CS301 – IT Solution Architecture

Project Report

Github Team Name:

Minimonies

Members:

Ansley Chua

Evelyn Chee

Kang Yi Qi

Marcus Kerh

Tay Qi Fang

Date: 10 November 2019

Page Count: 20 (Excl. Cover, TOC, Annex)

Table of Contents

Background and Business Needs

Official loans are often made in banks which requires excessive documents of your income and status. Not everyone can make a bank loan - especially the lower income families and students who do not meet the minimum wage requirement. In addition, those who need quick cash or small amount of loan would find it a hassle as preparing the necessary documents takes time with no guarantees that the bank would approve it. Hence, there is a need for a micro-loans system, Minimonies.

Minimonies is a platform that offers peer-to-peer loaning of a small amount of money to finance the needs and wants of individuals as well as SMEs. The loans, which may start from as low as $10, are requested by users to other individuals who wish to take up a loan.

Stakeholders

Stakeholder	**Stakeholder Description**
AWS API Gateway	An AWS service for creating, publishing, maintaining, monitoring, and securing REST
AWS CloudFront	Provides fast content network delivery service that securely delivers data and applications
AWS Cognito	Provides authentication, authorization, and user management for web and mobile apps.
AWS Elastic Load Balancer (Application & Network LB)	Elastic Load Balancing automatically distributes incoming application traffic across multiple servers
AWS Relational Database Service (RDS) AuroraDB	A high performing and highly available MySQL and PostgreSQL-compatible relational database
AWS Route53	An AWS web service to translate domain names to IP addresses
AWS S3	Provides storage service
AWS Simple Email Service (SES)	Provides a cloud-based email sending service
AWS Simple Queue Service (SQS)	Message queue service that follows a standard queue order.
Borrower	Borrower requests for a loan and repays the lender
DynamoDB	Key-value and document database that is high performing with built-in security, backup and in-memory caching
Lender	Lender provides funding to borrower
Loan Microservice	Manages loans related details
Notifications Microservice	Manages all notifications and acts as a gateway to send out notifications
Transaction Microservice	Manages all transactions from the loan
User Microservice	Manages users related details

Key Use Cases

Use Case Title - Login and view summary of loan transactions	
Use Case ID	**C01**
Description	User login to system, system integrates with AWS Cognito for authentication, system store the id in an in-memory cache and validate the presence of this id on every page.
Actors	Lender, Borrower, AWS Cognito
Main Flow of events	1. User log in to the system using their username and password. 2. System integrates with AWS Cognito for authentication. 3. AWS Cognito returns an authenticated ID to system. 4. System stores the ID in an in-memory cache 5. System validates the presence of this ID on every page 6. Information is returned to the system
Alternative Flow of events	**Flow 1 - Incorrect username/password** 1. User log in to the system using their username and password. 2. System integrates with AWS Cognito for authentication. 3. AWS Cognito returns authentication status to the system 4. Users to be redirected to the login page and be notified of incorrect credentials entered through an error message.
Pre-conditions	Users to have an existing account with Minimonies.
Post-conditions	Users to be able to see the home page.

Use Case Title - Add to Shopping Cart	
Use Case ID	**C02**
Description	Lender to view available items, then proceed to add items to the cart. System to store the inputs in a sticky session.
Actors	Lender, Loans Microservice
Main Flow of events	1. Lender to view the available items 2. System to send a request to API gateway 3. API gateway to forward request to Loans Microservice 4. Information returned to system 5. Lender to add items to cart 6. System to store selected item ID in sticky session 7. System to send a request to API gateway 8. API gateway to forward request to Loans Microservice 9. Information returned to system and display on shopping cart page
Alternative Flow of events	**Flow 1 - No available item** 1. Lender to navigate to view available items 2. System to send a request to API gateway

	3. API gateway to forward request to Loans Microservice 4. An error message will be displayed
Pre-conditions	Lender has an existing account and is already logged in.
Post-conditions	Lender to add an item to the cart.

Use Case Title - When current instance is down	
Use Case ID	**C03**
Description	Lender's cart items will be saved in the session. Items in the carts should remains the same when the main instance is down.
Actors	Lender, AWS Elastic Load Balancer, DynamoDB
Main Flow of events	1. Lender's cart items to be saved in the session 2. The instance running the website to be shut down 3. Instance to be directed to the next available instance decided using round robin 4. The website and lender's cart items are not affected
Alternative Flow of events	**<u>Flow 1 - Instance decided using round robin not available</u>** 1. Instance to be routed to the next available instance using round robin 2. Once instance is up, flow of events will continue from step 4 of main flow of events
Pre-conditions	Lender to have items in the cart
Post-conditions	Lender to have the same items in the cart

Use Case Title - Cache for faster website loading time	
Use Case ID	**C04**
Description	When a user accesses the website, request to route to AWS CloudFront and based on the cache in AWS CloudFront, it returns the information to the website.
Actors	AWS CloudFront, AWS ELB
Main Flow of events	1. User to enter username and password 2. Request route to AWS CloudFront 3. AWS CloudFront verifies information entered with cache 4. AWS CloudFront to send information back to website
Alternative Flow of events	**Flow 1 - Information required not available in CloudFront's Cache** 1. User to enter username and password 2. Request route to AWS CloudFront 3. AWS CloudFront verifies information entered with cache 4. Request to route to AWS ELB 5. AWS ELB verifies information entered with cache 6. AWS ELB to send information back to website 7. Request to route to AWS EC2 8. AWS EC2 to send information back to website
Pre-conditions	User to be existing user of Minimonies
Post-conditions	Correct information to be displayed on website

Use Case Title - Load Balancer & API Gateway	
Use Case ID	**C05**
Description	The user access the site via the API Gateway which sits behind the firewall in a private network and acts as a middleman. The load balancer behind the proxy helps balance the resource utilisation across multiple servers.
Actors	AWS ELB, AWS API Gateway
Main Flow of events	1. User key in the address to the website 2. Traffic hits the API Gateway which forwards to the load balancer 3. Load balancer routes to the least utilized server
Alternative Flow of events	**Flow 1 - Load Balancer Down** 1. User key in the address to the website 2. Traffic hits the main reverse proxy which forwards to the load balancer 3. Primary Load balancer fails, AWS scales out new load balancer and takes over the role as Load Balancer
Pre-conditions	Auto-scaling must be configured correctly
Post-conditions	Traffic should work as usual

Key Architectural Decisions

Architectural Decision - Microservices Architecture Style	
ID	**K01**
Issue	Decouple services,improve maintainability of environment and efficiency of deployment
Architectural Decision	Services are split into microservices to improve the maintainability of each services due to its independent nature. Any changes made will not affect the rest of the services which allows easier development and deployment. Furthermore, as the microservice encapsulates the backsend while only revealing the endpoints for developers to use, there is no need to understand how the system works but rather only the parameters to call the microservices. Thus, this improves the efficiency when developing and the scalability of the service.
Assumptions	NIL
Alternatives	Monolithic
Justifications	Monolithic architecture style is less scalable and harder to maintain as all the components are interconnected and interdependent. The entire system will have to be deployed although changes are made to one service.

Architectural Decision - Using AWS services for server environment	
ID	**K02**
Issue	The website should be highly available, high performing, scalable, and secure.
Architectural Decision	AWS services are easy to use and offers a wide variety of services to make the infrastructure secure, reliable, scalable and high performance. It also provides monitoring tool such as CloudWatch to monitor the availability of the system.
Assumptions	NIL
Alternatives	Microsoft Azure, Google Cloud
Justifications	AWS services are proven to be flexible, reliable and cost effective. It has an extensive documentation which makes it easy to use when performing web hosting services. It is also scalable and secure as it provides end to end software measure to secure the overall infrastructure

Architectural Decision - Using AWS Cognito for User Authentication	
ID	**K03**
Issue	Authentication to secure access rights within Minimonies website
Architectural Decision	AWS Cognito provides a wide range of services revolving around user management. This includes the multi-factor authentication service and the login authentication when checking against the user pool details. The user details will be stored securely in AWS Cognito which only authorised users are able to view its details. The access control helps to improve the security of the system and its reliability. Moreover, as most of the systems are deployed in AWS, AWS Cognito can support other AWS feature such as SES when emailing password reset details which will be easier for development.
Assumptions	NIL
Alternatives	NIL
Justifications	NIL

Architectural Decision - Using AWS SQS instead of AWS MQ	
ID	**K04**
Issue	To store user's information from Loans microservice to the queue, and for Notification microservice to retrieve information from the queue
Architectural Decision	AWS SQS is a lightweight, fully managed message queue and topic services that scale almost infinitely and provide simple, easy-to-use APIs
Assumptions	NIL
Alternatives	AWS MQ
Justifications	The team is building new applications in the cloud instead of moving an existing messaging application to the cloud.

Architectural Decision - Using Postman to write test cases	
ID	**K05**
Issue	To verify that the microservices are working before deploying the application
Architectural Decision	The team is focusing on AWS for the server side and hence, with Postman, we will be able to make use of AWS CodeBuild to have a CI system, along with CodeDeploy and CodePipeline.
Assumptions	NIL
Alternatives	Travis CI (PHP)
Justifications	Postman is able to integrate efficiently with AWS pipeline by using its CodeBuild services.

Architectural Decision - Developing API gateway to access individual services	
ID	**K06**
Issue	To provide access to client for the individual services
Architectural Decision	The team will develop API gateways using AWS API Gateway to access the individual services. This API gateway is able to provide client a unified point of entry without client knowing how the structure of the codes
Assumptions	NIL
Alternatives	Direct client server communication
Justifications	In order for ease of future development, we have decided to use API gateway so that we do not have to change the entire system when we edit the individual services. Moreover, we are able to hide the complexity of codes to our clients by only providing the endpoints for the services.

Development View

To achieve continuous integration and continuous development (CI/CD), our team has decided to utilise AWS CodePipeline.

Our team took on a concatenated process of a software development cycle where there are 3 main steps, which namely will be pulling from source, testing and deployment. Upon pushing our codes to Github, Github will trigger the AWS pipeline using WebHook which will pull the latest source codes from Github and store a copy on AWS S3. The codes will then be tested via AWS CodeBuild whereby the test cases will be run against the codes.

Once all test cases passed, AWS CodeDeploy will deploy all the codes to the individual production instances therefore achieving CI/CD.

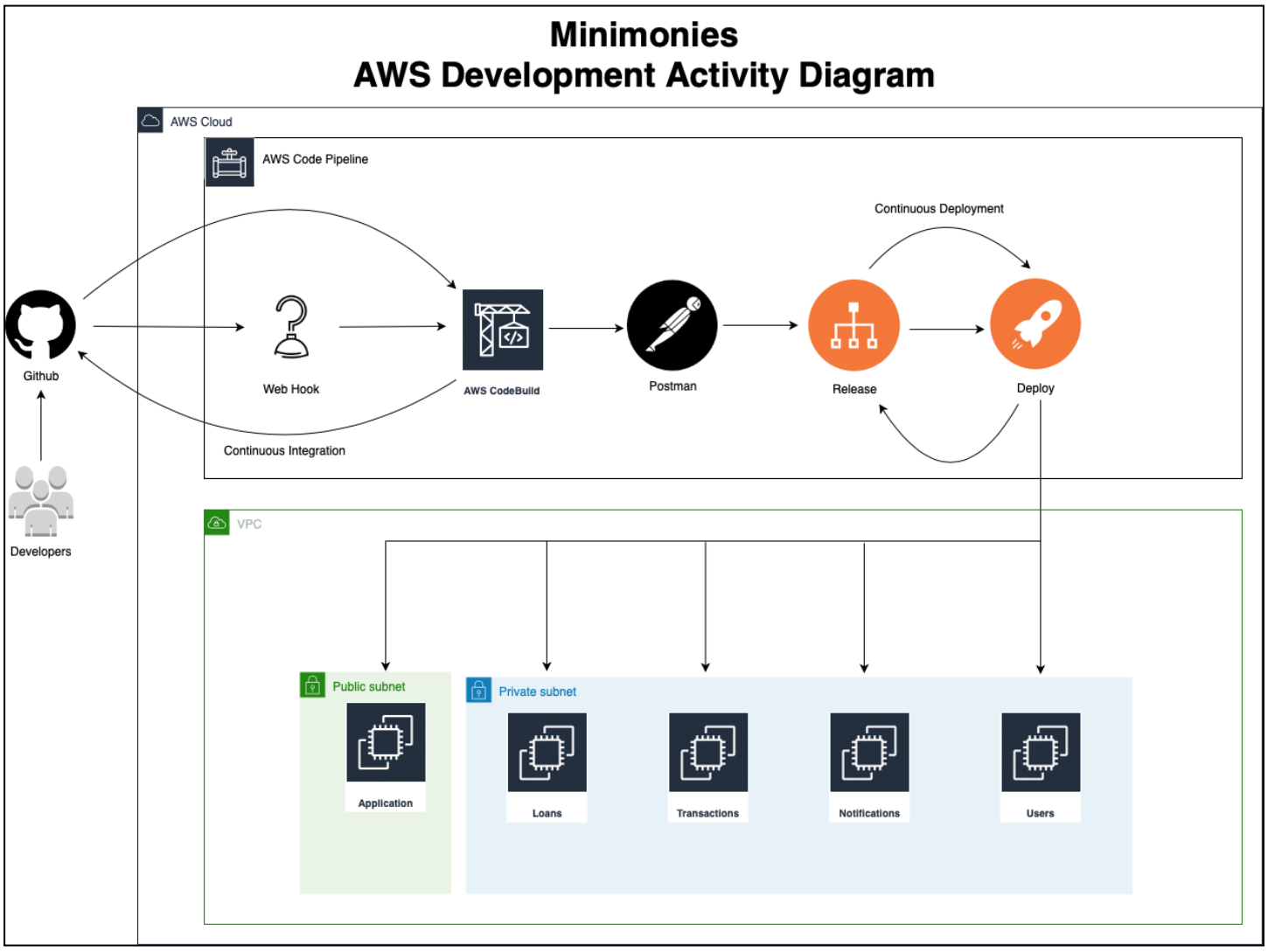

Figure DV0.1 - Development Activity Diagram

Solution View

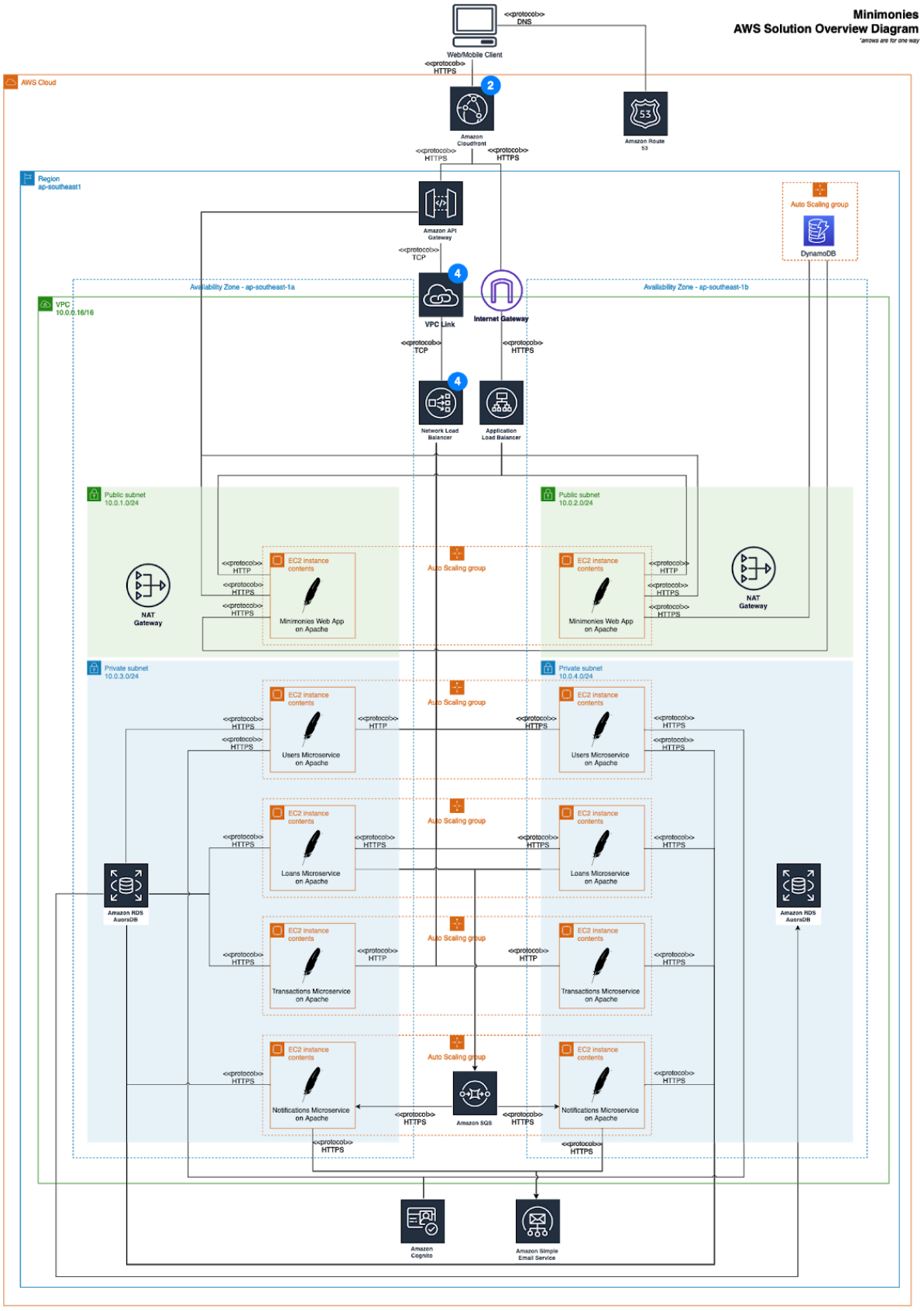

Figure SV0.1 - Overall Solution Diagram using AWS notations[1]

[1] For a clearer view, please refer to the Github repository report folder, `ITSA Solution Diagram View.svg`

Ease of Maintainability

In the proposed architecture, the team have included the microservices architecture style for our services as by decomposing the services to individual components. Changes made to a particular service will not affect other services, hence increasing maintainability of architecture. Failures incurred can be compensated quickly as well. This is similar for messaging queue, where messages are decomposed and stored in the queue and pulled when services are invoked.

The team have structured the codes such that they are decoupled as much as possible. The main reasons as to why we have decided to inherit the technique of decoupling is to make sure that each set of codes are not overly reliant on each other. A decoupled system allows changes to be made to the codes without having an affect on any other codes. Thus, increasing the maintainability factor in our system.

Design Patterns

As we are developing microservices, we researched on the industry standards on developing microservice and its endpoints. We found out that using a Facade design pattern is most suitable for this as it helps improve the maintainability of the microservices, where the facade design pattern can be used to manage codes better. Facade helps hide the complexity of the inner working of the microservices. For example, when a user needs to retrieve all the loan details, he/she simply needs to call the “findall()” method which will call the necessary methods such as the methods needed to start a database connection and prepare a SQL statement and query etc.

```php
class LoanGateway {

    private $db = null;
    private $loanId;
    private $db_name = "loans_db.loans";

    public function __construct($db, $loanId)
    {
                $this->db = $db;
                $this->loanId = $loanId;
    }

    public function findAll()
    {
        $statement = "SELECT * FROM loans_db.loans;";

        try {
            $statement = $this->db->query($statement);
            $result = $statement->fetchAll(\PDO::FETCH_ASSOC);
            return $result;
        } catch (\PDOException $e) {
            exit($e->getMessage());
        }
    }
```

Figure SV0.2 - Sample of codes

Integration Endpoints

Source System	Destination System	Protocol	Format	Communication Mode
Client Device	Amazon Route53	DNS	DNS Message	Synchronous
Minimonies Web Application	Amazon CloudFront	HTTPS	HTML	Synchronous
Amazon CloudFront	Application Load Balancer	HTTPS	HTML	Synchronous
Application Load Balancer	Minimonies EC2 Instance	HTTPS	HTML	Synchronous
Minimonies EC2 Instance	Amazon CloudFront	HTTPS	HTML	Synchronous
Minimonies EC2 Instance	DynamoDB	HTTPS	JSON	Synchronous
Amazon CloudFront	Amazon API Gateway	HTTPS	JSON	Synchronous
Amazon API Gateway	Network Load Balancer	HTTPS	JSON	Synchronous
Network Load Balancer	User Microservice	HTTP	JSON	Synchronous
Network Load Balancer	Loans Microservice	HTTP	JSON	Synchronous
Network Load Balancer	Transaction Microservice	HTTP	JSON	Synchronous
User Microservice	Amazon Cognito	HTTPS	JSON	Synchronous
Loans Microservice	Amazon SQS	HTTPS	JSON	Synchronous
Amazon SQS	Notification Microservice	HTTPS	JSON	Synchronous
Notification Microservice	Amazon SES	HTTPS	JSON	Synchronous
User Microservice	AuroraDB	HTTPS	Queries	Synchronous
Loans Microservice	AuroraDB	HTTPS	Queries	Synchronous
Notification Microservice	AuroraDB	HTTPS	Queries	Synchronous
Transaction Microservice	AuroraDB	HTTPS	Queries	Synchronous

Hardware/Software/Services Required

There are no procurement of physical hardware that is involved in this project. All hardware needed to power this project are launched using Amazon Web Services and therefore classified as an operating cost for services. Software involved are Open-source.

The following is a breakdown in the relevant cost involved in this project.

No	Item	Provider	Type	Quantity	License	Remarks	Charging Scheme	Unit Cost	Total Cost
1	Apache with PHP	Apache Foundation	Software	1	Open - Source		One-Time	$0.0000	$0.0000
2	API Gateway	AWS	Service	1	Proprietary	First 333 million $4.25	Block	$4.2500	$4.2500
3	Application Load Balancer	AWS	Service	1	Proprietary	$0.0252 per Application Load Balancer-hour (or partial hour) $0.008 per LCU-hour (or partial hour)	Hourly	$0.0332	$0.0332
4	Aurora Database	AWS	Service	1	Proprietary	$0.11 per GB-month $0.22 per 1 million requests	Block	$0.3300	$0.3300
5	AWS Linux AMI	AWS	Software	1	Open - Source		One-Time	$0.0000	$0.0000
6	CloudFront (Content Delivery Network)	AWS	Service	1	Proprietary	First 10 TB	Block	$0.1400	$0.1400
7	Cognito	AWS	Service	1	Proprietary	First 50,000 Free	Block	$0.0000	$0.0000
8	DynamoDB	AWS	Service	1	Proprietary	$1.25 per million writes $0.25 per million reads	Block	$1.5000	$1.5000
9	EC2 - t2.nano	AWS	Service	5	N.A	Subjected to AutoScale, min 5 instance	Hourly	$0.0073	$0.0365
10	Network Load Balancer	AWS	Service	4	Proprietary	$0.0252 per Network Load Balancer-hour (or partial hour) $0.006 per LCU-hour (or partial hour)	Hourly	$0.0312	$0.1248
11	Route53	AWS	Service	1	Proprietary	$0.5 per hosted zone	Block	$0.5000	$0.5000
12	Simple Email Service	AWS	Service	1	Proprietary	First 62,000 emails Free each month, $0.1 per 1000 emails	Block	$0.1000	$0.1000
13	Simple Queue Service	AWS	Service	1	Proprietary	First 1 million Requests Free, charged by per 1m req	Block	$0.4000	$0.4000

Sub-Total (Hourly) : $0.19

Sub-Total (Block) : $7.22

Total for 24 hours : $11.89

Availability View

Node	Redundancy	Clustering			Replication			
		Node Config.	Failure Detection	Failover	Repl. Type	Session State Storage	DB Repl. Config	Repl. Mod
EC2	Horizontal Scaling	Active - Active	Heartbeat	Load Balancer	N.A			
API Gateway	Horizontal Scaling	Active - Passive	Heartbeat	DNS	N.A			
CloudFront	Horizontal Scaling	Active - Active	Heartbeat	DNS	N.A			
Aurora DB	Horizontal Scaling	Active - Active	Heartbeat	DNS	DB	Database	Master - Slave	Synchronous
Dynamo DB	Horizontal Scaling	Active - Active	Heartbeat	DNS	Session	Client	Master - Slave	Synchronous

*Please refer to Annex (PV1.1 and 1.2) for more information on the scaling of DynamoDB.

Auto Scaling

To ensure high availability for this application, we configured auto-scaling of the systems using the Amazon EC2 Auto Scaling function. This function essentially provides better fault tolerance and better availability to the overall system.

Amazon EC2 Auto Scaling[2] can detect when an instance is unhealthy, terminate it, and launch an instance to replace it. This feature uses multiple Availability Zones where if one Availability Zone becomes unavailable, it will launch instances in another Availability Zone.

In the event of high traffic or when an instance unexpectedly terminates or suffers a downtime, Amazon EC2 Auto Scaling will launch a new instance to help handle the traffic. Vice versa, when there is low traffic, the system will scale in to reduce the number of instances. Therefore, Amazon EC2 Auto Scaling helps ensure that the application always has the right amount of capacity to handle the current traffic demand.

[2] "Benefits of Auto Scaling - Amazon EC2 Auto Scaling - AWS ...." https://docs.aws.amazon.com/autoscaling/ec2/userguide/auto-scaling-benefits.html. Accessed 9 Nov. 2019.

CloudFront - Content Delivery Network

The Content Delivery Network (CDN) helps increase the web application availability as web apps often spikes in traffic during peak periods of activity. By using Amazon CloudFront as a CDN, it can cache the web app's content in CloudFront's edge locations worldwide and which helps to reduce the workload on origin instances by only fetching content from the origin instance when needed[3]. This reduced workload on the origin instance helps increase the availability of the web app.

Session Stickiness

Session stickiness helps improve the web application's performance and availability as it helps minimize data exchange as servers within your network don't need to keep exchanging session data with DynamoDB, which takes a toll on DynamoDB when done in scale.

However, we note that servers can become overloaded if a specific sticky sessions by a user requires a high number of resources.

Sequence Diagrams

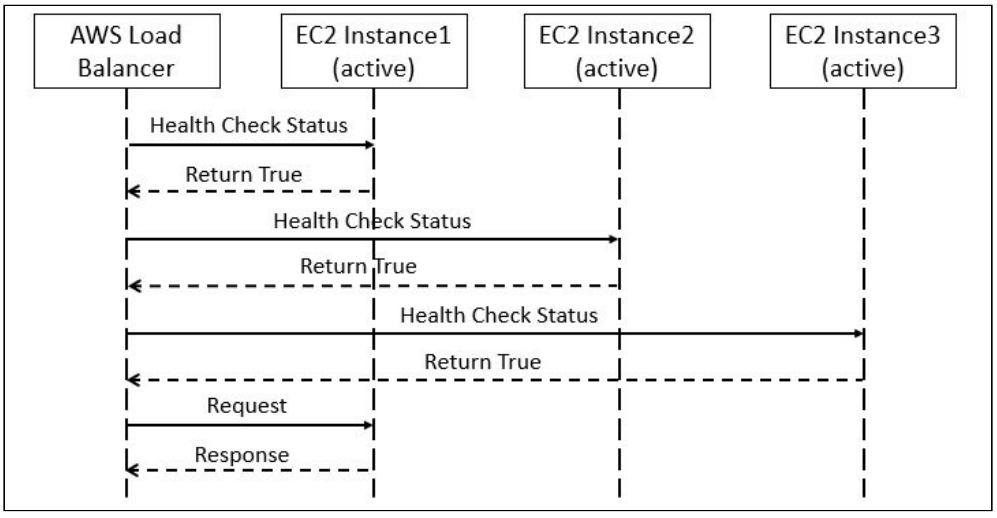

Figure AV0.1 - EC2 handled with no server failure

The load balancer will make heartbeat request to all instances to ensure all are up and running (Figure AV0.1). When requests are made, the load balancer will route the request to the instance depending on the preconfigured algorithm, which in this case we have chosen round robin to ensure that each instance has an equal chance.

[3] "Amazon CloudFront - Amazon Web Services." https://aws.amazon.com/cloudfront/features/. Accessed 09 Nov. 2019.

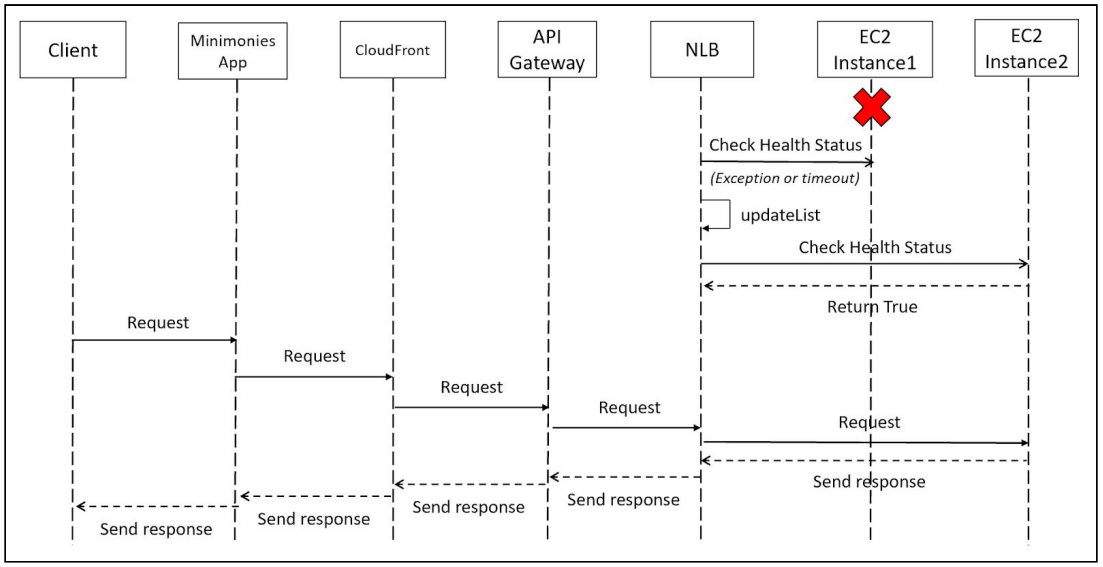

Figure AV0.2 - EC2 Instance fails before Request

When an EC2 instance is down (Figure AV0.2), the Network Load Balancer (NLB) will still behave as usual and send heartbeat request periodically to all instances and takes note of instances which did not send back a response. All user requests will then be passed to the available instances which have responded to the heartbeat request.

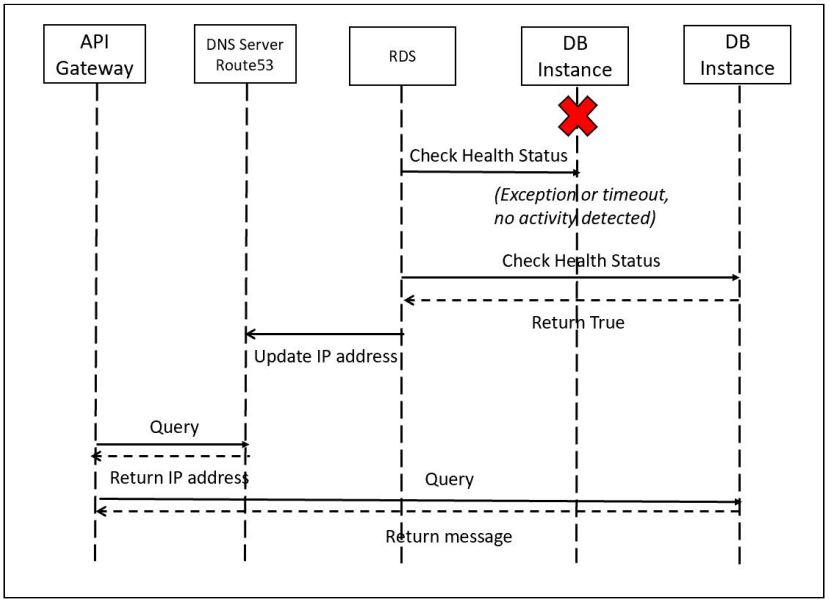

Figure AV0.3 - DB Instance of AuroraDB fails before Request

Amazon RDS is a web service to manage and scale relational databases in AWS Cloud. For this project, the RDS has been Multi-Availability Zone enabled, therefore in the event of a planned or unplanned outage of the DB instance, Amazon RDS automatically switches to a standby replica (Figure AV0.3). The failover works where the RDS service with automatically update the DNS record of the DB instance to point to the standby DB instance.[4]

[4] "High Availability (Multi-AZ) for Amazon RDS - Amazon ...." https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.MultiAZ.html. Accessed 9 Nov. 2019.

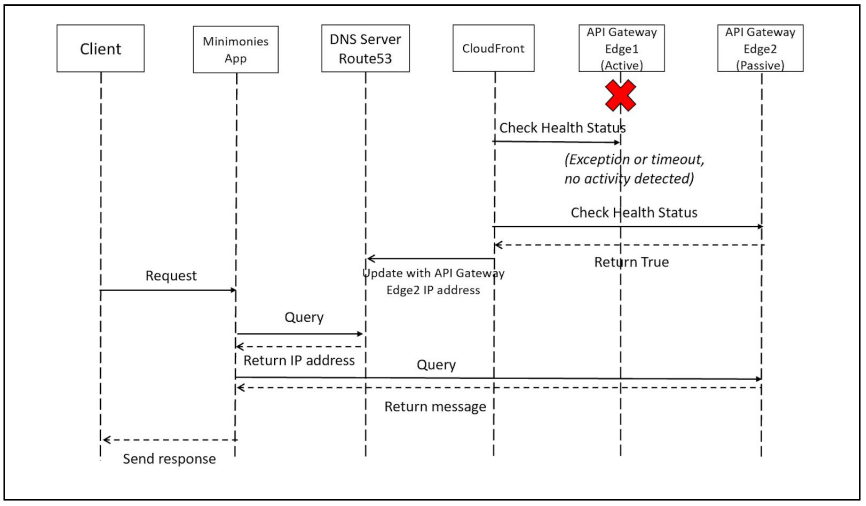

Figure AV0.4 - API Gateway fails before Request

As the API Gateway has been Edge Optimized by Cloudfront, CloudFront will frequently send a heartbeat to the API gateway across different availability zones to check their health status. In the event where the active API Gateway fails, CloudFront will update Route53 to the API Gateway that is available at another availability zone.

Security View

No	Asset/Asset	Potential Threat/Vulnerability	Possible Mitigation Controls
1	Server session data / User identity	Broken Authentication & Session Management - Integrity, Confidentiality If an attacker is able to gain access to the session identifiers and authentication credentials, the attacker is able to gain access to the current session and act as the user	Ensure that SSL is being used for all parts of the authentication of the application. In addition, we can store the user credentials in hashed form to prevent the attacker from knowing what the session details are
2	Lender / Borrower personal data	SQL Injection - Integrity, Confidentiality An attacker can potentially be able change values in the database by injecting malicious input into a SQL statement. This is normally done over a web application. SQL injection can allow an attacker to retrieve records from database or even modify records in the database	Ensure that all inputs are validated and prepared statements are evident in parameterized queries.
3	Server session data / User Identity	Cross Site Scripting (XSS) - Integrity, Confidentiality XSS main target is to execute malicious scripts on either the client or server side. Attacker can either send the malicious code directly to the server or create a malicious link for the user to click on which will redirect them to the website with malicious code. After the script is being executed, the attacker is able to access information such as session cookies or even run viruses on the victim’s machine.	A combination of validating input and sanitizing data. We can validate the input and ensure that the application is rendering the correct data. We can also check the server inputs and outputs to make sure that it is acting as it is expected to be.

4	User sensitive information	Insecure Direct Object References - Integrity This occurs when the system architect exposes the a reference of an object on the website as a form parameter. The attacker can use this parameter to access unauthorized data.	One way to mitigate this is to use indirect references. We should replace the references with random values. The mapping between the random values and random values are stored on the server. This will not expose the references to public.

Other Security Considerations

1. Mozilla Industry Standards

   Following the guidelines of Mozilla Observatory, the system is improved to be more secured for use. As shown in Annex (Figure SV.1.1), Minimonies has achieved an A+ for the security of the system by implementing the necessary security headers to address common web issues such as Cross Site Scripting.

2. AWS Virtual Private Cloud

   All microservices are deployed within an AWS Virtual Private Cloud (VPC) which provides enhanced security due to its isolation from the main features. The VPC also provide network access control to prevent unauthorised access to microservices. (Figure DV0.1)

3. Public/Private Subnet Segregation

   We configured our VPC with a public subnet and a private subnet as seen in Figure SV0.1. To enhance on our architecture security, we placed all services with potentially sensitive information in the private subnets, exposing on the main application on the public subnet. In addition, we set up a virtual private gateway via IPsec VPN tunnel to allow communication with our own network.

4. API Key for API Gateway

   An API key is required to access certain microservice with sensitive information such as user details to enhance the security of the system. Requests without API key will be denied access.

Performance View

No	Description of the Strategy	Justification
1	Content Delivery Network (CDN) CloudFront	CDN system has its distributed servers known as edge servers around the globe[5] that can act as a cache that can store almost any type of content like images, CSS, JavaScript, HTML.
2	Load Balancer (LB) with Auto Scaling	Load balancing evenly distribute network traffic across multiple instances[6] and prevents failure caused by overloading a particular resource. AWS Auto Scaling monitors the application and automatically adjust[7] the capacity of the system to maintain high performance. Integrating LB with auto scaling improves the performance of the system when traffic are assigned evenly to the instances.
3	HTTP Caching Client-side Cache	HTTP caching stores local copies of resources for faster retrieval[8] when the user accesses the resources the next time is required.

Performance Testing

For more information, please refer to the Annex (Figure PV1.1 to PV2.3) for the raw results of the JMeter Testing.

Test	Direct	Load Balancer	Content Delivery Network
30 Users, 30 Ramp-up	1618 ms	1029 ms	617 ms
100 Users, 1 Ramp-up	62536 ms	13177 ms	10844 ms

[5] "What is a CDN edge server? | Cloudflare." https://www.cloudflare.com/learning/cdn/glossary/edge-server/. Accessed 9 Nov. 2019.
[6] "How Elastic Load Balancing Works - AWS Documentation." https://docs.aws.amazon.com/elasticloadbalancing/latest/userguide/how-elastic-load-balancing-works.html. Accessed 9 Nov. 2019.
[7] "AWS Auto Scaling." https://aws.amazon.com/autoscaling/. Accessed 9 Nov. 2019.
[8] "Increasing Application Performance with HTTP Cache ...." 9 Oct. 2019, https://devcenter.heroku.com/articles/increasing-application-performance-with-http-cache-headers. Accessed 9 Nov. 2019.

Annex

Availability View

DynamoDB Benefits

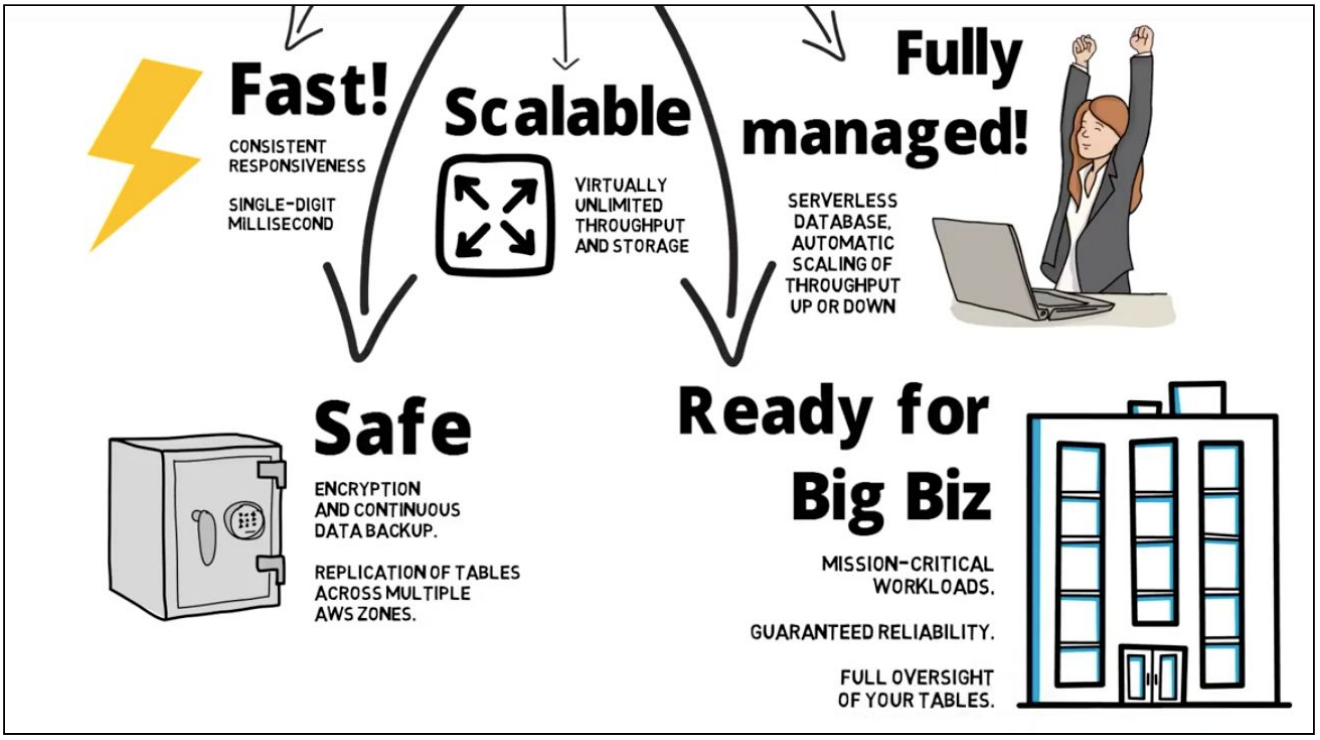

Figure AV1.1 - DynamoDB Benefits from AWS

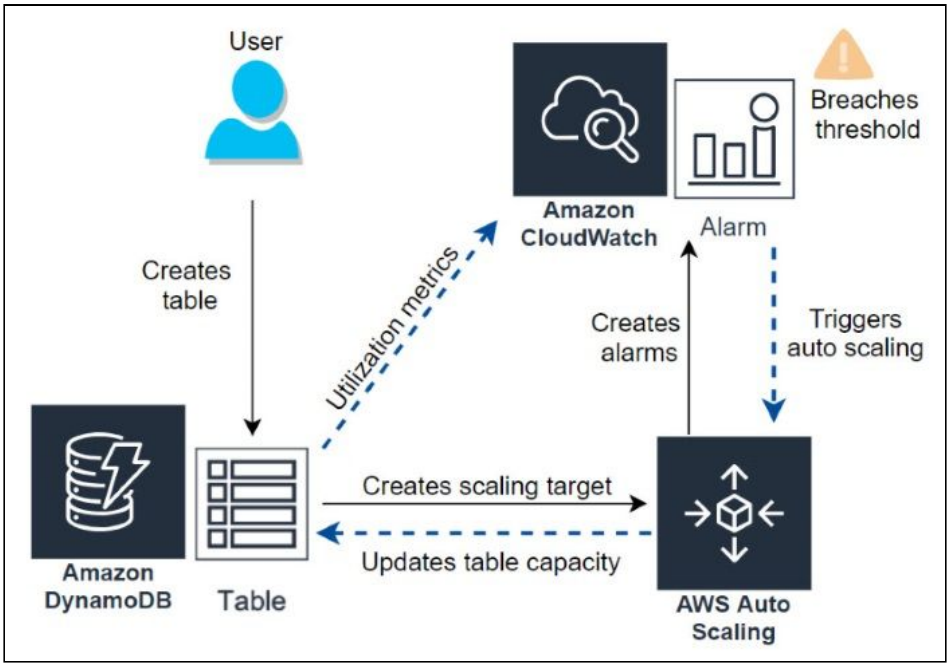

Figure AV1.2 - How does DynamoDB scale

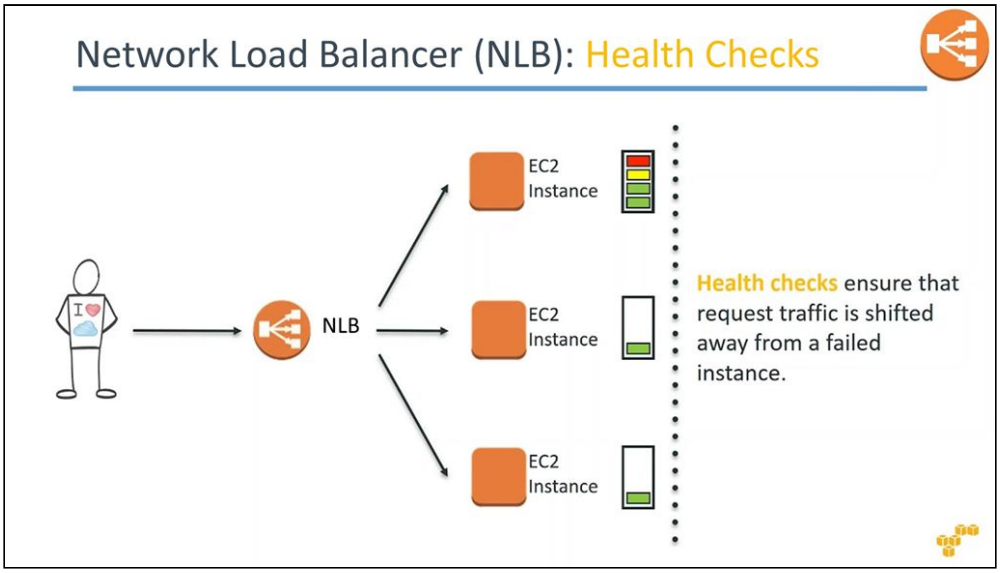

Figure AV1.3 - How does Network Load Balancer direct request traffic

DB identifier	Role	Engine	Region & AZ	Size	Status
minimonies-db	Regional	Aurora MySQL	ap-southeast-1	2 instances	Available
minimonies-db-instance-1	Writer	Aurora MySQL	ap-southeast-1a	db.t2.small	Available
minimonies-db-instance-1-ap-southeast-1b	Reader	Aurora MySQL	ap-southeast-1b	db.t2.small	Available

Figure AV1.4 - AWS RDS, AuroraDB master-slave configuration

Route 53 HealthChecks

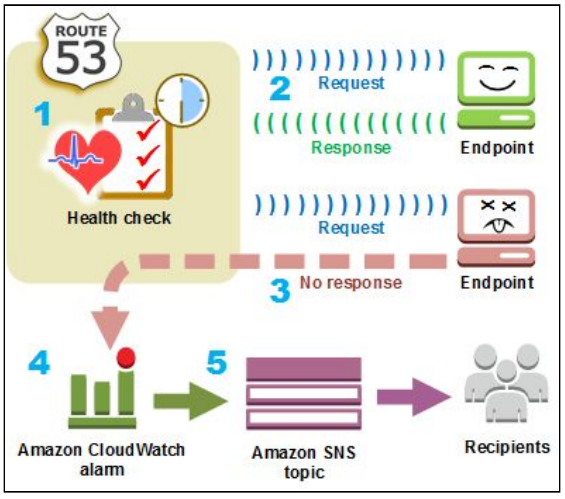

Figure AV1.5 - Route 53 Health Checks sequence

Security View

Mozilla Observatory Results

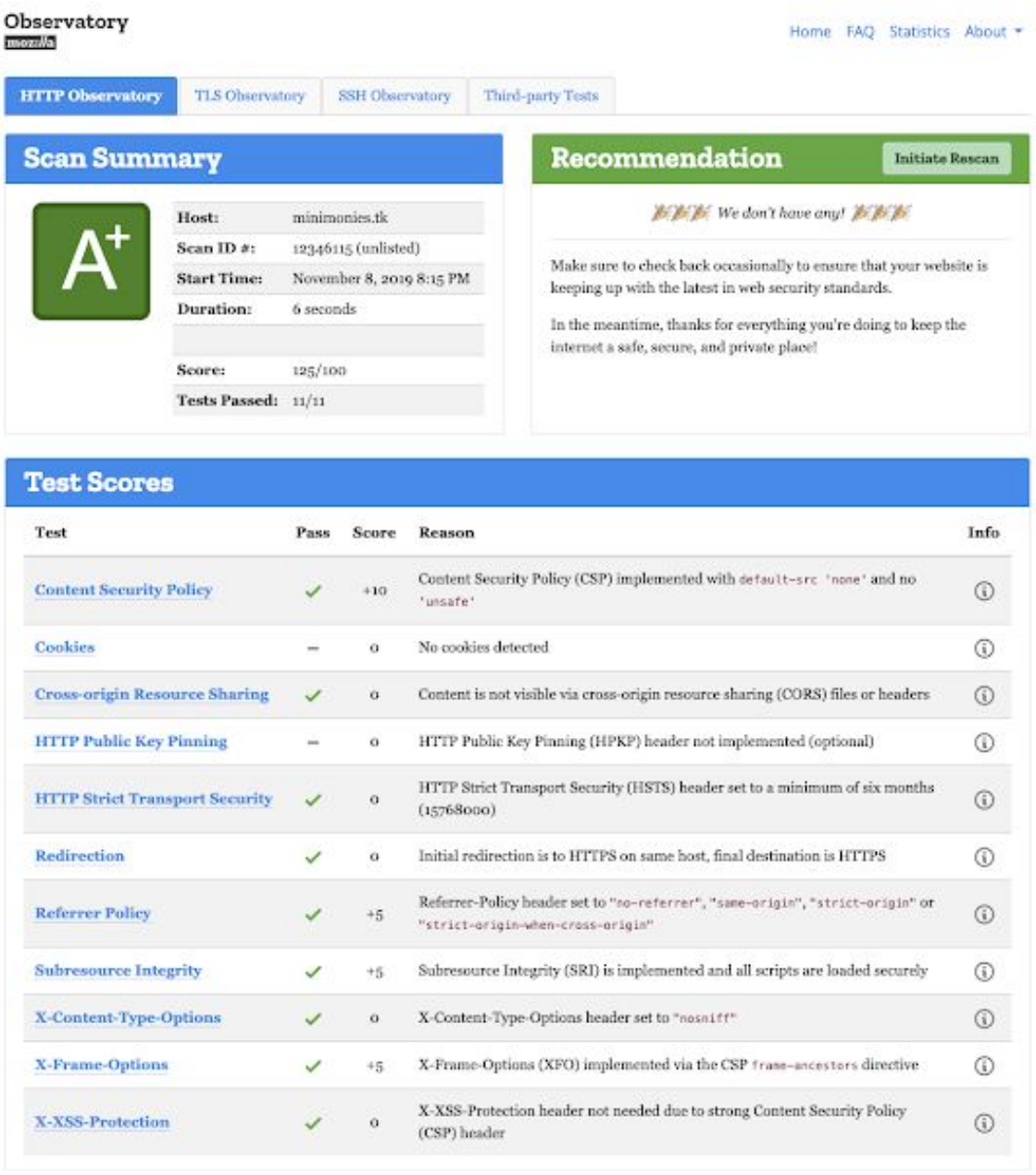

Figure SV.1.1 - Mozilla Observatory Result

Performance View

Raw Performance Testing JMeter Results

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
1 /	90	1428	193	10081	1348.55	1.111%	2.85018	318.52	10.93	114436.2
2 /login.php	90	1613	231	2597	606.07	0.000%	2.83948	317.90	16.59	114643.8
3 /lend/index.php	90	1768	208	2928	571.66	0.000%	2.84154	318.18	16.61	114662.4
4 /lend/viewAvailableLoans.php	90	1640	235	3948	702.15	0.000%	2.88277	322.90	16.89	114698.3
5 /lend/shoppingCart.php	90	1641	197	3185	653.92	0.000%	2.99601	335.56	17.53	114691.3
TOTAL	450	1618	193	10081	835.80	0.222%	13.74298	1538.39	74.85	114626.4

Figure PV.1.1 - JMeter test on Instance (EC2) 30 users, 30 ramp-up

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
1 /	90	486	139	3458	485.41	0.000%	4.30231	480.04	17.39	114254.2
2 /login.php	90	1165	208	11078	2175.60	0.000%	4.39840	491.96	26.82	114534.9
3 /lend/index.php	90	762	199	9682	1488.03	0.000%	4.43088	495.53	27.02	114520.5
4 /lend/viewAvailableLoans.php	90	1717	207	16012	3234.63	0.000%	3.94616	441.34	24.12	114525.2
5 /lend/shoppingCart.php	90	1014	207	13955	2200.48	0.000%	4.02199	449.87	24.56	114538.3
TOTAL	450	1029	139	16012	2160.87	0.000%	18.49492	2067.58	105.25	114474.6

Figure PV.1.2 - JMeter test on Load Balancer (AWS Application LB) 30 users, 30 ramp-up

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
1 /	90	505	226	1913	323.25	0.00%	8.31255	954.84	32.04	117623.8
2 /login.php	90	553	297	1825	266.43	0.00%	8.40022	974.12	49.25	118746.8
3 /lend/index.php	90	667	282	3277	483.32	0.00%	8.42933	977.8	49.43	118784.2
4 /lend/viewAvailableLoans.php	90	697	300	4098	545.72	0.00%	8.47218	982.5	49.79	118751.3
5 /lend/shoppingCart.php	90	666	287	3542	503.41	0.00%	8.60997	998.44	50.55	118745.8
TOTAL	450	617	226	4098	444.59	0.00%	37.24241	4310.9	203.57	118530.4

Figure PV.1.3 - JMeter test on Content Delivery Network (CloudFront) 30 users, 30 ramp-up

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
1 /	100	13522	412	96840	15281.71	0.000%	1.02407	114.39	3.93	114387.5
2 /login.php	100	154595	347	235530	77648.88	16.000%	.41432	39.57	2.06	97797.8
3 /lend/index.php	100	49628	440	230477	56846.98	7.000%	.40545	42.29	2.20	106817.5
4 /lend/viewAvailableLoans.php	100	19972	459	211103	41011.08	0.000%	.26663	29.87	1.56	114707.4
5 /lend/shoppingCart.php	100	74965	1025	208258	63866.40	3.000%	.22096	24.52	1.28	113615.1
TOTAL	500	62536	347	235530	75186.87	5.200%	1.10057	117.65	5.70	109465.1

Figure PV.2.1 - JMeter test on Instance (EC2) 100 users, 1 ramp-up

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
1 /	100	2997	1316	8682	1224.93	0.000%	10.16570	1134.52	41.09	114281.3
2 /login.php	100	8055	451	68682	16221.08	3.000%	1.38190	152.96	8.36	113343.4
3 /lend/index.php	100	20384	238	80976	21573.04	13.000%	1.20620	121.43	6.70	103091.2
4 /lend/viewAvailableLoans.php	100	22240	222	67864	23010.72	18.000%	1.22806	118.11	6.56	98486.9
5 /lend/shoppingCart.php	100	12207	199	60124	19141.37	9.000%	1.19916	122.08	6.73	104247.9
TOTAL	500	13177	199	80976	19446.28	8.600%	5.49765	572.80	29.24	106690.1

Figure PV.2.2 - JMeter test on Load Balancer (AWS Application LB) 100 users, 1 ramp-up

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
1 /	100	17081	1204	92088	19622.15	1.000%	1.08487	124.66	4.18	117668.4
2 /login.php	100	11731	292	63572	18611.15	5.000%	1.09769	121.00	6.14	112875.0
3 /lend/index.php	100	10274	290	60576	15667.95	0.000%	1.09907	127.47	6.44	118766.8
4 /lend/viewAvailableLoans.php	100	10006	296	80048	17056.43	1.000%	1.10273	127.95	6.48	118814.8
5 /lend/shoppingCart.php	100	5128	280	33166	10509.22	0.000%	1.10835	128.54	6.51	118761.4
TOTAL	500	10844	280	92088	17039.02	1.400%	5.34605	612.80	28.92	117377.3

Figure PV.2.3 - JMeter test on Content Delivery Network (CloudFront) 100 users, 1 ramp-up

References

https://www.striim.com/blog/2018/07/streaming-architecture-logical-replication-vs-streaming-data-integration/

https://medium.com/faun/deploy-a-php-application-on-ec2-with-github-and-aws-codepipeline-fb38cf204cbb

https://www.codeofaninja.com/2017/02/create-simple-rest-api-in-php.html

https://docs.aws.amazon.com/ses/latest/DeveloperGuide/send-email-smtp.html

https://phpunit.readthedocs.io/en/8.3/writing-tests-for-phpunit.html

https://aws.amazon.com/codedeploy/features/?nc=sn&loc=2

https://medium.com/@mwaysolutions/10-best-practices-for-better-restful-api-cbe81b06f291

https://developer.okta.com/blog/2019/03/08/simple-rest-api-php

https://aws.amazon.com/architecture/icons/

https://aws.amazon.com/blogs/database/amazon-dynamodb-auto-scaling-performance-and-cost-optimization-at-any-scale/

https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.MultiAZ.html

https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/welcome-health-checks.html