CS301 - IT Solution Architecture

AY 2021/22 Term 1

Final Project Proposal

Prepared for:
Professor OUH Eng Lieh

Done by:
Team 8 (Github: G2team8)

Name	Student ID
Ho Jing Yi	01375797
Lee Sean Jin	01337831
Lim Zhong Zhen Timothy	01348521
Soh Bai He	01375548
Tan Song Yuan Daniel	01332737

Background and Business Needs

COMO Group is a company headquartered in Singapore, with a wide variety of systems across different business units. At the heart of COMO Group lies ComoClub with an app that allows customers to redeem or purchase exclusive experiences and items across COMO's Business Units. This requires the application to leverage on and integrate with the existing systems in COMO.

Currently, each business unit system is separated from one another and there is no middleware to manage integration between these systems. A lack of integration creates information silos which creates inefficiencies and redundancies across the business.

Our proposed system architecture includes a middleware which serves to integrate and consolidate existing systems. It simplifies the connection between the app to multiple 3rd party APIs like Stripe, Memberson and 7Rooms. It is also designed to handle future integrations from other Business Units and to onboard new external APIs easily.

Stakeholders

Stakeholder	Stakeholder Description	Permissions
ComoClub's ops managers	In charge of liaising with the partner merchants of Club21, curate and upload pictures and descriptions of the experiences to Memberson CRM.	Read, Write
ComoClub's Maintenance team	In charge of scalability, availability and data security of our proposed middleware.	Read, Write
ComoClub's Developers	In charge of developing ComoClub's Club 21 application	Read, Write
App Users	Users of the Club21 application	N.A.
7rooms, Memberson CRM	External API providers	N.A.

Key Use Cases

Use Case Title: Login	
Use Case ID	1
Description	This use case describes how a user logs into the ComoClub app.
Actors	Users, Memberson CRM
Main Flow of Events	1. User enters his/her name and password in the frontend. 2. POST request made to /login endpoint with username and password. 3. Lambda function verifies if the user is registered in Memberson CRM. 4. Lambda function verifies if the user exists in AWS Cognito. If the user does not exist, create a new user in AWS Cognito. 5. The lambda function returns success response status to the frontend. 6. Frontend stores the user's first name, last name, mobile number and number of points in state. 7. If the lambda function returns a successful response, a login request is made to AWS Cognito via the SDK to authenticate the user. 8. On successful authentication, a pair of access tokens and refresh tokens are returned.
Alternative Flow of Events	If the user enters an invalid name and/or password, the app displays an error message. The user fails to login.
Pre-conditions	-
Post-conditions	The user is now logged into the app and directed to the app's homepage. The user's first name and number of points is displayed on the navigation bar. If the use case was unsuccessful, the state is unchanged.

Use Case Title: View list of experiences	
Use Case ID	2.1
Description	This use case describes how a user views the list of experiences available for booking.
Actors	Users, 7Rooms API Service, 7Rooms Middleware
Main Flow of Events	1. On successful login, the user is redirected to the /experience page. 2. GET request made to /sevenrooms/venue to retrieve a list of experiences. 3. Frontend displays a list of experiences to the user.
Alternative Flow of Events	7Rooms Middleware is unable to connect to the 7Rooms API endpoint. API error response returned.
Pre-conditions	User successfully logs in.
Post-conditions	-

Use Case Title: View details of an experience	
Use Case ID	2.2
Description	This use case describes how a user selects and views details of an experience (from various 3rd party endpoints).
Actors	Users
Main Flow of Events	1. User selects one of the experiences from the curated list of experiences. 2. User is directed to the /experience/{id} page. 3. Frontend displays details (from 2.1) of the selected experience to the user.
Alternative Flow of Events	-
Pre-conditions	User is currently at the /experience page.
Post-conditions	User to be at the /experience/{id} page.

Use Case Title: Book an experience with COMO points	
Use Case ID	3.1
Description	This use case describes how a user books an experience with their available COMO points.
Actors	Users, 7Rooms API Service, Memberson CRM, Memberson Middleware, 7Rooms Middleware
Main Flow of Events	1. User selects their desired date as well as the number of tickets to be purchased. 2. GET request made to /sevenrooms/venue/availability/{venueId}?date={chosenDate}&start_time=00:00&end_time=23:00&party_size={chosenPartySize} 3. Frontend displays time slots available based on date and number of tickets chosen 4. User selects their desired time and payment method “Pay with COMO credits”, then clicks “Proceeds to checkout.” 5. User is directed to /experience/{id}/checkout. 6. App verifies the user's number of available points against the points required for redemption by checking the user's available points against the number of points stored in Frontend state. 7. If points are sufficient, the app shows a summary of the user's booking details and requests the user to confirm purchase details. 8. User clicks on the “confirm payment” button. 9. The app makes a POST request to Memberson middleware (memberson/points/redeem) to deduct user’s points in and sends booking information to (sevenrooms/reservation/fail/{venueId}) which will forward to Memberson and 7Rooms respectively 10. The app displays success message and the user can check confirmed booking details in ‘My Bookings’ page
Alternative Flow of Events	Users do not have enough COMO credits and have to pay with a credit card.
Pre-conditions	User is currently at the /experience/{id} page.
Post-conditions	Booking successful. User to be at the /experience/{checkout}/success page.

Use Case Title: Book an experience with credit card	
Use Case ID	3.2
Description	This use case describes how a user books an experience with a credit card.
Actors	Users, 7Rooms API Service, Memberson CRM, Stripe, 7Rooms Middleware, Stripe Middleware
Main Flow of Events	1. User selects their desired date as well as the number of tickets to be purchased. 2. GET request made to /sevenrooms/venue/availability/{venueId}?date={chosenDate}&start_time=00:00&end_time=23:00&party_size={chosenPartySize} 3. Frontend displays time slots available based on date and number of tickets chosen 4. User selects their desired time and payment method “Pay by credit card”, then proceeds to checkout. 5. The app shows a summary of the user's booking details and requests the user to confirm purchase details. 6. Our Stripe middleware creates a Payment Intent. 7. User fills in payment details on the checkout form. 8. The app checks if a valid card number is inputted. 9. User clicks on the “confirm payment” button. 10. Stripe processes the payment. 11. Once payment is successful, the app sends booking information to our 7Rooms middleware which will forward to 7Rooms. 12. The app displays a success message and the user can check confirmed booking details in the ‘My Bookings’ page.
Alternative Flow of Events	Payment process failed.
Pre-conditions	User is currently at the /experience/{id} page.
Post-conditions	Booking successful. User to be at the /experience/{checkout}/success page.

Use Case Title: View a list of bookings made by the user	
Use Case ID	4
Description	This use case describes how a user views a list of confirmed booking details
Actors	Users, 7Rooms API Service, 7Rooms Middleware
Main Flow of Events	1. User clicks on "My Bookings" 2. User is directed to /booking 3. GET request made to /sevenrooms/reservation/search?venueId={venueId}&name={firstName} 4. Frontend displays a list of bookings made by the user
Alternative Flow of Events	7Rooms Middleware is unable to connect to the 7Rooms API endpoint. API error response returned.
Pre-conditions	User is logged in.
Post-conditions	User to be at the /booking page.

Sequence Diagram (Key Use Cases)

Refer to Appendix 1 for Sequence Diagram of key use cases and Appendix 2 for Frontend UI.

Quality Requirements

Maintainability	System must be able to handle daily deployments with no (<1 minute) downtime allowed. Automated health checks should be put in place for AWS Services.
Availability	System must be 99.9% available during normal operation hours.
	Mission critical systems must be able to recover from failure in less than 15 minutes (RTPO).
Performance	API round-trip time from a request to a response should be less than 2 seconds under normal operations with 70 concurrent users.
	API round-trip time from a request to a response should be less than 3 seconds under peak operations with 300 concurrent users.
Scalability	System must be able to support 300 requests per second during the 6pm to 7pm peak hour.
	System must be able to support 70 requests (on average) per second during normal operations throughout the day.
Security	Personal information should be encrypted for data at rest, motion and in use, closely following PCI, HIPAA, GDPR compliance.

Key Architectural Decisions

Architectural Decision: Microservices	
ID	AD1
Issue	Traditional architectures are designed such that all processes are tightly coupled and run as a single service. Due to the nature of a tightly coupled app, it is harder and more resource-intensive to implement changes, which makes the application harder to maintain as the app grows.
Architectural Decision	With a microservices architecture, our application is built as independent components that run each application process as a service.

Assumptions	None
Alternatives	Monolithic Application
Justification	By using microservices architecture, we are decoupling the backend and frontend of our application. As compared to a monolithic application, microservices application allows for faster development time. Microservices applications are also less prone to failures as functionalities are dispersed across different microservices, with each service running independently from each other. This results in applications that are more resilient and perform better as compared to a monolithic application. If the monolithic approach is taken we would lose the agility of developing new functionality or modifying existing functionality as each function will be tightly coupled with other functions. This drastically reduces the maintainability of our system which is undesirable.

Architectural Decision: Frontend managed by Amplify	
ID	AD2
Issue	Webpage is required to be high performing and scalable to cater to multiple users. It is expensive to host our static webpage on EC2 instances.
Architectural Decision	AWS Amplify offers a fully managed service for deploying and hosting static web applications with built-in CI/CD workflows, provides native support for React which our frontend is built on, and provides seamless integration with S3, Cloudfront, Route 53 and Cognito.
Assumptions	Application has to be a static webpage
Alternatives	Deploying the website on an EC2 instance
Justification	AWS Amplify is fuss-free, it seamlessly integrates our frontend application with S3, Cloudfront, Route 53 and Cognito for us automatically which reduces the complexity and difficulty of hosting our frontend as compared to other alternatives. AWS Amplify also ensures that our application is highly available with high performance through static webpage hosting via S3 and content delivery via CloudFront. If the alternative is chosen, we would have to manually configure and setup proper load balancing and redundancy to ensure high availability of our frontend application. We would also need to setup integration with AWS Cloudfront to ensure performance and AWS Cognito for user authentication. This would add additional complexity and workload in maintaining this bundle of services ourselves which is not as maintainable as using AWS Amplify.

Architectural Decision: Authentication with authorization tokens	
ID	AD3
Issue	Proper user control is important as user accounts are able to make payment and redeem points. Securing our API endpoints is essential to ensure that our systems are not compromised.
Architectural Decision	Our architecture ensures end to end user authentication where only registered users in Memberson CRM are able to access our services. Our Cognito user pool is synced with Memberson CRM's active users. AWS Cognito is integrated with AWS Amplify to secure frontend pages and is integrated with AWS API Gateway to secure API endpoints with OAUTH 2.0 authorisation. This prevents our client facing endpoints from being publicly accessible. Once AWS Cognito has authenticated the user, requests can

	be made to our API Gateway which will then route the request to our middleware microservices. Our microservices will then authenticate to external API endpoints on behalf of the user.
Assumptions	None
Alternatives	None
Justification	None

Architectural Decision: Infrastructure as code	
ID	AD4
Issue	A large-scale infrastructure almost always involves a variety of components and configurations. In addition, different teams within an organization may require similar infrastructures, but with slight variations. Although users can use infrastructure providers' command-line interfaces or user interfaces to spin up and configure components one by one, the end result is typically difficult to manage and maintain.
Architectural Decision	Terraform is used to provide infrastructure as code where it manages and provisions infrastructure through code instead of through manual processes. This allows replication of AWS resources to be done across multiple environments and regions rapidly and easily.
Assumptions	None
Alternatives	CloudFormation
Justification	Both CloudFormation and Terraform cover most features of AWS, but CloudFormation typically takes more time to implement support for new AWS features as compared to Terraform. In addition, Terraform also supports other cloud providers as well as third-party services. This allows COMO Club to have the flexibility to opt for multi-cloud solutions which will help to mitigate vendor lock-in in AWS services.

Architectural Decision: Elastic Container Service (ECS)	
ID	AD5
Issue	Compatibility issues across different environments (local, production) and OS. Horizontal scaling to meet scalability requirements.
Architectural Decision	Amazon ECS is a fully managed container orchestration service that will be used to easily orchestrate and scale the containers running in the cluster. Operational complexity of having to manually manage each microservice will be reduced through the use of Amazon ECS.
Assumptions	None
Alternatives	EKS
Justification	The maturity of ECS brings about smooth integration with Amazon SQS and AWS Secrets Manager to carry out business functionality more efficiently which will reduce latency between requests.

Architectural Decision: Amazon SQS

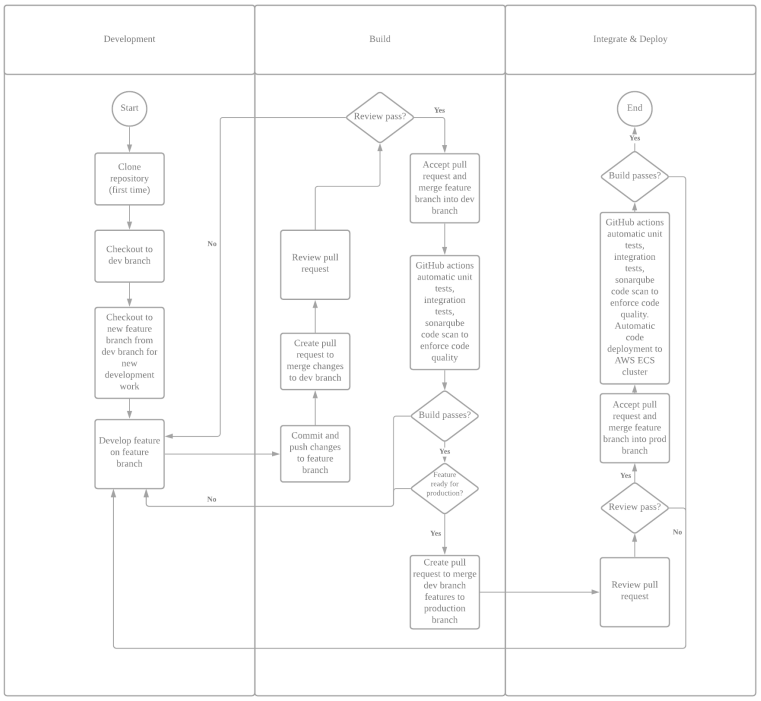
ID	AD6
Issue	The availability of external APIs services we used in our application is not within our control. When an external API goes down, it can cause our own services to fail as well.
Architectural Decision	In the scenario where the external APIs go down, each middleware container will push the incoming requests from the user to an AWS SQS Queue that will store requests in order when it detects that the external APIs are not available. Through this implementation, we aim to provide diminished service to the user instead of having no service at all when business critical external APIs become unavailable. In the scenario where our middleware containers perform aggregated API calls to the external APIs and a subset of these requests fail, the container instance will push the failed request details to the AWS SQS Queue to be re-tried again later. This provides diminished availability of functionality instead of unavailability when aggregated API requests in the middleware fail.
Assumptions	None
Alternatives	AWS MQ
Justification	Our use case only requires a simple queue functionality. We will not be taking advantage of the various routing types and more advanced functionality that AWS MQ provides. It is more cost efficient to use AWS SQS compared to AWS MQ.

Architectural Decision: VPC Endpoints	
ID	AD7
Issue	Without VPC Endpoints, servers running in private subnets will not be able to access AWS services such as AWS DynamoDB and AWS SecretsManager as they do not have internet access. If a NAT Gateway is provisioned, traffic will traverse the internet to connect to these AWS services which opens up attack surfaces which could compromise our system.
Architectural Decision	With VPC Endpoints, we will be creating a secure connection that is not exposed to the internet between our API gateway and our servers running in private subnets within the VPC.
Assumptions	None
Alternatives	NAT Gateway
Justification	Servers running in private subnets within a VPC do not have internet access. In order for these servers to reach AWS services such as AWS DynamoDB, AWS SecretsManager etc, a NAT Gateway will need to be provisioned to enable outgoing internet access. However, the negative to this method is that traffic will traverse the public internet which opens up opportunities for malicious attackers to intercept these outgoing requests. Since each AWS service will contain sensitive data such as Authentication tokens and credentials in the case of SecretsManager, traversing the internet to retrieve these values will be dangerous. It would be more desirable for servers in the private subnet to connect to AWS services via a VPC Endpoint which guarantees that traffic will not traverse the public internet.

Development View

Development Strategy

The team uses 2 environments, production (main branch) and development (dev branch). The team also adopts GitHub for code management and versioning. New features are developed by branching out from the existing dev branch. Once the feature is ready, a pull request will be made to merge the feature branch into the dev branch. Once merged, automated unit tests, integration tests and code quality scanning will be done using GitHub actions *(Please refer to Appendix 3 Figure 3.1 and 3.2 for GitHub actions pipeline)*. If the build passes and the dev environment is finalised and stable for production, a pull request will be created to merge the dev environment into prod. Once the request is approved, automated unit testing, integration testing and code quality enforcement will happen before the code is pushed to the AWS ECS cluster. *(Please refer to Appendix 3 Figure 3.3 for sonarqube code scan output.)*



Deployment Process

Our team adopts GitHub actions for continuous integration and continuous deployment (CI/CD). On code push to the production environment (main branch), GitHub actions will be triggered to run a pipeline of steps to do unit tests, integration tests, sonarqube code quality checks and finally deployment when all previous steps pass. These steps are predefined in the .github/workflows/prod.yaml file.

In the pipeline's deployment step, we utilise maven wrapper to generate a .jar file of our application. This .jar file will then be used to create a Docker image based on the Dockerfile specifications. Next, authentication will be done to AWS via CLI which allows the GitHub actions runner to connect to our AWS account. The Docker image built will then be pushed into AWS Elastic Container Registry (ECR) with the "latest" version tag.

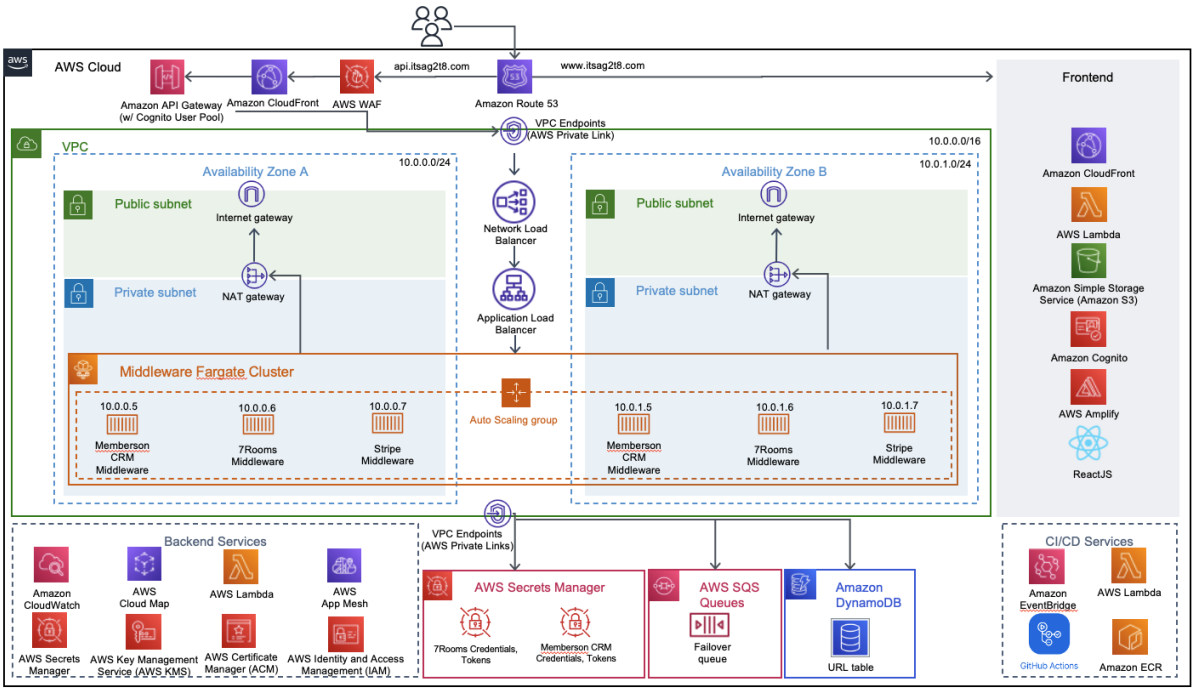
Once the image is pushed to ECR, the runner will retrieve the latest Task Definition from AWS and update it to target the newest image pushed to ECR. This will then trigger ECS to spin up new containers with the latest ECR image. We previously configured the ECS service to have Minimum healthy percent of 100% and Maximum percent of 200%, which configures new ECS containers with the newest image to spin up and reach a ready state before it stops containers with the old image. This ensures that there is no downtime for each of our containers as there will always be containers ready to accept incoming requests during the deployment process.

Solution View

Network Diagram

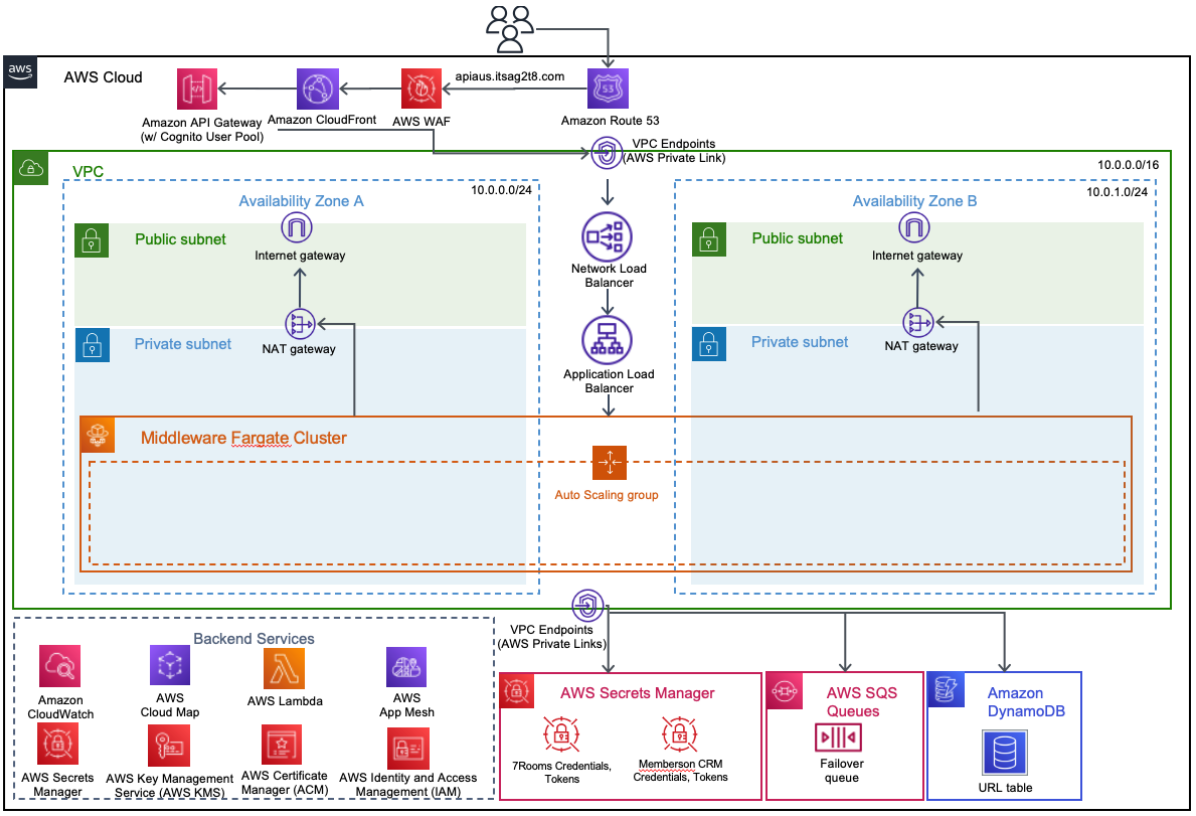
Production Environment

Region: ap-southeast-1, Singapore (active configuration)



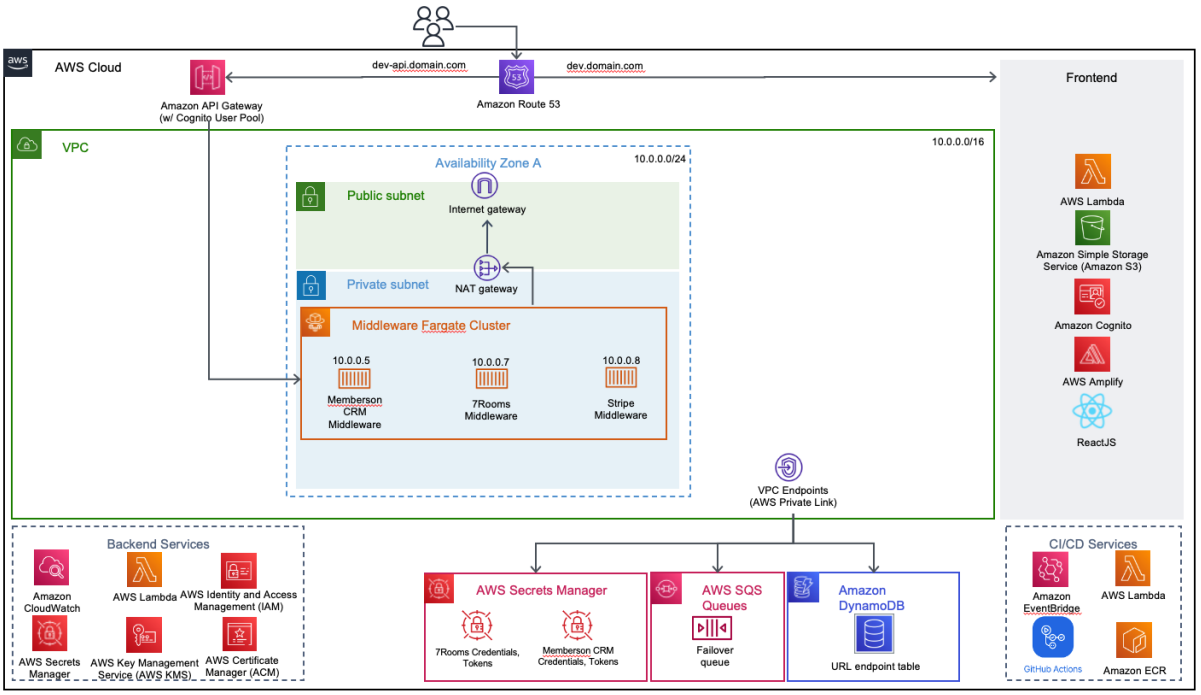
Disaster Recovery Environment

Region: ap-southeast-2, Sydney (passive configuration)



Development Environment

Region: ap-southeast-1, Singapore (active configuration)



Architecture design

The team adopts a microservice style architecture design which focuses on decoupling each middleware container based on its functionality and interactions with the 3rd party API services such as Memberson CRM. By ensuring that each middleware container only has 1 specific functionality, we are able to decouple them from each other as each container runs independently from the other. This improves maintainability as future development work on one microservice will not affect the others.

The team also utilises an API gateway to manage the middleware endpoints. In the future, when there are changes in client facing API endpoints, the API gateway can be easily altered to change the client facing endpoint URL while still being able to route requests to the middleware containers running in the private subnets. This decouples the client facing endpoints from the container exposed endpoints which offers greater flexibility and control over client exposed endpoints to COMO Club.

When new 3rd party API services are onboarded, new middleware containers can be easily deployed to ECS using our terraform and CI/CD configuration templates. The API Gateway can also be easily configured to route traffic to the new middleware container via simple point and click methods on the AWS console. This speeds up the onboarding process as the time taken for a new middleware to go from source code to a functioning production endpoint as there are less pains of having to manually configure AWS infrastructure or deploy containers.

Authentication and access control is also implemented at the API Gateway using AWS Cognito. This ensures that middleware containers can be developed independently on development environments or localhost environments easily without having the added complexity of having to manage security and access control on the source code level.

To further enhance maintainability, AWS Cloudwatch will be used to monitor logs. Each of our middleware servers are configured to use Log4j2 with customised logging. We utilise multiple logging levels to ensure traceability for our developed functionalities. As a base guideline, logging is done when:

1. Application starts
2. Application ends
3. Decision making points
4. Trigger points
5. Action points (E.g. retrieving data from DynamoDB)
6. Receiving data
7. Sending data

To further improve the clarity of logs generated, we utilise a custom formatted logging pattern consisting of:

1. Name of application
2. Logging level
3. Timestamp in ISO-8601 format
4. Class and function name
5. Stack trace when logging exceptions

This allows developers to easily track down bugs which will improve maintainability. In addition, our architecture can be easily scaled up to integrate the AWS managed Elasticsearch, Logstash, and Kibana (ELK) stack to analyse logs, create visualizations for application and infrastructure monitoring, faster troubleshooting and security analytics which will further improve the maintainability of the source code.

Design patterns

(Refer to Appendix 4 for code implementation)

Singleton (Creational Pattern)

We implemented Singleton to ensure that those classes that require access to the same class (for e.g. DynamoDB, RestTemplate, SecretsManager, WebConfig, WebClientConfig) will be connecting to the same instance of that class.

Single Responsibility Principle (SRP - Solid Design Principle)

By implementing SRP, we ensure that each class only has one responsibility and is only handling a single concern. This makes our application easier to maintain since if an error occurs, it will be easier to find. For example, in our Memberson Middleware, we separated our functionality into different classes (Customer, Points and Rewards). Same for our Sevenroom Middleware (Reservation, Venue). Each class in our middleware will only do one job and have only one reason to change.

Open-Closed Principle (OCP - Solid Design Principle)

By implementing OCP, we ensure that our classes are open for extension but closed for modification. Rather than using concrete classes, we used interfaces or abstract classes. Any new functionality we need can be added by creating new classes that implement the interfaces.

Integration Endpoints

Source System	Destination System	Protocol	Format	Communication Mode
Client Device	Amazon Route53	DNS	DNS Message	Synchronous
Amazon CloudFront	Amazon S3	HTTPS	HTML	Synchronous
Amazon CloudFront	API Gateway	HTTPS	JSON	Synchronous
API Gateway	Network Load Balancer (via VPC Endpoints)	HTTP	JSON	Synchronous
Network Load Balancer	Application Load Balancer	HTTP	JSON	Synchronous
Application Load Balancer	Elastic Container Service (Fargate)	HTTP	JSON	Synchronous
Elastic Container Service (Fargate)	Amazon DynamoDB (via VPC Endpoints)	HTTPS	JSON	Synchronous
Elastic Container Service (Fargate)	Amazon Secrets Manager (via VPC Endpoints)	HTTPS	JSON	Synchronous
Elastic Container Service (Fargate)	Amazon SQS (via VPC Endpoints)	HTTPS	JSON	Asynchronous
Amazon EventsBridge SQS Lambda Trigger	AWS Lambda	HTTPS	JSON	Asynchronous
AWS Lambda	Amazon DynamoDB (via VPC Endpoints)	HTTPS	JSON	Synchronous
AWS Lambda	Amazon Secrets Manager (via VPC Endpoints)	HTTPS	JSON	Synchronous
AWS Lambda	Amazon SQS (via VPC Endpoints)	HTTPS	JSON	Asynchronous
AWS Lambda	Network Load Balancer (via VPC Endpoints)	HTTP	JSON	Synchronous
AWS Lambda	Amazon Cognito	HTTPS	JSON	Synchronous

Development Budget

Production Environment

Full cost breakdown

Region	Service	Monthly Cost (USD)	First 12 Months Total Cost (USD)	Description
Asia Pacific (Singapore)	Amazon API Gateway	12.93	155.16	HTTP API requests units (exact number), REST API request units (thousands), Cache memory size (GB) (None), WebSocket message units (thousands), Average message size (32 KB), Requests (per hour), Requests (100 per day)
Asia Pacific (Singapore)	DynamoDB on-demand capacity	0.03	0.36	Average item size (all attributes) (1 KB), Data storage size (0.1 GB)
Asia Pacific (Singapore)	DynamoDB Streams	0	0	Number of DynamoDB Streams GetRecord API requests (10 per month)
Asia Pacific (Singapore)	Amazon Virtual Private Cloud (VPC)	86.26	1035.12	Number of NAT Gateways (2)

Asia Pacific (Singapore)	Amazon Virtual Private Cloud (VPC)	75.93	911.16	Number of interface VPC endpoints per region (4)
Asia Pacific (Singapore)	Amazon Simple Queue Service (SQS)	0.50	6	DT Inbound: Internet (0 GB per month), DT Outbound: Not selected (0 TB per month), Standard queue requests (million per month), FIFO queue requests (1 million per month)
Asia Pacific (Singapore)	Amazon CloudWatch	11.49	137.86	Number of Custom/Cross-account events (10), Standard Logs: Data Ingested (1 GB), Logs Delivered to CloudWatch Logs: Data Ingested (1 GB), Logs Delivered to S3: Data Ingested (1 GB), Number of Lambda functions (4), Number of requests per function (60 per hour)
Asia Pacific (Singapore)	AWS Secrets Manager	6.16	73.92	Number of secrets (4), Average duration of each secret (30 days), Number of API calls (30000 per day)
Asia Pacific (Singapore)	Amazon Cognito	15	180	Advanced security features (Enabled), Number of monthly active users (MAU) (300)
Asia Pacific (Singapore)	Amazon Elastic Container Registry	5	60	DT Inbound: Internet (1 TB per month), DT Outbound: Not selected (0 TB per month), Amount of data stored (50 GB per month)
Asia Pacific (Singapore)	AWS Web Application Firewall (WAF)	9.60	115.20	Number of Web Access Control Lists (Web ACLs) utilized (1 per month), Number of Rules added per Web ACL (4 per month), Number of Rule Groups per Web ACL (per month), Number of Managed Rule Groups (per month), Number of Rules inside each Rule Group (per month)
Asia Pacific (Singapore)	AWS Fargate	89.97	1079.64	Average duration (1 days), Amount of ephemeral storage allocated for Amazon ECS (20 GB), Number of tasks or pods (8 per day), Operating system (Linux)
Asia Pacific (Singapore)	Amazon Route 53	4.90	58.80	Hosted Zones (1), Basic Checks Within AWS (4), Number of domains stored (2)
Asia Pacific (Singapore)	Application Load Balancer	41.76	501.12	Number of Application Load Balancers (1)
Asia Pacific (Singapore)	Network Load Balancer	35.92	431.04	Number of Network Load Balancers (1), Processed bytes per NLB for TCP (4 GB per hour), Average number of new TCP connections (100 per second), Average TCP connection duration (5 seconds)

Asia Pacific (Singapore)	Amazon CloudFront	1.80	21.60	Data transfer out to internet (4 GB per month), Data transfer out to origin (2 GB per month), Number of requests (HTTPS) (1 million per month)
Asia Pacific (Singapore)	AWS Key Management Service	7	84	Number of customer managed Customer Master Keys (CMK) (1), Number of symmetric requests (2000000)
Asia Pacific (Sydney)	Amazon Elastic Container Registry	5	60	DT Inbound: Not selected (0 TB per month), DT Outbound: Not selected (0 TB per month), Amount of data stored (50 GB per month)
Asia Pacific (Sydney)	AWS Secrets Manager	1.60	19.20	Number of secrets (4), Average duration of each secret (30 days), Number of API calls (million per month)
Asia Pacific (Sydney)	Amazon Simple Queue Service (SQS)	0.50	6	Data transfer cost (0), Standard queue requests (million per month), FIFO queue requests (1 million per month)
Asia Pacific (Sydney)	Amazon API Gateway	0	0	HTTP API requests units (exact number), Average size of each request (34 KB), REST API request units (millions), Cache memory size (GB) (None), WebSocket message units (thousands), Average message size (32 KB), Requests (0 per month)
Asia Pacific (Sydney)	Application Load Balancer	21.32	255.84	Number of Application Load Balancers (1)
Asia Pacific (Sydney)	Network Load Balancer	20.59	247.08	Number of Network Load Balancers (1), Processed bytes per NLB for TCP (0.5 GB per hour), Average number of new TCP connections (per second), Average TCP connection duration (seconds)
Asia Pacific (Sydney)	AWS Fargate	1.43	17.16	Operating system (Linux), Average duration (1 days), Amount of ephemeral storage allocated for Amazon ECS (20 GB), Number of tasks or pods (4 per month)
Asia Pacific (Sydney)	AWS Lambda	0	0	Architecture (x86), Architecture (x86), Number of requests (3 per day)
Asia Pacific (Sydney)	Amazon Virtual Private Cloud (VPC)	43.13	517.56	Number of NAT Gateways (1)
Asia Pacific (Sydney)	Amazon CloudFront	0.09	1.08	Data transfer out to internet (0.5 GB per month), Data transfer out to origin (0.3 GB per month), Number of requests (HTTPS) (10000 per month)
Asia Pacific (Sydney)	Amazon	11.41	136.89	Number of Custom/Cross-account

	CloudWatch			events (10), Standard Logs: Data Ingested (1 GB), Logs Delivered to CloudWatch Logs: Data Ingested (1 GB), Logs Delivered to S3: Data Ingested (1 GB), Number of Lambda functions (4), Number of requests per function (60 per hour)
Asia Pacific (Sydney)	AWS Key Management Service	7	84	Number of customer managed Customer Master Keys (CMK) (1), Number of symmetric requests (2000000)
Total		**516.32**	**6,195.79**	**-**

Development Environment

Full cost breakdown

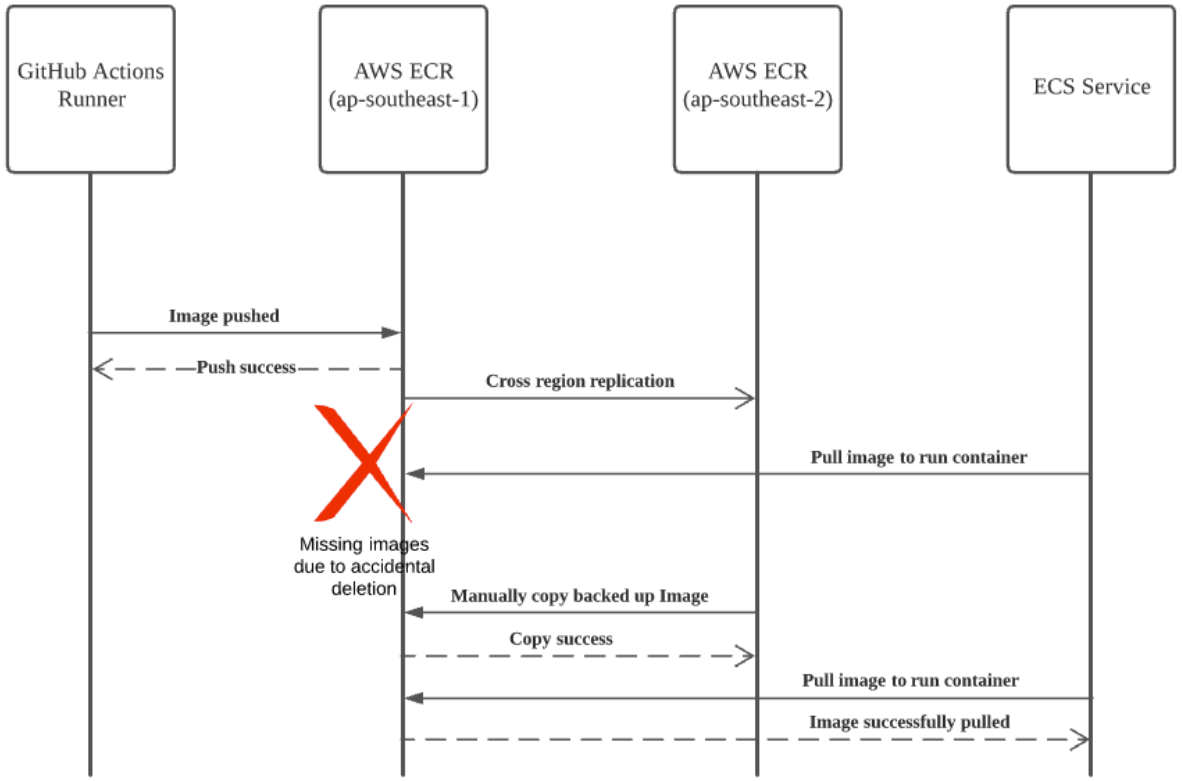
Region	Service	Monthly Cost (USD)	First 12 Months Total Cost (USD)	Description
Asia Pacific (Singapore)	Amazon Route 53	1.40	16.80	Hosted Zones (2), Basic Checks Within AWS (1)
Asia Pacific (Singapore)	Amazon API Gateway	0	0	HTTP API requests units (exact number), REST API request units (exact number), Cache memory size (GB) (None), WebSocket message units (thousands), Average message size (32 KB), Requests (per month), Requests (1000 per month)
Asia Pacific (Singapore)	DynamoDB on-demand capacity	0.03	0.36	Average item size (all attributes) (1 KB), Data storage size (0.1 GB)
Asia Pacific (Singapore)	Amazon Virtual Private Cloud (VPC)	43.36	520.32	Number of NAT Gateways (1)
Asia Pacific (Singapore)	Amazon Virtual Private Cloud (VPC)	9.54	114.48	Number of interface VPC endpoints per region (1)
Asia Pacific (Singapore)	Amazon Simple Queue Service (SQS)	0.50	6	Data transfer cost (0), Standard queue requests (million per month), FIFO queue requests (1 million per month)
Asia Pacific (Singapore)	Amazon CloudWatch	4.40	52.80	Number of Metrics (includes detailed and custom metrics) (10), Standard Logs: Data Ingested (1 GB), Vended Logs: Data Ingested (1 GB), Number of Custom/Cross-account events (3)
Asia Pacific (Singapore)	AWS Secrets Manager	4.82	57.84	Number of secrets (12), Average duration of each secret (30 days), Number of API calls (100 per day)
Asia Pacific	Amazon Cognito	0.05	0.60	Advanced security features (Enabled),

(Singapore)				Number of monthly active users (MAU) (1)
Asia Pacific (Singapore)	Amazon Elastic Container Registry	5	60	DT Inbound: Internet (50 GB per month), DT Outbound: Not selected (0 TB per month), Data transfer cost (0), Amount of data stored (50 GB per month)
Asia Pacific (Singapore)	AWS Fargate	44.98	539.76	Average duration (1 days), Number of tasks or pods (4 per day), Amount of ephemeral storage allocated for Amazon ECS (20 GB)
Asia Pacific (Singapore)	AWS Lambda	0	0	Number of requests (50000)
Total		**114.08**	**1,368.96**	**-**

Availability View

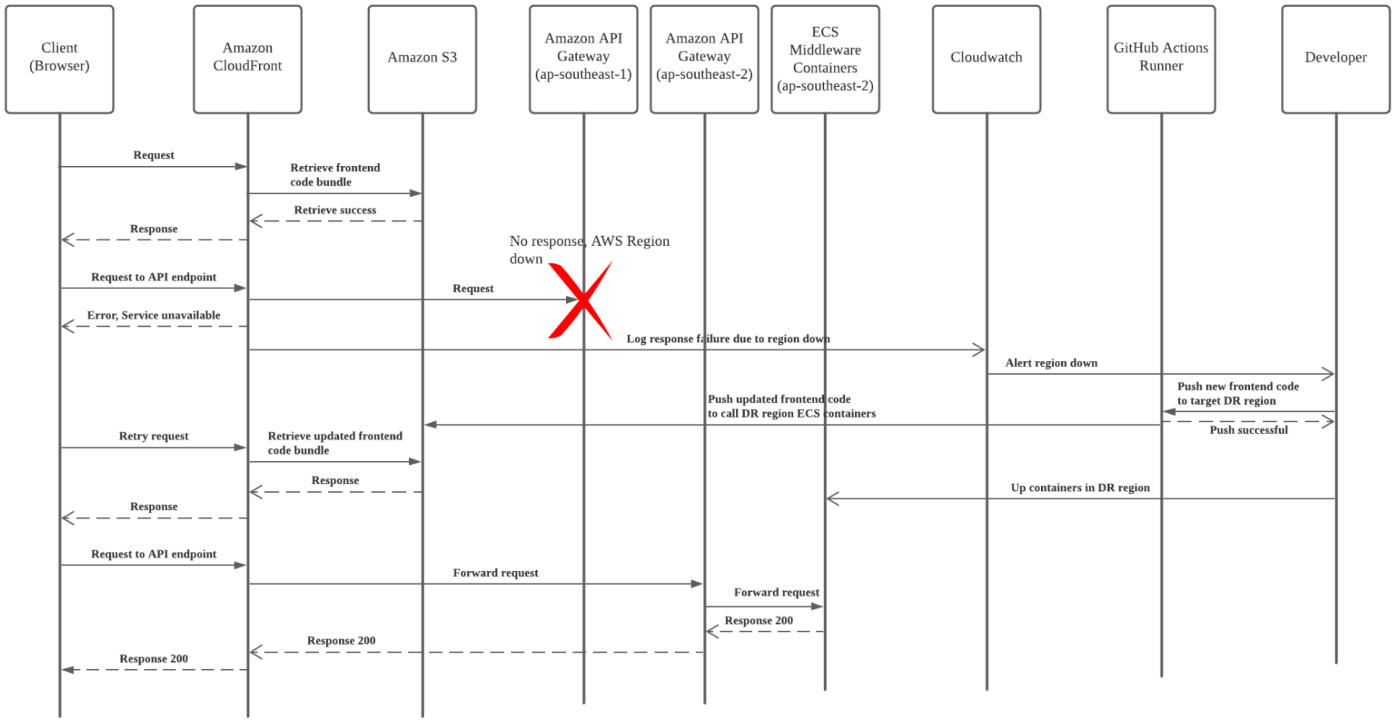
Node	**Redundancy**	**Clustering**			**Replication (if applicable)**			
		Node Config	**Failure Detection**	**Failover**	**Repl. Type**	**Session State Storage**	**DB Repl. Config.**	**Repl. Mode**
Application Load Balancer	Horizontal scaling (fixed 1 instance per AZ)	Active-active	Ping (Health check)	Network Load Balancer	N.A.			
Elastic Container Service (Fargate)	Horizontal auto scaling (based on ECSServiceAverageMemoryUtilization, ECSServiceAverageCPUUtilization, ALBRequestCountPerTarget(	Active-active	Ping (Health check)	Application Load Balancer	N.A.			
AWS DynamoDB	Horizontal Auto Scaling	Active-active	Heartbeat	DNS	DB	Database	Master-Slave	Asynchronous
AWS DynamoDB	Horizontal Scaling (Cross region replication via Global Tables	N.A.			DB	Database	Master-Slave	Asynchronous
Amazon ECR	Horizontal Scaling (Cross region replication)	N.A.			DB	Database	Master-Slave	Asynchronous
Amazon Secrets Manager	Horizontal Scaling (Cross region replication)	N.A.			DB	Database	Master-Slave	Asynchronous

Cross Region Replication



Disaster Recovery

Pilot light Active-Passive configuration.



Autoscaling

ECS Service

We implemented autoscaling for ECS services to ensure high availability and scalability of the application. Autoscaling handles the traffic load by scaling in and out appropriately, this ensures that the backend services are also fault tolerant.

The application load balancer can help detect if the Fargate tasks are healthy by checking the health status of each endpoint. If the health check fails, the task is considered unhealthy: it will be deregistered from the target group,

terminated, and a new task will be launched to replace it. We deployed the ECS service across multiple availability zones, if an availability zone is down, there would be instances in the other availability zone that would keep the application running which ensures high availability. Depending on the traffic tasks will be scaled up/down automatically. Auto-scaling ensures that we use only what we need, resources and capacity are provisioned to handle the current demand, saving on cost in the long term.

DynamoDB

We used DynamoDB On Demand Capacity to automatically adjust the throughput capacity to actual traffic patterns to ensure high availability of our backend databases. When there is a sudden surge in traffic, On Demand Capacity allows a table or a global secondary index to increase its provisioned read and write capacity. When the traffic decreases, DynamoDB scales down and reduces throughput so that we do not have to pay for unused allocated capacity. On Demand Capacity offers simple pay-per-request pricing for read and write requests so that we only pay for what we use, making it easy to balance cost and performance.

For an item up to 4 KB in size, one read capacity unit corresponds to one strongly consistent read per second, or two eventually reads per second. If we are expecting a 100 times load during peak periods, we must scale up our read and write capacity by 100 times.

Security View

No	Asset/ Asset Group	Potential Threat/ Vulnerability pair	Possible Mitigation Controls
1	All backend services	As many different AWS services need to communicate with each other, traffic traversing public networks may risk the confidentiality of certain data or services.	All internal traffic is routed through subnets within a Virtual Private Cloud, and ingress connections are only allowed through the API gateway (after necessary authentication). A NAT gateway is deployed to allow services to egress traffic for communication with 3rd party APIs.
2	Microservice containers	The number of microservices deployed presents a large attack surface for external parties to exploit and compromise confidentiality.	An API gateway is used to encapsulate the microservices and prevent discovery of resources deployed within the private subnet.
3	Microservices	Unauthorised users calling APIs will result in a breach of confidentiality of data.	AWS Cognito is implemented at the API gateway level to ensure only users with valid accounts and credentials can access the microservices.
4	Microservices	Attackers may send requests with malicious payloads intended to harm the integrity of data.	AWS Web Application Firewall is configured in front of the API gateway with rules to block common attack patterns under the OWASP Top 10 security risks, including cross site scripting and known bad inputs.
5	Microservice containers	Attackers may perform denial of service attacks by flooding the APIs with an unusually large number of requests, affecting the availability of microservices.	AWS Web Application Firewall also filters out excessive traffic coming from a single location, or bots identified through their user agent and an AWS-managed IP reputation list.
6	User data	Confidentiality of user data can be compromised by man-in-the-middle attacks outside of the AWS network to steal user data.	Connections to AWS Cloudfront are forced to HTTPS only, so that all traffic between the VPC and public internet is encrypted with SSL.

7	External APIs	Credentials and authentication tokens used for calling external APIs may be inevitably leaked if hardcoded or stored in configuration files, impacting confidentiality.	AWS Secrets Manager is used to secure sensitive secrets and ensure that services can only access them programmatically when needed and they are not stored or revealed elsewhere.
8	Internal services	If one of the services' code is compromised, it may be used to improperly access other AWS resources or perform unintended actions which can affect both confidentiality of data and integrity of the infrastructure.	Identity and Access Management is configured for individual services and functions using the principle of least privilege, so that they are only able to read or modify other AWS resources that they are explicitly allowed to.
9	External APIs	Endpoints for external APIs stored in DynamoDB or secrets stored in Secrets Manager may be exposed if physical security of data residing in AWS is compromised.	As a last line of defence, encryption at rest is enabled for DynamoDB and Secrets Manager using the AWS Key Management Service to prevent data stored from being read by third parties.
10	Microservices	In the event that buggy/broken code is pushed by a developer, the availability of that specific microservice may be affected.	Unit tests, code scanning and build checks are implemented in the CI/CD pipeline to ensure that only properly working code is deployed to production and does not cause downtime.

Performance View

No	Description of the Strategy	Justification
1	Load Balancer with Auto Scaling (implemented)	Traffic will be assigned evenly to instances by load balancers based on their availability and the Auto Scaling monitors the application and adjusts the capacity of the system thus improving the performance of the system.
2	CloudFront caching (implemented)	With CloudFront caching, more objects are served from CloudFront edge locations, which are closer to COMO Club's users. This reduces the load and reduces latency.

Using the use-cases as a workflow (Login → View Experience → Book Experience).
With Auto-scaling and caching implemented. We achieved a timing of 2125ms with 300 concurrent users.

Performance Testing

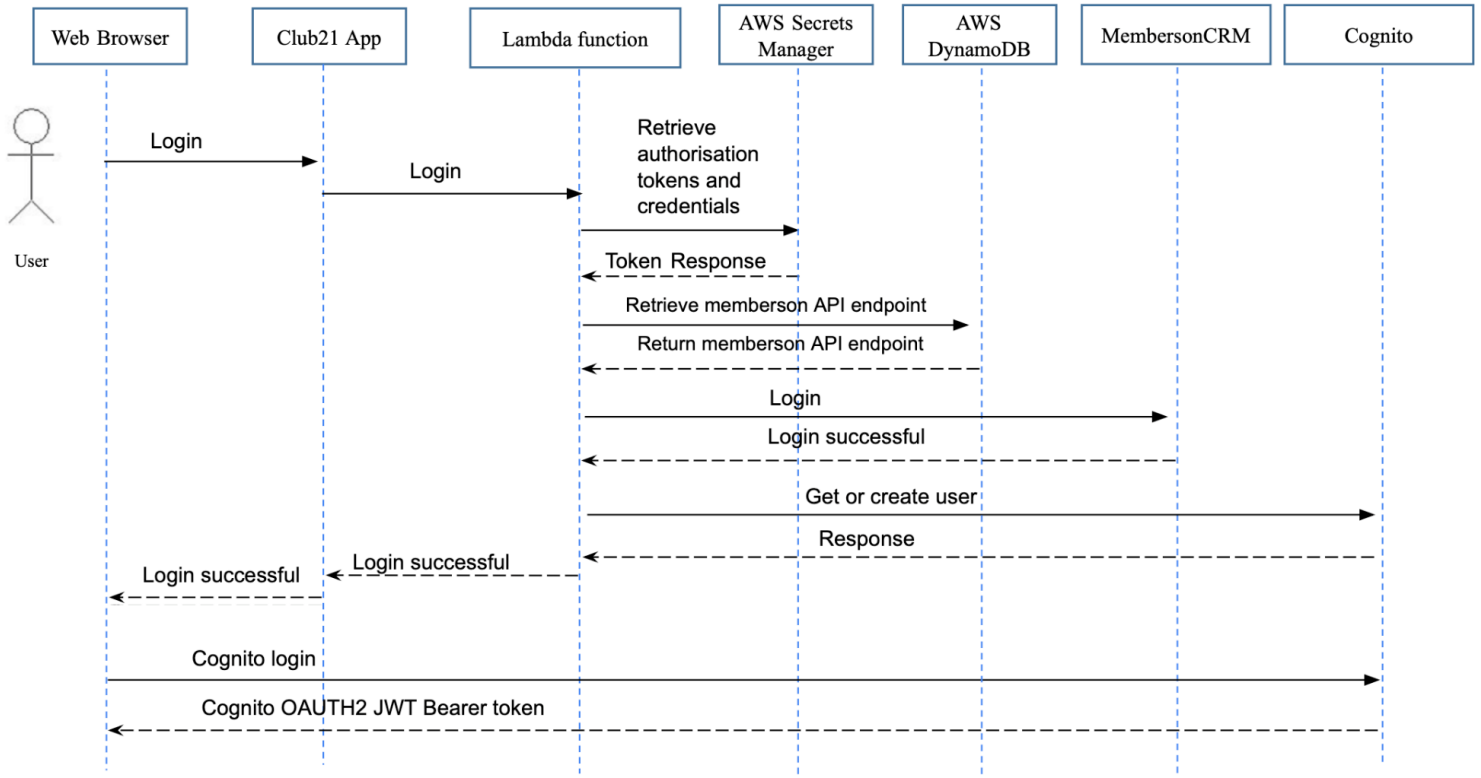
Test	Base	CloudFront Cache
70 Users | 1 Ramp up	2015ms	1011ms
300 Users | 1 Ramp up	7425ms	3055ms
300 Users | 30 Ramp up	5258ms	324ms

Please refer to Appendix 5 for Jmeter Performance testing results.

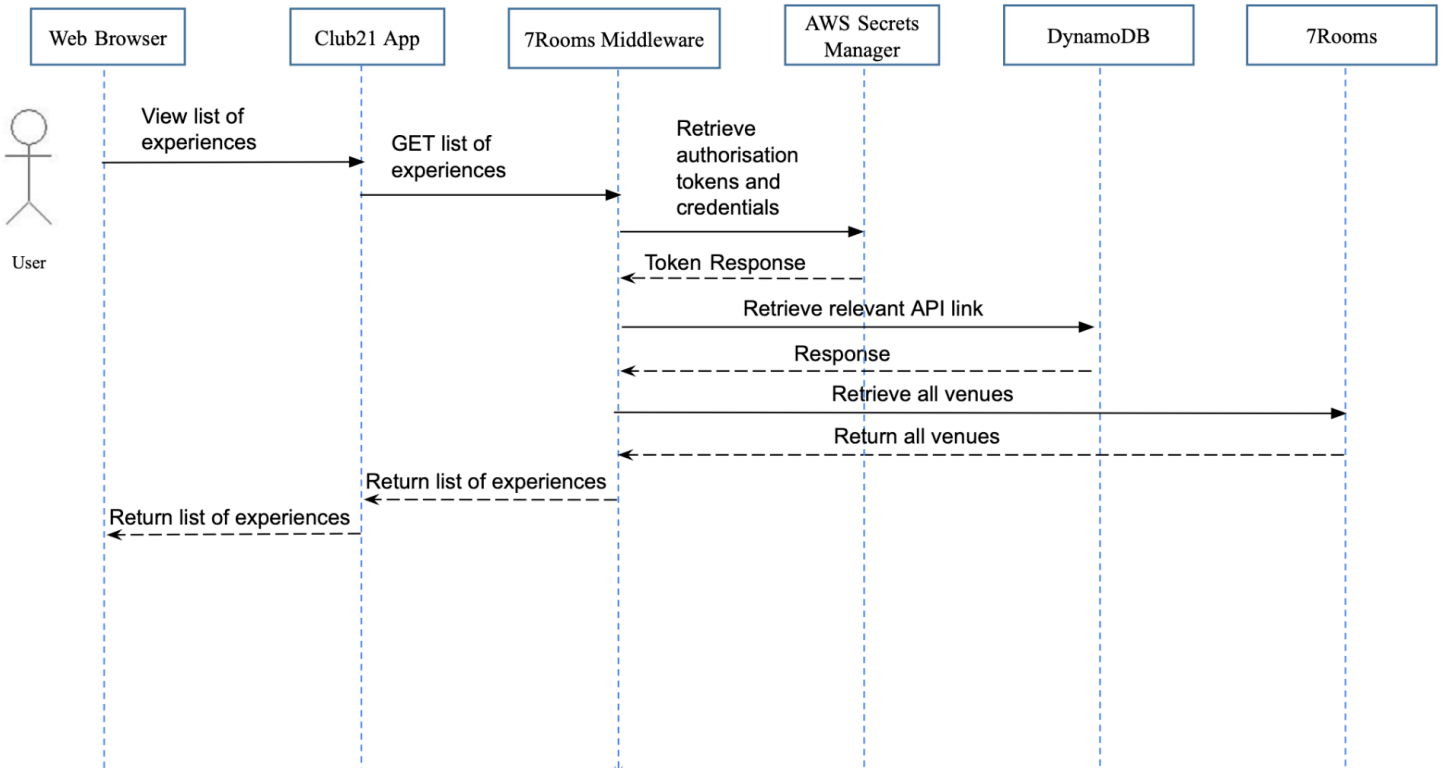
Appendix

Appendix 1: Sequence Diagrams

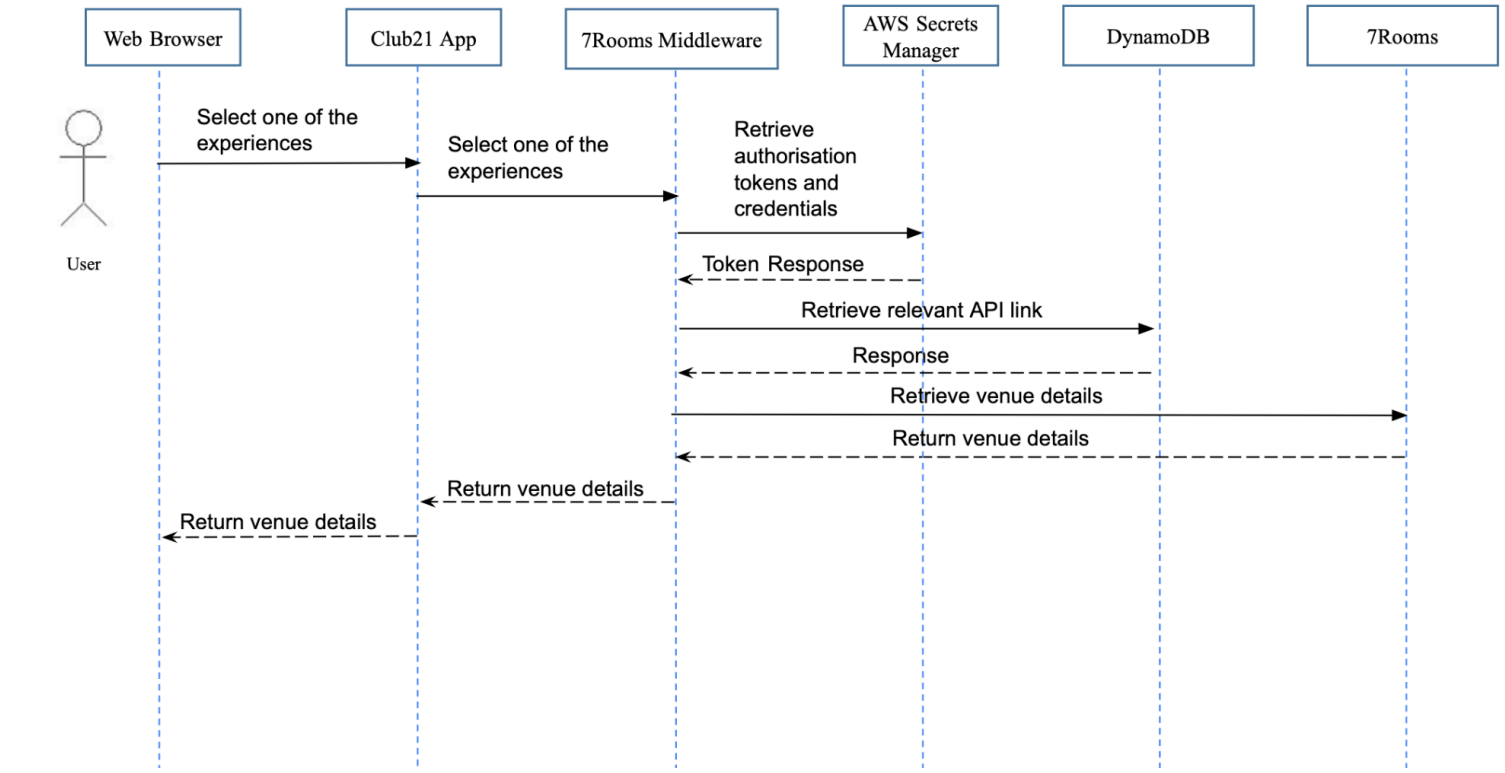
Use Case 1: Login

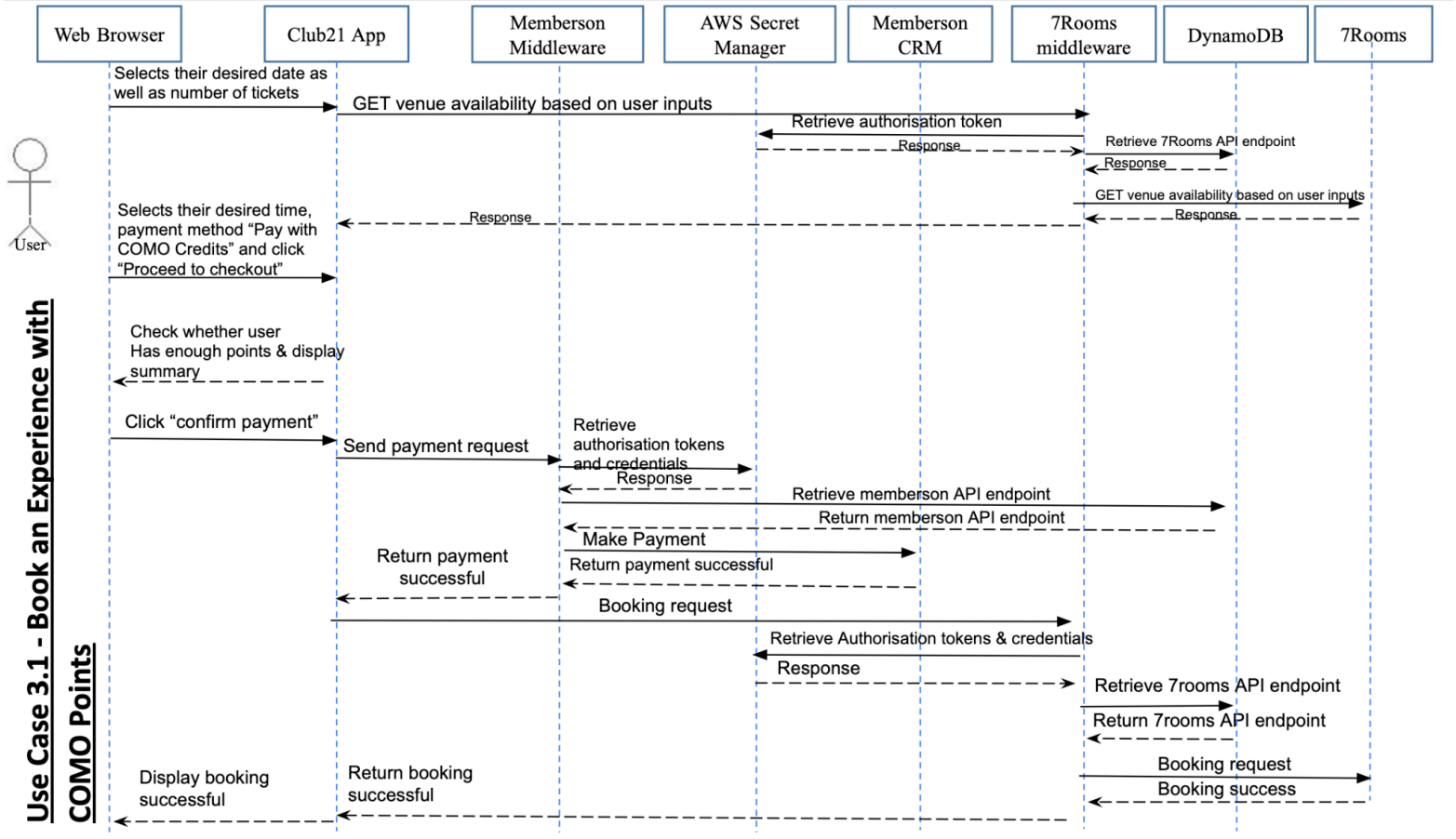


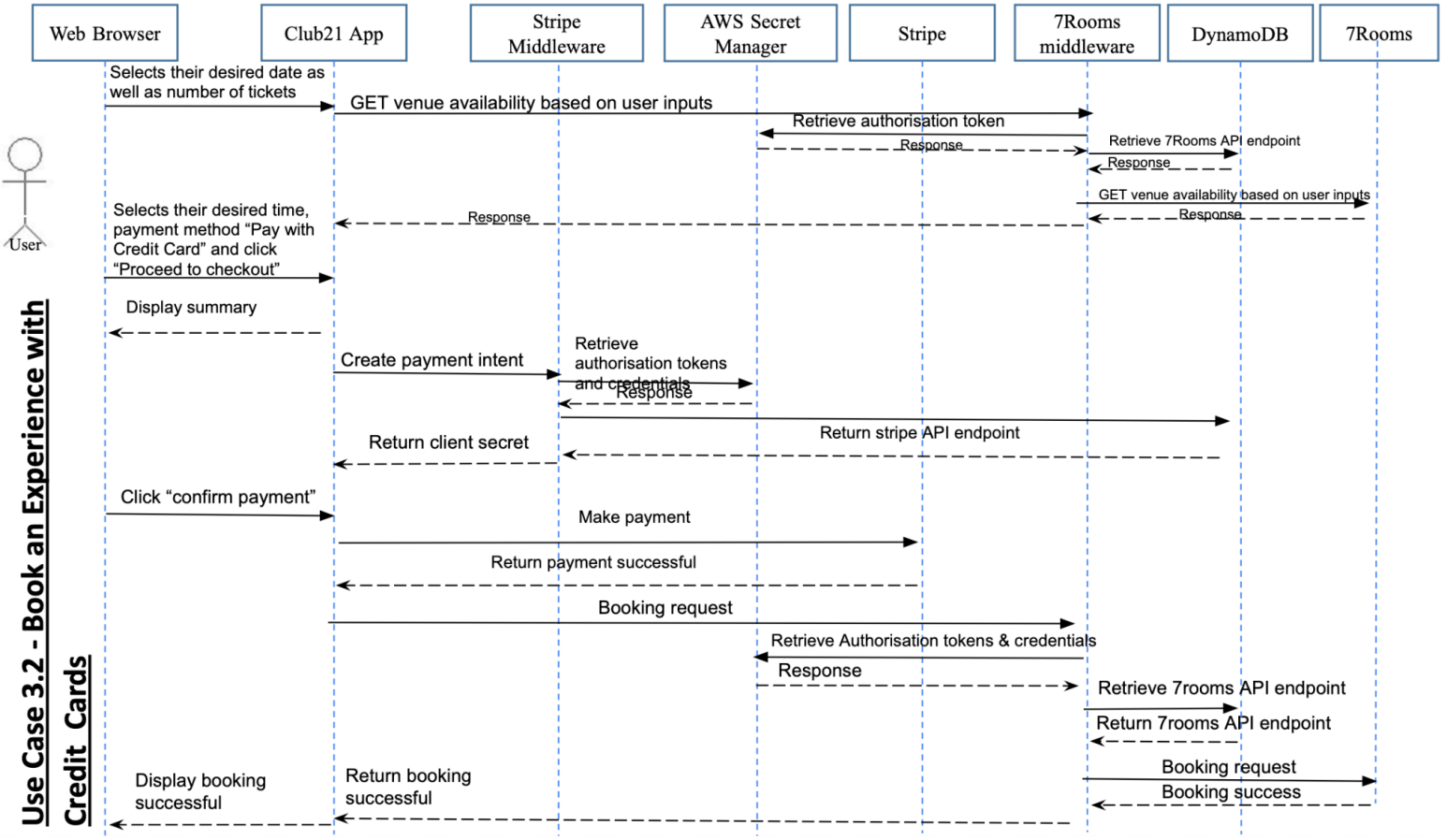
Use Case 2.1: View list of experiences



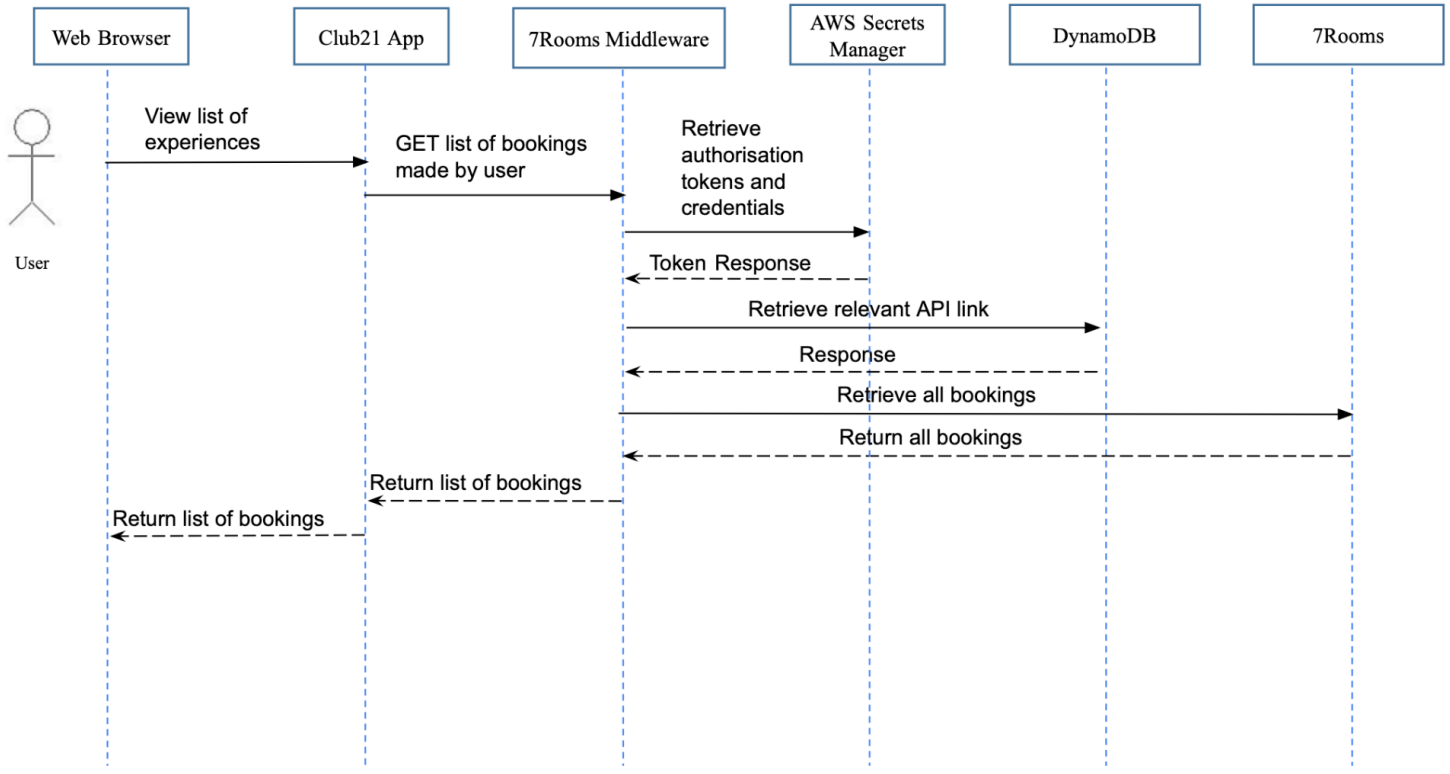
Use Case 2.2: View details of an experience







Use Case 4: View list of user's bookings



Appendix 2: Frontend UI

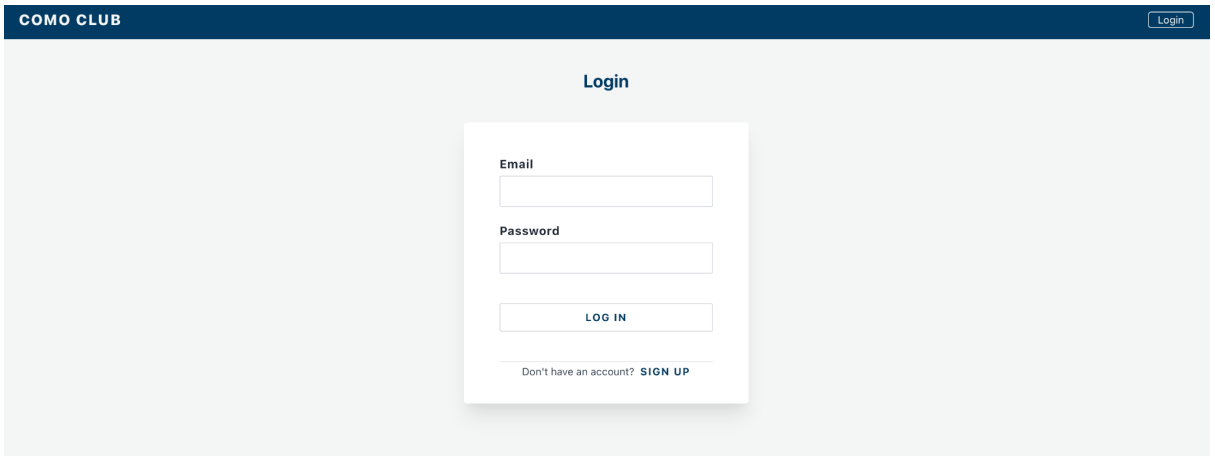


Figure 1.1: Login Page

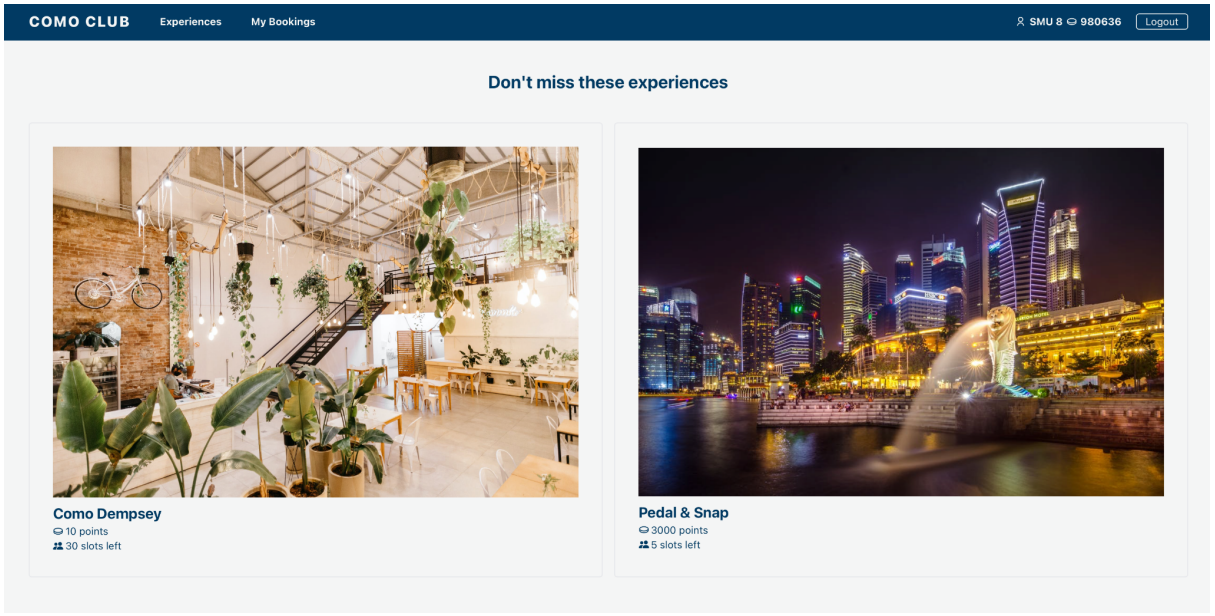


Figure 1.2: Homepage where users can view experiences for booking

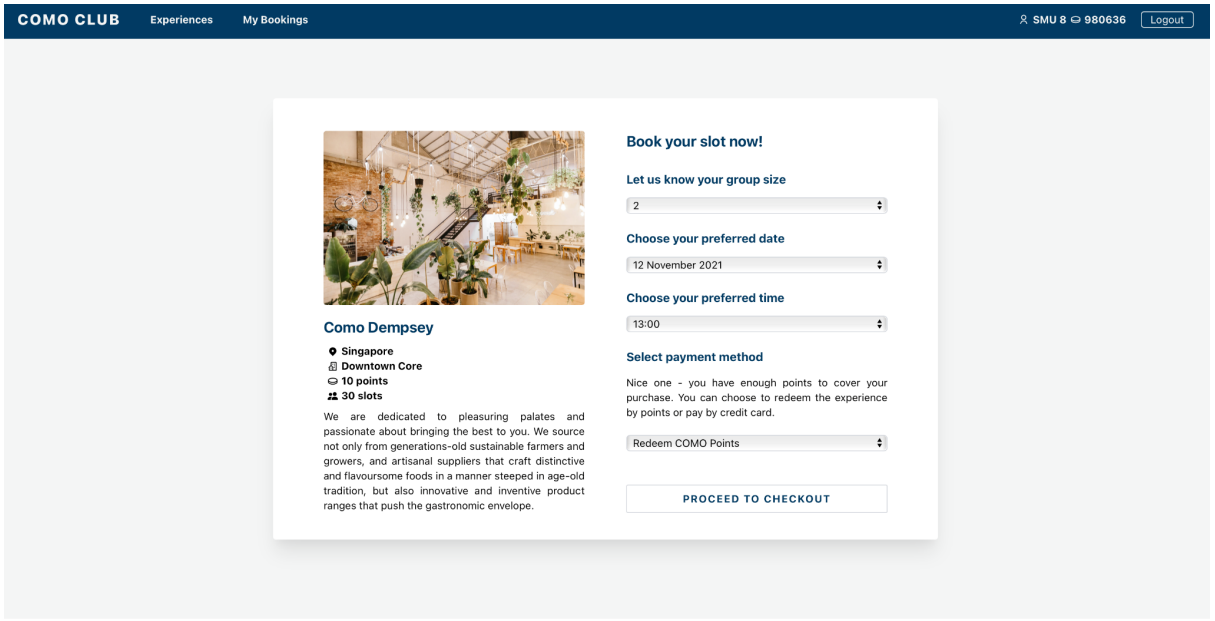


Figure 1.3: Booking page where users can select group size, date, time, payment method

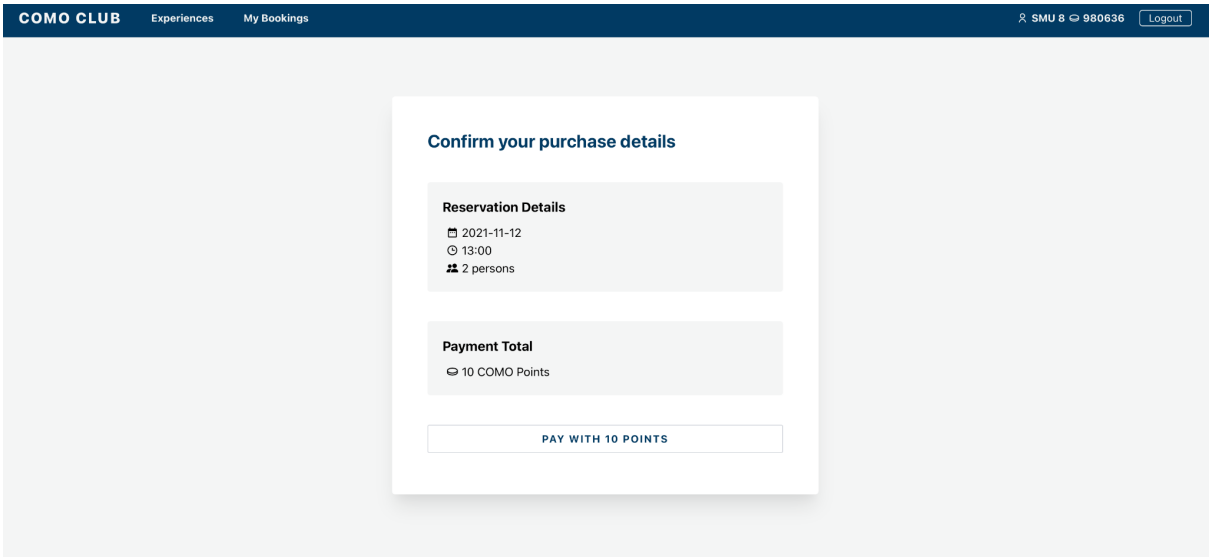


Figure 1.4.1: Checkout page where users pay by points

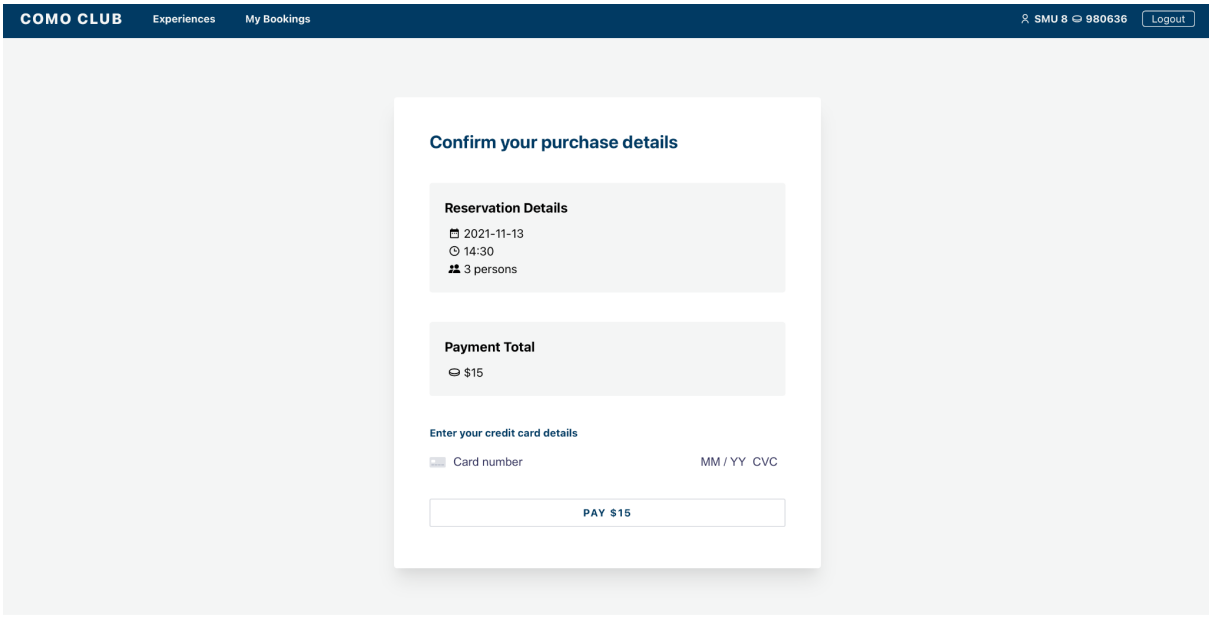


Figure 1.4.2: Checkout page where users pay by credit card

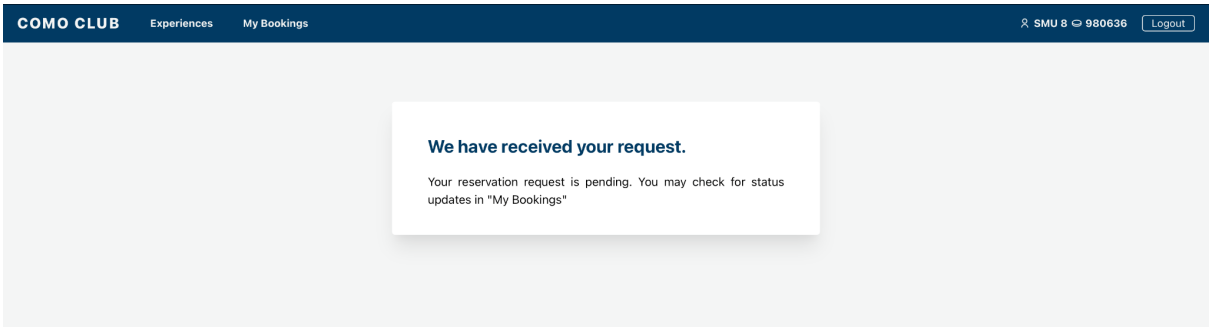


Figure 1.5: Checkout success page

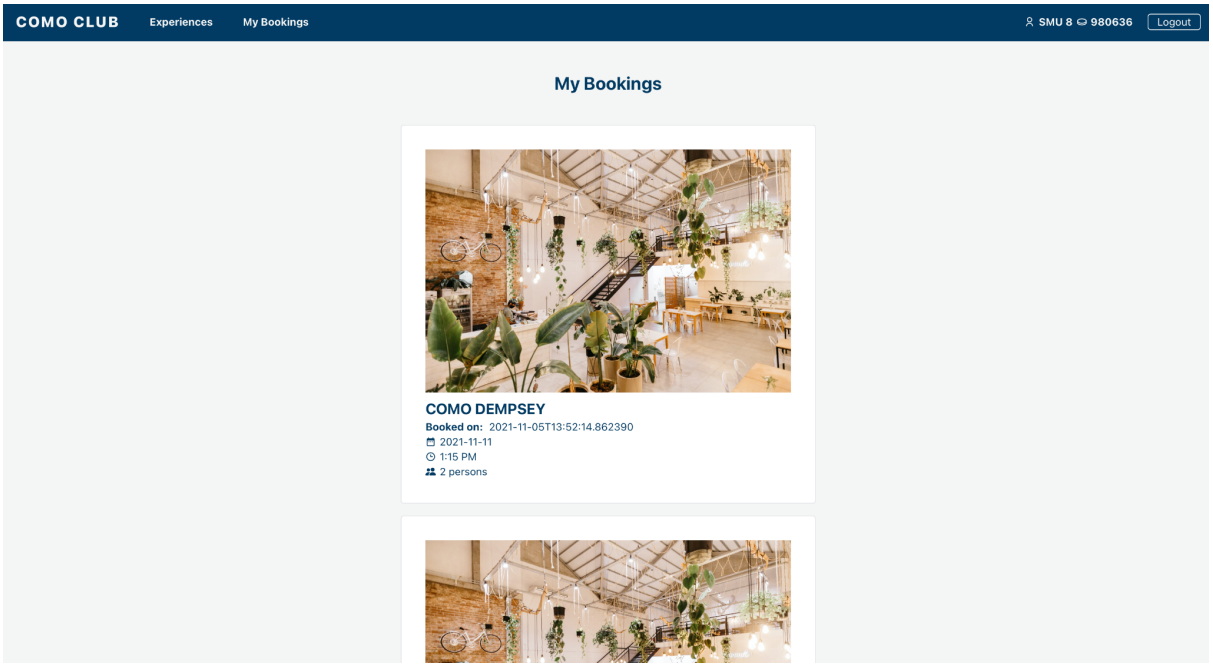


Figure 1.6: My bookings page where users can view confirmed booking details

Appendix 3: CICD

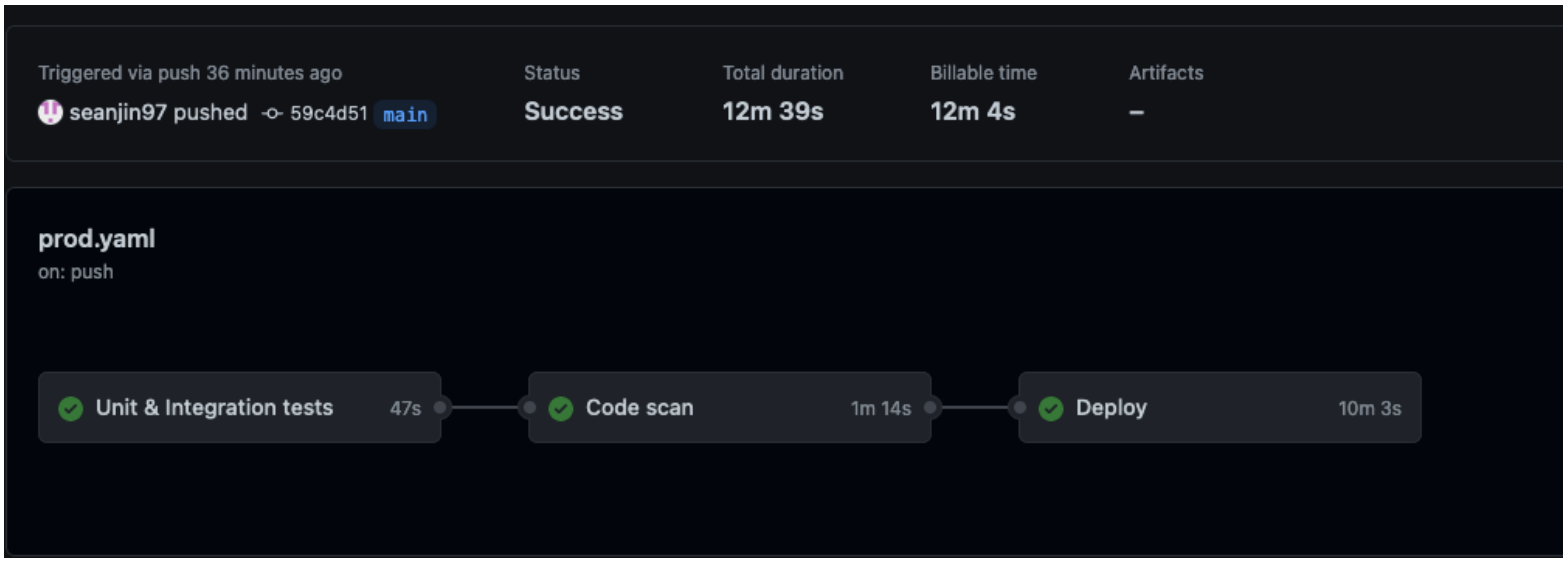


Figure 3.1 GitHub actions CI/CD stages

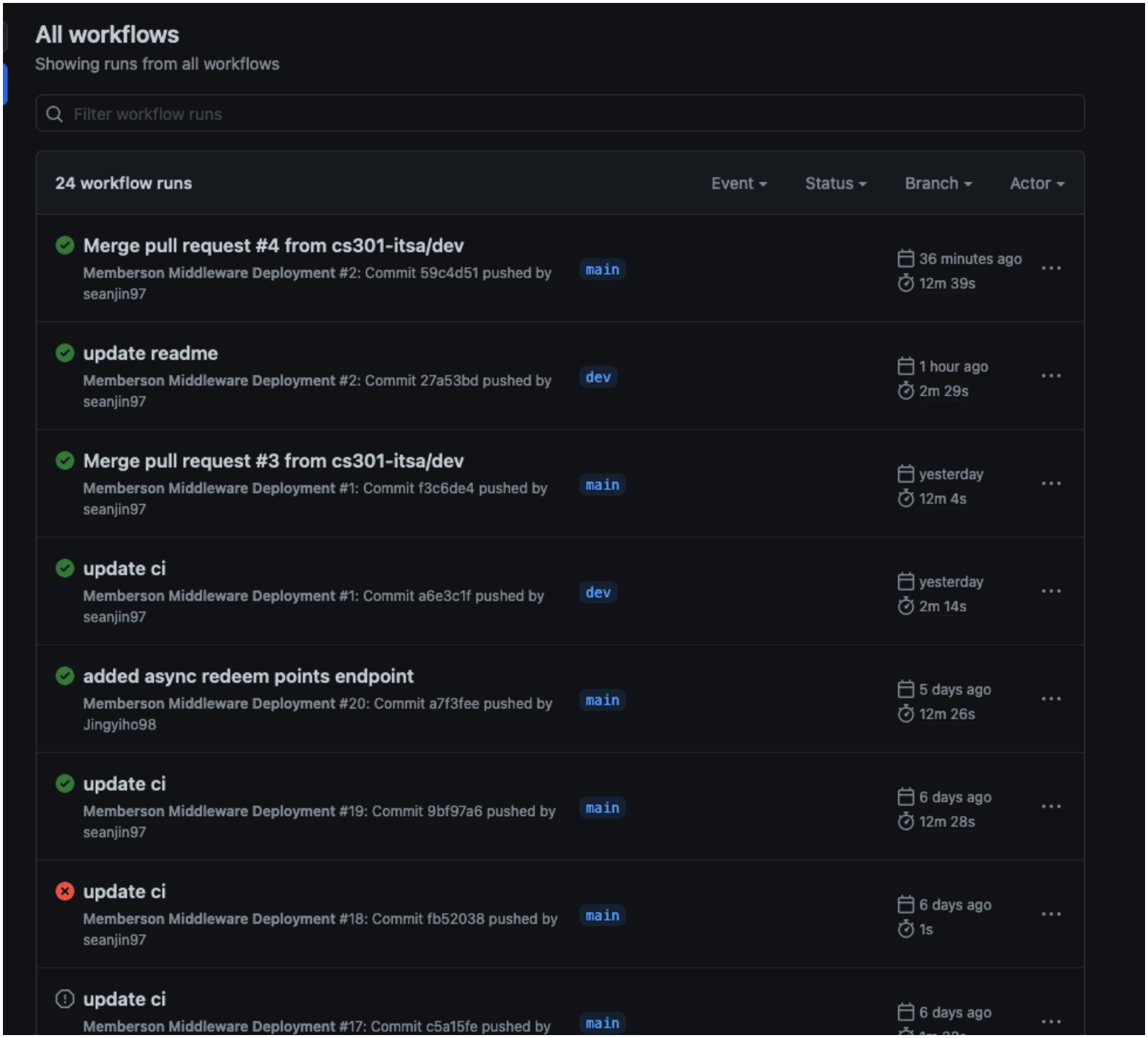


Figure 3.2: GitHub actions run history

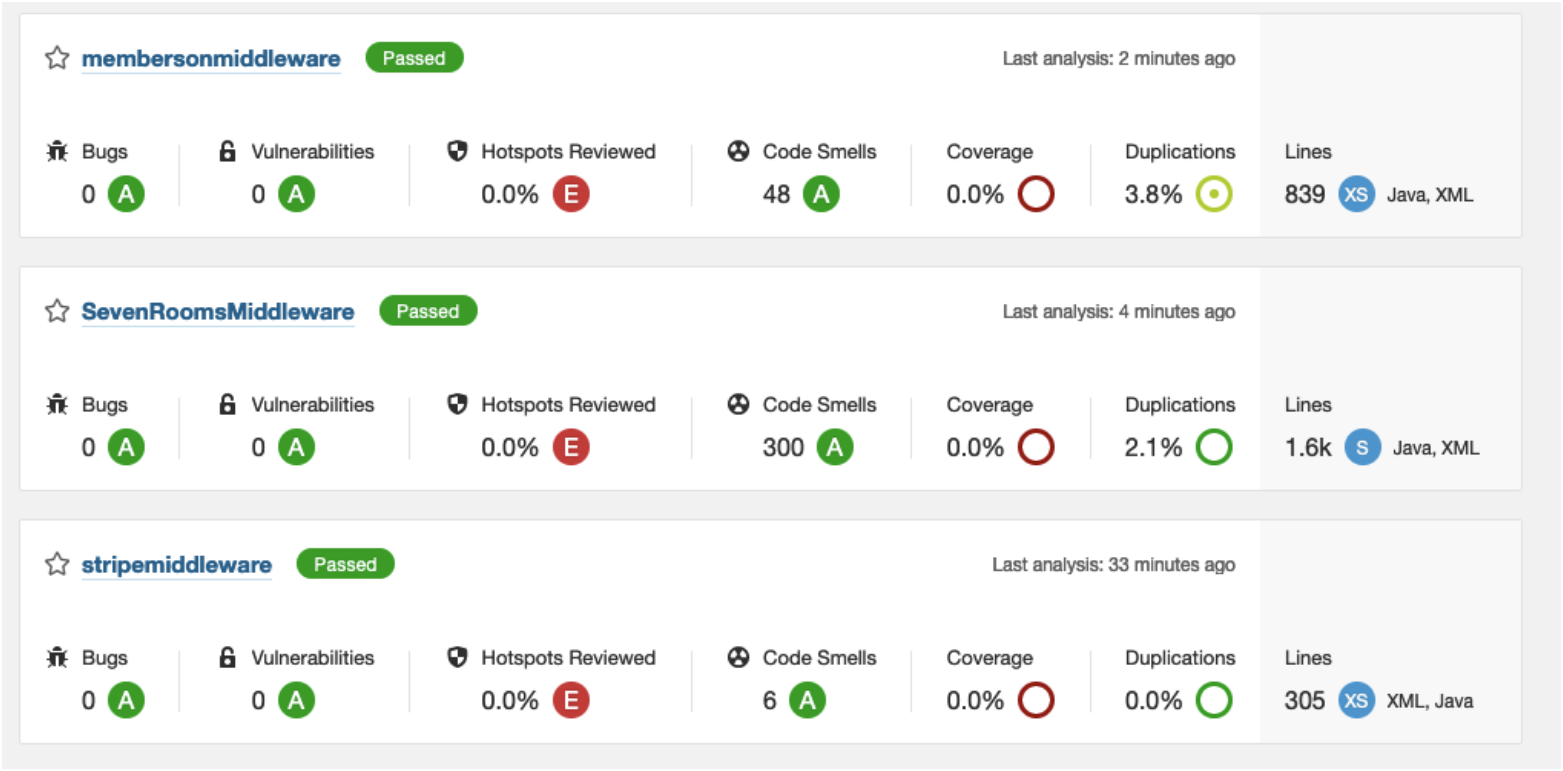


Figure 3.3: Sonarqube code scan output

Appendix 4: Design Patterns

```java
@Configuration
public class DynamoDBConfig{

    @Value("${amazon.aws.accesskey}")
    private String amazonAWSAccessKey;

    @Value("f22tF8EVVdQGBNAri4uh711qL3r5X+sN6fEZ8UBy")
    private String amazonAWSSecretKey;



    @Bean
    public DynamoDBMapper dynamoDBMapper() { return new DynamoDBMapper(buildAmazonDynamoDB()); }

    private AmazonDynamoDB buildAmazonDynamoDB() {
        return AmazonDynamoDBClientBuilder
                .standard()
                .withCredentials(
                        new AWSStaticCredentialsProvider(
                                new BasicAWSCredentials(
                                        amazonAWSAccessKey,
                                        amazonAWSSecretKey
                                )
                        )
                )
                .build();
    }

}
```

Figure 4.1: Singleton Implementation (DynamoDBConfig)

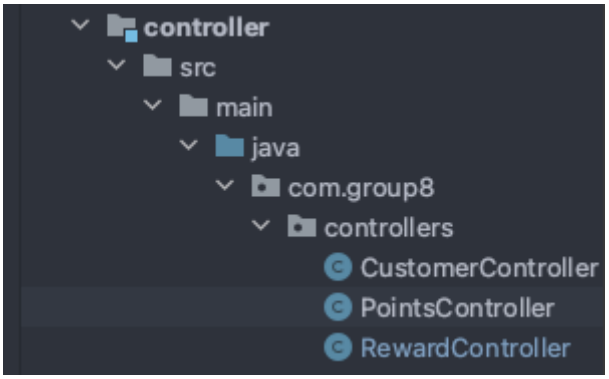


Figure 4.2: Single Responsibility Principle Implementation (Memberson Middleware)

```
public interface PointsAPI {

    ResponseEntity<?> redeemPoints(redeemPoints redeemPoints) throws URISyntaxException;

    String API_BASE_PATH = "/points";
    String API_REDEEM_POINTS = API_BASE_PATH + "/redeem";
    String API_PATH_WITH_ID = API_BASE_PATH + "/{customerNo}";
}
```

```
@RestController
public class PointsController implements PointsAPI {
```

Figure 4.3: Open-Closed Principle Implementation

Appendix 5: Performance View

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received ...	Sent KB/...	Avg. Bytes
getCustomerProfile	70	5687	2259	8725	1852.62	0.00%	7.7/sec	19.52	0.00	2596.0
searchReservationRequest	70	2306	1311	4325	742.13	0.00%	5.8/sec	213.73	7.92	37650.0
createReservationRequest	70	1743	963	3866	641.17	0.00%	6.0/sec	4.20	8.88	719.0
getVenueAvailability	70	1168	435	2496	571.22	0.00%	7.6/sec	88.37	10.63	11930.0
getAllVenues	70	1050	425	2341	514.46	0.00%	8.4/sec	19.99	10.59	2426.0
redeemPoints	70	138	46	961	176.17	0.00%	8.1/sec	5.20	0.00	659.0
TOTAL	420	2015	46	8725	1992.68	0.00%	23.8/sec	217.26	21.88	9330.0

Figure 5.1: Baseline Performance, 70 users 1ramp up

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received K...	Sent ...	Avg. ...
getCustomerProfile	70	328	30	731	211.74	0.00%	48.5/sec	120.00	0.00	2531.3
getAllVenues	70	276	93	545	107.02	0.00%	78.6/sec	181.83	98.59	2370.0
getVenueAvailability	70	184	17	319	99.81	0.00%	88.9/sec	1031.38	124.65	11874.0
redeemPoints	70	2692	479	4666	1172.02	0.00%	14.5/sec	9.35	0.00	660.0
createReservationReq...	70	2538	1076	5158	1071.73	0.00%	7.6/sec	5.34	11.30	719.0
searchReservationReq...	70	51	23	88	15.62	0.00%	8.8/sec	324.24	12.04	37594.0
TOTAL	420	1011	17	5158	1313.83	0.00%	39.4/sec	357.33	36.13	9291.4

Figure 5.2: CloudFront Caching Performance, 70 users 1ramp up

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
searchReservationRequest	300	7139	1140	26988	4821.89	0.00%	4.8/sec	176.50	6.54	37650.0
createReservationRequest	300	7105	956	23662	4596.65	0.00%	5.0/sec	3.50	7.40	719.0
getAllVenues	300	4845	454	9771	2590.99	0.00%	7.9/sec	18.82	9.97	2426.0
getVenueAvailability	300	6333	453	17853	3742.52	0.00%	5.6/sec	64.89	7.81	11930.0
getCustomerProfile	300	445	141	2224	332.31	0.00%	9.8/sec	24.83	0.00	2596.0
redeemPoints	300	132	43	1435	213.29	0.00%	5.6/sec	3.61	0.00	659.0
TOTAL	1800	4333	43	26988	4431.78	0.00%	27.4/sec	249.71	25.15	9330.0

Figure 5.3: Baseline Performance, 300 users 30 ramp up

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received K...	Sent ...	Avg. ...
getCustomerProfile	300	41	8	862	101.02	0.00%	10.0/sec	24.84	0.00	2535.7
getAllVenues	300	51	25	351	33.15	0.00%	10.3/sec	23.78	12.89	2370.0
getVenueAvailability	300	19	9	59	8.90	0.00%	10.3/sec	119.23	14.41	11874.0
redeemPoints	300	283	47	4824	632.53	0.00%	10.3/sec	6.62	0.00	660.0
createReservationReq...	300	1511	962	3403	560.96	0.00%	10.0/sec	7.03	14.88	719.0
searchReservationReq...	300	38	18	199	19.56	0.00%	10.4/sec	381.86	14.18	37594.0
TOTAL	1800	324	8	4824	641.04	0.00%	57.6/sec	522.75	52.85	9292.1

Figure 5.4: CloudFront Caching Performance, 300 users 30 ramp up

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received ...	Sent KB/...	Avg. Bytes
getCustomerProfile	300	18559	2335	29085	8303.22	15.33%	10.0/sec	22.49	0.00	2298.5
searchReservationRequest	300	7243	1196	25593	5179.42	2.00%	5.0/sec	178.78	6.76	36912.0
createReservationRequest	300	7762	1026	24486	5314.69	0.67%	5.0/sec	3.54	7.49	719.7
getVenueAvailability	300	5257	432	20358	4084.61	1.00%	5.3/sec	60.78	7.38	11819.0
getAllVenues	300	4561	419	12377	3301.45	2.67%	7.5/sec	17.54	9.47	2381.0
redeemPoints	300	1170	43	10108	2334.74	0.00%	5.3/sec	3.42	0.00	659.0
TOTAL	1800	7425	43	29085	7451.93	3.61%	25.7/sec	228.79	23.54	9131.5

Figure 5.5: Baseline Performance, 300 users 1 Ramp up

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Recei...	Sent K...	Avg. B...
getCustomerProfile	300	1688	8	3791	1183.65	0.00%	64.0/sec	158.59	0.00	2535.5
getAllVenues	300	4460	34	5757	909.22	0.00%	30.6/sec	70.97	38.46	2371.0
getVenueAvailability	300	1807	12	2983	324.65	0.00%	27.7/sec	321.74	38.88	11875.0
redeemPoints	300	2724	408	6067	1505.83	0.00%	23.4/sec	15.07	0.00	660.0
createReservationRequest	300	6894	1692	15719	3895.04	0.33%	12.3/sec	8.64	18.26	719.3
searchReservationRequest	300	759	82	3844	427.31	0.00%	13.6/sec	499.49	18.55	37595.0
TOTAL	1800	3055	8	15719	2753.01	0.06%	63.8/sec	578.73	58.51	9292.6

Figure 5.6: CloudFront Caching Performance, 300 Users 1 Ramp up

Appendix 6: Springboot Test

```java
@SpringBootTest
@Import(TestConfig.class)
public class ReservationControllerTest {
    @Autowired
    RestTemplate restTemplate;
    @Autowired PropertiesReader propertiesReader;
    @Autowired SecretManagerUtils secretManagerUtils;
    @Autowired DynamoDBMapper dynamoDBMapper;
    @Autowired ReservationController reservationController;

    @Mock DynamoDB mockDynamoDb;

    @Test
    public void test_createReservation(){
        Mockito.when(dynamoDBMapper.load(DynamoDB.class, hashKey: "sevenrooms_create_reservation")).thenReturn(mockDynamoDb);
        Mockito.when(mockDynamoDb.getApi_link()).thenReturn("http://localhost:8080/{venue_id}");
        Mockito.when(secretManagerUtils.getSevenRoomsSecretValue()).thenReturn("sevenroom_secret");
        ResponseEntity<ReservationRequestDTO> reservationRequestDTOResponseEntity = ResponseEntity.ok(new ReservationRequestDTO())
        Mockito.when(restTemplate.exchange(
                any(URI.class), any(HttpMethod.class), any(HttpEntity.class), eq(ReservationRequestDTO.class)
        )).thenReturn(reservationRequestDTOResponseEntity);


        ResponseEntity<?> actual = reservationController.createReservation(
                venue_id: "abc",
                new ReservationRequest(
                        date: "11112021",
                        time: "17:00",
                        party_size: 5,
                        first_name: "John",
                        last_name: "Doe",
                        phone: "81112222"
                )
        );
        ResponseEntity<?> expected = null;
        assertEquals(HttpStatus.OK, actual.getStatusCode());
        assertEquals( expected: "ReservationRequestDTO", actual.getBody().getClass().getSimpleName());
    }
```

Figure 6.1 SpringBootTest using Mockito

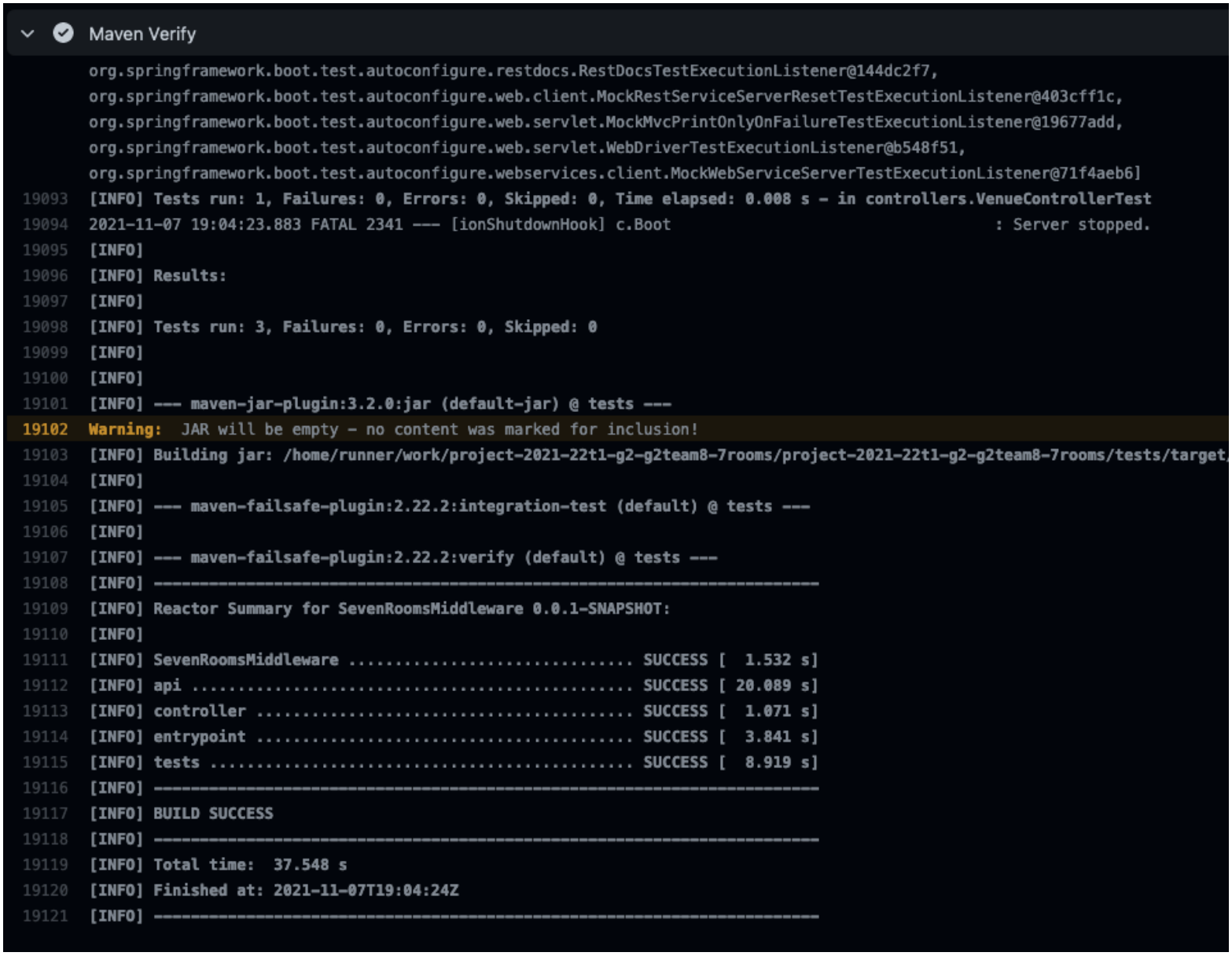


Figure 6.2 SpringBootTest integration tests pass on GitHub actions pipeline

Appendix 7: AWS

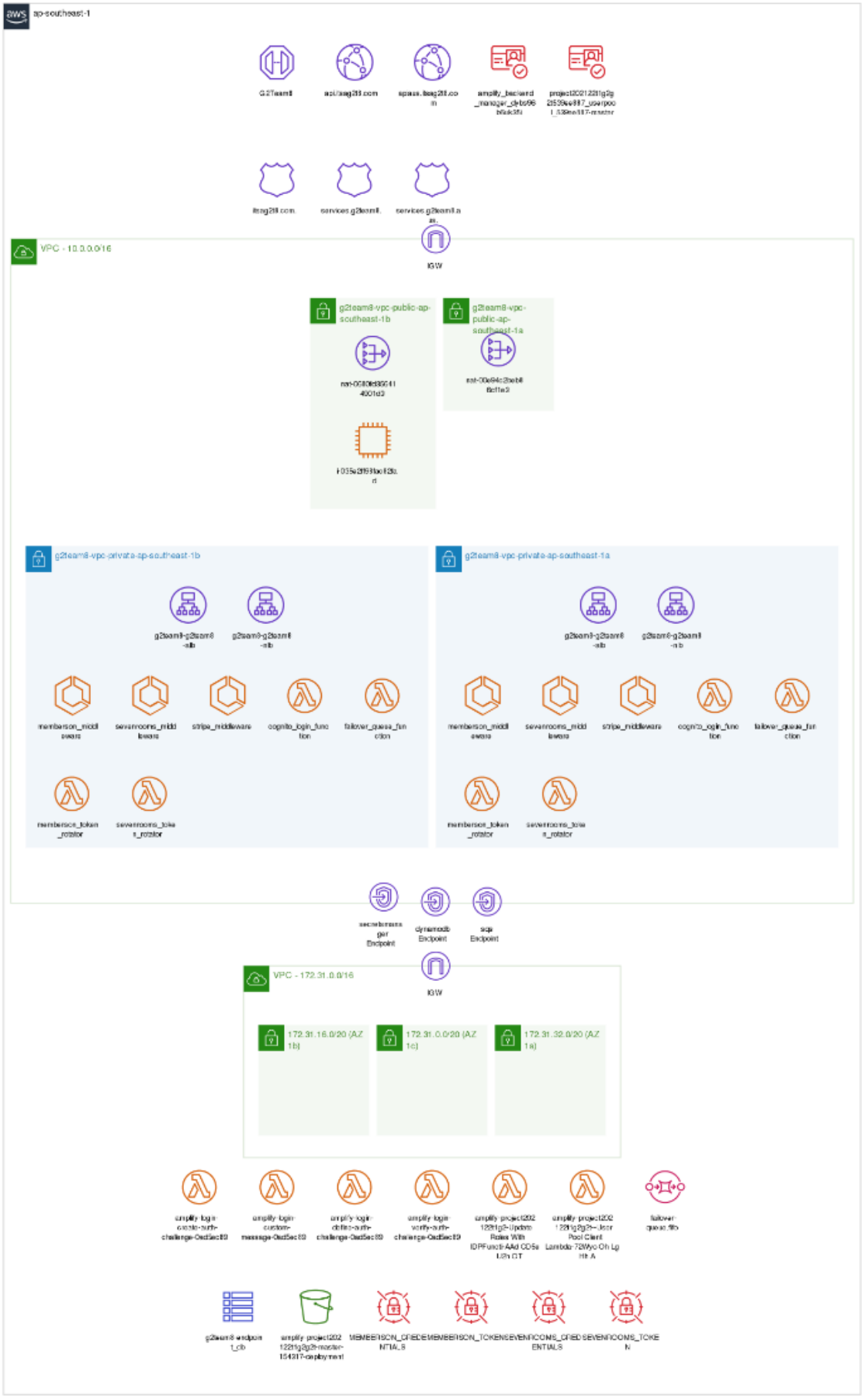


Figure 7.1: InfViz Generated Architecture Diagram

https://drive.google.com/file/d/1E-VOYWJgq0cN_RuwAs_BXhmg6P4wmYr5/view?usp=sharing *(Please use SMU account credentials to access the report)*

Appendix 7.2: InfViz Generated AWS report for main region (ap-southeast-1)

https://drive.google.com/file/d/1Ogpmcq-bfAnZakU03hXkhjhHC6mUE33K/view?usp=sharing *(Please use SMU account credentials to access the report)*

Appendix 7.3 InfViz Generated AWS report for DR region (ap-southeast-2)

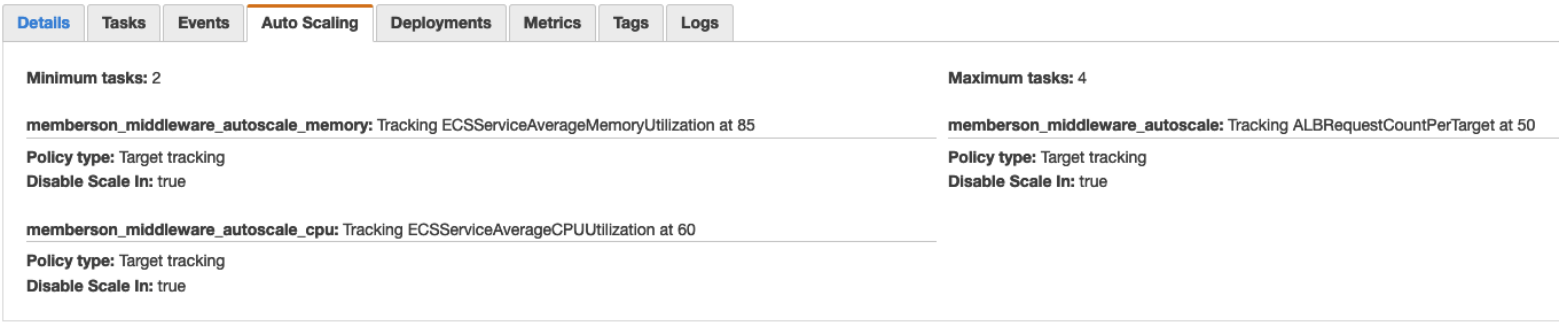


Figure 7.4 ECS Service Auto Scaling Configurations

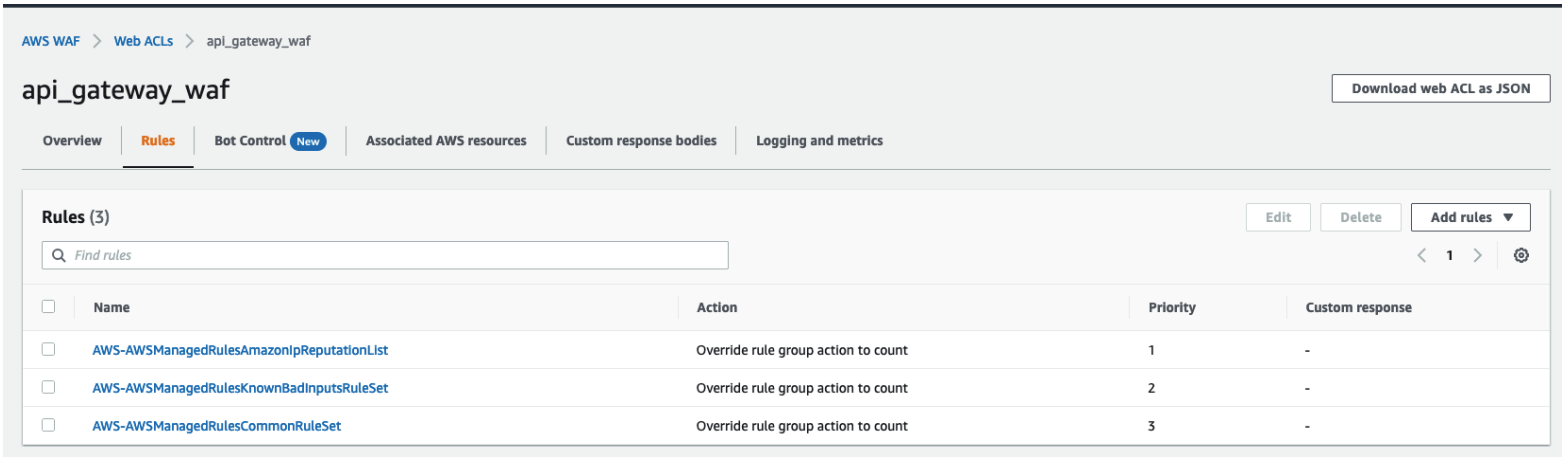


Figure 7.5 AWS WAF Configuration