CS301 - IT Solution Architecture

2021-22T1 G2 Team 1

Project Report

Prepared for:

Professor Ouh Eng Lieh

Prepared by:

Bryan Lee Min Yuan	(01373682)
Christopher Lim Sheng Yong	(01347024)
Justina Ann Wong Juan-Wen	(01384010)
Ko Hui Ning	(01384225)
Pang Huan Shan, Shawn	(01366212)

Background and Business Needs

COMO group's current collection of systems are siloed across all business units and there is no middleware to manage system integration. Hence, there is a need for a platform to consolidate and connect the different sections and allow for a seamless user flow during experience redemption. The above mentioned process of integration should also provide standardisation and documentation of the different isolated systems for future expansion and further development of requirements as per the application roadmap.

As such, the project would need to:

1. Define an enterprise architecture design that defines, standardizes and documents all elements of the respective business entities, taking into account the need for a highly available system.
2. Strategise and manage the APIs to ensure modularity in our approach (for easy maintenance) and set the stage for future ease of scaling of the solution.

Stakeholders

Stakeholder	**Stakeholder Description**	**Permissions**
COMOClub Members	Users of the system	Read/write data through the Club21 application.
Developers	Developers of user-facing services and backend systems in COMO Group.	Read/write access to source code and documentation. Read/write access to CI/CD pipelines. Read access to COMO Group source code and documentation.
Operations	Administrators who primarily focus on maintaining the health of services deployed on AWS and ensure the integrity of data on the EDW.	Read/write admin access to the AWS organization. Read/write admin access to database systems and configuration. Read access to COMO Group source code and documentation.
Security Team	Developers who monitor systems for malicious activity and conduct simulated cybersecurity attacks in search of vulnerabilities in the system.	Read access to system logs. Read access to COMO Group source code and documentation.

Key Use Cases

Use Case Title	Create an Experience Booking by redeeming COMO points [Appendix 1.1 - diagram]
Use Case ID	1
Description	The user makes the Experience Booking via the Club21 application and pays via COMO points. The user sees a screen confirming his/her booking at the specified date and time.
Actors	User
Main Flow of events	1. User selects experience. 2. The mobile application calls the middleware which makes a GET request to 7Rooms to retrieve available dates and times (/venues/availability). 3. User selects the preferred date and time.

	4. User views the amount of existing COMO points stored within the mobile application. 5. User selects COMO points option. 6. User reviews and confirms purchase. 7. The mobile application sends a grouped transaction to the middleware to make a booking with COMO points (PUT /book). The grouped transaction contains the request bodies needed for external API calls to make-booking (7Rooms) and redeem-points (Memberson). 8. Behind the scenes, the middleware makes a PUT request to 7Rooms (/venues/{venue_id}/book) to confirm the reservation. 9. Then, the middleware makes a POST request to Memberson to deduct the appropriate number of COMO points (/api/transaction/redeem-point). 10. User views confirmation screen.
Alternative Flow of events	<u>Middleware detects that 7Rooms is down at step 2:</u> 1. As the request to 7Rooms is a GET request, it is discarded by the middleware instead of being sent to the message queue. 2. The middleware replies the mobile application with a 502 Bad Gateway response <u>At step 7, the middleware detects that 7Rooms/Memberson is down:</u> 1. The middleware pings the external 7Rooms/Memberson and receives a 5XX error, thus detecting that it is down. 2. As the request is a PUT request, it is published to a request exchange to be consumed when the external services are all up. 3. The middleware responds with a 202 ACCEPTED and the system informs the user that a confirmation will be sent when the booking has been successfully processed (booking is still pending). 4. Middleware continuously pings 7Rooms/Memberson, checking for activity. 5. When the middleware receives a successful response from 7Rooms/Memberson, it a. Consumes and processes the queued booking from the requests queue. b. Upon success, it will add the successful record to the notifications exchange. c. The mobile application will consume the success message from the notifications queue and update the user on the booking confirmation.
Pre-conditions	User is logged in and viewed available experiences. User has decided on the experience for which he/she wants to make a booking. User has sufficient COMO points to pay for selected Experience Booking.
Post-conditions	User has a successful booking saved in 7Rooms. User has the correct number of points deducted from his/her account.

Use Case Title	View experiences [Appendix 1.2 - diagram]
Use Case ID	2
Description	The user should be able to view experiences that are suitable based on membership type. The experiences should also be updated with information on availability, where applicable.
Actors	User
Main Flow of	1. User logs in. The mobile application sends a grouped transaction to the middleware

events	(/login). The grouped transaction contains the request bodies needed for external Memberson API calls to get customer number, get profile token and get membership information. a. Get customer number: Middleware sends a POST request to 'api/profile/search-simple'. b. Get profile token: Middleware sends a POST request to '/profile/{customer_no}/signin'. c. Get member record and membership tiers: Middleware sends a GET request to 'profile/{customer_no}/ memberships'. 2. User selects an experience from the experiences page, mobile application calls the middleware to a. Make a GET request to 7Rooms API to retrieve venues ('/venues'), thus retrieving all experiences from BUs/Brands. b. Calls 7Rooms API to retrieve venue availability for the selected venue through a GET request ('/venue/{venue_id}/ availability'). The app updates the selected venue offering with available dates and times. 3. The user sees a list of experience offerings which are upcoming, available and suitable to the membership tier.
Alternative Flow of events	<u>Login is unsuccessful at step 1, due to wrong password:</u> ● The middleware returns 401 Unauthorised. ● User can still view experiences that are available to public (i.e. below e-member tier) <u>7Rooms API fails at step 2b:</u> ● The app lists the venues from step 2a, but details about availability will be unavailable.
Pre-conditions	Users have an account with a valid membership tier, i.e. e-member, regular, classic, prestige or elite.
Post-conditions	Users have a viewable list of experiences, with their booking availability updated.

Use Case Title	Item redemption using COMO points [Appendix 1.3 - diagram]
Use Case ID	3
Description	The user has selected an item of their liking from the list of redeemable items and wishes to checkout the item with Como points.
Actors	User
Main Flow of events	1. User reviews the selected item(s). 2. The mobile application makes a call to the middleware to check if the user has sufficient points and if the item(s) are in stock. The middleware makes a POST request to the Memberson CRM Reward API to check if reward is redeemable ('/reward/check-redeemable'). 3. The user clicks on 'Pay with Como Points'. 4. The mobile application sends a grouped transaction to the middleware (POST /redeem_item). The grouped transaction contains the request bodies needed for external Memberson API calls to redeem points and redeem the reward. a. Redeem points: Middleware makes a POST request to

	'/api/transaction/redeem-point'. b. Redeem reward: Middleware makes a POST request to '/reward/redeem'. 5. User sees that the item has been successfully redeemed.
Alternative Flow of events	<u>At step 4, the middleware detects that Memberson is down:</u> 1. The middleware pings Memberson and receives a 5XX error, thus detecting that it is down. 2. As the request is a POST request, it is published to a request exchange to be consumed when the external services are all up. 3. The middleware responds with a 202 ACCEPTED and the system informs the user that a confirmation will be sent when the redemption has been successfully processed (redemption is still pending). 4. Middleware continuously pings Memberson, checking for activity. 6. When the middleware receives a successful response from Memberson, it a. Consumes and processes the queued redemption from the requests queue. b. Upon success, it will add the successful record to the notifications exchange. c. The mobile application will consume the success message from the notifications queue and update the user on the booking confirmation. <u>Memberson CRM Reward API fails at step 4b due to 400 'Could Not Complete' error:</u> 1. The middleware makes a POST request to Memberson CRM Cancel Redeem Points API. This refunds the points redeemed at step 4a. (/cancel-instant-redemption). 2. The user receives a 400 Could Not Complete response, the item could not be redeemed.
Pre-conditions	User is logged in and views rewards which are eligible to their membership type.
Post-conditions	User successfully makes an item redemption, which should be registered to their member number. Points used for redemption will be deducted from the account.

Quality Attributes

- Maintainability
  - A 3-tiered architecture (web, application, and database) abstracts the different services into individual tiers, enabling the notion of separation of concerns and the ease of maintainability for each tier.
  - Implement CI/CD pipeline to automatically test, build and deploy our application to improve software testability.
  - Implement a microservice architecture with Fargate instances on Elastic Container Service (ECS), with each microservice having its own independent repository and CI/CD pipeline to test, build and deploy.
- Availability
  - Redundancy is introduced by horizontally scaling our middleware microservice instances with ECS service auto scaling and coordinating the redundant instances with active-active clustering. If one instance goes down, the other instances are still available and can still serve requests. For production, we propose using a 3-node RabbitMQ cluster deployment in an active-active configuration on Amazon MQ for high availability.
  - Application load balancers provide high availability by distributing traffic amongst a target group of instances, while ensuring they are healthy and running by automatically detecting failures in them.

    - Multiple Availability Zones (AZ) isolates failures (e.g. power loss in an AZ). When one AZ becomes unavailable, the load balancer will redirect traffic to the instance on the other available AZ.
- Security
    - Separation of public and private subnets. The public web subnet has publicly accessible instances (i.e. reverse proxy), and the private application subnet blocks direct access from the Internet to its instances.
    - Configure a chain of security groups that are specific to each tier, such as allowing port 3306 for databases and port 80/443 for web servers. These security groups ensure that only desired traffic can flow between tiers, thus reducing the attack surface in the event of a compromised client.
    - Enable data encryption in transit (using SSL/TLS) using an SSL certificate issued by AWS Certificate Manager and attach it to our internet-facing load balancer for SSL termination.
    - Configure AWS Identity Access Management (IAM) users, roles, and policies to provide authorised stakeholders (e.g. developers, operations, and security team) with granular access to AWS services relevant to them.
    - Configure AWS CloudWatch alarms to monitor for potential threats and anomalous behaviour.
- Performance
    - Cross-zone load balancing in the ALB equally distributes traffic across the registered middleware targets in the auto scaling group across all AZs, avoiding the case of overloading our middleware instances on one AZ.
    - Auto scaling monitors our middleware instances and automatically scales the number of instances to cope with fluctuating workloads.

Architectural Decisions

<table>
<tr><td colspan="2">Architectural Decision - Microservices architecture</td></tr>
<tr><td>ID</td><td>1</td></tr>
<tr><td>Issue</td><td>Decoupled services, ease of deployment, scalability.</td></tr>
<tr><td>Description</td><td><u>Implementation</u>
Our middleware is built on a microservices architecture. It has the following services:
Composite
Acts as an orchestrator that communicates with healthcheck proxy service (see below) to carry out business logic.
Example: Booking a venue would call 7Rooms twice (i.e. authentication + make-booking) and CRM once (i.e. to deduct COMO points). (Appendix 1.1)
[Maintainability] Implements the adapter pattern to handle mixed message formats to accommodate external endpoint requirements.
Healthcheck Proxy Services (Atomic Service)
Handle calls to external API such as Memberson, CRM Rewards and 7Rooms.
[Availability] Polls external API for health and updates RabbitMQ if service is online.
[Maintainability] Extensible and easily attachable to external APIs and composite services.</td></tr>
</table>

	• **[Modifiability]** Each healthcheck proxy service in the microservice middleware can be developed independently, allowing for split-stack development and greater developer velocity. • **[Modularity]** Each healthcheck proxy service can be scaled independently depending on usage. For example, if 7Rooms API is called more than CRM API, we can scale up the 7Rooms adapter service to meet the high usage. **AMQP-WS Proxy Service (Atomic Service)** • **[Availability]** Allow web clients to subscribe to the required AMQP brokers via WebSockets. • **[Security]** Hide broker credentials from client applications.
Assumptions	-
Alternatives	Monolithic architecture.
Justifications	Although monoliths are less complex and easier to implement at the start, the components will get progressively harder to scale and maintain as the application grows due to tight coupling. Given COMO's desire to onboard new Brands and Business Units, we propose that microservices is a more robust architecture that supports COMO's growth. Monoliths would require re-deploying the entire service whenever changes are made, whereas micro-services only require re-deploying the affected services. **[Availability]** On the other hand, microservices allow for a greater degree of fault tolerance, as services fail independently. In case of error, we can still provide degraded service instead of no service.

Architectural Decision - CI/CD	
ID	2
Issue	Proper application life-cycle management is important to ensure developers are well-equipped to make frequent code changes that can be efficiently and safely deployed without risking code breakage.
Description	We implement a CI/CD pipeline via GitHub Actions, which carries out the following tasks: 1. GitHub Actions run component tests. 2. GitHub Actions build the docker image and push it to ECR. 3. AWS ECS pulls the image from ECR and deploys the Fargate instance.
Assumptions	-
Alternatives	Manual testing, release and deployment. Different CI/CD tools for the implementation (eg. GitLab, Jenkins).
Justifications	A manual deployment process (testing, release and deployment) slows down the development of new features and reduces the release frequency. The CI/CD pipeline

	enables upgrades to introduce smaller units of change, hence making them less disruptive and decreasing the frequency of pushed code causing errors. The project required that the projects be hosted on GitHub, hence GitHub Actions was the easiest to integrate for use with our system. **[Maintainability]** CI/CD pipeline allows the development team to make frequent and reliable code changes in an automated, testable environment.

Architectural Decision - API Integration pattern for B2B	
ID	3
Issue	The main purpose of the middleware is to coordinate between the COMO mobile application and the external services used, increasing maintainability of the system as it continues to scale to include more entities.
Description	**Healthcheck Proxy Service** ● Each external API is mapped to a healthcheck proxy service within the middleware, which pings the external API periodically to check health. **Composite Service** ● The composite service contains adapter objects which adapt requests from the client application into the format accepted by each external API. ● The composite service also defines certain endpoints which logically group multiple external API calls together, creating a cleaner interface for the client application to interact with. ● **[Maintainability]** Composite implements a facade pattern, providing a unified interface to a set of interfaces. The frontend application is decoupled from the backend services.
Assumptions	-
Alternatives	Front-end applications coordinate and call each service individually.
Justifications	For the middleware use case, API Integration is necessary (as opposed to other modes of communication) as this is the integration pattern chosen by the external services. Furthermore, the middleware requires transaction level data transfer and communicates with external services synchronously, making it the most appropriate integration pattern to use. **[Modifiability]** The adapter objects can be easily modified when changes to the external APIs occur, abstracting these changes away from the composite logic and client applications which only need to have knowledge of the middleware's API. **[Availability]** The integration pattern decouples the middleware from the external APIs by ensuring that the composite service is able to process requests and send responses back to the mobile application even when the external APIs are down.

Architectural Decision - Message integration pattern for internal communication	
ID	4
Issue	Allows for (1) message persistence and (2) asynchronous communication when needed.
Description	The middleware uses a RabbitMQ broker for 3 main functions: 1. for the composite service to receive health updates from healthcheck proxy services; 2. for the composite service to queue requests for future processing upon encountering failure of external API(s); and 3. for the client application to receive updates upon successful processing of queued requests, once external API(s) recover.
Assumptions	-
Alternatives	REST.
Justifications	REST does not support asynchronous communication and message persistence, thus failing to satisfy some of the above use cases (i.e. notifications and caching incoming requests) as well as messaging does. **[Availability]** The message integration pattern decouples the middleware and external APIs and ensures that the middleware is always able to return a response upon a request from the mobile application even if one of the external APIs which it needs to call to process that request is down. **[Reliability]** Messages in the queue are persistent so if the broker were to be shut down while the message is in queue, ready to be processed, the message will still be processed normally when the broker comes back up.

Architectural Decision - Defence in Depth Pattern with 3-Tiered Architecture	
ID	5
Issue	Chaining security groups by splitting services across multiple tiers.
Description	Our middleware is split into 3 tiers: data tier, application tier, web tier. To achieve defence in depth, we configured a chain of security groups that are specific to each tier, such as allowing port 3306 for database tier and port 80/443 for web server tier.
Assumptions	-
Alternatives	Single point of defence.
Justifications	A single point of defence may be easier to implement, but it also risks becoming a single point of failure. For the single point of defence to be effective, it must be up and running constantly (i.e. 100% availability), which is infeasible. **[Security]** The chained groups ensure that only desired traffic can flow between tiers, thus

	reducing the attack surface in the event of a compromised client. **[Maintainability]** A tiered architecture abstracts the different layers and is more modular and reusable. Hence, it makes the system easier to implement. **[Modularity]** Enforces separation of concerns by abstracting the different layers into discrete web-app-data tiers.

Architectural Decision - Network Traffic Management (Load Balancing/Reverse Proxies)	
ID	6
Issue	Security, high availability
Description	To handle incoming traffic from the Internet, we use reverse proxies to forward requests from publicly accessible instances in the public subnet to private applications within the private subnet. This provides a layer of security by protecting core services which handle sensitive customer data from being directly accessed via the Internet. To ensure high availability, AWS application load balancers are configured to perform health checks and to distribute traffic amongst a target group of reverse proxy instances. Load balancers also support cross-zone implementation across multiple Availability Zones(AZ). For example, in the event of power loss in one AZ, the load balancer will redirect traffic to the instance on the other available AZ.
Assumptions	-
Alternatives	-
Justifications	-

Architectural Decision - Clustering/Resilience Patterns (AZs, ECS Cluster)	
ID	7
Issue	The middleware is highly available in accordance with ISO 25010 principles. We incorporate redundancy to prevent single points of failure in our application and use clustering to coordinate between the redundant components.
Description	We auto scale the middleware ECS microservice instances horizontally across two AZs. If the instances in one AZ go down, our internal load balancer will be able to direct traffic to the other zone. We also run multiple instances of the Fargate instances in our ECS Cluster (active/active configuration) which enables parallel execution of workloads. This will also allow us to dynamically respond to changing traffic loads by adding or removing instances from the cluster.
Assumptions	-
Alternatives	Vertical scaling
Justifications	Choosing to vertically scale our solution instead would introduce a single point of failure as

	each microservice will run on a single instance. Furthermore, there is a limit to how much the instance's performance can be improved by adding resources such as CPU and storage, as opposed to horizontal scaling where ephemeral instances can be added or removed when necessary. However, it would be possible to incorporate vertical scaling into our horizontally scaled system if it is deemed beneficial as the middleware takes on more loads in future.

Architectural Decision - AmazonMQ Broker with EBS Volume as a Data Tier	
ID	8
Issue	Data storage
Description	Each Amazon MQ broker instance communicates with an Amazon EBS storage volume, which provides block level storage optimized for low-latency and high throughput. This feature allows us to store the messages published to the exchange in the EBS storage, which can be later consumed by the application and web tiers. For demo purposes, our team has opted to use a single-instance RabbitMQ broker. For actual production, we propose using a cluster deployment[1] of three RabbitMQ broker nodes managed by a Network Load Balancer. This cluster deployment ensures high availability as the NLB is able to manage traffic at layer 4, including sudden and volatile traffic patterns.[2]
Assumptions	-
Alternatives	RDS DB
Justifications	In our proposal, we originally proposed RDS DB. Upon consultation with the client and our architectural decision to use message integration for internal communication, we decided that it is more appropriate to use the built-in RabbitMQ compatibility with EBS Volume.

Proposed Budgets

Actual Development Budget

Activity/Hardware/Software/ Service	**Description**	**Cost**
Designing the cloud architecture	High level overview of the cloud architecture design of the deliverable with enterprise architecture diagrams	**50 man-hours**
Implementing service logic	Produce the source code and simple documentation of the middleware to integrate the different siloed systems.	**100 man-hours**
Configuring an ECS cluster with 4 services running 2 Fargate instances each, and	Each instance utilises 0.5GB task memory and 0.25 vGPU	1013 hrs memory: 4.50 USD 507 hrs vCPU: 20.50 USD **Total: 25 USD**

[1] https://docs.aws.amazon.com/amazon-mq/latest/developer-guide/rabbitmq-broker-architecture-cluster.html

[2] https://aws.amazon.com/elasticloadbalancing/network-load-balancer/

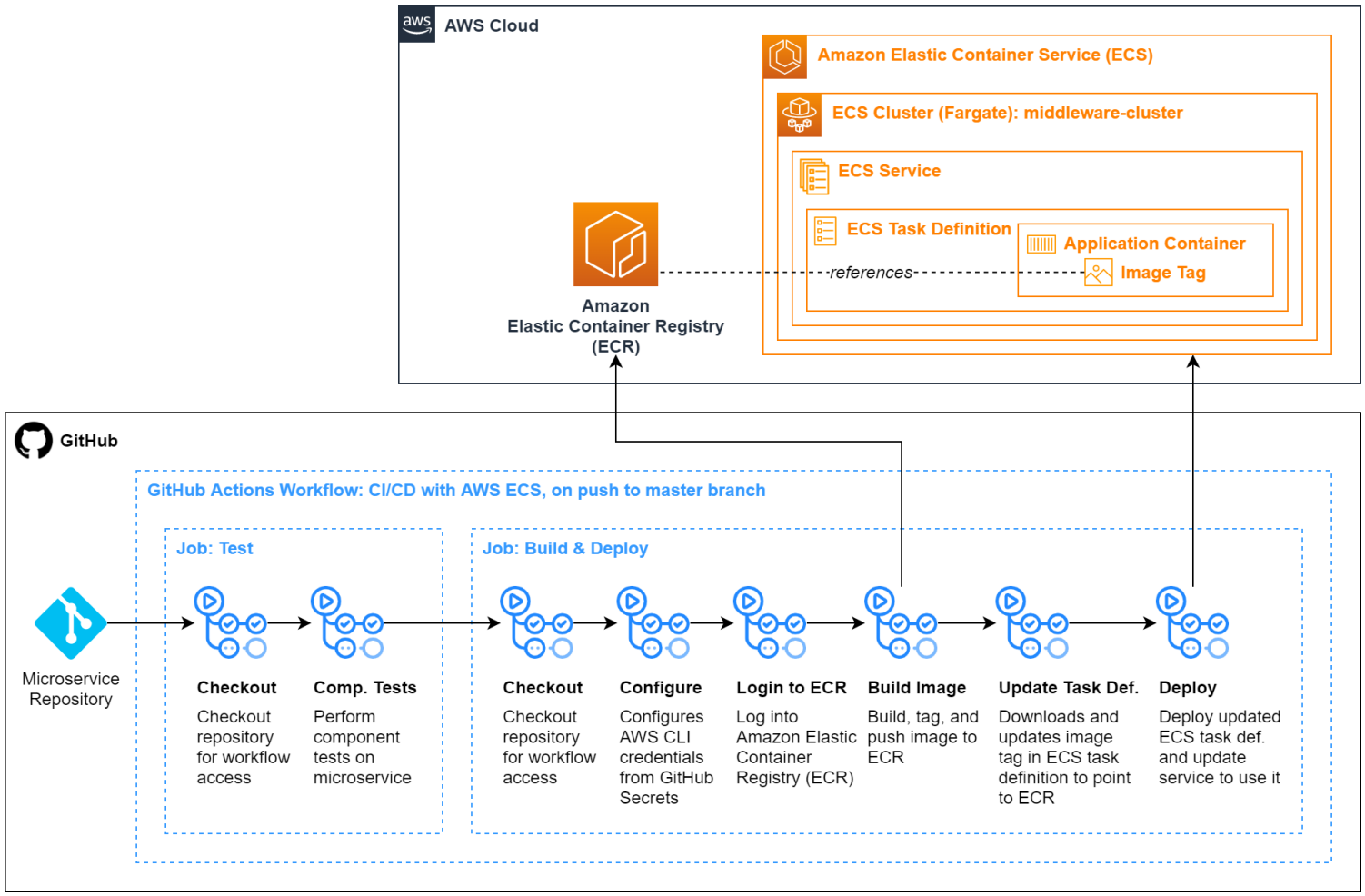
frontend on a single instance		
Running of 4 tg2.micro AWS EC2 instances	2 NAT instances in the public subnet to enable the middleware to access the internet, and 2 instances to host proxy servers on the public subnet	6469 hrs of use: 75.05 USD 91.525 GB storage: 9.15 USD **Total: 84.20 USD**
Running of 4 AWS Application Load Balancers	To route traffic between the proxy servers and between the middleware instances	5.761 LCU-hrs: 0.05 USD 2479 hrs of use: 55.78 USD **Total: 55.83 USD**
Running of 1 mq.t3.micro broker instance	Message broker service for RabbitMQ to queue pending bookings and microservice health checks.	783 hrs of use: 21.59 USD 21 GB storage: 2.10 USD **Total: 23.69 USD**
Configuring Amazon Route 53	To route end users and do routing within the system between services using the DNS service.	2 Hosted zones: 1.50 USD 827 DNS queries: 0.00 USD 963 Intra-AWS-DNS-queries: 0.00 USD **Total: 1.50 USD**
Monitoring with AWS CloudWatch	Configure alarms to detect and warn the team of any vulnerabilities and anomalous behaviour.	2.3 alarms: 0.23 USD 13.842 metrics: 4.15 USD 0.137GB log data: 0.07 USD 0.01GB log storage: 0.00 USD **Total: 8.90 USD**
Developing the frontend application	Code out the different UI interfaces and define functions to call the middleware app	**50 man-hours**
Implementing a CI/CD pipeline	Write unit and integration tests and design CI/CD pipelines for automated deployment.	**10 man-hours**
Total	**199.12 USD, 210 man-hours over the course of Oct-Nov 2021**	

Proposed Production Budget

Activity/ Hardware/ Software/ Service	**Description**	**Cost**
Implementing service logic	Produce the source code and documentation of middleware, configure it to integrate the siloed systems.	**200 man-hours**
Implementing CI/CD pipeline	Write unit, integration and component tests, and design CI/CD pipelines for automated deployment.	**150 man-hours**
Configuring an ECS cluster with 4 services running 2 Fargate instances each	**Availability, Maintainability** To provide scalable on-demand computing capacity, round the clock availability, and server accessibility. The use of cloud computing resources reduces the effort of maintaining the entire system.	35.55 USD per service per month, **142.20 USD** per month total

Running of 2 t4g.medium AWS EC2 instances to host proxy servers	**Security** To allow for secure connection and data transfer between middleware microservice instances and the internet.	18.40 USD per instance per month, **36.80 USD** per month total
Running of 2 NAT gateway instances, estimated traffic of 30GB/month	**Availability** To allow outbound connectivity from EC2 instances to the internet.	34.20 USD per instance per month, **68.40 USD** per month total
Running of 4 AWS Application Load Balancers with traffic of 0.04GB/hr each	**Performance, Availability** To distribute incoming traffic across multiple EC2 instances. Ensures optimal performance of the server and can perform fail-over in event of failure, hence ensuring availability of the system.	16.66 USD per instance per month, **66.64 USD** per month total
Running of a cluster of 3 AWS RabbitMQ mq.m5.large single-instance brokers	**Availability** To allow the composite microservice to check for the health of the external services and store pending and processed bookings such that the composite microservice can continue to be available to process bookings even when external services are down.	240.24 USD per instance per month, **720.72 USD** per month total
Configuring Elastic Container Registry	**Availability** Automated container image deployment without the need for provisioning infrastructure. Allows systems to distribute images faster, reduce download times and improve availability using scalable, durable architecture.	0.10 USD per GB per month **0.10 USD** per month total
Configuring Amazon Route 53	**Availability, Performance** Routes traffic between the system and the external network and enables discovery of microservices. Automatically scales to handle large query volumes. 2 hosted zones (1 for domain, 1 for service discovery).	0.50 USD per hosted zone 0.40 USD per 1000000 queries **1.40 USD** per month total
Monitoring with Amazon CloudWatch	**Availability, Performance, Security** Monitors operational data and logs to check on the health of running instances. Detects anomalous behaviour to alert the team for intervention. Can also complement the use of auto scaling through alarms to alert server scaling.	0.30 USD per month per 10000 metrics, 3 USD per dashboard, 0.10 USD per standard resolution alarm metric, 0.30 USD per high resolution alarm metric, 0.50 USD per GB logs, 1 USD per million events **18.02 USD** per month total
Using the AWS Key Management System	**Security** Enables creation and management of cryptographic keys. Allows for digitally signed operations to ensure the integrity of the data persisted.	1 USD per key per month, first 20000 requests per month are free, **1 USD** per month total
Total	**1055.28 USD, 350 man-hours per month**	

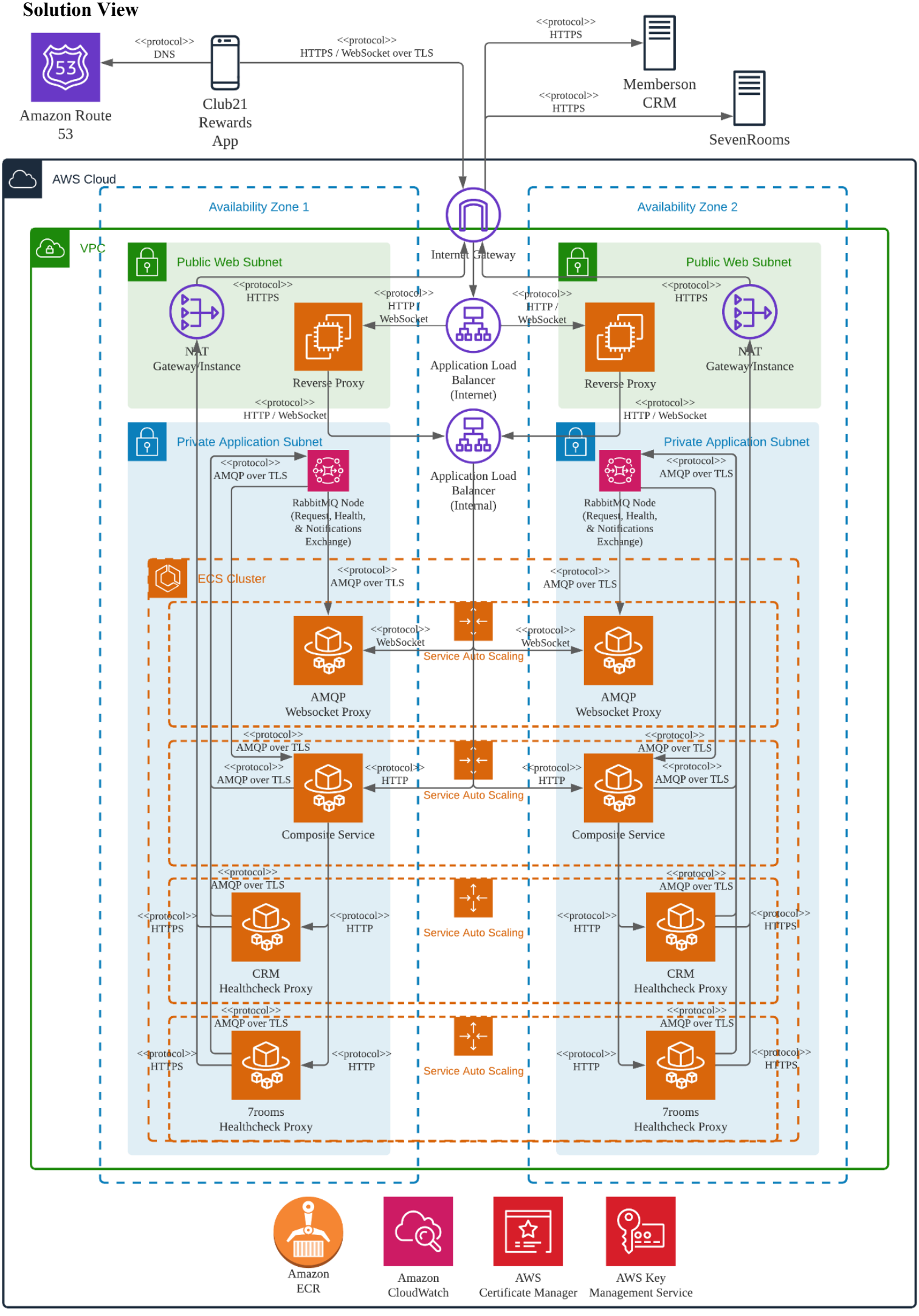
Development View



We implemented a CI/CD pipeline/workflow using GitHub Actions to automatically perform component tests, building of our microservice containers, and deploying them straight to AWS Elastic Container Service. Every microservice in our middleware cluster has its own individual pipeline to deploy the microservice into its respective ECS service. This enables us to deploy each microservice independently, further promoting loose-coupling amongst the microservices during deployment.

For each microservice repository, upon pushing code commits to the master branch on GitHub, GitHub Actions runs the CI/CD workflow to perform component tests to ensure they all pass (healthcheck proxy microservice), build a Docker image using the Dockerfile present in the repository, uploads the image to Amazon Elastic Container Registry (ECR), downloads the existing task definition of the microservice from ECS, updates the image tag to point to the new image stored on ECR, and finally uploads the updated task definition to ECS to redeploy the microservices. If any job fails, the entire workflow will fail and will not proceed with the deployment of the microservice. The developers will then have to rectify the issues brought up and make a new commit to run the CI/CD workflow again.

Solution View



Integration Endpoints

Source System	Destination System	Protocol	Format	Communication Mode
Club21 Rewards App	Amazon Route 53	DNS	DNS	Synchronous
Club21 Rewards App	IGW, Internet-Facing ALB	HTTPS	JSON	Synchronous
Club21 Rewards App	IGW, Internet-Facing ALB	WebSocket (TLS)	JSON	Asynchronous
Internet-Facing ALB	Reverse Proxy	HTTP	JSON	Synchronous
Internet-Facing ALB	Reverse Proxy	WebSocket	JSON	Asynchronous
Reverse Proxy	Internal ALB	HTTP	JSON	Synchronous
Reverse Proxy	Internal ALB	WebSocket	JSON	Asynchronous
Internal ALB	Composite Service	HTTP	JSON	Synchronous
Internal ALB	AMQP-Websocket Proxy	WebSocket	JSON	Asynchronous
Composite Service	Broker	AMQP (TLS)	JSON	Asynchronous
Composite Service	CRM Healthcheck Proxy	HTTP	JSON	Synchronous
Composite Service	7Rooms Healthcheck Proxy	HTTP	JSON	Synchronous
CRM Healthcheck Proxy	Broker	AMQP (TLS)	JSON	Asynchronous
CRM Healthcheck Proxy	NAT, IGW, Memberson	HTTPS	JSON	Synchronous
7Rooms Healthcheck Proxy	Broker	AMQP (TLS)	JSON	Asynchronous
7Rooms Healthcheck Proxy	NAT, IGW, 7Rooms	HTTPS	JSON	Synchronous
Broker	Composite Service	AMQP (TLS)	JSON	Asynchronous
Broker	AMQP-Websocket Proxy	AMQP (TLS)	JSON	Asynchronous

Ease of Maintainability

A monolithic middleware was initially proposed to handle COMO's middleware layer. However, with maintainability, flexibility and extensibility in mind, we shifted focus to implementing a microservice architecture. We did this by defining Fargate task definitions and creating Fargate services in a cluster on Elastic Container Service (ECS). As mentioned in our key architectural decisions, each microservice can be developed independently, with each having its own Git repository and dedicated CI/CD pipeline to test, build and deploy, leading to greater development velocity, less development friction, and less coupling between the microservices. Furthermore, on ECS, each microservice receives a "sidecar" log router container to handle routing of all output application logs and feed them into CloudWatch for centralised monitoring. All of which increases the ease of maintainability of the middleware.

As the middleware heavily involves calling external services (i.e. Memberson [CRM & Rewards], 7Rooms, and future expansions), we naturally have to write code to interface with third party services. To ensure that

maintainability is kept, we introduced the *adapter* design pattern into our codebase by converting the interface of our class into another interface that is compatible with an external service. We created an adapter for each external service we call. In our case, we have two adapter objects in our composite booking service to enable compatibility with the two external services (Memberson and 7Rooms). This promotes the single responsibility principle as it separates the logic of converting data (into what the external services expect) away from our main business logic (e.g. making a booking in our booking composite service).

CloudFormation

We created CloudFormation templates that configure the necessary infrastructure resources for the middleware. This includes a "main.yml" root stack template that will create all necessary nested stacks and resources for the middleware infrastructure. Nested stack templates are statically hosted on S3 and referenced through their URL.

The templates are concerned with the overall infrastructure of the application, including the VPC, the Internet gateway, subnets, routing tables, security groups, NAT instances, reverse proxies, load balancers, CloudWatch logging, the message broker, and the ECS cluster. For services within the ECS cluster, we believe that services and task definitions should be created individually by the developers of the microservices. This increases developer ownership and allows for a more flexible deployment strategy for each microservice team. To deploy our services, we manually created task definitions then configured our CI/CD pipelines to automatically create new revisions and update the services whenever necessary. See appendix 3 for additional steps to be carried out on the base infrastructure to recreate the middleware.

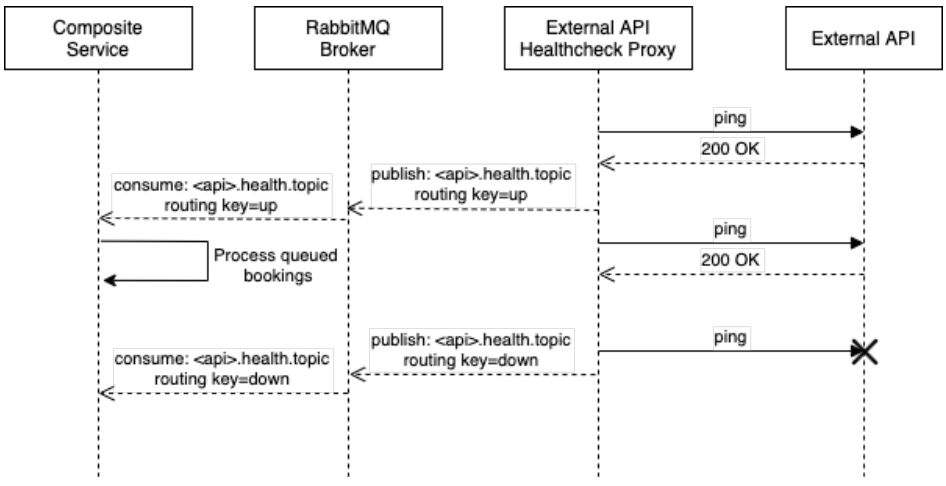
Flexibility at Adapting to New Changes

We developed the middleware with COMO's mission of being "simple yet all-encompassing" in mind, and with the main goal of the middleware being the orchestrator of various systems that will manage system integration across business units and provide many different services. As of current development, only Memberson and 7Rooms are called to handle our use cases. However, we understand that more services will need to be added in the future, e.g. Stripe payment services, other loyalty programme systems, appointment/calendar booking service, etc. Our healthcheck proxy service was created with extensibility in mind to handle the integration of these future services. The healthcheck proxy (project-2021-22t1-g2-g2team1-healthcheck-proxy), as mentioned, handles calls to external APIs and polls those APIs for health checking to determine whether the external service is online or offline. The healthcheck proxy uses environment variables to adapt to different services, for example, providing the environment variable "ENDPOINT" will direct the healthcheck proxy towards the external service that is hosted on that endpoint. This enables us to use the same healthcheck proxy codebase for each external service, including future services that COMO intends to integrate with the middleware. When we need to integrate a new external service, we can simply define a new continuous deployment pipeline job with service-specific environment variables to handle the deployment of the healthcheck proxy for the new external service, with no change of code needed. This is how our current healthcheck proxies for Memberson and 7Rooms external services are currently being deployed using the same codebase, simply by defining two continuous deployment pipeline jobs within the same workflow of healthcheck proxy's code repository (see the two deploy jobs at the bottom of cicd.yml). Hence, the way we wrote the healthcheck proxy promotes reusability of code, making the code more maintainable as it can be modified in a single location, and enabling the middleware to be extensible and flexible at adapting to new external service integrations.

Availability View

Node	Redundancy	Clustering			Replication
		Node Config.	Failure Detection	Failover	
Composite Service	Horizontal scaling across 2 availability zones	Active-Active	Load balancer pings the service Monitoring with alerts by Amazon CloudWatch	Load balancer routes traffic to the other available instance	NA - Sessions and data management are not the concern of the middleware
Healthcheck Proxy Services	Horizontal scaling across 2 availability zones	Active-Active	Load balancer pings the service Monitoring with alerts by Amazon CloudWatch	Load balancer routes traffic to the other available instances	NA - Sessions and data management are not the concern of the middleware
RabbitMQ Broker	**Existing:** Single-instance broker (to limit cost) **Proposed:** Cluster deployment for high availability	**Proposed:** Active-Active	**Proposed:** Load balancer pings the service (managed by Amazon MQ) Monitoring with alerts by Amazon CloudWatch	**Proposed:** Queue mirroring failover mechanism: if master node goes down, another node will be promoted to master	**Proposed:** Queue mirroring (managed by Amazon MQ)
External APIs	NA	NA	Details below	Details below	NA

External API Failure Detection - Healthcheck Proxy Mechanism

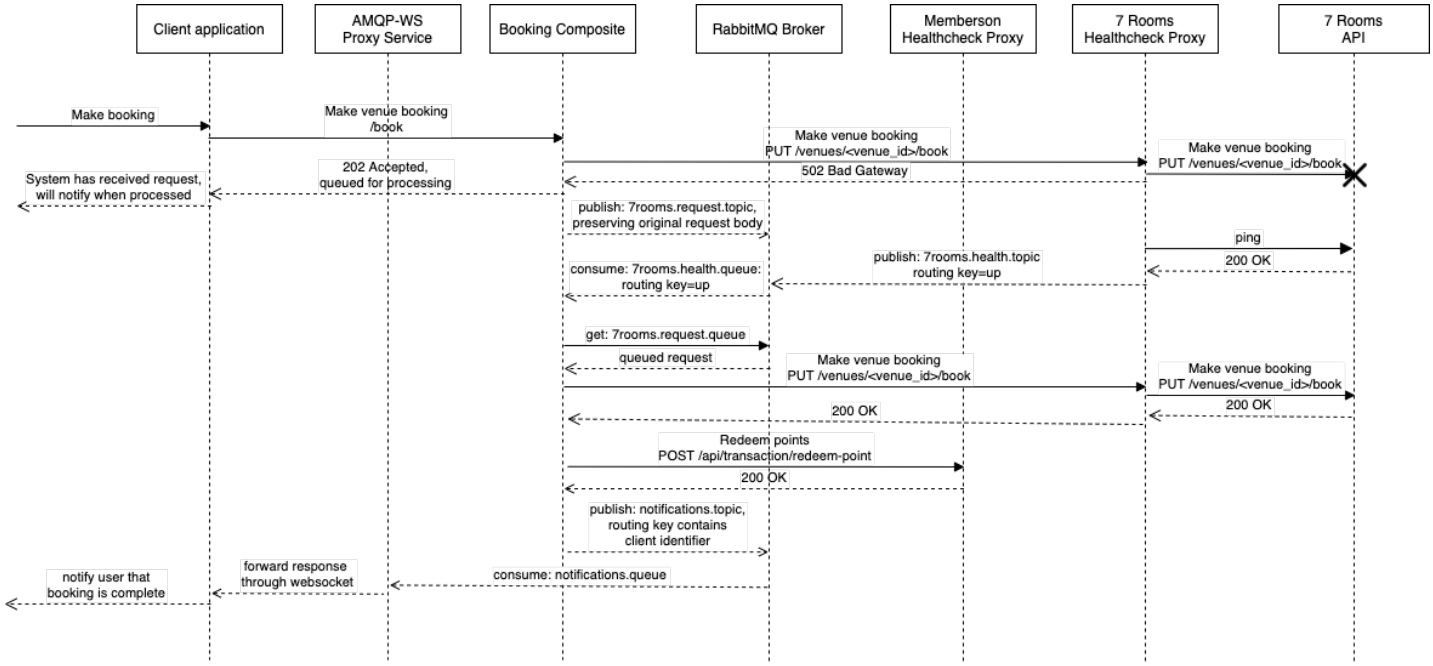


Each external API has its own healthcheck proxy service, which pings the external API at a constant interval. The healthcheck proxy declares an exchange on the RabbitMQ broker on start, which it uses for event-driven communication with the composite service. If a ping timeout occurs, the healthcheck proxy publishes a message to the exchange using the routing key "down". When the healthcheck proxy detects that the external API is back up via a 200 OK response from a ping, it publishes a message to the exchange using the routing key "up". The messages have no payload and are differentiated by their routing keys.

External API Failover - Composite Failover Mechanism (using Use Case 1)

The composite service handles failover if any of the external API(s) involved in the request are down for PUT and POST requests:

1. The healthcheck proxy returns a 502 Bad Gateway response to the composite service.
2. The composite service returns a 202 ACCEPTED response to the client application instead of a 200 OK response, informing the client application that the request cannot currently be processed, but has been queued for future processing.
3. For each external API involved, the composite service runs a separate thread listening for health updates from the RabbitMQ broker. When a health update is received, informing the composite service that the external API is back up, the composite service proceeds to process the queued requests involving that external API.
4. When a queued request has been processed successfully, the composite service publishes the full response to a public RabbitMQ broker which is located in the public subnet. The client application listens for completed requests sent to the public RabbitMQ broker and consumes completed requests, notifying the user that their request has been completed successfully.



Security View

No	Asset/ Asset Group	Potential Threat/ Vulnerability Pair	Possible Mitigation Controls
1	Application Tier/Data Tier	**[Threat]** Malicious or unwanted access to sensitive resources within the VPC - Confidentiality, Integrity **[Vulnerability]** Loosely-defined security group configurations, Lack of process visibility	Use of reverse proxies in the public subnet to control incoming traffic. Provides extensibility for further filtering of traffic, such as IP-based restrictions for non-Singapore traffic. Security group configuration for reverse proxy: Block unnecessary protocols; only allow ports 80(http) and 443 (https) for reverse proxy. Configured the security group to restrict inbound and outbound traffic on the principle of least privilege - allowing only connections on approved ports and IP

		across microservices	addresses. Setup user activity logging on Amazon CloudWatch, which can be used to detect malicious activity. Implemented SSL termination on the internet-facing load balancer with an issued AWS Certificate Manager certificate for the domain como.itsag2t1.com.
2	User and message data persisted in EBS Volume	**[Attack]** DDOS attack on publicly accessible MQ brokers - Availability **[Weakness]** Public RabbitMQ broker	Encryption at rest: AmazonMQ uses AWS Key Management Service to provision an AWS-managed key for server-side encryption of MQ data. Protecting against DDOS: Brokers are created within the VPC with no public accessibility. Security group: Block unnecessary protocols; only allow port 5671 for AMQP.
		[Attack] Hacker attack - Integrity, Confidentiality **[Weakness]** Lack of encryption for data at rest and in transit	Encryption in transit: Data sent from the Club21 App to our internet-facing load balancer, along with the data sent from our microservices to external services is over HTTPS Encryption in transit: Data sent between Amazon MQ brokers is encrypted using SSL/TLS
3	Confidential Secrets in Application Data	**[Attack]** Hackers compromising the container images of the system to access any sensitive data inside **[Weakness]** Storing of sensitive information/credentials (ie. secrets) inside application data	We use GitHub Actions to push the container images to the ECR. By default, ECR uses server-side encryption to encrypt container images at rest using an AES-256 encryption algorithm.

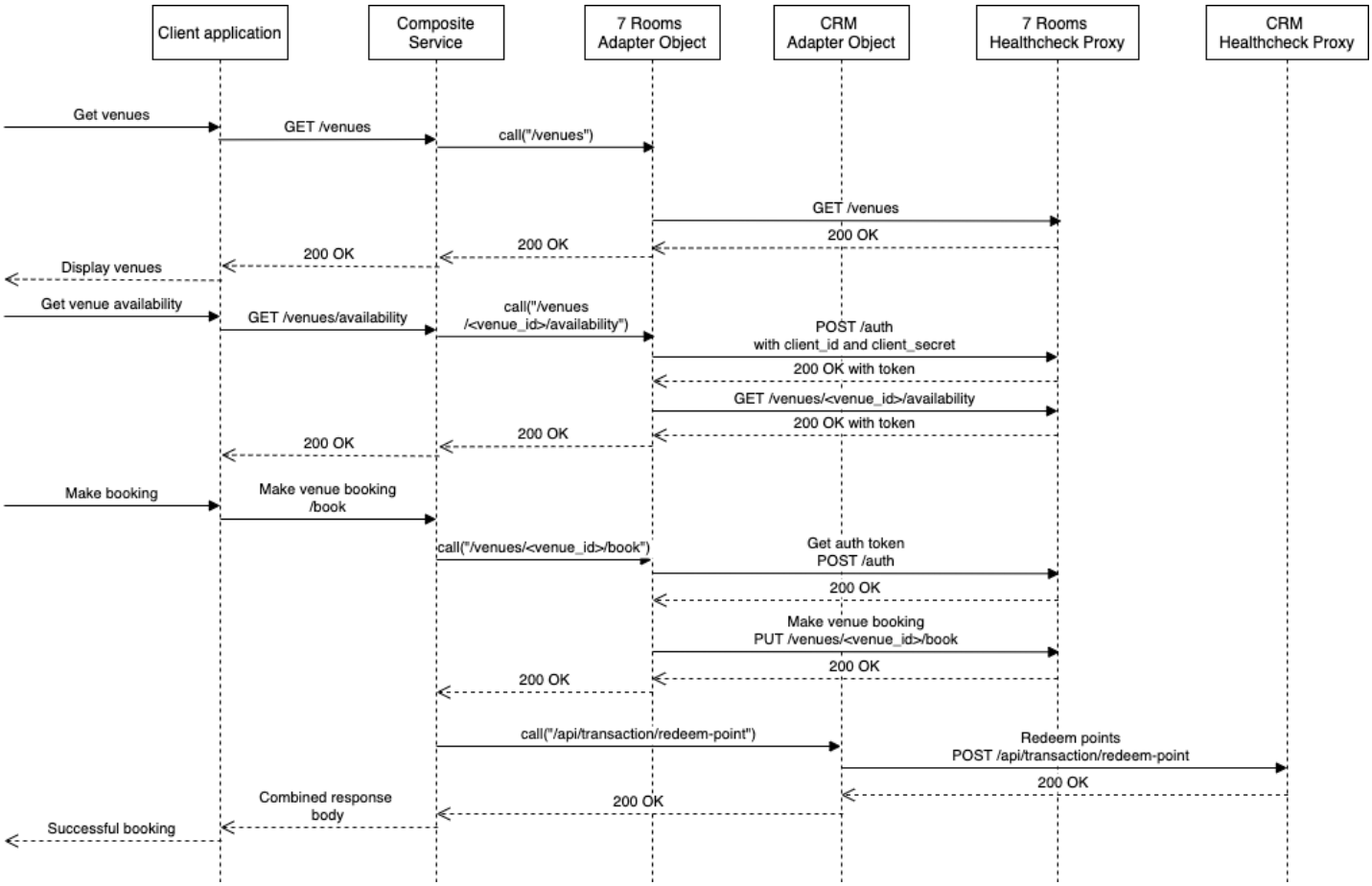
Performance View

No	Description of the Strategy	Justification	Performance Testing (Optional)
1	Centralised Cluster on AWS	Middleware is centralised on Amazon ECS, and all related microservices are provisioned within the same region. This ensures that latency within the middleware is kept to a minimum, thus making requests within our cluster and between composite and adapter services will be performant.	Baseline: Make a direct call to the external API endpoint Test: Make a call to the external API endpoint through the middleware Evaluation: take the difference in times to measure middleware latency
2	ALB cross-zone load balancing	Cross-zone load balancing at the request level (Layer 7) to equally distribute traffic across middleware targets in the auto scaling group, thus reducing overloading of any single middleware instance.	-
3	ECS Service Auto Scaling	ECS service auto scaling monitors application load on our microservice to automatically scale in and out the number of instances according to fluctuating traffic (based on CPU percentage), thus providing steady and predictable performance.	See appendix for ECS service auto scaling during a simulated high workload test on our composite service
4	HTTP Caching	Web application is server-side-rendered. This reduces initial page load times and allows for static content to be cached at edge networks.	-
5	[Not implemented] Parallelised execution of non-dependent transactions	We propose that we could parallelise grouped transactions within a request on the composite service, given that some transactions are non-dependent. For an example of a grouped transaction, when calling '/book', there are two non-dependent transactions: 'redeem-points' and 'make-booking'. The composite can execute each discrete task on separate threads to asynchronously execute each action.	Baseline: Serialised execution of grouped transactions Test: Parallelised execution of grouped transactions

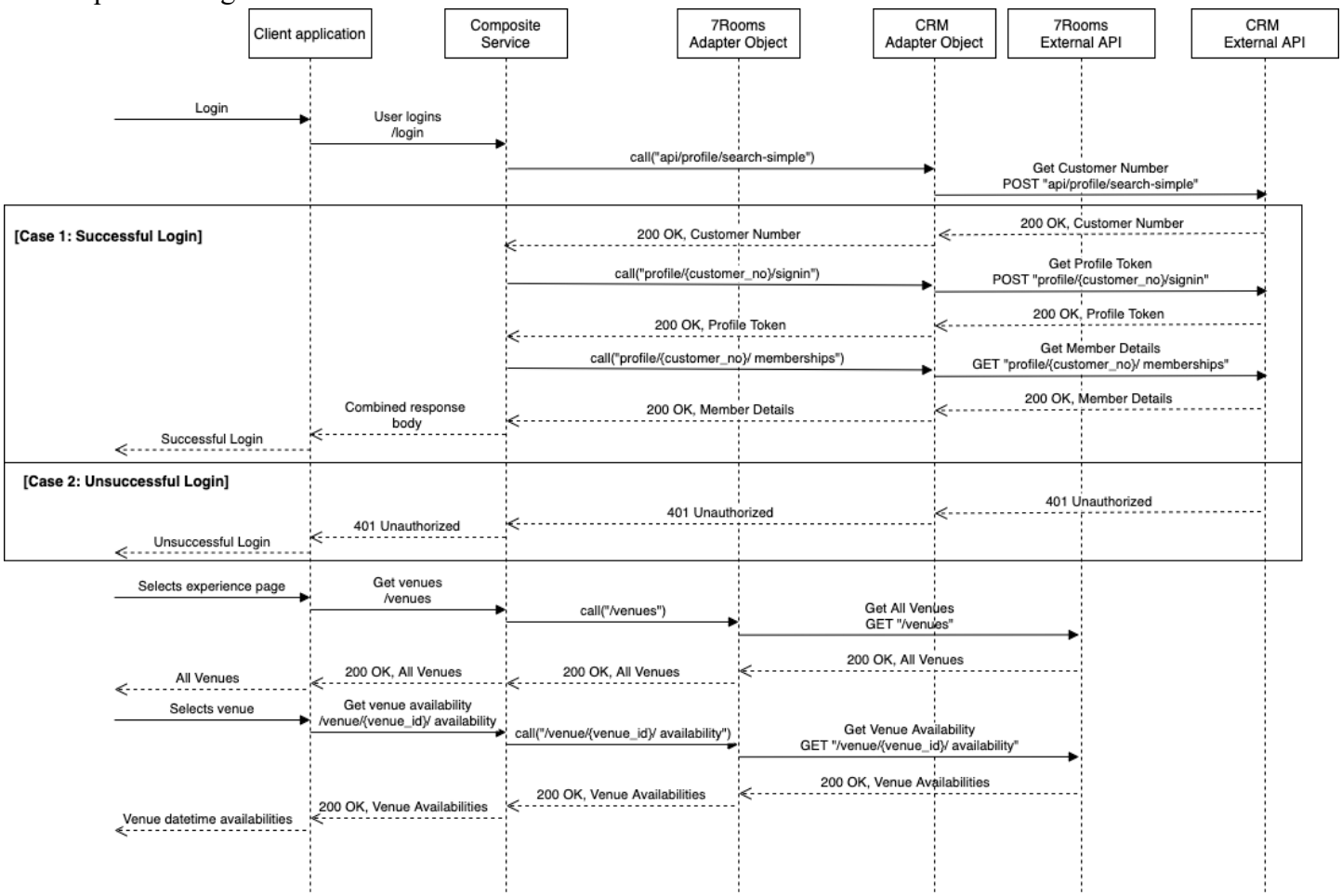
Performance tests were conducted to compare the performance between sending requests to our middleware versus sending requests to the external services directly. See appendix 2.1 for results.

Appendix

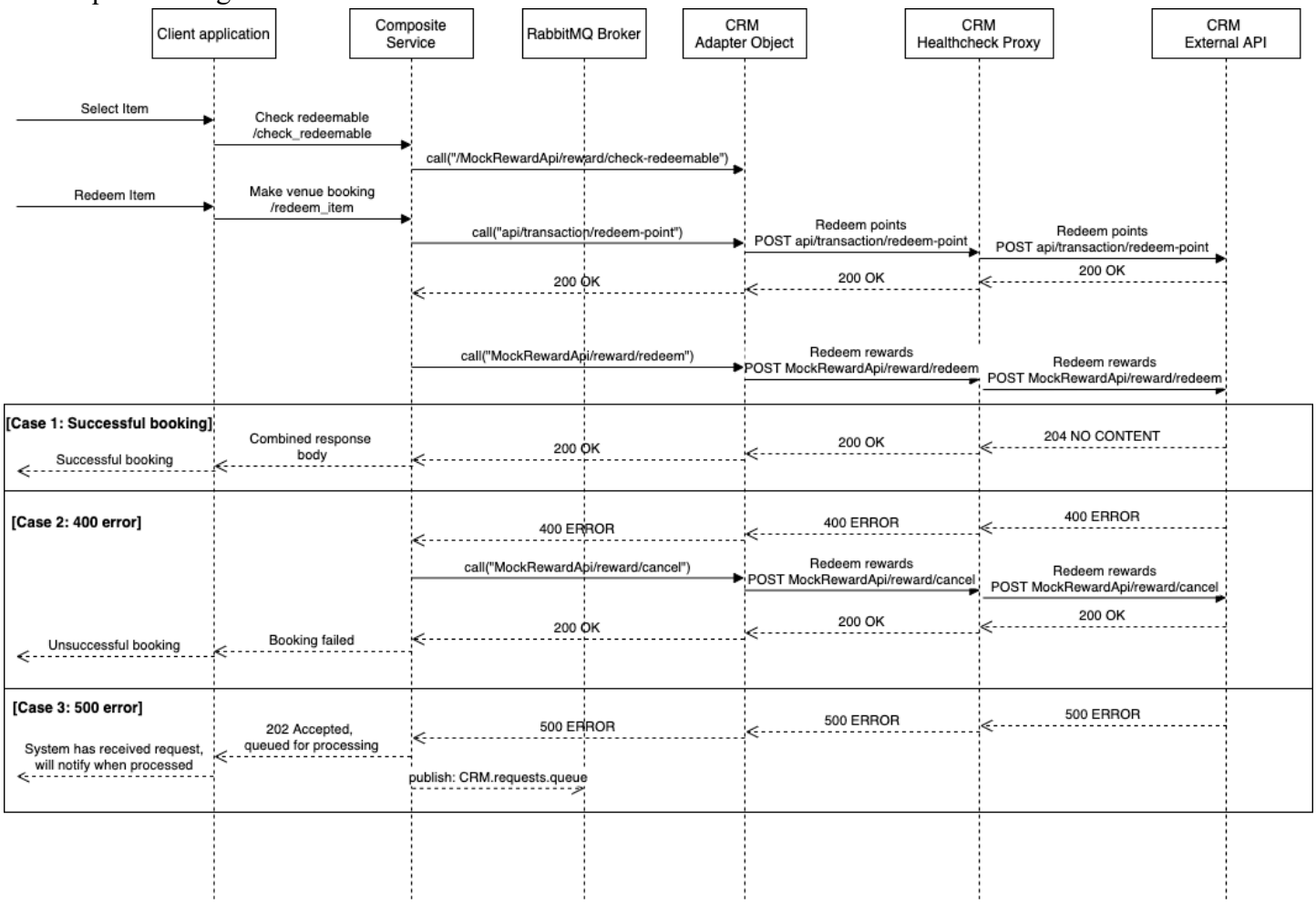
1.1. Sequence Diagram - Use Case 1 (Refer to Availability View for failover scenario)



1.2. Sequence Diagram - Use Case 2



1.3. Sequence Diagram - Use Case 3



2.1. Performance Tests

The average latency for different actions is measured against two configurations:

- calling the middleware service with 2 service instances per microservice; and
- calling the external services directly.

Functional requirements are matched wherever possible. If a functional requirement comprises multiple external services, the overall latency for the middleware is compared against the sum of latencies for each external service called.

2.1.1. Test 1: Authentication with Memberson CRM

Test configuration	**middleware (ms)**	**direct (ms)**
30 threads, 1 sec ramp-up, 3 loops	3306 *(736%)*	449
90 threads, 1 sec ramp-up, 1 loop	5981 *(774%)*	773

Middleware, **30** threads, 1 sec ramp-up, loop count of **3 = 3306ms:**

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec
Auth (Composite)	90	3306	1865	4534	608.99	0.00%	7.4/sec	1.77
TOTAL	90	3306	1865	4534	608.99	0.00%	7.4/sec	1.77

Middleware, **90** threads, 1 sec ramp-up, loop count of **1** = **5981ms**:

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec
Auth (Composite)	90	5981	1916	10570	2483.11	0.00%	7.8/sec	1.86
TOTAL	90	5981	1916	10570	2483.11	0.00%	7.8/sec	1.86

Memberson CRM directly, **30** threads, 1 sec ramp-up, loop count of **3** = **449ms**:

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec
CRM Auth (Direct)	90	449	33	1390	396.41	0.00%	35.5/sec	11.09
TOTAL	90	449	33	1390	396.41	0.00%	35.5/sec	11.09

Memberson CRM directly, **90** threads, 1 sec ramp-up, loop count of **1** = **773ms**:

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec
CRM Auth (Direct)	90	773	74	1756	502.51	0.00%	43.7/sec	13.67
TOTAL	90	773	74	1756	502.51	0.00%	43.7/sec	13.67

2.1.2. Test 2: Cancel a reservation with 7Rooms through our middleware vs directly

Test configuration	**middleware (ms)**	**direct (ms)**
30 threads, 1 sec ramp-up, 3 loops	1977 *(163%)*	1210
90 threads, 1 sec ramp-up, 1 loop	4341 *(238%)*	1822

Middleware, **30** threads, 1 sec ramp-up, loop count of **3** = **1977ms**:

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received ...	Sent KB/sec	Avg. Bytes
Cancel Re...	90	1977	1068	2474	350.95	0.00%	11.9/sec	5.83	3.68	503.0
TOTAL	90	1977	1068	2474	350.95	0.00%	11.9/sec	5.83	3.68	503.0

Middleware, 90 threads, 1 sec ramp-up, loop count of **1** = **4341ms**:

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Cancel Reservations ...	90	4341	1585	7138	1513.98	0.00%	11.1/sec	5.44	3.44	503.0
TOTAL	90	4341	1585	7138	1513.98	0.00%	11.1/sec	5.44	3.44	503.0

7Rooms directly, **30** threads, 1 sec ramp-up, loop count of **3** = 703 + 507 = **1210ms:**
(Cancellation requires two calls)

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB...	Avg. Byt...
Get Auth	90	703	494	1232	282.08	0.00%	22.3/sec	8.32	13.77	382.0
Cancel Reservations	90	507	491	580	17.95	0.00%	25.7/sec	10.90	11.35	434.0

7Rooms directly, **90** threads, 1 sec ramp-up, loop count of **1** = 1196 + 626 = **1822ms:**
(Cancellation requires two calls)

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Get Auth	90	1196	1016	1422	100.45	0.00%	40.7/sec	15.20	25.15	382.0
Cancel Reservations	90	626	499	704	65.24	0.00%	54.0/sec	22.88	23.83	434.0

2.2. ECS service auto scaling

CPU utilisation spikes to a simulated high-threshold of >5% on our composite Fargate instances

New composite Fargate instances spawned on ECS after CloudWatch alarm is alerted from the CPU spike

Task	Task definition ...	Container insta...	Last status
2c72f614a9a54...	composite:33	--	ACTIVATING
5605d1381c6a4...	composite:33	--	RUNNING
84e80d71fac34...	composite:33	--	ACTIVATING
dbcd0cb3f3304...	composite:33	--	RUNNING

3. Additional steps on top of our base CloudFormation infrastructure stack

Visit the repository containing the README and CloudFormation stacks on GitHub: https://github.com/cs301-itsa/project-2021-22t1-g2-g2team1-infrastructure. Start by creating an S3 bucket named cf-templates-$AWS_REGION-$AWS_ACCOUNT_ID and upload all templates into it. Create the stack on CloudFormation by using the main.yml template. Note, this setup is for development purposes, not production.

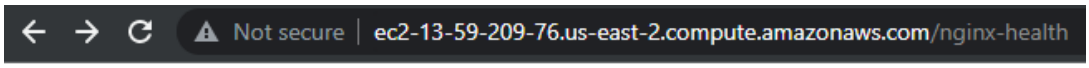
3.1 Configure NGINX reverse proxies on both availability zones

1. SSH into the COMO-ReverseProxyA with ssh ubuntu@<reverse-proxy-a's ip> and sudo vim /etc/nginx/sites-enabled/default and replace with the following:

```
server {
        listen 80 default_server;
        listen [::]:80 default_server;

        root /var/www/html;

        server_name _;

        location / {
```

```
                # First attempt to serve request as file, then
                # as directory, then fall back to displaying a 404.
                # try_files $uri $uri/ =404;
                proxy_pass      http://internal-COMO-ALBIn-18CU0587D0IY2-977318805.us-east-2.elb.amazonaws.com;
        }

        location /nginx-health {
                # access_log off;
                return 200 "reverse proxy 1 healthy\n";
                add_header Content-Type text/plain;
        }

        location /ws/ {
                proxy_pass      http://internal-COMO-ALBIn-18CU0587D0IY2-977318805.us-east-
2.elb.amazonaws.com/amqp-ws/;
                proxy_http_version 1.1;
                proxy_set_header Upgrade $http_upgrade;
                proxy_set_header Connection "Upgrade";
                proxy_set_header Host $host;
        }
}
```

2. Visit http://<ec2-reverse-proxy-a-ip>.us-east-2.compute.amazonaws.com to verify nginx works and that your modification appears:




```
reverse proxy 1 healthy
```

3. Repeat steps for COMO-ReverseProxyB.

3.2 Configure the composite microservice

1. Create ECS Task Definition for each microservice
    a. Fargate instance type
    b. Task definition name: composite
    c. Task role: ecsTaskExecutionRole-COMO
    d. Task execution role: ecsTaskExecutionRole-COMO
    e. Task memory: 0.5GB
    f. Task CPU: 0.25 vCPU
    g. Container
        i. Name: composite
        ii. Image: amazon/amazon-ecs-sample (this is a “placeholder” php app, will be overwritten when doing CI/CD)
        iii. Port mappings: 80
        iv. Environment files: s3://environment-bucket-us-east-2-<unique id generated by cloudformation>/production.env
        v. Environment variables:
            1. PYTHONUNBUFFERED: 1 (for python services)
        vi. Log configuration, disable auto-configure CloudWatch Logs:
            1. Log driver: awsfirelens
            2. Name: cloudwatch
            3. log_group_name: ecs-como-log-group
            4. log_stream_prefix: from-fluent-bit-
            5. log_key: log
            6. region: us-east-2
            7. auto_create_group: true
    h. A new container definition called log_router appears above, click on it

        i. Log configuration, disable auto-configure CloudWatch Logs:
            1. Log driver: awslogs
            2. awslogs-group: firelens-container
            3. awslogs-region: us-east-2
            4. awslogs-stream-prefix: firelens
            5. awslogs-create-group: true
    i. Create
2. Create ECS service in COMO-cluster
    a. Name: composite
    b. Launch type: FARGATE
    c. Task definition: composite, revision 1 (latest)
    d. Service name: composite
    e. Number of tasks: 2
    f. Min healthy percent: 100
    g. Max percent: 200
    h. Deployment circuit breaker: Disabled
    i. Deployment type: default, rolling update
    j. Task tagging configuration: default
    k. Cluster VPC: COMO-vpc
    l. Subnets: COMO-SubnetPrivA and COMO-SubnetPrivB
    m. Security groups, new:
        i. Minimally allow TCP port 80 from internal ALB, adjust according to the business logic of this microservice
    n. Auto-assign public IP: Disabled
    o. Load balancer type: ALB
    p. Load balancer name: COMO-ALBIn
    q. Container name : port: composite:80:80, Add to load balancer
        i. Production listener port: 80:HTTP
        ii. Target group name: create new, default
        iii. Target group protocol: HTTP
        iv. Target type: ip
        v. Path pattern: /composite*
        vi. Evaluation order: 1-100, lower, the higher the priority, use the next lowest priority, if there are already existing rules
        vii. Health: /composite/health
    r. Enable service discovery integration
        i. Create new private namespace, or select existing namespace “como”
        ii. Create new service discovery service
        iii. Enable ECS task health propagation
        iv. DNS record type: A
        v. TTL: 300 seconds
    s. Configure Service Auto Scaling to adjust your service’s desired count:
        i. Minimum number of tasks: 2
        ii. Desired number of tasks: 2
        iii. Maximum number of tasks: 4
        iv. IAM role for Service Auto Scaling: ecsAutoscaleRole
        v. Scaling policy type: Target tracking
        vi. Policy name: cpu60
        vii. ECS service metric: ECSServiceAverageCPUUtilization

viii. Target value: 60
ix. Scale-out cooldown period: 150
x. Scale-in cooldown period: 300
xi. Disable scale-in: Unchecked

t. Create

3. Create ECR repository, /composite
    a. Visibility settings: Private
4. On GitHub, go to the composite microservice's repository, Settings, Secrets, and ensure your AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY are added.
5. Make a commit to the main branch of the composite repository to trigger a workflow run. The microservice will be automatically deployed onto ECS.

Repeat 3.2 for the other microservices: healthcheck-proxy (crm-adapter, 7rooms-adapter [two microservices deployed from a single repository's workflow]) and amqp-ws.