# A Serverless Solution for Fair Exchange System
## Report of CSC8109 Group Project

Christopher Clark, Tong Zhou, Weicheng Yang, and Junyan Wang

School of Computing, Newcastle University
Newcastle upon Tyne, NE4 5TG

## 1 Introduction

## 2 System Specification

According to the requirements described in CSC8109 Group Project Specification, this project is aimed to design and develop a fair exchange system on the AWS cloud. The general process is defined as the following picture[1]:

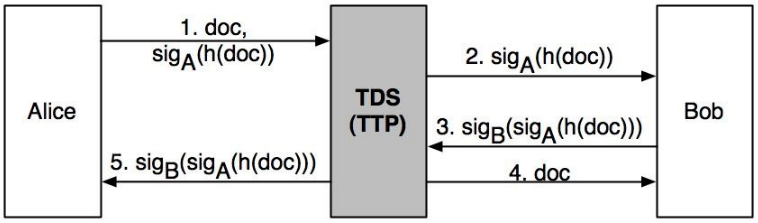


Fig. 1: Process

Before we start to discuss the design of the system, firstly we define some words that will generally be used within our system to represent the components in the fair exchange system:

- **User:** any party who use our system to conduct any fair exchange processes;
- **Sender:** the user who initiates a new exchange request to send some document to another party;
- **Receiver:** the user who passively receive an exchange request from another party;
- **TDS (Trusted Document Store):** a third-party server that works as a fair operator that make sure no party during the exchange can have an advantage over the others, also called a Trusted Third Party(TTP);
- **EOO (Evidence Of Original):** the signature of the document encrypted by the private key of the sender. This signature can be used as evidence of the original document that is confirmed and signed by the sender;

- **EOR (Evidence Of Receipt):** the signature if the EOO that encrypted by the private key of the receiver. This signature can be used as evidence that the receiver has already got the target document;
- **Transaction:** an entire life cycle from when sender send the document with EOO to system till receiver got the document with validated EOO and, at the same time, sender got the validated EOR from the receiver. Any exchange aborted during any point of the exchange can also be regarded as a transaction;

As is shown in the given diagram, there are three parties involved within this system, which is a sender(Alice), a receiver(Bob) and TDS. The sender is trying to send a document to the receiver, at the same time exchange the EOO with EOR. The TDS works as a fair third party which makes sure that no party can have an advantage over the other one, in another word, either receiver get the document with the EOO and at the same time sender get the EOR, or none of them get anything from the other party.

### 2.1 Workflow

To be specific, there are five steps in the entire workflow.

1. The sender sends the document and EOO to TDS.
2. TDS send the signature to Receiver.
3. The receiver sends the EOR to TDS.
4. TDS send the document to Receiver.
5. TDS send the EOR to Sender.

But, as a trusted third party, several validation steps that not listed in the diagram should also be included within the process:

- Between step 1 and step 2, TDS need to validate whether the EOO match with the document, so that sender cannot cheat by sending a fake EOO that does not match with the document;
- Between step 3 and step 4, TDS need to validate whether EOR matches with the EOO that stored in TDS, so that sender cannot cheat by sending a fake EOR that is not generated by the given EOO.

Moreover, both the sender and receiver should be available to abort the transaction manually if they want to terminate the process. But according to fair exchange principles, there are several restricts that should be applied to the manually abort process, which is that neither sender or receiver can abort a transaction after receiver got the document. Also, they cannot abort a transaction that has not been created.

### 2.2 Key Generation

During the exchange process, the system frequently needs to generate some signature as proof that some user has confirmed of some specific operation. Thus, we will need some secured key for every user in our system before any transaction been conducted. We can ask the user to provide their private key every time before the transaction, but that will be easier if we assign a pre-generated private key for every user when a new user is registered in our system.

### 2.3 Authentication

To distinguish the sender and receiver from all users, we need to have some user authentication mechanisms that only allow identified user to use our system, and also we need to have the registration process before a new user start to use our system. The registration process should include the following procedures:

- Get user information include:
  - Email
  - Full name
  - Password (may be kept by third-party authentication service)
- Verify the email by sending a verification code to the email address.
- Generate a private key and a public key so that it can be used as the unique signa-ture of a particular user.

After the user has been created in our system, the user can log in to our systemwith the email address and password set by this user.

### 2.4 Exchange process

To provide service to the user with all kind of devices and platform, we are goingto design a browser-based system that allows users from all over the world to accessour system and communicate with TDS server.

## 3 Background Research

The background research is going to be broken down into 3 main sub-sections. It will talk at a high level at a high level about the architectural styles that are offered by AWS. This is turn will reflect on the language that we have chosen, which will logically follow on how the Cryptography Libraries/Technologies that were used.

- Technology Alternatives/Choices Within AWS
- Cryptography Libraries

### 3.1 Technology Alternatives/Choices within AWS

This was the first major hurdle that we had to consider when designing our system. This section is not going to talk about the specific AWS services that were used within he making of this project, more so; the choice that is seen as the most viable will influence the type of technologies within AWS that we intend to utilize.

There are two routes that can be taken when looking at the subject of architecture. It can be looked at from the perspective of how the code base isgoing to be structured. Is it a valid option to go for a monolithic code base, ora fine grained microservices based architecture when it comes to structuring the codebase. However, upon looking into the two main types of architecturethat are popular with current cloud services, it was seen that this decision is somewhat made for you.

In reference to the decision being made for you in terms of how you architectyour overall cloud service. This is in reference to an extremely popular cloud architecture known as Serverless.

### 3.2 What is Serverless?

Serverless is a somewhat new concept within Cloud Computing. This is not necessarily the concept of not using servers. More so, it is the concept of not having to manage your own servers, and having an only pay for what you use mentality. The main benefits of serverless are as follow:

1. No Server Management
2. Flexible Scaling
3. Pay for Value
4. Automated High Availability

With Serverless it isn't necessarily all or nothing either. AWS offers Serverlesscomponents such as databases, web servers, and code bases. To get a betteridea of what this means in all 3 contexts. A serverless database would only be running when it is being used; so if it is being written and read from, you will only pay when it is active. When it is lying dormant, you are not paying for that time. In contrast to having a standard server set up with a; database,application container (Jboss / tomcat) and web server; which may be running 24/7 without being used.

In reference to Serverless not having to be all or nothing, there may be use cases where a serverless database and application container(something to run your code) is fine. However, you may need to control access to who can access you website (such as it only being accessible from Newcastle University). This is something that is not offered by Serverless, and will be mentioned in the later sections; so using an EC2 server may be the way to go.

Other drawbacks of not utilizing a serverless architecture can include If you want an EC2 instance running then you need to pay someone to apply software patches, check the disk space, make sure there is enough CPU etc. This costs money - which you don't need if you are using a serverless architecture. Also if you go from 2 users of your platform to 1,000,000 then you will need to scale your EC2 instances which can be pain. They will required load balancers and autoscaling groups.

### 3.3 Cryptography Libraries

Our cryptography research can be split up into 3 main sections.

- What kind of keys did we choose
- What kind of verification algorithm did we use

There are more elements to be discussed in terms of cryptography, in terms of how did we handle key storage and generation? But this is more appropriately discussed in section 3. So within the spec, it was suggested to look at the following components of the JCA library :

- KeyStore for representation of a (file-based) certificate and key store (for management of certificates and public/private key pairs). You can use the Java command line keytool utility to generate a keystore, keys and associated certificates.
- Signature for generation and verification of signatures over data.
- SignedObject for encapsulation of a signature and its associated object(data).
- Cipher for encryption and decryption of data.
- MessageDigest for generation and verification of secure hashes of data.
- SecureRandom for generation of secure pseudo random numbers.

At the point of deciding on how to implement our encryption, the group had agreed that they would feel comfortable doing a lot of the implementation in Javascript. Firstly because of the front end implementation being done in this language, but also the fact that using Serverless technologies such as Lambda lend themselves very well to using this Language.

So the challenge at this point then became, finding something that implements the above within JavaScript, and do we need all of these featuresing the first place?

Through looking at various node packages, we found various keys within the npm store, however there was not a lot of official verification on these, so wedidn't feel these would necessarily be safe.

In the end we ended up using the official Cryptography library offered by Node. This allowed us to generate keys (RSA/DSA) that could be used for the generation of signed Objects backed by the X-509 standard, similar to the JCA libraries. We opted to use the RSA keys, as they verify much faster than DSA keys. And the speed at which DSA keys can create a signature is not that much faster than an RSA key.

In terms of actual encryption of our data, it was assumed we would be using secured means of communication backed by HTTPS/SSL mentioned in the spec, as well as using a storage repository that offers encryption at rest such as S3; the encryption of our data wouldn't be necessary.

## 4 Design Decisions

This section is going to deal with the selection of specific selection of AWS Services. What they are, and why they were chosen. How they were applied to the algorithm is going to be discussed in section 5.

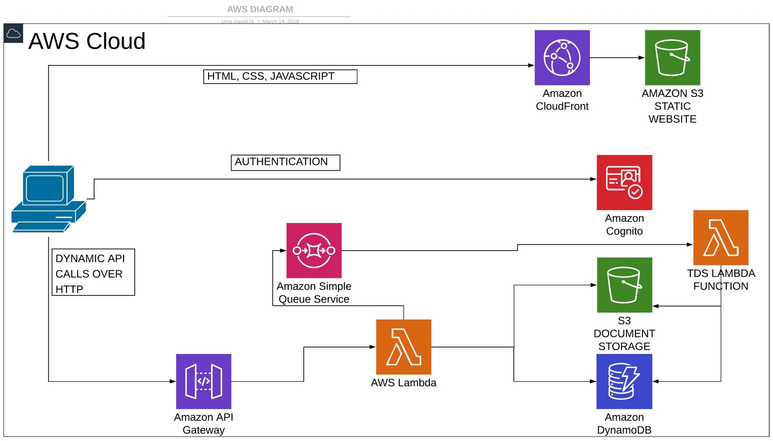


Fig. 2: AWS Cloud

This diagram gives a relatively good idea of the high-level interaction of our service. This will give a high-level description of what each service is for. Later on, in the section, more detail will be given on what each service is, what is it used for, and why it was chosen over alternatives. As mentioned within the previous section. The architecture we have chosen is that of a 100% serverless architecture. Briefly what this meant was explained in section 2, however it will become clearer as it is explained in further sections, exactly why it is being utilized.

As mentioned within the previous section. The architecture we have chosen is that of a 100% serverless architecture. Briefly what this meant was explained in section 2, however it will become clearer as it is explained in further sections, exactly why it is being utilized. In terms of a brief description of what this diagram means.

- Website's static assets (HTML/CSS/JAVASCRIPT) are being stored in an S3 bucket, as opposed to an EC2 container being served over Cloudfront
- Website's Authentication is being backed by an AWS service called Cognito
- Access to AWS services is being granted through an AWS component called API-Gateway.
- Backend Logic is being executed by AWS Lambda (Calling of Amazon API's, handling of creating/verifying signatures)

- Storage of assets (Documents/Signatures/Keys) is being handled by S3
- DynamoDB is used to keep track of the state of the transactions.

### 4.1 Breakdown of AWS services

As a side to the start of this section, EC2 is going to be compared against multiple components and use cases, as it has multiple. For example, it can be used to serve front end assets in the form of a web server, it can also be used to house databases/application containers.

### 4.2 How is our content being served to the user?

When I say how is our content being served to our user. This can be interpreted as what technology are we using in place of a web-server to serve our front end logic to the user.

We have opted to use S3 static website hosting served through an AWS service called Cloudfront. There are a lot of benefits to doing it this way. First of all S3 offers a multitude of benefits including:

- Allows for highly available front end 99.99%
- Auto load balanced
- Low Cost only pay for your what you use

To expand upon these bullet points. There is no server do worry about in the traditional sense here. You will only pay for the amount of times someone requests static assets from your bucket to display the website.

There is no need to worry about load balancing as this is all handled by S3.There has been one major recorded outing in the entirety of AWS's lifetime, also known as the day ”the internet went down”. So they can guarantee 99.99% availability.

The reason we are using CloudFront; is that out of the box S3 static websites do not offer the protection of HTTPS and SSL. Serving our website through this method offers us these protections, as well as more control over what can be sent to our bucket; such as the controlling of allowed HTTP operations(such as only allowing GET and HEAD requests, as opposed to just post).

There are also the downsides of using something like an EC2 server to bear in mind as well when choosing how our assets are served.

If you want an EC2 instance running then you need to pay someone to apply software patches, check the disk space, make sure there is enough CPU etc, etc. Also if you go from 2 users of your platform to 1,000,000 then you will need to scale your EC2 instances which can be pain. They will required load balancers and autoscaling groups.

This comparison can be seen as Static Website vs Dynamic Website. Static websites are gaining a lot of traction as per the above answers. Having a dynamic website typically means database, application container (jboss / tomcat) and web server. All of the problems as pointed out in the above statements. If your content doesn't need dynamic database driven content, then it can be far cheaper to have static content served from S3.

Static S3 websites are not a bullet proof solution however, CloudFront does not offer the same amount of control as having a server hosted on EC2. So for example, if a use case required that the IP address only be accessible from Newcastle University; this isn't something CloudFront/S3 can offer you. So this may be a good use case to consider using EC2.

### 4.3 How are AWS Services being accessed?

There are multiple ways AWS services can be accessed.

1. They can be accessed directly over a RESTful interface using Amazon API Gateway.
2. They can be accessed programmatically using the AWS SDK; contained-within a conventional code stack.
3. Or a combination of the two can be used. Amazon API Gateway can be used to call Lambda Functions (these will be talked about in more detail further on in the section) that then perform operations upon AWSServices

Each of these options have their merits and drawbacks.

Let's examine the first point, accessing the AWS service directly over REST. This can often be seen as a good use case, if you are wanting to call one service; and that is it. For example, if all you want to do is read something from a bucket, this makes sense. The problem with this approach relies on if your goal involves 1 or more operations relying on each other. Say for example, you want to do a database lookup, that database lookup will tell give you an S3 document location, you then want to copy that document from one bucket to another. This could be many REST requests and coordinating them can quickly become a pain.

This above drawback to just using REST calls can be solved by utilizing the AWS SDK within code. If you have a logical operation that relies on multiple AWS services working together, it is easy to call each API separately and get the overall result you need; synchronizing them is not a challenge. However, the drawback with this approach comes with having to provide your full AWS details (Secret Key etc) somewhere in the code. While there are secure ways of doing this, such as inserting them as environment variables during a DevOps process, it is an all too common occurrence for people to publish their secret keys to GitHub, and have their AWS accounts abused.

A compromise between these two solutions is using API Gateway to call Lambda functions that in turn call your AWS services. Your API call can be secured by an API key (it can be secured further using JWT's provided by theCognito login service, but this will be discussed later in the section), no need to provide your full AWS details. And since the Lambda function is inside yourAWS account, it can securely call services from there. Lambdas can have calls to multiple calls to AWS services inside of them. The benefits and drawbacks of Lambda will be discussed later on in this section.

### 4.4 How is backend logic performed?

Backend logic is done (primarily) with code. Our backend logic consists primarily of reading and writing to a database (to keep track of our transactions); as well as reading and writing from BLOB storage(signatures, document). As well as the creation and verification of signatures. We identified two main approaches that we could take. We could either have a codebase written in a language of our choice hosted on an application server on EC2. Or we could utilize the Runtime as a Service platform known as Lambda that is offered by AWS. We have already seen the benefits of securing our AWS info by utilizing Lambda, but it has other benefits.

**What is Lambda?** Lambda can be described in the following four bullet points:

- Stateless functions
- "Single Use"
- Allows for Runtime as a Service
- Implemented in almost all languages
- Secures Code

It is a stateless function, which means that you SHOULD write it in a way thatit is stateless; meaning that all information that is needed to execute the function, should be contained within it, it should rely on no outside state.

Lambda functions are designed to be single use (used for only one purpose) this is a term that can be used loosely however, single use really means that the code should not be very resource intensive. If the code is doing multiple things, but it is not incurring heavy memory usage, this is fine.

It allows for the concept of Runtime as a Service. This means that you only pay for the total amount of time it takes for the code to execute. Code takes 1 second to run? You only pay for 1 second.

It is implemented in all major languages(Java, JavaScript Python, C#, Go etc).It should be noted though, while all these languages offer the exact same API access. It is easier to work with Lambda in languages such as Python and JavaScript than it is for languages such as C# and Java (You can't work in the Lambda GUI using these languages)

Finally, it secures code. As stated above, your AWS Credentials are automatically accessed from the Lambda function. Any sensitive information such as keys or database access credentials can be securely stored inside your Lambda code.

**Why Lambda?** As mentioned above, your other option for backend logic is hosting your runtime on an application server. This means that it will be running 24/7, andif it is idle you will only pay for what you use. The benefits of using Lambda are similar to why you would use a static S3 bucket to host your content as opposed to an EC2 server. It comes with a lot of complex configuration and management when it comes to scaling your application.

If you want an EC2 instance running then you need to pay someone to apply software patches, check the disk space, make sure there is enough CPU etc, etc. This costs money - which you don't need if you are using pure lambdas.

Also if you go from 2 users of your platform to 1,000,000 then you will need to scale your EC2 instances which can be pain. They will required load balancers and autoscaling groups.

With Lambdas you don't have any such problems you write your code and upload it. If no one uses your service you pay 0 cents. If 1,000,000 users come along tomorrow then they are instantly scaled and you pay for what you use.

**Lambda isn't bulletproof** While a lot of benefits were listed above, Lambda is far from bulletproof. There are certainly use cases where hosting your code on an EC2 instance could be the way to go. For example, if you have software that requires access to a lot of third party API's, or it cannot be guaranteed that the code will be stateless, EC2 hosted code may be the way to go.

Finally, going back to the "single use" point, if you have an operation that is going to be very memory intensive, and it is not feasible to split it up into several Lambda functions. EC2 hosted code may again be the way to go. Lambda allows for a maximum of 3GB of memory to be utilized on every invocation.

### 4.5 How Is Data Being Persisted?

This section will look at how data is persisted in two ways. How are we keeping track of the state of each transaction (Database)? How are we storing assets such as signatures and documents (S3)?

**Database Selection** There are three main choices within the AWS ecosystem to use when selecting a database.

1. Amazon RDS (Relational Database Service) this has support for PostgreSQL, MariaDB, Oracle, MySQL, and Microsoft's SQL Server
2. Amazon's DynamoDB NoSQL Database
3. Amazon's Aurora SQL Database

It first makes sense to compare Amazon RDS with DynamoDB and Aurora. This is because Aurora and DynamoDB have out of the box support for serverless architecture (you only pay for the amount that you use).

When using Amazon RDS there is inherent complexity with managing (not as much complexity as if you were managing it without RDS, but complexity nonetheless).

RDS has inherent complexities in provisioning and running the service. RDS is designed to be run 24/7 and, in order to run it well, will require constant monitoring (in the form of alerts) to be probing RDS to ensure it is healthy. DynamoDB/Aurora has none of this (or rather there is, but AWS take care of that for you). It's therefore cheaper and more reliable to run easily. RDS has it's place and if we had a bigger service we would probably need to look at it.

Also there is a cost consideration. Running your own RDS instance 24/7 can incur large costs over the life span of the service we are running. This isn't the case for DynamoDB/Aurora.

**Aurora or DynamoDB?** Aurora and Dynamo have the following in common:

- They are both serverless.
- Both scale very well.
- Both support ACID transactions.

The main draw of using something like Aurora is that it has support for PostgreSQL and MySQL. So if you have an already existing database that uses MySQL or PostgreSQL it makes a lot of sense to switch to Aurora.

The reason we have decided to use DynamoDB is that we have no pre-existing database infrastructure that we want to use, as well as it providing almost single digit response times at almost any scale.

The main use case that would mean using Aurora or DynamoDB is if you are wanting to return large amounts of data (as in millions of rows of data). If youare returning this amount of data, it makes sense to use Aurora. However, at most in this type of application we would be returning thousands (the historyof transactions for a user).

**BLOB Storage Selection** In terms of BLOB storage, the main options offered by AWS are Amazon S3 and EFS. This choice was a relatively straight forward one. EFS is only offered when using an EC2 instance, which we have decided not to do. AWS S3 offers the following benefits:

- Allows for highly available front end 99.99%
- Encryption at Rest
- Low Cost only pay for your what you use

S3 has almost no down sides when it comes to using BLOB storage, and many of the benefits were listed within the using S3 as a static website.

### 4.6 User Security

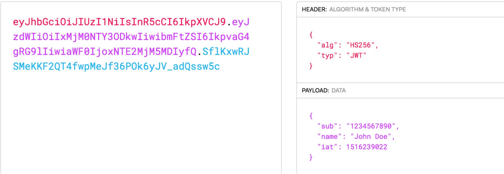
There are two aspects of user security within our system. The login system and how we secure our private keys

**Login System** In terms of user security, there are two approaches to take. It is possible to

- Write your own login system
- To use an authentication API such as Passport.js,
- To use AWS Cognito

AWS Cognito provides out of the box basic authentication features such as MFA, email confirmation upon signing up etc.When using something like AWS Cognito it takes all the complexity out of managing sensitive items such as user passwords. The main benefit of Cognito that applies to our system is the concept of JWT's. Within our system JWT's have two uses.

- Firstly, it allows us to uniquely identify our users through the use of a sub attribute. JWT's can be decoded into attributes which we see below. Within the implementation section we will see how the sub is used to uniquely and securely identify users when it comes to the TDS for storing documents and for storing keys.
- Secondly, within the API gateway section, it was mentioned that API keys can be used to secure access to our AWS services. When using Cognito, it allows us to use our JWT's as our API keys. This means that it is only possible to access AWS services using API gateway if the user is logged in.



```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ
zdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJ
SMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

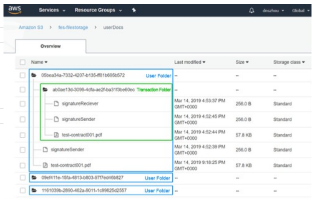
HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

(a) JWT



| Name | Last modified | Size | Storage class |
|---|---|---|---|
| 05bea34a-7332-4207-b135-ff81b695b572 User Folder | -- | -- | -- |
| ab0ae13d-3099-4dfa-ae2f-ba31f3be60ec Transaction Folder | -- | -- | -- |
| signatureReciever | Mar 14, 2019 4:53:37 PM GMT+0000 | 256.0 B | Standard |
| signatureSender | Mar 14, 2019 4:52:45 PM GMT+0000 | 256.0 B | Standard |
| test-contract001.pdf | Mar 14, 2019 4:52:44 PM GMT+0000 | 57.8 KB | Standard |
| signatureSender | Mar 14, 2019 4:52:39 PM GMT+0000 | 256.0 B | Standard |
| test-contract001.pdf | Mar 14, 2019 9:18:25 PM GMT+0000 | 57.8 KB | Standard |
| 09ef411e-15fa-4813-b803-97f7ed46b827 User Folder | -- | -- | -- |
| 1161039b-2890-462a-9011-1c99825d2557 User Folder | -- | -- | -- |

(b) S3 Folder

**Key Management** In terms of managing keys within our system. At a high level we are storing them in S3, uniquely identified by a sub. This process will be explained in more depth within Section 5.

In terms of keys being stored within S3. They are secured from anyone outside the AWS account e.g. the users. However, it is insecure to store plaintext keys within S3, as they can be seen by the S3 bucket administrator.

In terms of keys being stored within S3. They are secured from anyone outside the AWS account e.g. the users. However, it is insecure to store plaintext keys within S3, as they can be seen by the S3 bucket administrator.

**Notification Services** There are two choices when building a notification service within AWS. We can use

- SQS (Simple Queue Service)
- SNS (Simple Notification Service)

SNS pushes messages in near real-time to all subscribers. SQS is a queueing system. Messages can't be received by multiple receiver sat the same time. Any one receiver can receive a message, process and delete it. Other receivers do not receive the same message later. SNS uses a publish subscribe architecture, which means that multiple subscribers might subscribe to a topic, when a message is sent to that topic; all subscribers will receive the message.

We have chosen to use SQS for primarily two reasons.

Our use case is that each message will only ever be sent to one person, so it doesn't make a lot of sense to use a publish-subscribe mechanism. We currently have two queues' set up. One to execute Step 2 of the protocol (Alice sends signature to Bob). Another queue to execute Step 4/5 of the protocol (Document sent to Bob, Signature sent to Alice, transaction set to resolved).

Secondly, we initially thought with the speed at which SNS can process messages, even though our architecture didn't require publish subscribe, if SNS can send 25,000 a second, and SQS can do maximum 3000. If it's so much faster, shouldn't we just choose SNS?

However, SQS recently introduced the ability to execute a lambda function every time a message appears in the queue. Previously, the system had to poll the queue every X seconds to process the message, thus could be possibly. With the introduction of this most recent feature, sometimes referred to as event driven architecture; this makes SQS in a lot of use cases superior to SNS.

You have observability with SQS. If the Lambdas take 30 seconds to run, then with SNS you could only process a job once every 30 secs. If you drop 10k jobs on SQS you can get 10k Lambdas to spin them up and the total timewill be 30 secs.

## 5 Implementation overview and Challenges

### 5.1 Challenges

During the four weeks development, we have met many challenges. Thanks to our good teamwork, we've accomplished this work perfectly. The challenges can be divide into three parts, which are cloud configuration, lambda programming and security. The details are as follows,

- **Cloud Configuration**. As we use AWS serverless infrastructure, this is our first time to build a serverless system. Everything is new to us. In order to develop the application efficiency, we separate the configuration into several parts. For example, Tong take part in user system like Cognito. Because of the good communication and cooperation, we solve this problem quickly.
- **Lambda Programming**. As we use Lambda function as our back-end architecture, we don't need to use Java even through Java is the only language we learned from university. Since Chris, Tong and Junyan have experience of Javascript, we chose Node.js as our backend language. It is because not only we are familiar with Javascript, but also the language consistency between front-end and back-end.
- **Security Issue**. Security is obviously an important work in every software development. In order to make our system as safe as possible, we apply many

different strategies to protect both the user and the transactions data. For instance, we utilize HTTPs for each api to make sure all the api are security connection. And we apply private key and public key to manage our user system.

## 6 Test plan

Testing is a very important stage in the life cycle of software development and therefore a substantial amount of time was dedicated to test this system. Table 1 shows the result of API test, Table 2 shows the result of functional test and Table 3 provides the statistic data each API.

### 6.1 Test strategy

According to the testing process, this project has been strictly tested the three stages:

- **API Test**. API testing is a type of software testing that involves testing application programming interfaces (APIs) directly and as part of integration testing to determine if they meet expectations for functionality, reliability, performance, and security. We make a form for testing all APIs
- **Functional Test**. Functional testing is a quality assurance (QA) process[6] and a type of black-box testing that bases its test cases on the specifications of the software component under test. We also make a form for testing all functions.
- **Performance Test**. In software quality assurance, performance testing is a testing practice performed to determine how a system performs in terms of responsiveness, stability under a particular workload, scalability, reliability and resource usage. We utilise JMeter[4] as our test software which that can be used as a load testing tool for analysing and measuring the performance. We tested all APIs with multiple users and threads to make sure our system is highly accessible under high demand of transactions

### 6.2 Test result

We conducted extensive test including api tests, functional tests and performance tests.

Table 1: API Test Result

| Name | Expected response | Actual result | Pass |
|---|---|---|---|
| uploadDocument | the location of the document | the location url of the uploaded document | Yes |
| generateEooSignature | the eoo of the document | the eoo signature of the document in BASE64 format | Yes |
| generateEorSignature | the eor of the document | the eor signature of the document in BASE64 format | Yes |
| creatTransaction | whether transaction created successfully or not | Boolean true or false | Yes |
| confirmTransaction | whether transaction created successfully or not | Boolean true or false | Yes |
| getAllUsers | all users | a list of users | Yes |
| getTransactionDetails | the imformation of the transaction | a json format data which contains the details | Yes |
| register | the location of the document | the location url of the uploaded document | Yes |
| verifySignature | the result of validation | Boolean true or false | Yes |

The api test result form includes the api name, the expected response, the actual result and whether the api pass the test or not.

Table 2: Functional Test Result

| Name | Expected response | Actual result | Pass |
|---|---|---|---|
| Test upload function | file has been in S3 bucket | file has been in S3 bucket | Yes |
| Test infromation function | lastest infromation appears | lastest infromation appears | Yes |
| Test sender's abort transaction function | both pages are aborted | Boolean true or false | Yes |
| Test EOR not match | transaction aborted | transaction aborted | Yes |
| Test when the receiver does not receive EOO, the sender aborts | transaction aborted | transaction aborted | Yes |

The functional test result form includes the funtion name, the expected response, the actual result and whether the function pass the test or not.

Table 3: Performance Test Result

| Requsets | Response Time(ms) | Throughput | Network Recieved(KB/s) | Network Sent (KB/s) |
|---|---|---|---|---|
| Abort | 268.0 | 3.73 | 2.0 | 8.26 |
| CheckReciever | 446.0 | 2.24 | 0.71 | 5.03 |
| GetAllUsers | 2029.0 | 0.49 | 0.37 | 1.07 |
| GetInbound | 159.0 | 6.29 | 2.65 | 13.64 |
| GetOutbound | 196.0 | 5.10 | 2.15 | 11.07 |
| GetTransactionDetails | 232.0 | 4.31 | 4.36 | 9.68 |
| GetUnreadInbound | 142.0 | 7.04 | 5.32 | 15.31 |
| RetrieveURL | 100.0 | 10.00 | 11.71 | 22.21 |
| ReturnAndSaveEOR | 840.0 | 1.19 | 0.85 | 2.66 |
| Upload | 36.0 | 27.78 | 34.18 | 60.36 |
| Total | 444.8 | 2.24 | 1.65 | 4.94 |

This statistic form shows all APIs performance like response time, throughput and network

## 7 Critical Evaluation

According to the development process and feedback of each group, this section will provide the insight of our system and other groups' work.

### 7.1 Self-evaluation

During our entire development, we choose serverless infrastructure after comparison of each framework. The details of serverless structure can be seen in Section 3. The serverless programming do help us a lot, the aspects of serverless design affected our project are as follows,

- **Construction Cost**. As we utilize AWS services, AWS provides a whole set of serverless services, like Lambda, S3 bucket and DynamoDB. In Lambda, we can easily invoke S3 and DynamoDB data by the specified libraries. This setting saves us a lot of developing time.
- **Development Efficiency**. On the other hand, we utilize node.js as our Lambda function programming language. Since we apply web browser as our system client, the front-end logic programming language which is Javascript has matched the node.js. So, it reduces our learning costs.
- **Auto Scalability**. As Lambda is a FaaS(Function as a Services) which provide by AWS, it can start instance replicas as much as possible when needed without deployment and configuration delays. It depends on our stateless service.
- **Low Portability**. It is difficult to migrate our project to other cloud service providers, because all of our tasks was developed in AWS serverless service. If our system become a huge project, the serverless may not be a good choice.

### 7.2 Relative-evaluation

According to the infrastructure of our system, there are many different aspects between our project and other groups' project.

- **Group One**.
- **Group Two**.

## 8 Group working

For any team who work together to deliver a specific product or research progress, group working is always a crucial part of the entire life cycle, especially for software development teams. Software products are so complicated that any tiny issue can crash the whole system, but software products nowadays are more and more complex that is not possible for any single developer to finish all of the work. It is especially crucial that teamwork should be standardized and constructed with specific rules.

### 8.1 API Definition

Firstly, we set a rule to define all our back-end API in a standard way with three components:

- **Input:** the information that is designed to pass into the system
- **Process:** the internal workflow that is designed to be developed for this API
- **Output:** the value or information that should be returned to the outside system

### 8.2 Scrum Framework

During our project, we used the Scrum framework[2] as a tool to collaborate all our members and improve the efficiency of delivery.

Conventionally, develop team take some typical approach, like waterfall model, to manage their entire life-cycle of development, which means the development is following one direction from the conceptional analysis, function design, coding, testing to finally deploying and maintenance. But with the development of the software engineering industry, now more and more band new technologies are taking the place of old-style platforms of programming. More and more development team are required to keep pace with new technologies and cope with the changing requirements to follow the trend of the market. They have to find a different way to conduct their entire software production life-cycle.

Another reason for not using the waterfall model is because it takes time to understand the requirement, especially before you start to work on it[5]. Only after you processed to the central part of the project can you get a general idea of what you are going to make, but with waterfall model you do not have a chance to reverse the initial design and choose another path to conduct your project until you finish most of the product?

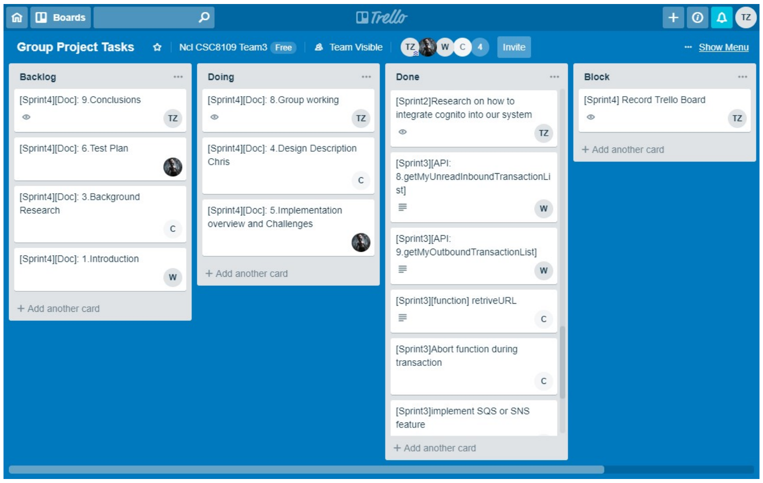
To solve these problems, we decided to take Scrum as our model during the entire developing life-cycle. Just like XP (Extreme Programming), Scrum is a kind of agile Methodology that especially suited for small develop team from3 9 members, and break the entire development work into small pieces within time-boxed iterations, which is called "Sprints." Since we only have about four weeks to develop our project, we divided the entire lifecycle into four individual Sprints. During every week we had one planning meeting on Monday, one retrospective meeting on Friday and daily stand-up meetings every day.

– **Scrum Planning** is designed to discussing what we are going to deliver at the endof this sprint and decide which task we are going to finish within this sprint. Every jobshould be described clearly so that every member can understand is.

– **Daily Scrum** s a meeting conducted by all developing team on every workdayduring the iteration. Every group member should speak one by one, and there are onlythree topics on this meeting: what you did yesterday, what you are going to do todayand what issue or blocker during the work. This meeting is restricted to be within 15minutes, which means there is no need to describe every detail of the problem and justlet everyone else know the status of all tasks.

– **Sprint Retrospective** means at the end of every sprint; the team will talk and eval-uate the progress we made during this sprint, and talk about any improvements we canmake in the next sprint.

We also used Trello[3] as our list-making application to help us list the status of every task undergoing, and we can easily drag them to different columns represent different status within our sprints.



Group Project Tasks

Ncl CSC8109 Team3 Free

Team Visible

Backlog

- [Sprint4][Doc]: 9.Conclusions TZ
- [Sprint4][Doc]: 6.Test Plan
- [Sprint4][Doc]: 3.Background Research C
- [Sprint4][Doc]: 1.Introduction W

Doing

- [Sprint4][Doc]: 8.Group working TZ
- [Sprint4][Doc]: 4.Design Description Chris C
- [Sprint4][Doc]: 5.Implementation overview and Challenges

Done

- [Sprint2]Research on how to integrate cognito into our system TZ
- [Sprint3][API: 8.getMyUnreadInboundTransactionList] W
- [Sprint3][API: 9.getMyOutboundTransactionList] W
- [Sprint3][function] retriveURL C
- [Sprint3]Abort function during transaction C
- [Sprint3]implement SQS or SNS feature

Block

- [Sprint4] Record Trello Board TZ

Fig. 3: Trello Interface

### 8.3 Task List

All the tasks completed during our project is listed in the table below:

Table 4: Contribution of each member

| Task | Owner |
|---|---|
| (Sprint1) Design front-end webpage for creating new exchange transaction | Tong |
| (Sprint1) Research on encryption library with Node JS | Christopher |
| (Sprint1) Research on how to do CRD with Lambda, DynamoDB anduse Postman to test it | Weicheng |
| (Sprint1) Research on upload process using lambda and S3 bucket | Weicheng |
| (Sprint1) API: 1.uploadDocument | Junyan |
| (Sprint1) API: 6.getAllUsers | Tong |
| (Sprint1) API: Login with Cognito system | Tong |
| (Sprint2) API: 10.getTransactionDetails | Tong |
| (Sprint2) API: 11.register | Tong |
| (Sprint2) API: 12.verifySignature | Christopher |
| (Sprint2) API: 2.generateEooSignature | Christopher |
| (Sprint2) API: 3.generateEorSignature | Christopher |
| (Sprint2) API: 4.creatTransaction | Junyan |
| (Sprint2) API: 5.confirmTransaction | Junyan |
| (Sprint2) Design confirm Transaction web page | Tong |
| (Sprint2) Design the APIs to be created in our system | Tong |
| (Sprint2) Generate a private key and public key(within API 11.register) | Christopher |
| (Sprint2) Research on how to integrate Cognito into our system | Tong |
| (Sprint3) API: 8.getMyUnreadInboundTransactionList | Weicheng |
| (Sprint3) API: 9.getMyOutboundTransactionList | Weicheng |
| (Sprint3) RetriveURL | Christopher |
| (Sprint3) Abort function during the transaction | Christopher |
| (Sprint3) Implement SQS or SNS feature | Christopher |

And Table 5 is a general conclusion of the contribution of each group member:

Table 5: Contribution of each member

| Name | Contribution | Total percentage |
|---|---|---|
| Tong Zhou180275186 | Workflow design, teamwork assigningand front-end and back-end programming. | 25% |
| Junyan Wang170127567 | Back-end and front-end programming, S3,and API operations. | 25% |
| Weicheng Yang170152145 | Front-end and Back-end programming,Functional and performance test. | 25% |
| Christopher Clark180614318 | Cloud architecture, back-end program-ming, and encryption algorithms | 25% |

## 9 Conclusion

## References

1. Csc8109 requirement. `https://homepages.cs.ncl.ac.uk/nick.cook/csc8109/`.
2. Scrum development. `https://en.wikipedia.org/wiki/Scrum_(software_development)`.
3. Trello. `https://trello.com/about`.
4. Emily H Halili. *Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites*. Packt Publishing Ltd, 2008.
5. Jeff Sutherland. Agile development: Lessons learned from the first scrum. *Cutter Agile Project Management Advisory Service: Executive Update*, 5(20):1–4, 2004.
6. Chien-Wei Wu, WL Pearn, and Samuel Kotz. An overview of theory and practice on process capability indices for quality assurance. *International journal of production economics*, 117(2):338–359, 2009.

# 10 Appendix