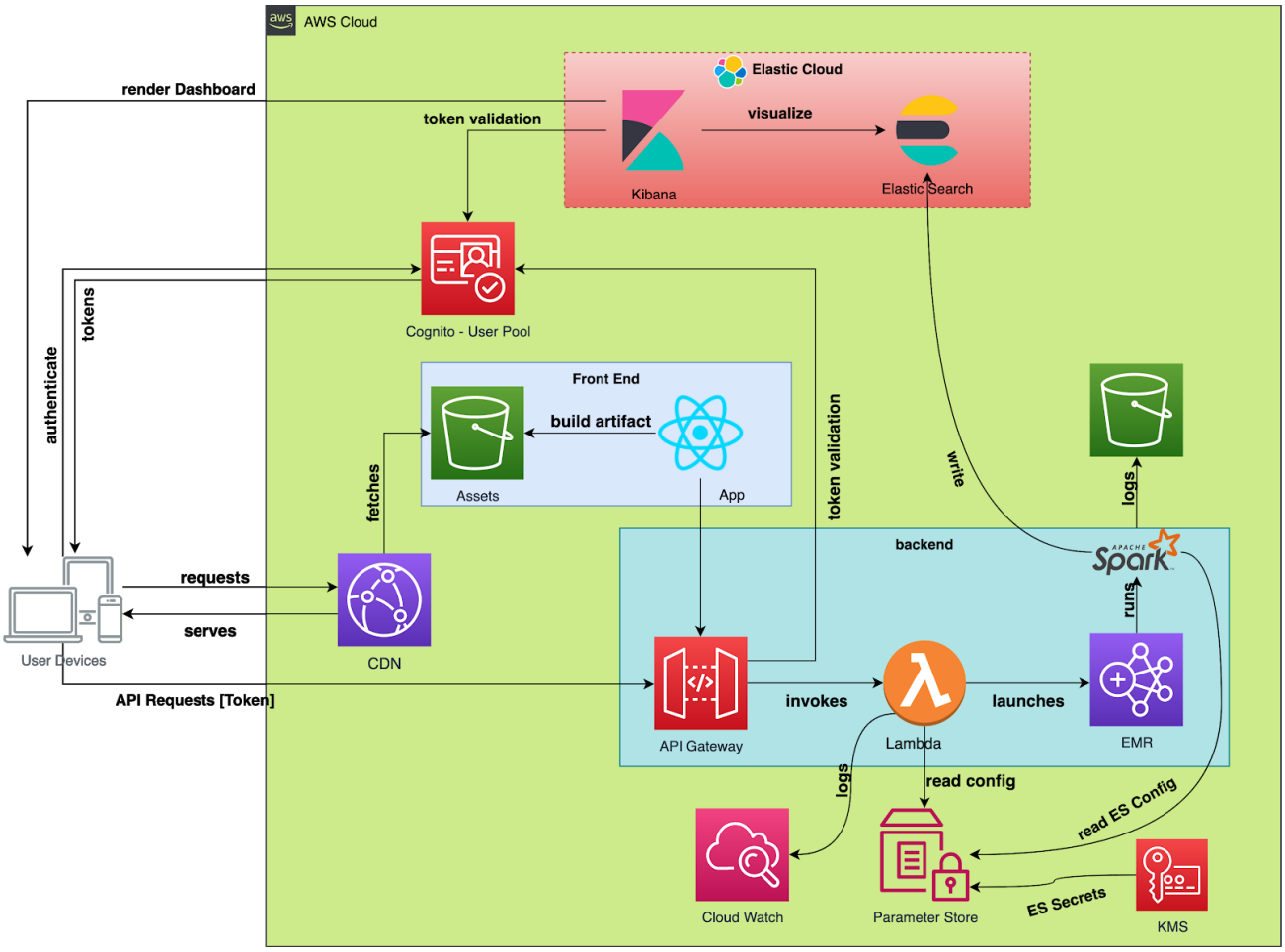


Figure 3: Current Revamped Architecture with Spark

Another notable missing feature is the absence of a tracking and notification mechanism for users to know the status of their simulation. For the MVP we wanted to focus on devops and agile software development methodologies and hence the timeline for these features are pushed. The architecture we have is by no means a complete one but it reflects our current state, and time invested in best practices like Infrastructure as Code, CI/CD, unit testing which ensured that we can churn out features at a faster pace in the future.

Our current architecture is designed to scale out as needed to support demand. Before we look at the different components involved in depth, the entire flow would look as follows:

1. User types in our app URL

    a. This URL for now is the Cloudfront URL subdomain, but ideally we would buy a domain name and configure Route53 for custom domains.
2. DNS requests are resolved and the request reaches cloudfront.
3. Cloudfront serves a cached copy of the page in location closer to the user requesting the application.
    a. If no data is found is cache, cloudfront fetches it from the S3 bucket and caches it.
4. The single page application (SPA) is rendered by the user client.
5. Users provide their credentials.
6. SPA authenticates against AWS cognito and on successful authentication tokens provided are stored in client local storage.
7. Users enter a page to specify simulation parameters.
8. Button click triggers request including the token obtained in step 6 to the API Gateway.
9. API gateway authenticates the provided token against cognito.
10. On successful authentication, API gateway invokes our lambda.
11. On invocation lambda reads configuration from parameter store, if configuration values are not already cached.
12. Lambda launches an EMR cluster with the specified configuration.
13. Simulation results are written to ElasticSearch running in ElasticCloud. This is subsumed by AWS as ElasticCloud instances are managed and run within AWS cloud.
    a. Spark application reads ES secrets like username and password from Parameter store and decrypting with KMS
14. User requests a kibana for the dashboard.
15. Request is authenticated against Cognito using OpenID connect.
16. User views dashboard.

METHODOLOGY

Agile

We follow the essence of agile development methodology, in which we have weekly sprint meetings and daily updates to understand the state of tasks we have picked up and for calling out blockers and help required.

We use Github Projects for project management making it easier for us to integrate with and automate task tracking. For example an issue created is automatically sent to the

backlog lane and when an issue is assigned to a team member, the story wall also reflects this. Likewise, when a PR is made against an issue, the task moves to in-progress lane, so make a draft PR as soon a task has been picked up to give others visibility and also request for reviews. Finally when a PR is merged, the card or task moves to 'Done' lane. Reviews of PR are done religiously and only merged after all the feedback is taken into account.

12 Factor

I. **Codebase**:

"One codebase tracked in revision control, many deploys"

A. We use Github for version control and went with a mono repo approach (one codebase for frontend and backend) for operational efficiency. A mono repo ensured that everyone in the team was aware of the commits flowing in and made issue tracking and project management a breeze. In the long run if this becomes untenable we always have the option to move back to a multi repo strategy.
B. CI / CD pipeline ensures that commits get deployed, giving rise to one codebase and many deployments paradigm.

II. **Dependencies:**

"Explicitly declare and isolate dependencies"

A. For the python backend we isolate our code base using requirements.txt file and use pip to install it, which is the recommended workflow in the python ecosystem.
B. For the ReactJS frontend we use npm as a build as a dependency tool, which again is the standard in the JS ecosystem.
C. For our Spark module, since it is written in python, we use a separate requirement.txt for tracking dependencies only for this app.

III. **Configuration:**

"Store config in the environment"

A. All our code resources read configuration from a configuration store, in our case we use AWS Parameter Store.
B. The path to read the config in the parameter store is injected via environment variables.
C. We don't have any secrets in our application as all permissions are managed via IAM roles and policies, but if we did have it, the application can contact KMS to decrypt these values.

IV. **Backing services:**

"Treat backing services as attached resources"

A. Resources can be switched with at will and can be deployed with minimal intervention.
B. Backing stores in our case are S3 buckets and we can switch them out with ease.

V. **Build, release, run:**

"Strictly separate build and run stages"

A. We have defined build, release and run processes.
B. We use Code Build for our build process, a web hook triggers a build as soon as a commit is pushed or PRs are merged with the master branch.
C. After build, the release process includes deploying artifacts to AWS lambda with a new version.
D. For the front end, the build assets are pushed to S3 bucket which is version controlled.

VI. **Processes:**

"Execute the app as one or more stateless processes"

A. All of our components follow the process model for scaling and hence are stateless.
B. For example, we have an auto scaling group configured for Spark which due to its in-memory architecture and computing model works without any problems.
C. Since we use a lambda for our processing, our backend has to be stateless.

VII. **Port binding:**

"Export services via port binding"

A. We don't have any components that are exposed using a service, as we use API Gateway for lambda innovation using ARN.
B. In Spark we can configure port numbers for configuring history servers and other components.

VIII. **Concurrency:**

"Scale out via the process model"

A. We will use a scale-out option when the load increases due to the increased number of points in the simulation using the Spark compute model.
B. For the backend, since we use lambda and they scale using processes and we can scale out and in as need be.

IX. **Disposability:**

"Maximize robustness with fast startup and graceful shutdown"

A. Use of lambdas guarantees we have fast startup and shutdown.

X. **Dev/Prod parity:**

"Keep development, staging, and production as similar as possible"

A. For now we don't have different environments for development, staging and production given the timeline for MVP.
B. But since we use IaC paradigm, setting up different environments is simple.

XI. **Logs:**

"Treat logs as event streams"

A. All of our components write the log to stdout and from there redirected to Cloudwatch.
B. EMR clusters redirect their logs to S3 buckets as a process and this can then be redirected to Cloudwatch in the future.

XII. **Admin processes:**

"Run admin/management tasks as one-off processes"

A. We don't have any admin process and all configuration is done using requirements file or in the case of EMR using a bootstrap script.

COMPONENTS

In this section, we attempt to provide a snapshot of the tools we use to solve the problem statement and how we use it. We follow the Architectural Decision Record model to present our arguments and reasoning. Although ideally we should have captured this in the Version Control System at the point decisions were taken, we thought that this format was best to capture our thought process and present them for this report. In the future, we would make it a point to follow this process rigorously to help newcomers get acquainted with the project easily.

CDN

Status: CloudFront was used as our CDN solution to provide fast and secure delivery of frontend assets.

Context: Content Delivery Network (CDN) is a geographically distributed group of servers which work together to provide fast delivery of Internet content [2].

A CDN makes the transfer of assets needed for loading Internet content(HTML pages,

javascript files, stylesheets, images, and videos) fast. The majority of the web traffic of big companies like Facebook is served via a Content delivery Network [2].

In our case, we see the web application to be accessed by all over the world, given the nature of our service, we want to make content available as close to the users as possible.

Decision: We chose Amazon CloudFront as our CDN solutions for the following reasons:

- It is fast and global. CloudFront is largely scaled and distributed with 216 points of presence.
- It is secure. CloudFront provides both network and application level protection. It has built-in protections such as AWS Shield Standard providing us DDoS protection available for free. Cloudfront can be integrated down the line with AWS WAF to protect against common web attacks like SQL injection and admin page scanning. Further we can restrict access to static IPs if we decide to open it up only to specific paying organizations.
- Highly programmable. CloudFront features are customisable for specific applications and also supports integration with other tools. Possibility of using Lambda@Edge to push backend logic closer to the user makes the whole package alluring.
- Integration. As we use S3 to host our static assets we can use origin identity access policies to restrict that only cloudfront can read from our S3 bucket and the bucket does not need to be public, this means in the future we can securely add geo restrictions if need be via cloudfront and also this ensure cached data is used thereby saving S3 data transfer costs.

Consequences: CloudFront helped us to improve the performance of our web application. Even though it is just a project app which won't have huge traffic, with CloudFront's scaling possibilities and widely distributed network, our app will work fast for users accessing from any part of the world. Also, we don't have to worry about the security issues thanks to CloudFront's high level of protection.

React App

Status: React was used to build a single-page frontend web application.

Context: There was a need to choose a framework which is not too complicated to pick up for someone who does not necessarily have experience in web development or javascript in general. React is time tested and has a wide community. Moreover, Libraries to integrate with our authentication provider is another important

consideration. As react is modular we can reuse components available as open source with minimal integration overhead.

Decision: Initially, our plan was to leverage AWS Amplify with React to create the frontend. AWS Amplify is a development platform for building mobile and web applications. It also allows you to build a single-page web application. However, it uses AWS CloudFormation for provisioning required infrastructure like Cloudfront and CodeBuild pipelines, and moreover Amplify "Getting Started" guide recommends creating a role with Administrator access. This is not possible in many organizations, albeit it was to make the onboarding process easier, we thought it takes control away from the user as resources are created in an opaque fashion. Since we were already using Terraform as our IaC solution, we opted to create infrastructure required to deploy react ourselves following react documentation. This gives greater control and visibility.

The decision made in favour of React is not limited to being able to use Terraform. Here are the advantages of using React:

- Declarative. In React we declare that we want a header with a message, we don't tell it how to do it.
- JSX. No need to separate HTML templates from Javascript, because both are part of the view concern.
- Render on the server. Server side rendering is one of the most important features which makes a JS app run faster.

Consequences: React allowed the team which had no significant frontend development experience to create a web-app in a time bound manner; once all the infrastructure was set up.

Cognito

Status: AWS Cognito was used as an identity provider and relied on for authentication thereby securing authenticated pages of our web-app and facilitating token based authentication in the backend.

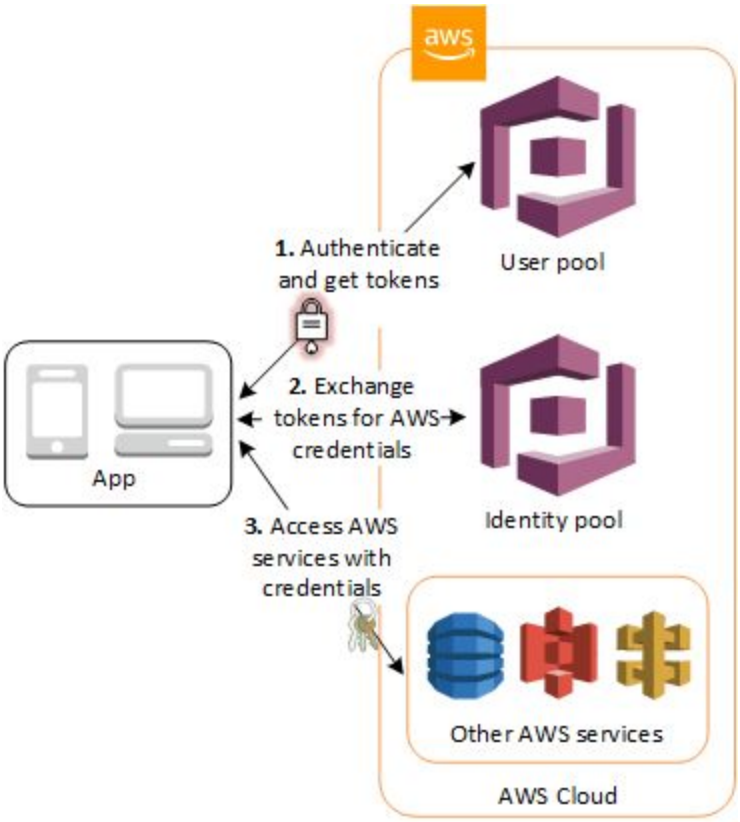
Context: To run a data and computationally intensive application, a lot of cloud resources are required and cost a lot of money. Authentication is needed to run our simulation in a secure way and to prevent any potential attackers from overusing our cloud resources which can potentially cost thousands of Euros. Authentication is a hard process to get right, as the risk of getting it wrong has compound effects and in extreme cases would make us liable for legal proceedings, hence we attempt to transfer risk and

technological know-how to a well known and audited IDaaS solution.

While there were many authentication as a service options available like Auth0 and IBM Security Identity, we opted for AWS Cognito as it is a fully managed service where we don't need to worry about the low level implementations of the methods used for authentication.

Decision: We opted to use AWS Cognito for the following reason:

1. *Less steep learning curve:* It is a fully managed authentication as a service which takes care of all the details of token based authentication. So, a much less learning curve is required by us, and we can focus on the development of the application. Below is shown how it works:



2. *State of the art:* It uses OAuth 2.0, which is very secure and fulfills our needs.
3. *Scalability:* With its user directory, called user pools, it is highly scalable, to even millions of users.
4. *Sign in using federation:* It supports social and enterprise federation. So, if in future we want to facilitate sign in social media websites like facebook, we don't have to make many changes to our authentication infrastructure.

5. *Integration:* It easily integrates with other AWS services like API Gateway and also with React and Kibana, which we used for visualisation.
6. *Cost effective:* The cost of using Cognito is minimal as it uses pay per use model. For 50,000 to 100,000 monthly active users, cost is 0.0055$ per month.

Consequences: Cognito helped us to implement authentication in our webapp quickly and gave us a gentle introduction to the world of authentication and security. However, one of the challenges it introduced is that all the resources in the website have to be individually authenticated by Cognito.

API Gateway:

Status: AWS API gateway is used to expose backend resources.

Context: With our cloud infrastructure running the app, we wanted users to have a front door into it for users to send and receive data. If we have an API gateway, not only can we regulate and monitor the user requests in case of errors but also we can provide a better UI for users to do so. Such a gateway also helps us to decouple the components of our cloud infrastructure. AWS API gateway was our best choice to accomplish this task.

Decision: We had various options for choosing API gateway like Azure API gateway, which is more suitable if we were using Azure Cloud services, and Express API gateway, which is an open source service. However, we choose AWS API Gateway for the following reasons.

1. *Efficient API development:* It supports RESTful API, which suffices our needs. It also supports API versioning, which could be useful for us in the future, if we want to integrate new features in our app for AB testing.
2. *High Performance:* With so many edge locations of AWS, the latency in the API calls is the lowest possible. Most often we deploy the API gateway in the same region as our compute resources. Scalability is also not an issue with the API gateway as it is fully managed by AWS.
3. *Cost effective:* It is a very cost effective solution, in which we pay per use. It could be as low 0.90$ per million API calls.
4. *Easy monitoring:* With its easy integration with AWS CloudWatch we could easily visualize and monitor the requests by users and trace the errors when they occur.
5. *Security*: With AWS IAM roles, the access to AWS API Gateway can be controlled and be only given to the developers in charge for working on it. With its OAuth 2.0 support, it can also be easily integrated with Cognito, the authentication service we are using to provide a secure gateway into our cloud resources to our

customers.

Consequences: AWS API gateway was fairly easy to use and was successfully integrated with Cognito and Lambda. It also helped us to learn how state of the art serverless architectures work.

Lambda

Status: AWS Lambda was the microservice used to take care of the setting up computing resources in a serverless fashion.

Context: From ground zero we were in favour of proceeding with a serverless architecture. The reasons being we wanted to concentrate on writing the code logic and worry less about the computational intricacies. The plan was to offer 'Function-as-a-Service' in the form of a web application.

The possible options which were considered were AWS Lambda, AWS ElasticsBeanStalk(EBS) and AWS Elastic Container service(ECS).

Decision: AWS Lambda was the first choice which popped in mind due to the fact it was depriving the task of managing the infrastructure to manage while spinning up a new Lambda. Secondly, using Lambda follows the 'Only pay for what you use' pricing model. Since the simulation execution was product owner dependent, it was not going to be running 24/7. Thus, ElasticBeanStalk did not make much sense due cost inflation. Additionally, the product owner expected the model to scale out as in the when required on the complexity of the simulation. With Lambda's 'Auto Scaling feature' this request was not an integration issue.

Lambda allows you to bring your bring code and present them as 'artefacts' on which the computation is performed. In our case we scripted the invocation of lambda and presented the zipped code as an artefact.

Finally, as mentioned earlier we are using 'API Gateway' for authentication, response code mapping, routing HTTP request, etc. As it can be safely inferred that the API Gateway helps provide a great interconnectivity with other microservices within the AWS environment. Lambda connects very well with API Gateway as compared to ECS.

Consequences: The integration of AWS Lambda was a smooth process and undertaken via a script making it further simpler.

EMR

Status: Integration was done as planned to aid the parallel processing for the simulation.

Context: To run the simulation as fast as possible, we intended to parallelize the processing by making use of a cloud data platform for processing the vast amount of data. This requirement popped during the sprint cycle being executed in week 3, where the product owner put forward the request of scale out processing per simulation request. To avoid falling in the trap of nitty gritty details during setting up Spark Clusters and to focus our attention on enhancing the features of the app, we went with a cloud solution for running Spark.

Decision: Since the simulation is web application, traditional on-premises solutions were a complete no-no and focus was to find an appropriate cloud service suiting our use case. Among different options in this ever-developing cloud infrastructure, we decided to move forward with AWS EMR since from the bottom up we have been invested in integrating a variety of AWS services, thus support for this integration would be an easier option as compared to the others. Secondly, we envisioned the simulation to have a diminutive life cycle, executed occasionally as per the product owner's request. AWS EMR delivers the cloud feature of 'elasticity' as the number of instances can be increased or decreased automatically using Auto Scaling (which manages cluster sizes based on utilization) and you only pay for what you use. In our case we set up auto scaling to be dependent on the available free memory in the cluster if when it falls below 25% of the total memory, we scale out and use *spot instances* to reduce costs. Additionally, AWS EMR maintains the storage via S3 buckets and computation independently making the scaling process easier. With no headaches about infrastructure provisioning the time taken for deployment would be reduced to a couple of minutes.

Consequences: This approach actually helped us to match the sprint deadlines and deliver a product satisfying the product owners expectation supplementing our theoretical knowledge of Apache Spark gathered from other academic courses.

Parameter Store

Status: Backend reads configuration from AWS parameter store.

Context: To adhere to the 12 factor methodology and as a best practice it is essential to separate code and config, this helps us update configuration without requiring redeployments. For example, changing the database URL to point to a new instance does not require deployment with changes, rather updating the new value in the config store

is enough. Moreover, it encourages best practices like not checking in secrets into version control.

Decision:

1. There is a need for a solution that will abstract the complexity of maintaining different versions of a configuration, this is needed for audit purposes and to rollback any breaking configuration changes. Further the solution has to be highly available, as configuration stores can be constructed as a single point of failure.
2. A simple way to achieve this is to use a highly available database as a configuration store, but we will have to deal with operational details like version management and value encryption, moreover granting permissions to only a select rows which would be used by them is hard.
3. In AWS Parameter Store we find all the functionalities we want in a configuration management system, it's highly available and manages different versions of configurations, moreover it integrates with AWS KMS so that secrets can be encrypted and stored as values.
4. The application can then query Parameter Store for configuration under a particular path and this is managed via IAM roles and policy and if the application also has permission to the KMS key, encrypted values can also be read.

Consequences: We integrated Parameter Store successfully in the backend lambda and used it to control which version of the simulation spark application and what instance types are to be used for. We are ideating on options to parameterize configuration for the frontend and feel that injecting these values during build time is the only option for now.

Cloud Watch

Status: Components where possible ingest logs as streams into cloudwatch.

Context: Observability is a key factor needed to analyze behavior of applications and identify errors when they occur. Logging is a crucial step to achieve observability.

Decision: We decided to use AWS CloudWatch for our monitoring and observability needs, other alternatives we considered are ElasticSearch and Splunk; ElasticSearch is an open core search engine built on top of Apache Lucene and is very good at text searches, Splunk is proprietary solution well known for their Splunk Processing Language and is a full feature suite for logging and monitoring, whereas we shy away from them because of cost reasons. Important considerations while making a choosing monitoring solution

is completeness of the solution and accoutrements like alerting, dashboarding. And in these categories CloudWatch, it provides logging for both infrastructure and application, most AWS components have native integration with CloudWatch making integration easy, shipping logs to an external service is easier said than done and will often involve reading from CloudWatch and shipping to an external service.

CloudWatch also provides an alerting mechanism using CloudWatch alerts in production that would be beneficial to track repeated failures of lambda and wake up on support tech. It is a managed solution and insights services from CloudWatch facilitate searching logs interactively using a custom query language. Further creating dashboards to monitor metrics is an easy process in CloudWatch. Importantly cloudwatch supports rolling dumping of old logs into cheaper storage like S3, this functionality is referred to as tiered storage and is essential to keep costs low and performance high.

Consequence: We configure all our components to dump log streams to CloudWatch where possible, one exception to this is EMR, wherein the logs are written directly to S3 given the sheer volume of logs that could be generated.

Kibana

Status: For our visualisation needs, we used Kibana dashboard by Elastic. It was successfully integrated into our simulation web-app.

Context: It is not enough to write a program to do simulation, unless we can show it to the user in an insightful manner. For this, we needed to use a visualization tool to display the results of simulation in a user-friendly way. We used Kibana to accomplish this.

Decision: With many visualisation tools available in the market, the main ones being AWS QuickSight, Tableau, and Kibana, all of them capable of showing beautiful plots, we opted for Kibana for the following reasons:

1. *User friendly python API:* Kibana loads data from ElasticSearch Index. With ElasticSearch API for python, it was quite straightforward to send our simulation data to ES Index. Also, with the Kibana dashboard API we could automate the generation of dashboard programmatically. Our initial choice was AWS QuickSight, since we are using all the other services from AWS but it was more complicated to build a PoC with it, owing to its low-level api and no automatic integration with storage services like S3.
2. *Support Vega:* Vega is a visualisation language which specifies the property of the plot in a json like format. With Kibana we could write scripts to generate the plots

with the ElasticSearch data. This would be useful for us, we want to automate the generation of the plots and deployment of the dashboard.

3. *Real time dashboard:* To show our simulation plots, we needed to build a real time dashboard with a high refresh rate. With Kibana, we could set the refresh rate easily to 1s or more. On the other hand, with QuickSight, the refresh rates are typically available for 1 hour to more.
4. *Integration with Cognito:* Kibana can easily connect with Cognito as it supports OpenAuth 2.0.
5. *Integration to react app:* Kibana can generate a html script to embed an iframe into a react app, our chosen tool for front-end, so our job becomes much easier.

Consequences*:* We could successfully integrate Kibana with our react web-app to show real-time simulation plots. One challenge was learning to write Vega-lite charts in Kibana but with its extensive documentation, we could accomplish this in a short time.

Simulation App

Introduction

The simulation App is a stand-alone python program that was built to simulate the spread of the infectious disease. The simulation is driven by different parameters that helps us understand the spread of the disease under different environments. In particular, the simulation App supports following parameters:

- **simID:** ID for the current simulation instance.
- **N**: Population size to run the simulation.
- **x_limit, y_limit**: Defines the size of the 2d world in which the simulation is run.
- **dist_limit**: Determines the distance threshold that can infect two people.
- **mov_rate**: Sets the rate of the motion of the people in a 2d-world.
- **kill_prob**: Probability with which a person with a disease dies, determines the deadliness of the virus.
- **hosp_prob**: Probability with which a person with a disease is hospitalized.
- **status_type**: Role distribution of the population. By default, 70% of the population is working, while 10% are Children, students and Old/Retired each.

The output produced by the simulation is shown in the figure below. The snapshot is taken on Day 32.

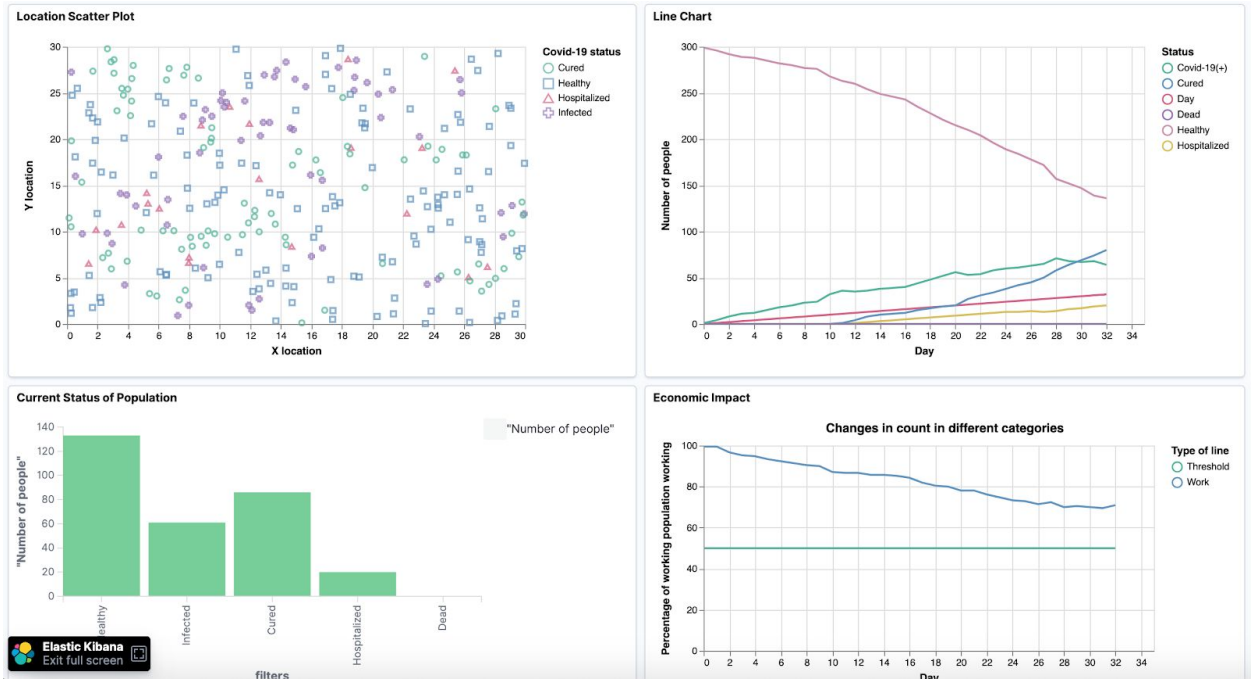


Figure3: Simulation Snapshot at Day 32

Schema and DataStorage

The data for the simulation is stored in the pandas data frames to keep track of the state of the world as well as the historical statistics to understand the dynamics of the pandemic. The choice of pandas dataframes have been chosen keeping in mind the scalability of the simulation by importing the App to Spark in the later stages. Spark has support for dataframes and the conversion logic from pandas to spark can be done fairly easily. The two data frames used are *covid_df* and *stats_df*. As mentioned earlier, the covid_df is used to keep track of the current day status while the stat_df keeps the historical information as the simulation moves forward in time. Thes structure/schema of the data frames to store the relevant information is shown below:

X (float)	Y (float)	Covid-19 (String)	Day (Int)	status (String)	work_hours (Float)
2.112	3.123	Positive	3	Student	0

Covid_df dataframe: Schema and an example data row

simID (string)	Day(int)	Healthy (int)	Covid-19(+) (int)	Hospitalized (int)	Cured (int)	Dead (int)	Work (int)
"sim_id"	4	285	15	0	0	0	96.67

stats_df dataframe: Schema and an example data row

The plot shown in figure 3, contains three different subplots. The first subplot is generated using the covid_df dataframe while the plots 2 and 3 are generated using the stats_df.

Scaling the Simulation

The biggest limitation that we identified in the simulation is the computational bottleneck to run the simulation over large data points. For this, we decided to spread the across the multiple nodes to scale it to a large number of datapoints. It was decided to use spark to run the simulation over multiple nodes, because of its good support in python and dataframes as well as the availability of excellent documentation and tutorials.

Apache Spark

Apache Spark is a large-scale distributed processing Engine to build data processing applications at scale. It is currently one of the fastest-growing data processing platforms, due to its ability to support streaming, batch, imperative (RDD), declarative (SQL), graph, and machine learning use cases all within the same API and underlying compute engine. Spark can efficiently leverage larger amounts of memory, optimize code across entire pipelines, and reuse JVMs across tasks for better performance. Spark is one of the most widely used data processing engines used today, and this was one of the main factors that led us to choose spark to build our simulation at scale.

The performance comparison between Spark and Hive (another very popular data processing Engine built on top of Hadoop) has been shown in the figure below. The figure has been taken from the Facebook Engineering experiments on 60+ TB data processing jobs[1].

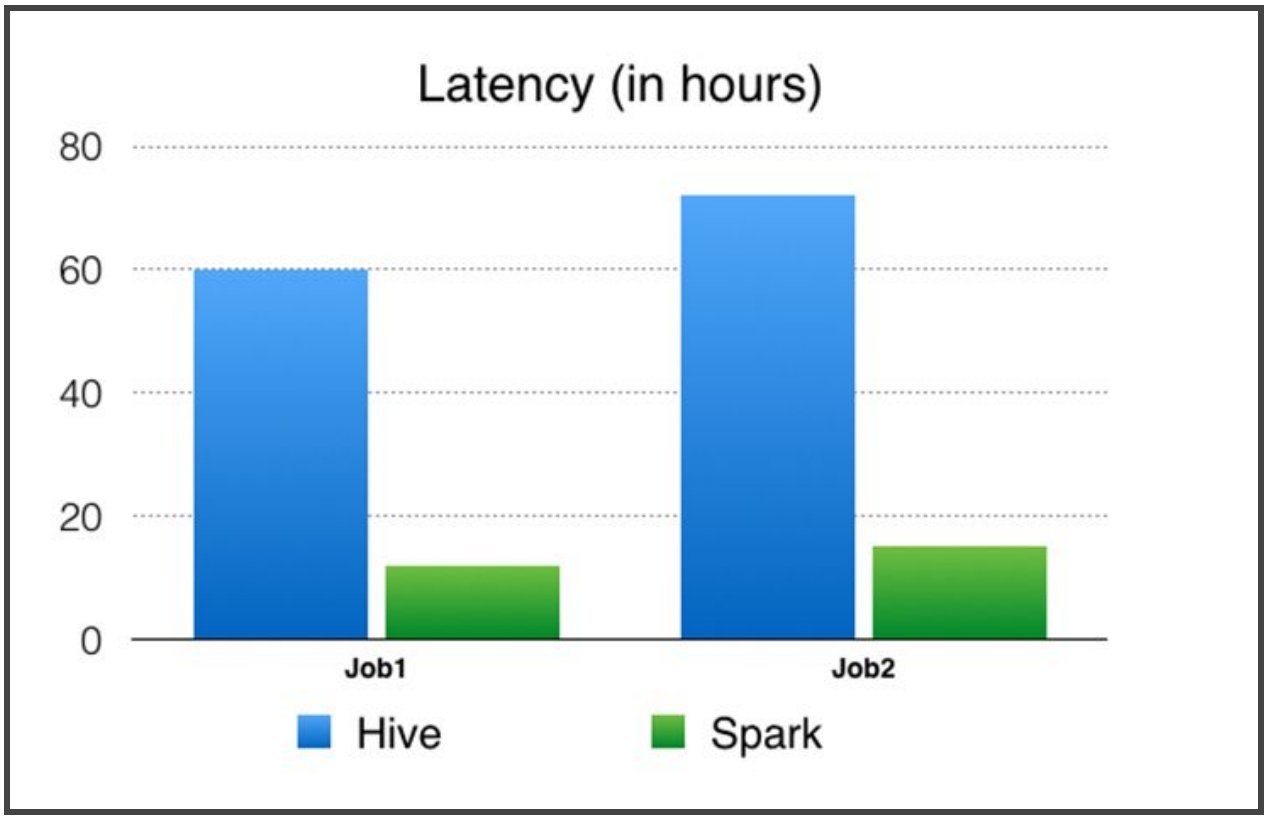


Figure: Performance comparison between Spark and Hive

Spark internals for Scalability

The scalability of spark applications is attributed to Resilient Distributed Datasets (RDDs). an RDD is a read-only, partitioned collection of records. RDDs can only be created through deterministic operations on either (1) data in stable storage or (2) other RDDs.

These operations are called transformations to differentiate them from other operations on RDDs. Examples of transformations include map, filter, and join. RDDs do not need to be materialized at all times. Instead, an RDD has enough information about how it was derived from other datasets (its lineage) to compute its partitions from data in stable storage. This is a powerful property: in essence, a program cannot reference an RDD that it cannot reconstruct after a failure. Finally, users can control two other aspects of RDDs: persistence and partitioning. Users can indicate which RDDs they will reuse and choose a storage strategy for them (e.g., in-memory storage). They can also ask that an RDD's elements be partitioned across machines based on a key in each record.

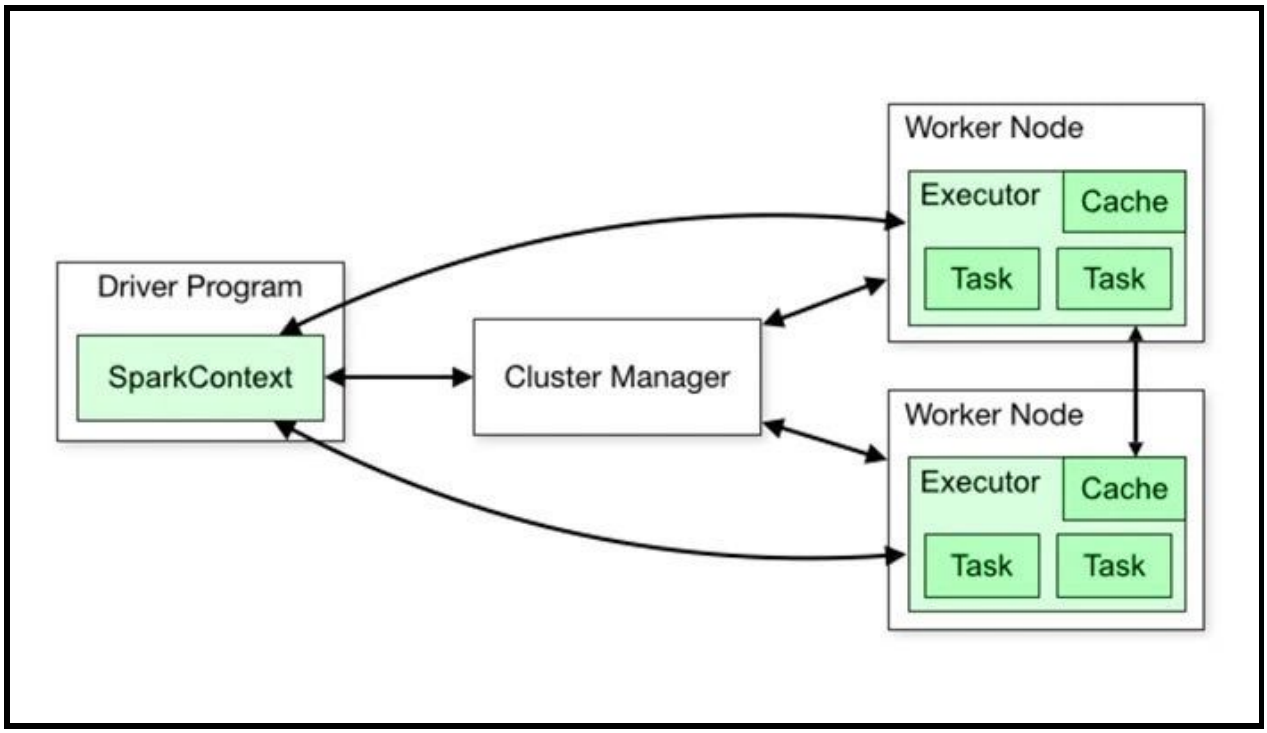


Figure: Spark Architecture

Another important component of a spark application is Directed Acyclic Graph (DAG). DAG in Apache Spark is a set of Vertices and Edges, where vertices represent the RDDs and the edges represent the Operation to be applied on RDD. In Spark DAG, every edge directs from earlier to later in the sequence. On the calling of Action, the created DAG submits to DAG Scheduler which further splits the graph into the stages of the task. DAG provides fault tolerance in the spark applications, by re-executing the DAG in a job in case of failures.

The spark application is managed by a Driver program that schedules the jobs to the different workers and executors. These workers execute the jobs independently and the final results are stored on the distributed file system such as HDFS.

DevOps

CI / CD

Status: It has successfully been integrated.

Context: Establishing an effective and efficient continuous integration and continuous

delivery (CI/CD) pipeline is critical for applications. A pipeline helps you automate steps in your software delivery process, such as initiating automatic builds and then deploying to the production environment.

Decision: Initially, the proposed services were AWS CodeBuild and Travis CI. We decided to go with the AWS CodeBuild since it provides continuous scaling, by virtue of which we have instantaneous processing of each submitted build and can run separate builds concurrently, which means the builds are not left waiting in a queue. Additionally, the pricing model is "Pay-per-minute", thereby minimizing cost as we are not billed for the idle build server capacity. Moreover, because the service is from AWS, all permissions to dependent services are granted using IAM permissions instead of injecting service credentials if used by an external service.

Agile methodology dictates that every commit needs to be tested and integrated, hence there is a need for this build to be automatically triggered, that is for every commit pushed to the master branch of source repository, a build needs to be triggered. We made use of webhooks to achieve this. Webhooks are a "notification" mechanism, a way to inform subscribers about updates, in our case a new commit. The trigger can also be time based where CodeBuild polls frequently for changes. This is the traditional 'PULL' oriented model whereas webhooks are push models. In our case we consumed the 'Push' request on GitHub webhook primarily since we used GitHub for code integration as polling means delay in build being triggered.

Moreover for the team to know about the current status of the build pipeline we include the build batch in our README. The build batch is dynamically created per build and hence is unique per build. It is accessible via a nonrestrictive URL, making it feasible for anyone to track and audit the status of the Project.

Consequences: AWS CodeBuild is a fully managed continuous integration service that compiles source code, runs tests, and produces software packages that are ready to deploy. With CodeBuild, you don't need to provision, manage, and scale your own build servers. CodeBuild scales continuously and processes multiple builds concurrently, so your builds are not left waiting in a queue. Additionally, having chosen GitHub over AWS S3 for code integration did fall in our favour since the 'Build Badge' feature was currently unavailable for the S3 since it worked on transferring the artifact as part of the CodePipeline.

Terraform

Status: Terraform was used as a IaC solution to create and configure the resource we

had used during the project. Using Terraform saved us from using the AWS console for all our infrastructure creation and made it reproducible.

Introduction: IaC has revolutionised the IT business by reducing the cost, increasing the speed(faster execution) and mitigating the risks which were there when managing a server was a full-time job. Infrastructure as Code (IaC) is infrastructure management(networks, virtual machines, load balancers) in a descriptive model, which uses the same versioning used by DevOps team for source code. Every time one applies an IaC model, it generates the same environment. Infrastructure as Code is a crucial practice and it is used together with continuous delivery.

Decision: There are quite a number of available IaC solutions such as:

- Chef
- Puppet
- Ansible
- SaltStack
- CloudFormation
- Terraform

It is not an easy task choosing the one solution which will fit the best for your goal. First of all, we need a provisioning tool(designed to provision the servers as well as the rest of the infrastructure). Chef, Puppet, Ansible, and SaltStack are configuration management tools. Another advantage of Terraform is declarative style which means we just write the code of the end state we want and the IaC tool is responsible for achieving that as opposed to procedural style where the user has to write the step-by-step code of how to achieve the desired state. As an example of declarative style you can see an excerpt from our Terraform file configuring API_Gateway:

```
resource "aws_api_gateway_rest_api" "rest_api" {
  name        = "api_gateway_rest"
  description = "Simulation Serverless Application"
  endpoint_configuration {
      types = [
      "REGIONAL"]
  }
}
```

So, it seems like our choice should fall between CloudFormation and Terraform. However, CloudFormation has a vendor lock meaning that it can be used only with AWS services.

Apart from everything mentioned, the immutable infrastructure paradigm is another strong side of Terraform. It reduces the possibility of a phenomenon known as configuration drift. It also allows the user to know exactly what software is running on a server which provides easy deployment of any previous version of software whenever the user wants.

Consequences: Choosing Terraform allowed us to automate and ease the process of configuring and setting up the services we have used during the project at different stages. It helps us move to a different AWS account or spin up new environments easily.

Time Allocation And Learnings:

Sprint 1:

Akash

Tasks:

- Literature review about simulation - 4 hours
- Research about AWS services - 4 hours
- Discussion with team mates - 3 hours
- Set up communication channels - 1 hour

Learnings:

- Learned about various services on the cloud and how they interact with each other.
- Learned about serverless vs self-managed architectures.
- Learned about pros and cons of different databases like DynamoDB, ElasticSearch and S3 and which one is suitable for storing our Simulation Data.

Anant

Tasks:

- Literature review about simulation - 6 hours
- Research about cloud architecture and use cases- 5 hours
- Discussion with team members - 3 hours
- Research microservices provided by AWS - 3 hours

Learnings:

- Understood the problem at hand and the plausible solutions to the same.
- Learned about different approaches to cloud architecture and dependence on the nature of transactions[stateful/stateless].
- Learned about pros and cons of different services for integration, visualization across providers and the minute differences.

Emir

Tasks:

- Literature review about simulation - 3 hours
- Research about AWS services - 5 hours
- Discussion with team members - 3 hours
- Set up communication channels - 1 hour

Learnings:

- Learned about the technologies used for virus simulations.
- Learned about 12-factor methodology
- Learned about pros and cons of different databases like DynamoDB, ElasticSearch and S3.
- Learned about agile practice.

Haroon

Tasks:

- Literature review about simulation - 8 hours
- Research about different open source implementations - 4 hours
- Discussion with team mates - 3 hours
- Set up communication channels - 1 hour

Outcome:

Shortlisted two repos to use as reference for the simulation.

Learnings:

- Learned about the different simulations that people were working on and built intuition about the simulations.
- Learned about pros and cons of different data models for simulations

- Learned about pros and cons of different databases like DynamoDB, ElasticSearch and S3 and which one is suitable for storing our Simulation Data.

Nithish

Tasks:

- Discussion to identify scope of project - 3 hours
- Ideate on possible architectures and visualizations - 3 hours
- Understand cost implications of the architecture proposed - 3 hours
- Setup repository, project walls and automation in Github - 2 hours

Learnings:

- Conflict resolution when dealing with dissent on architectural choices.
- Experiment with Cloudcraft for making pricing decisions.

Sprint 2:

Akash

Tasks:

- Python project set up - 4 hours
- Implementing framework for unit tests - 2 hours
- Setting up automation on git repo like Code standardisation and requirements.txt - 2 hours
- Team meetings - 2 hours

Learnings:

- Understood what components are required in a typical python project.
- Learned the importance of unit tests in CI/CD automation and how to use pytest.
- Learned about git configurations and how to automate certain processes.

Anant

Tasks:

- Research about Code integration and code deploy - 2 hours
- Setting up Code Pipeline on AWS- 5 hours
- Setting up CodeBuild on AWS - 4 hours
- Discussion with team members - 2 hours

Learnings:

- Learned the need and working of Pipeline.
- Learned how to construct a CodeBuild project.

Emir

Tasks:

- Research about Terraform - 6 hours
- Setting up Terraform locally - 3 hours
- Terraform to provision Lambda and triggering it via API Gateway - 8 hours
- Discussion with team members - 2 hours

Learnings:

- Learned about IaC
- Learned about Terraform
- Learned about provisioning Lambda via Terraform
- Learned about triggering Lambda via API Gateway

Haroon

Tasks:

- Worked on building the initial data model in code for the simulation - 4 hours
- Implemented random walks, infect, initialize simulation and plot methods - 8 hours
- Discussion with team mates - 3 hours
- Set up communication channels - 1 hour

Learnings:

- Understood the overall structure of a python project and how to implement the whole simulation as a package
- Learned about the importance of probabilities and randomization to simulate a pandemic.
- Learned about the git workflow in a team environment

Nithish

Tasks:

- Task detailing for functionality to be developed in this sprint - 2 hours
- Support and provide context on Terraform and its purpose - 3 hours
- Pair on CodeBuild setup - 3 hours
- Team meetings - 1 hour

Learnings:

- Explaining complex topics in an easy and digestible way without getting bogged down in details.
- Navigating with a pair without giving away answers thereby assist in their learning process.

Sprint 3:

Akash

Tasks:

- Set up lambda in AWS and unit test on Lambda - 4 hours
- Set up API gateway - 2 hours
- Team meetings - 1 hour

Learnings:

- Learned about the importance and need of API gateway for a Cloud Application.
- Learned how to connect API gateway with lambda function.

Anant

Tasks:

- Researched about EMR Cluster - 4 hours
- Setting up EMR Cluster though UI and script- 6 hours
- Discussion with team members - 1 hour

Learnings:

- Real world application of big data processing through EMR.
- Functioning and importance of EMR.

Emir

Tasks:

- Research about AWS Amplify - 5 hours
- Research about React - 5 hours
- Deploying a web-page - 9 hours
- Discussion with team members - 1.5 hours

Learnings:

- Learned about AWS Amplify
- Learned about React
- Learned about building web pages.

Haroon

Tasks:

- Implemented features to kill, hospitalize and cure people who are infected- 8 hours
- Enhanced the data model to assign status to people and working hours and simulated the economic impact of the pandemic - 6 hours
- Team meetings - 1 hours

Learnings:

- Learned about the importance of flexible data models to add features in the later stages.
- Improved logical and programming skills by implementing logic for different features as mentioned above.
- Understood how a developer should be ready to accommodate the features from a customer during the project implementation.

Nithish

Tasks:

- Task detailing for functionality to be developed in this sprint - 2 hours
- Distill and understand working on AWS Amplify to simplify process - 3 hours
- PR reviews and feedback - 4 hours
- Team meetings - 1 hour

Learnings:

- Cloudformation and it's working along with Amplify.
- IaC for Cloudfront and S3 solution along with Origin Access Identities.

Sprint 4:

Akash

Tasks:

- Understanding the OAuth 2.0 - 2 hours
- Set up Cognito - 4 hours
- Team meetings - 2 hours

Learnings:

- Learned how basic authentication flows work.
- Discovered that tools like Amplify exist to automate front-end design for future reference so that I can focus on more interesting things while building an app.

Anant

Tasks:

- Writing and deploying a sample python spark application- 6 hours
- Team meetings - 2 hours

Learnings:

- Learned and implemented a pyspark app locally.
- Integration of the pyspark app through AWS Lambda on EMR cluster.

Emir

Tasks:

- Connecting UI to AWS Cognito - 5 hours
- Sending simulation parameters to Lambda - 5 hours
- Discussion with team members - 1 hour

Learnings:

- Learned about AWS Cognito and User pools
- Learned about sending data to Lambda via API Gateway
- Learned about JSX.

Haroon

Tasks:

- Literature review on spark and spark dataframes - 4 hours
- Implemented simple programs with spark to understand the API - 2 hours
- Setup the basic structure of the spark simulation codebase - 3 hours
- Team meetings - 2 hours

Learnings:

- Learned about the spark dataframes, RDDs and other features provided by spark.
- Learned about the python API for spark.
- Understood the distributed programming paradigms and best practices to write a program that can run in a cluster with many nodes.

Nithish

Tasks:

- Task detailing for functionality to be developed in this sprint - 2 hours
- Pairing on EMR and Running Spark application - 4 hours
- Read configuration from SSM - 3 hours

Learnings:

- Instill importance of separating code and configuration.
- Enabling auto-scaling in Spark based on current available memory.

Sprint 5:

Akash

Tasks:

- Experiment with QuickSight - 4 hours
- Build Visualisation PoC with Kibana - 8 hours
- Team meetings - 1 hour
- Work on the report - 4 hours

Learnings:

- Pros and cons of using quicksight and kibana.
- Visualization language Vega-lite.
- How to use Elastic-search API to automate CRUD operations.
- Automate creation of Kibana visualizations using Kibana-dashboard API.

Anant

Tasks:

- Improving user interactions on front end of app - 6 hours
- Team meetings - 1 hour
- Work on the report - 6 hours

Learnings:

- React Javascript to add features like error message, handling reactive events and future possibility of rendering of outputs instead of redirection.

Emir

Tasks:

- Writing the report - 8 hours
- Making minor changes on the page - 2 hours
- Integrating on git - 3 hours
- Discussion with team members - 2 hours

Learnings:

- Learned about Git in the context of team projects

Haroon

Tasks:

- Converted the simulation codebase to spark - 4 hours
- Helped with the integrations with the cloud pipeline - 2 hours

- Helped integrate Kibana for the data visualization of simulations - 3 hours
- Worked on report - 5 hours
- Team meetings - 1 hours

Learnings:

- Learned more about the spark and distributed programming models.
- Learned about integration of python programs with different cloud components.
- Learned a bit about Kibana and its usefulness for data visualizations
- Learned to demonstrate the work by documentation and reports.

Nithish

Tasks:

- Report - 6 hours
- KMS integration with EMR for ElasticSearch credentials - 3 hours

Learnings:

- EMR instance profile roles configuration.
- Terraform modules for KMS integration.