CS301 Team3 Final Report
GitHub Team name: cs301-team 3
Team members (name and matriculation): Carmen (01386295), Jie Mi (01392286), Jiong Yu (01389061), Jolene Loh (01392238), Lindy (01387183), Nikki (01390645)

CONTENT

1. Background and Business Needs

COMOGroup's Como Club's mission is to create a holistic, integrated lifestyle offering with curated experiences that recognise and reflect their members above all else.

COMO Group's vast system and infrastructure span across disparate and loosely-integrated Business Units (BUs). The systems are siloed across all BUs without middleware to manage system integration. This project aims to create a solution architecture for the middleware to manage the system integration, with beneficial software patterns and architecture styles that achieves maintainability, availability, security and performance qualities.

Como Club's drivers include membership signups, products sold, points redeemed and bookings.

2. Stakeholders

Stakeholder	Stakeholder Description	Permissions
Member	- Earn points - Book experiences - Redeem items using points - Pay with card - Change details like password and payment type	- View experiences/items, - Create bookings, - redeem items, - view wallet, - log in, - update password and payment type
Guests	- Become a member - Browse experiences/items	- Register for member (create account) - View experiences - View items
Merchants	- Merchants offering services/items to members	- Connect with COMOClub to create, update and delete experiences

3. Key Use Cases

Use Case Title - Experience Booking for 7 Rooms	
Use Case ID	1
Description	Experience Booking allows members to book and access a wide array of services.
Actors	Members, Merchants, 7rooms, MembersonAPI, COMO Credit
Main Flow of events	1. Member to select the experience to book 2. Member to indicate the date/time based on availability via the 7 rooms booking engine 3. Member will pay with their points 4. MembersonAPI to check whether member has enough points to allow for successful booking 5. Member to review and confirm the booking details 6. Once confirmed, the acknowledgement will be shown to the Member
Alternative Flow of events	1. Member to select the experience to book 2. Member to indicate the date/time based on availability via the 7 rooms booking engine and pay using points 3. Member will pay with their points

	4. MembersonAPI to check whether member has enough points to allow for successful booking 5. Member has insufficient points
Pre -conditions	1. The customer has to be a COMOclub member 2. Has the COMO Club mobile application downloaded
Post -conditions	1. Member able to book the experience through COMO Club 2. If paid using COMO Credit, accounts and transactions will be balanced 3. If booking fails, member will not be able to book the experience through COMO Club and no transactions will take place

Use Case Title - Item Redemption	
Use Case ID	2
Description	Item redemption allows members to redeem items using COMO credit
Actors	Members, Merchants, MembersonAPI, COMO Credit
Main Flow of events	1. Member to select the item for redemption 2. Member to select the quantity and pay using points 3. Member will pay with their points 4. MembersonAPI to check whether member has enough points to allow for successful redemption 5. Member to review and confirm the redemption details 6. Once confirmed, the acknowledgement will be shown to the member
Alternative Flow of events	1. Member to select the item for redemption 2. Member to select the quantity 3. Member will pay with their points 4. MembersonAPI to check whether member has enough points to allow for successful redemption 5. Member has insufficient points
Pre -conditions	1. The customer has to be a COMOclub member 2. Has the COMO Club mobile application downloaded
Post -conditions	1. Member able to redeem the item through COMO Club 2. If paid using COMO Credit, accounts and transactions will be balanced 3. If redemption fails, member will not be able to redeem the item through COMO Club and no transactions will take place

Use Case Title - Creating a new membership account	
Use Case ID	3
Description	Being a COMO Club member allows app users to enjoy exclusive deals to redeem items specially curated for them.
Actors	Guests, MembersonAPI
Main Flow of events	1. User decides to become a member 2. User clicks the Register button from the login page. 3. User will input their personal information and register 4. User has become a member
Alternative Flow of events	1. User decides to become a member 2. User clicks the Register button from the login page 3. User will input their personal information 4. User enters a password that is too short. 5. User sees an error on the screen and inputs another password 6. User will input their personal information and register 7. User has become a member
Pre -conditions	1. Download the COMO Club mobile application or visit the COMO club website
Post -conditions	1. MembersonAPI adds a new member account 2. Member can log in to their account

4. Quality Requirements

Software Product Quality	How does our solution architecture achieve these requirements?
Performance	The application will respond to requests in under 5 seconds. API requests will be served in under 2 seconds. These qualities will be achieved with the use of AWS. Load balancing will be done across multiple availability zones/regions (closest to where most of COMO Club's customers and BUs reside) to provide lower latency and a better experience for customers. Caching will be done to increase response times. The following information will be cached: 1) User's session information (for cross-region replication in the event of a failover) 2) User's membership details (for quicker reference) 3) Experience and item listings from Memberson (for quicker reference, cache shared across users, updated when user makes a booking / redemption.
Availability	Ensured with the help of redundant components. Redundant components deployed across regions and availability zones allows users to connect to redundancies, if main servers and availability zones become unavailable. We also implemented load balancers on AWS, which ensures that if 1 instance goes down, there would be a back up ready.
Security	Utilisation of security groups to control inbound and outbound traffic to instances receiving/processing requests from app and APIs. Data would be encrypted in transit between: 1) Public subnets ←→ Private subnet (with AWS encryption solutions, default AWS services security controls) 2) Public subnets ←→ App (with SSL)
Maintainability	Use a CI pipeline to allow one click deployments. Deployments to be checked against automated test cases and health checks to ensure the system runs as expected and buggy changes do not affect the technology value stream. Allowing the application to add and deploy new features while keeping maintenance at low costs. Three different instances handling app communication, API requests and retrieval, and processing logic allow for ease of integration with any new APIs / businesses.

5. Key Architectural Decisions

Architectural Decision 1 - Instance Set Up	
ID	1
Issue	Security of connecting the service that receives the user's interface information and the service that calls the external APIs.
Architectural Decision	Setting up the Receive API service and Receive App service in public subnets with access to the internet through NAT gateways and an internet gateway, and a Process App service in a private subnet to connect the other two services. This allows us to protect the more sensitive service (Process App) within a more secure layer.
Assumptions	We assume that the application will have a lot of traffic from customers, thus requiring an instance type that accommodates that nature the best.
Alternatives	There are multiple instance types in AWS that can be used with alternatives like R3, M3, C3, etc. instance types. However, they are all geared towards different purposes. M3 for example is suitable for a wide range of applications from database to servers, while C3 is used for intensive computation in web servers, gaming and analytics.
Justification	We used the t2.large as the instance type due to its Burstable Performance Instances that provide a baseline level of CPU performance with the ability to burst above the baseline during spikes in demand which is more suited in comparison to the other instance types. This is because of the nature of our application (used for leisure purposes, so after normal working hours and during holidays), so workloads can be more spikey in nature.

Architectural Decision 2 - Load Balancing	
ID	2
Issue	Improving application availability and performance
Architectural Decision	We load balance our instances within the same VPC using application load balancers and also outside the VPC and across regions with the help of AWS' Route53. This allows us to distribute the workload/application traffic, improving availability and latency.

	Also allows the system to be available across multiple availability zones and regions.
Assumptions	We assume that the application by the company will be globally available and thus require an aws load balancing system that will manage the traffic from multiple regions and zones.
Alternatives	The alternative would have been clustering in Elastic Container Service which combines the multiple servers to function as a single entity/resource.
Justification	With elastic load balancing (ELB) we could add and remove EC2 instances as our needs change without disrupting overall flow of communication. Load balancing can be more simple to deploy in established environments with different types of servers because clustering usually requires identical servers within the cluster.

Architectural Decision 3 - Caching and Scaling	
ID	3
Issue	Improving application efficiency & reduce resource wastage
Architectural Decision	S3 Buckets were used to cache commonly used information to reduce the number of repeated calls. We also used auto scaling to provision and tear down instances as required, to dynamically increase/reduce capacity in response to current traffic demand.
Assumptions	Constant variation in demand/traffic, e.g. higher user traffic during peak periods like holidays.
Alternatives	The alternative to cache would have been Redis or Memcached in Amazon ElastiCache; both provide a similar service of an in-memory key/value store. They are both designed to provide O(1) performance for all GET and SET operations. What this means in practice is that GET and SET calls maintain a constant speed as the cache size grows.
Justification	However, the alternatives can be inconsistent to caching, and have constant evictions if there is not enough memory. For example, Because Memcache uses an LRU algorithm and the Drupal Memcache module, by design, doesn't actually flush out data, some evictions are to be expected. Thus, S3 Bucket is far better due to its consistency in its reliable and durable data storage.

6. Development View

ReceiveAppRequests (RAR)

Main focus: Processes UI interaction, renders UI, formats frontend requests into JSON, forwards frontend requests

RecieveAppRequests listens to the user interaction with the user interface of the application and communicates to the backend the data that has to be sent to ProcessAppRequests. We chose to follow the Spring Boot web framework for the application because it lets us create standalone applications that run on their own, without relying on an external web server. Furthermore, it provides production-ready features such as metrics, health checks, and externalized configuration and automatically configures Spring and 3rd party libraries whenever possible. This framework is done similarly for ProcessAppRequests and ReceiveApiRequests.

RecieveAppRequests focuses on tackling our 3 use cases: Experience Booking, Item Redemption and Member Registration. Each use case has its own controller, service and object respectively. The controllers help to manage the data received from the user interface. The data collected is directed to the RARListener controller which handles forwarding frontend JSON information to ProcessAppRequests.

Outgoing request parameters: identifier (associated with a particular request), action (HTTP method), URI (end of the API endpoint), data (body of request bound for API, may be null).

ProcessAppRequests (PAR)

Main focus: Processes requests, handles cache storage, determines next route for requests

ProcessAppRequests is the main logic centre of our entire architecture, and is in charge of handling the communications between RecieveAppRequests and ReceiveApiRequests. This includes determining where ReceiveAPIRequests should be forwarding the requests to; and mapping out the various requests as required in our use cases.

Essentially, the JSON information that is collected will be processed for its respective purpose as defined in the “identifier” (e.g. confirmRedeemItem) and “action” (i.e. POST, GET) attributes in the JSON. After identifying the purpose, it will send the JSON received to the corresponding apis or user interfaces. Key-value pairs stored include identifiers and vendors, vendors and base URLs. There may be variations of vendors, eg. two different Memberson base URLs were used, /api (mapped to key “Memberson”) and /MockRewardApi (key “MembersonMock”). Cache keys are also stored here. For this implementation, we stored the list of items and venues returned from the API.

ProcessAppRequests also has conditional checks for certain cases, such as when the request is to redeem an item (a points balance check is to be carried out), or when the request is to set a user password in Memberson (the additional profile token header is required, this would be added to the outgoing request).

Outgoing request parameters: action (HTTP method), URL (full API endpoint for a specific request), vendor (string, used by ReceiveAPIRequests to format API calls correctly), data (body of request bound for API, may be null), profile token (if needed).

ReceiveApiRequests (RAPI)

Main focus: Communicates with the 2 APIs: MembersonApi and 7RoomsApi. Sends and receives data from API to send back to ProcessAppRequests

The job of RecieveApiRequests is to communicate with external APIs such as MembersonApi and 7RoomsApi to perform actions such as Bookings and Item Redemptions. ReceiveAPIRequest will convert the JSON request forwarded from ProcessAppRequest into the format expected by the various external APIs. It then sends the requests to the external APIs, receives the responses and forwards back to ProcessAppRequests.

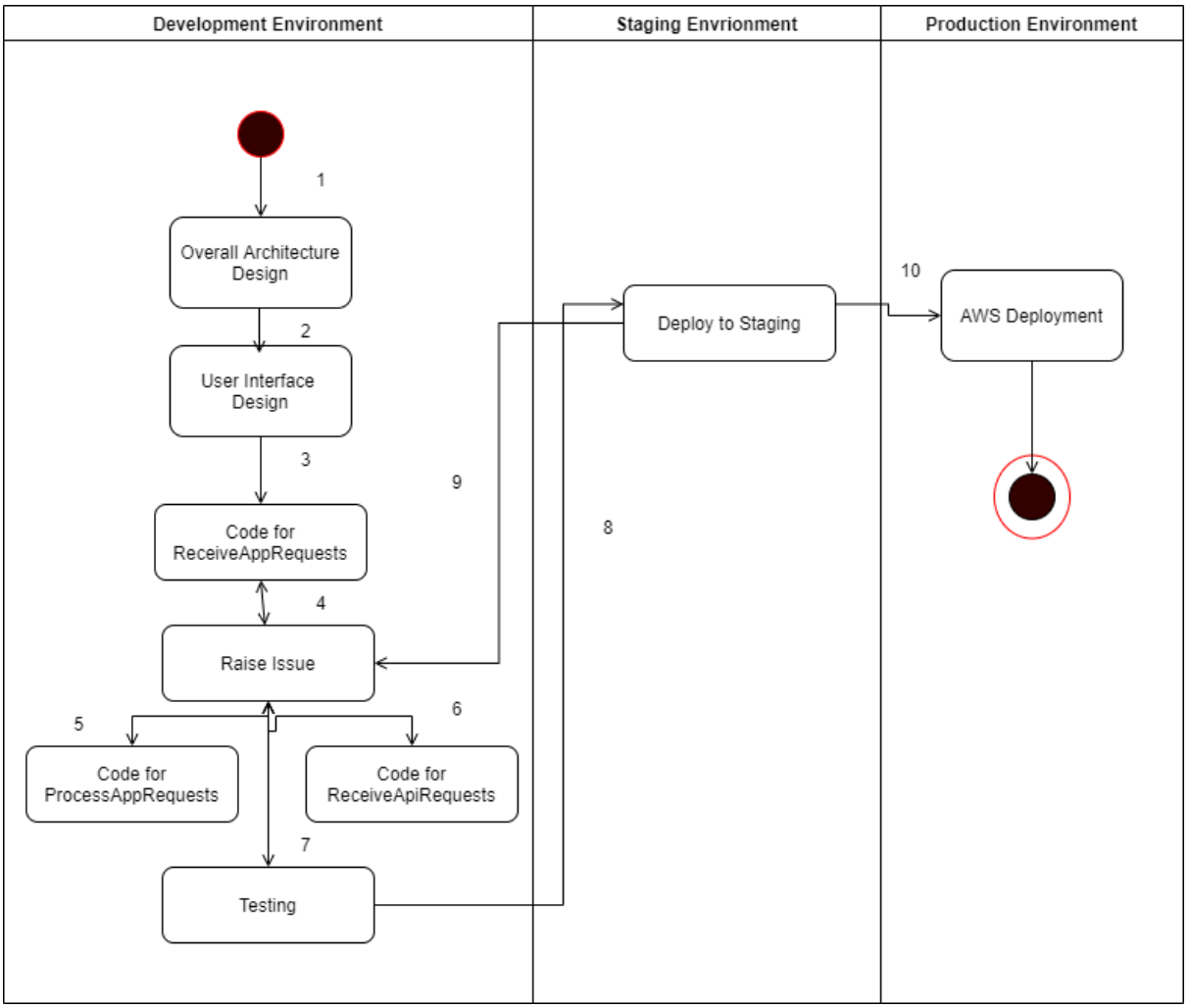
Outgoing requests are API calls and are accompanied by HTTP headers (eg. SvcAuth) and the request body.

Extending the Application

Integrating with a new API:

a) New methods would be written in ReceiveAppRequests for any new functionality.
b) These methods would be keys in ProcessAppRequests's configuration file, mapped to the vendor of the new API. The vendor (and its variations) would be added to another map, with corresponding base URLs.
c) Include any cache keys for responses that will be cached.
d) Any specific cases for certain requests would be written to be handled in ProcessAppRequests with any accompanying helper methods.
e) New request methods would be written in ReceiveAPIRequests according to the format required by the new API (eg. authorization headers).

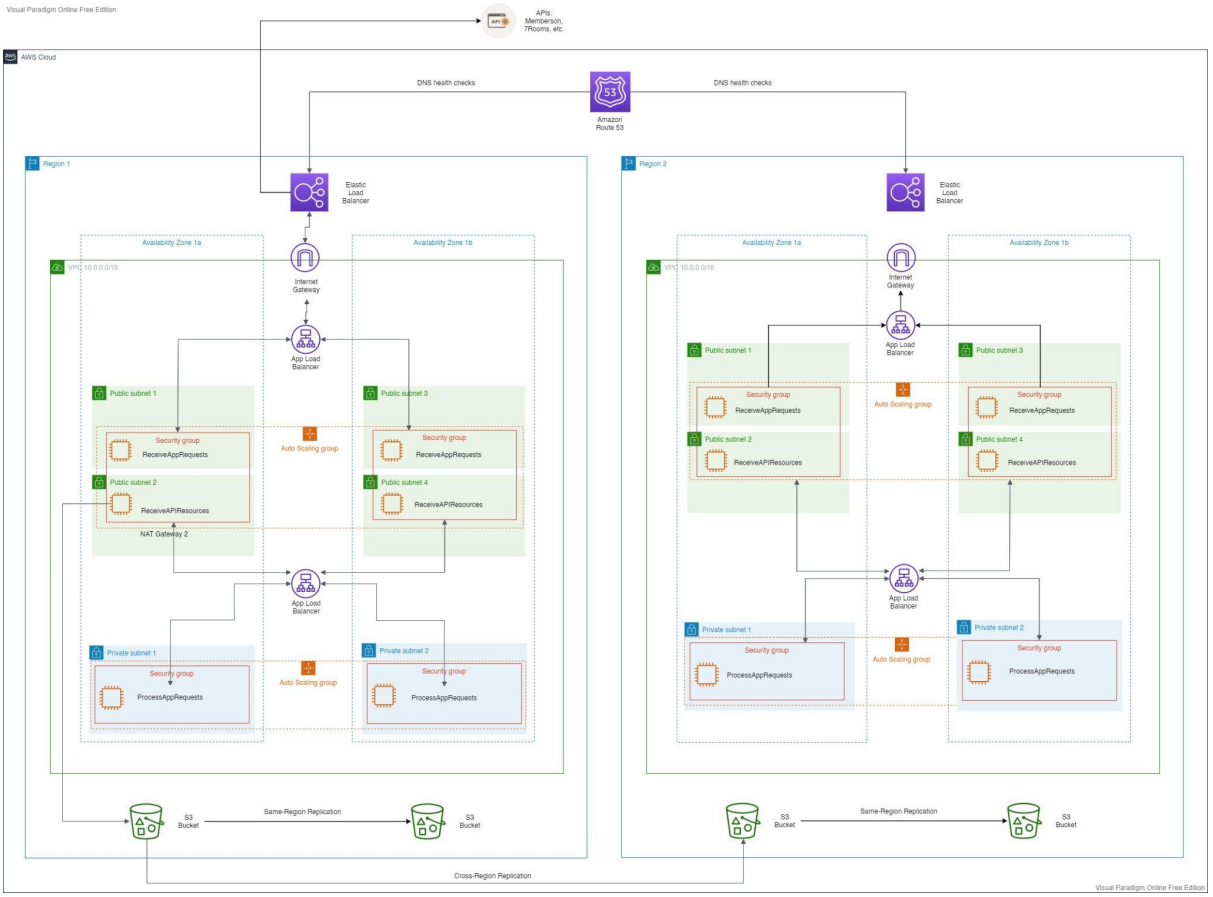
Activity Diagram



1. Developers/Architects design the overall architecture design (1)
2. Developers/UI/UX design the User Interface design (2)
3. Developers implement the features/code in the first service, ReceiveAppRequests (3)
4. Developers test the code and resolve any issues found (7)
5. Developers implement the code for ProcessAppRequest (5)
6. Developers test the code, including its integration with ProcessAppRequest (7)
7. Developers implement the code for ReceiveAPIRequests (6)
8. Developers test the code, including its integration with the other two services (7)
9. Developers deploy to staging environment(8) and if there are issues, fix them
10. Developers deploy to AWS.

7. Solution View (Maintainability)

AWS Diagram: Our overall Solution Architecture



We aimed to achieve maintainability via CI/CD through GitLab. With the GitHub repository, it can easily be configured to also point to a repository in GitLab. Under GitLab, we would be able to achieve CI through .gitlab-ci.yml. By configuring a build and release stage, every time a commit is made the code would be pushed through the pipelines and be containerized in the end. We can also add in a test stage to run maven integration tests for more automation.

CD could be achieved through setting up a deploy stage in .gitlab-ci.yml and configuration of AWS ECS. During CI, a docker container image is created. By pulling this image from ECS's task definition, we can easily upkeep the automation of deployment of our code via service tasks.

Source System	Destination System	Protocol	Format	Communication Mode
User Interface	ReceiveAppRequests	HTTP	HTML	Synchronous
ReceiveAppRequests	ProcessAppRequests	HTTP	JSON	Synchronous
ProcessAppRequests	Cache	HTTP	JSON	Synchronous

			(file)	
ProcessAppRequests	ReceiveApiRequests	HTTP	JSON	Synchronous
ReceiveApiRequests	MembersonAPI	HTTPS (external API)	JSON	Synchronous
ReceiveApiRequests	7RoomsAPI	HTTPS (external API)	JSON	Synchronous
ReceiveApiRequests	RewardsAPI	HTTPS (external API)	JSON	Synchronous

8. Proposed Budget

a. Development Budget

Activity / Hardware / Software / Service	**Description**	**Cost**

Working hours	- Time spent implementing:
	  - New Software: key features of the app
	  - Software Modification
	  - Software Integration
	- Creating Middleware to connect Front-end and Back-end of the COMOclub Application
	- Designing Solution Architecture
		6 weeks of labour by a team of 6. Duties consists of:
		- Project Manager
		- UI/UX
		- Architect
		- Developer
		- QA Tester

Amazon Web Services	- Services used to host and run the middleware application in a development environment	Generally $400 total for AWS
Designing Solution Architecture	- Designing, describing, and managing the solution engineering in relation to specific business problems	2 hours x 4 weeks x 6 students

Developing Security	- Using SSL to communicate with AWS resources - Time spent implementing TLS	1 hours x 4 weeks x 6 students
Developing CI	- Time spent creating and testing - Time spent iterating	2 hours x 4 weeks x 6 students
Developing CD	- Time spent developing & deploying	3 hours x 4 weeks x 6 students

b. Production Budget

Activity / Hardware / Software / Service	**Description**	**Cost**
Virtual Private Cloud (VPC)	- NAT Gateways for this project were removed - VPC alone has no additional charges	- $0 USD per month
EC2 Instance Saving Plan	- 12 Instances - Instance type - 100% Utilization/month - Storage: 30GB	- $24 USD per month
Elastic Load Balancing	- No. of Application Load Balancers: 4	- $65.70 USD per month
S3 Bucket Caching	- Amazon Simple Storage Service: 2 S3 buckets - S3 Standard storage: 30GB	- $6.60 USD per month

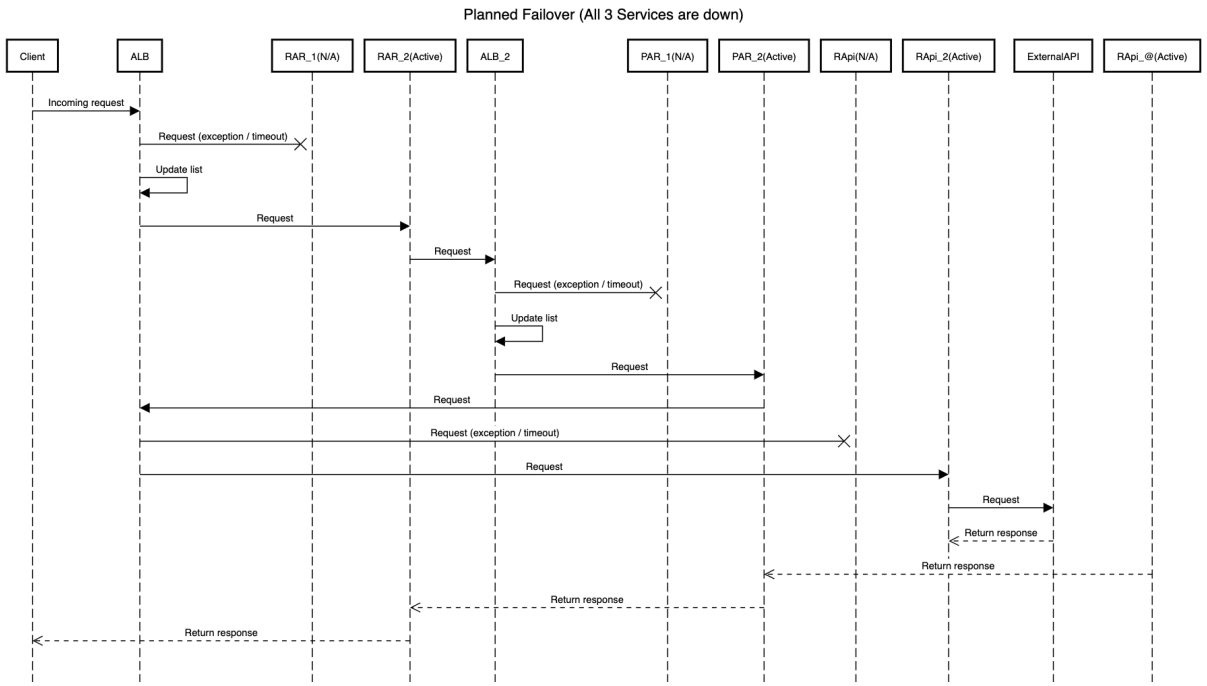
9. Availability View

We configured our system to use application load balancers across availability zones within a VPC. We also make use of Elastic load balancers and Route53 to ensure availability across regions.

Node	Redundancy	Clustering		
		Node config.	Failure detection	Failover
RAR	horizontal	active-passive	ping	load-balancer
PAR	horizontal	active-passive	ping	load-balancer

RAPI	horizontal	active-passive	ping	load-balancer

Sequence Diagram with failover
Scenario: All 3 services are down.



10. Security View

No	Asset/Assets	Potential threat/Vulnerability Pair	Possible Mitigation Controls
1	Software	SQL Injection affects the confidentiality of the system. The account of the users may be compromised if the information is not properly hashed.	We applied the SHA-512 Hash Function to encrypt the passwords.
2	Software	DNS Spoofing/Cache Poisoning, this threatens the availability of our system if one of the instances is under attack. For example, our PAR service is where majority of our code logic is at and thus we need to secure the communication transit and	We modified the settings of the subnets' security groups to control and restrict all inbound and outbound traffic to the services hosted on the EC2 instances.

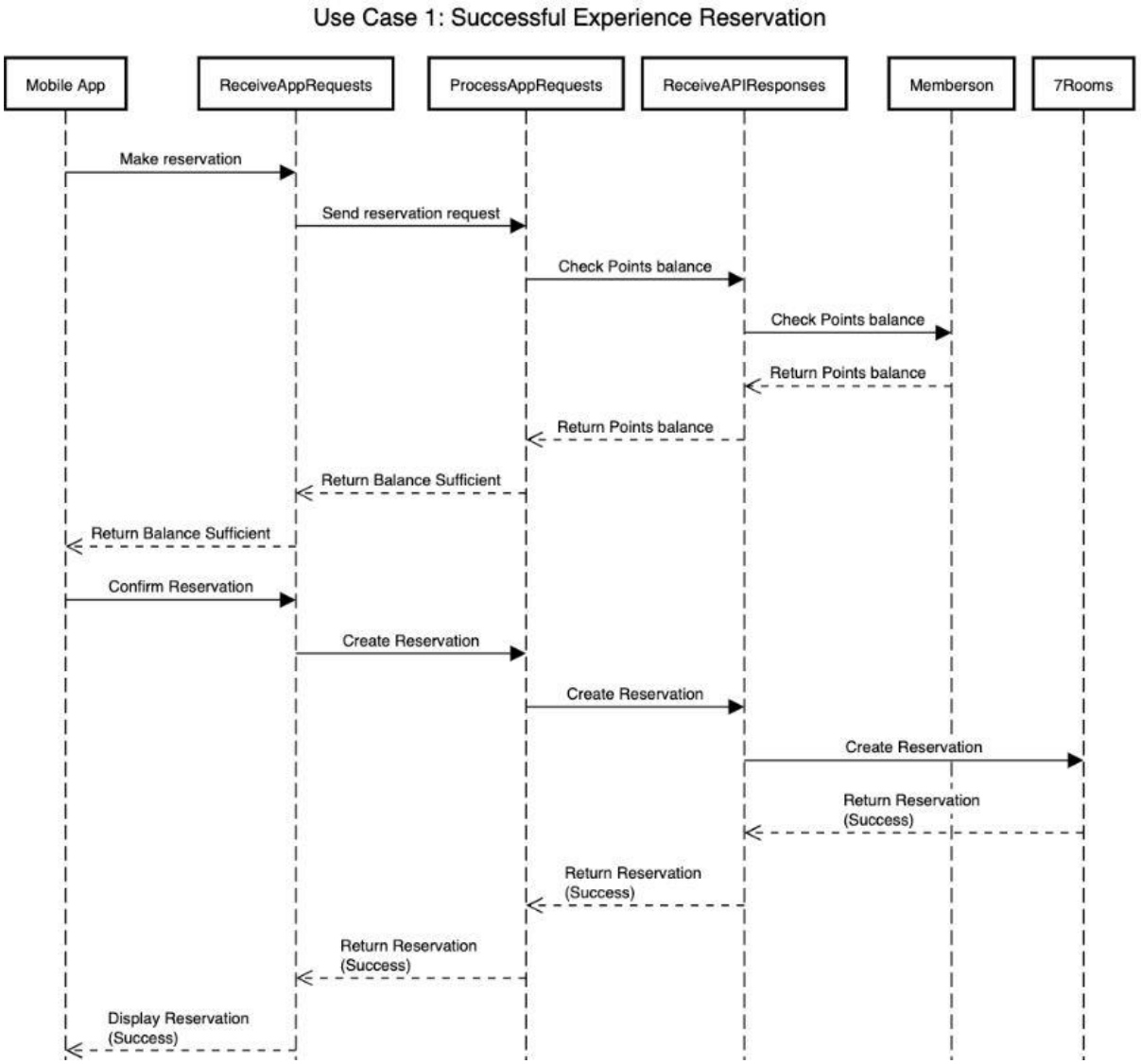
		prevent it from being accessible to the internet to reduce the attack surface	
3	Software	Man in the Middle Attack affects the communication between the user and the web application. This directly affects the integrity of the system. For example, if a user would like to book a venue but is not communicating with the legitimate server.	Digital Signature (not implemented) Transport Layer Security (not implemented)

Refer to the AWS Diagram in the Solution View to see how our security components fit in the architecture.

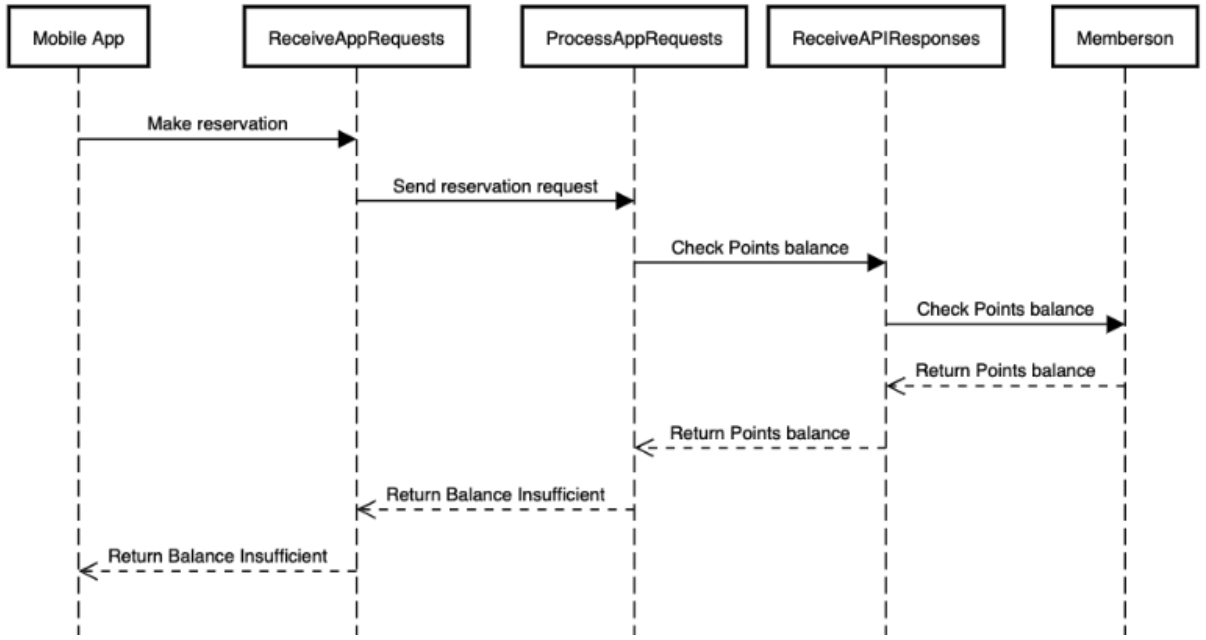
11. Performance Views

No	Description Strategy	Justification
1	Caching	An S3 bucket was used to cache commonly used data. The alternatives such as Memcache can be inconsistent to caching, and have constant evictions if there is not enough memory. For example, because Memcache uses an LRU algorithm and the Drupal Memcache module, by design, doesn't actually flush out data, some evictions are to be expected. Thus, S3 Bucket is far better due to its consistency in its reliable and durable data storage. The S3 bucket also comes with benefits such as a simulated file hierarchy and versioning.
2	Auto-Scaling	It ensures that our instances will always have enough capacity to handle the traffic. It also helps with cost management since the less we use, the lesser we pay. We do not have to constantly pay for a larger capacity as well.

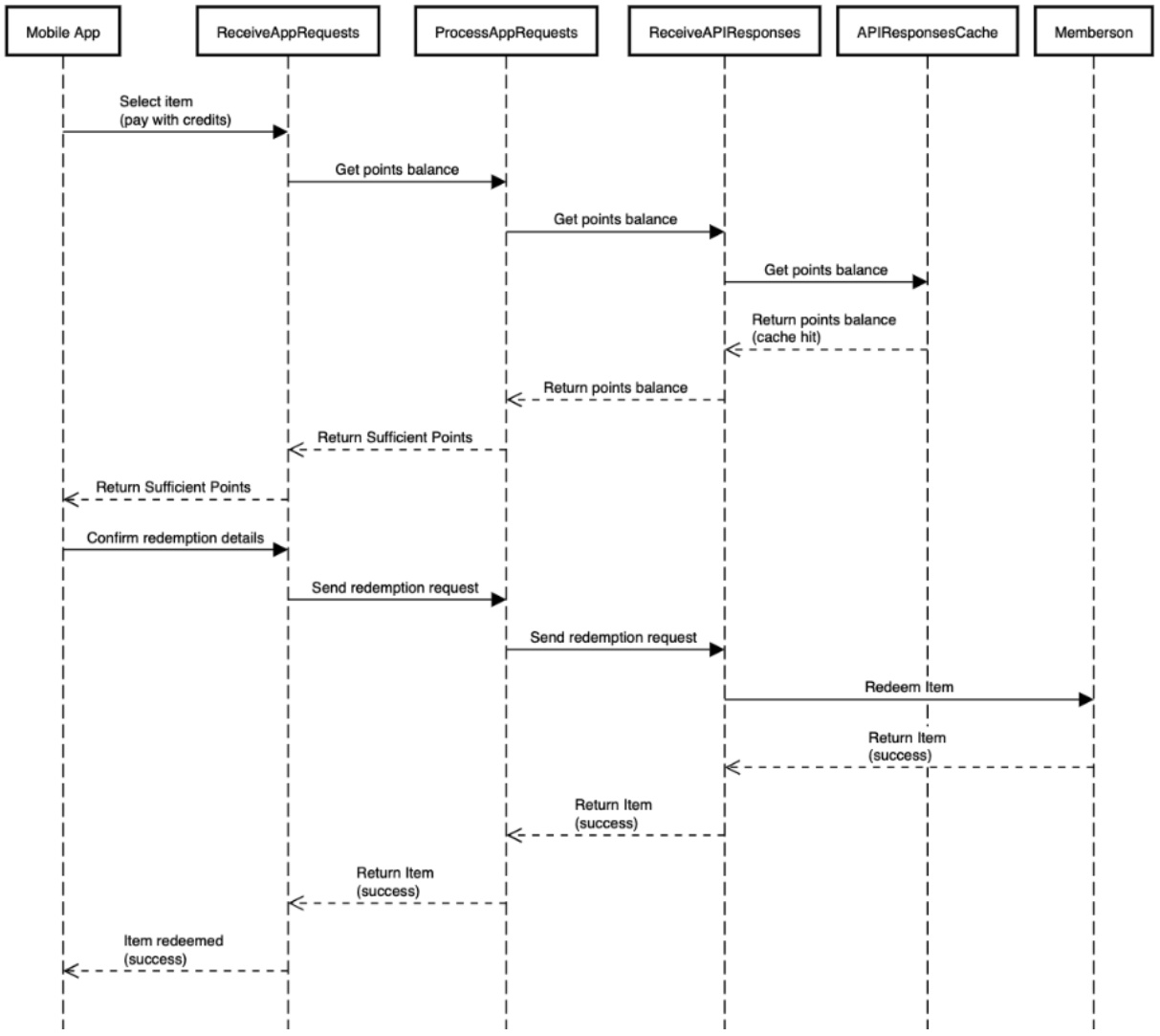
Use Case Sequence Diagrams



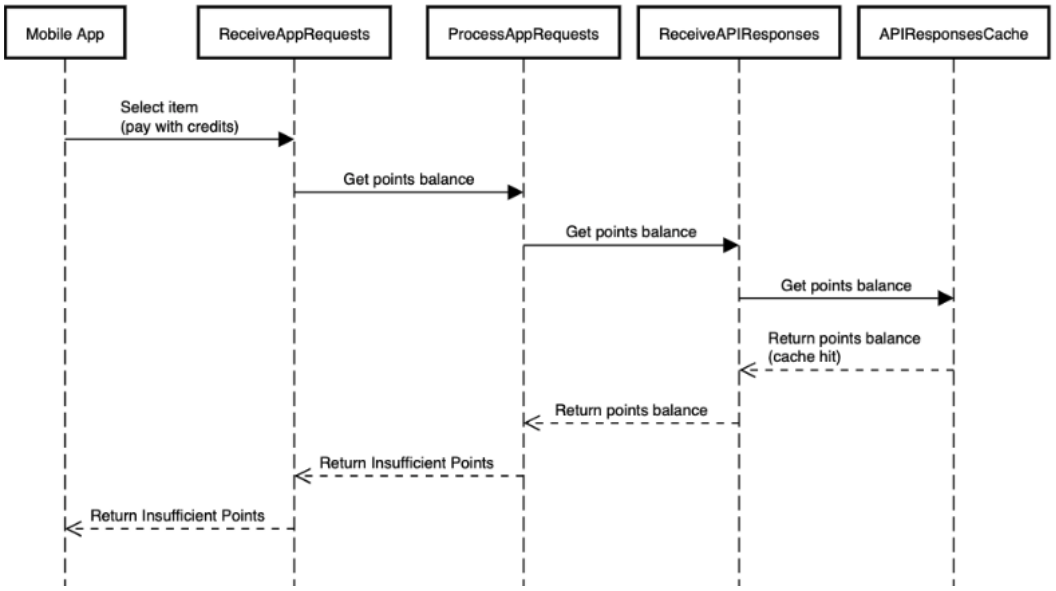
Use Case 1: Unsuccessful Experience Reservation (Insufficient Points)



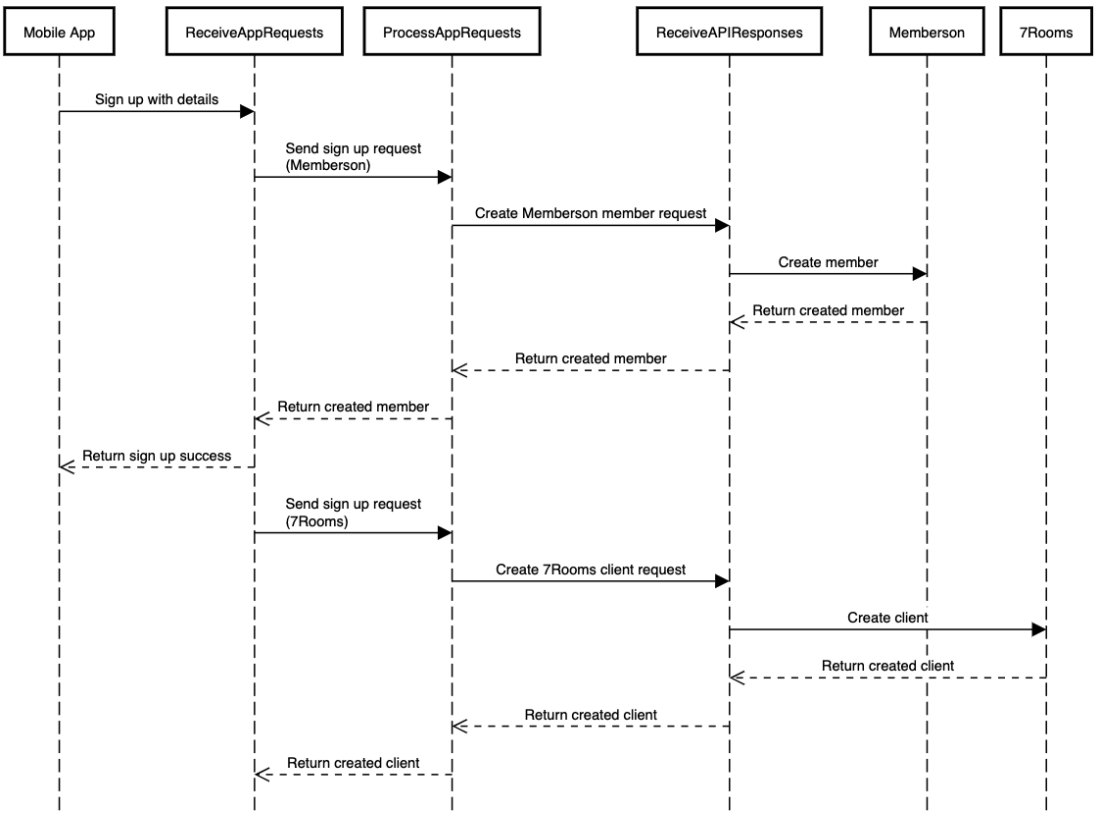
Use Case 2: Item Redemption (Sufficient Points Balance)



Use Case 2: Item Redemption (Insufficient Points Balance)



Use Case 3: Creating New Member



Use Case 3: Creating New Member (Password too short)