CS301 IT Solution Architecture
G1 Team 8

Jasmine Toh Ming Fang - 01373797
Ong Jun Xiang - 01337747
Ng Wen Jie - 01334000
Looi James - 01332928
Chu Wei Hao - 01360083

1. Background & Business Needs

Ascenda enables hotel booking platforms on behalf of partnered hotels, banks, airlines, and loyalty programs, allowing customers to earn & redeem hotel night stays. As an aggregator, Ascenda collects pricing results from many different hotel partners, of which customers will be able to view, earn and redeem hotel night stays on the hotel booking platform.

Ascenda's hotel booking platform is an ecommerce website, hence, **speed** is deemed to be the most important business needs for a pleasant user experience. For Ascenda to remain competitive, the hotel booking platform would have to produce results on the customers' browsers almost instantaneously. Furthermore, during peak periods and especially in post-pandemic times, the hotel booking platforms must be extremely **scalable** to meet the larger spikes in demand. Additionally, Ascenda deploys daily, and no downtime is allowed during deployment, meaning our solution must be **maintainable**. Ascenda also strives to ensure that the payment information, personal information, and services are **secure**.

Hence, our solution will have to consider the tradeoff between speed and accuracy of results to meet the abovementioned Ascenda's business needs.

2. Stakeholders

Stakeholder	Stakeholder Description	Permissions
Business Stakeholders		
Ascenda	Ascenda acts as the middleman, providing the brokerage/bridging service between hotels and end-users	- Read: Global Access - Write: Global Access
Hotels	Ascenda's clients and partners. Ascenda collects pricing results from different hotel partners and displays their information on the hotel booking platform.	N.A.
Customers	Customers interact with the platform to perform various functions, such as retrieving accurate hotel information and displaying the lowest price by hotel and/or by destination.	- Read: Get latest available hotel information - Write: Booking & payment information
IT Stakeholders		
AWS	AWS provides Ascenda with the infrastructure and services to deploy and manage the applications. As an IT stakeholder, Ascenda is expected to use their services legally and to perform all the necessary security configuration and management tasks	N.A.

Ascenda's Development Team	Ascenda's Development Team is responsible for the development of features that fulfill the needs of the hotel booking platform.	- Read: Global Access - Write: Global Access
Ascenda's Maintenance Team	Ascenda's Maintenance Team is responsible for the availability, scalability, and security of the solution.	- Read: Global Access - Write: Global Access

3. Key Use Cases

Use Case Title – Search Destination by Location	
Use Case ID	01
Description	Customers will be able to **input a location** to query for hotels located in specified location. This is significant as Ascenda aims to provide a fast and accurate search of available hotels for their customers.
Actors	System, Customer
Main Flow of events	1. Customer inputs location data in the search bar. 2. The frontend will call the Ascenda's API. 3. Backend takes in relevant parameters and queries the database to retrieve the relevant hotel results. 4. Backend microservice returns JSON result to the frontend. 5. Customer's frontend displays available hotels
Alternative Flow of events	Invalid Location. Error Message, UI will display the "No hotel available in the specified location" message.
Pre-conditions	Ascenda API endpoint must be available. System needs to prepare location list. Port connection from microservice to database established. Customer input form displayed.
Post-conditions	System webpage displays correct/relevant hotel search results

Use Case Title – Autocompletion of User's Destination Search Input	
Use Case ID	02
Description	System will be able to **autocomplete Customer input** for the location field. This is an important feature to help improve the customer's user experience greatly.
Actors	System, Customer
Main Flow of events	1. Customer types in text in the search bar 2. Input text will be autocompleted by the system by providing suggestions on the webpage
Alternative Flow of events	System unable to recognise text input and is not able to provide autocompletion
Pre-conditions	System needs to prepare location list.

	Port connection to database established. Customer input form displayed. Recommender system operational.
Post-conditions	System webpage displays correct/relevant autocomplete results

Use Case Title –View Available Rooms	
Use Case ID	03
Description	Customer should be able to **view all available rooms** based on a hotel, dates of stay, number of rooms, number of guests. This functionality provides customer to sort available rooms based on the various input parameters.
Actors	System, Customer
Main Flow of events	1. Customer inputs parameter data in the frontend UI 2. The frontend will call the Ascenda's API using the Customer's search query. 3. Backend takes in parameters and queries the database to retrieve the relevant hotel rooms results. 4. Backend microservice return JSON result to frontend. 5. Customer's frontend UI displays available hotel rooms.
Alternative Flow of events	No available room, UI will display message, "No rooms available".
Pre-conditions	Ascenda's API endpoint is available Ports are configured to accept data packets
Post-conditions	Customer's client page displays all available rooms

Use Case Title – View Hotels by Price	
Use Case ID	04
Description	System renders a **listing view of all available hotels and cheapest room of each hotel from Ascenda's API**. This use case is important because customers would be interested in the cheapest and most accurate hotel room prices.
Actors	Database
Main Flow of events	1. Customers send search query. 2. Backend retrieves all available hotel results sorted by price. 3. Ascenda's API returns result as a JSON response. 4. The response would be rendered back on the client's side.
Alternative Flow of events	No results, display error message on UI, "Error, please try again"
Pre-conditions	Valid search parameters
Post-conditions	Returns result in ascending price

Use Case Title – Login To System	
Use Case ID	05
Description	**Customer logs in to their account to view their relevant details**. This provides users a personalised experience as their history with the system can be utilised for future interactions (suggestions/discounts)
Actors	System, Customer
Main Flow of events	1. Customer lands on the website's homepage which prompts log in to access system functionality. 2. Customer inputs username and password and logs in. 3. Customer is redirected to booking page / page where they left off.
Alternative Flow of events	User input wrong login credential, UI displays error message, "Username/Password incorrect, please try again"
Pre-conditions	System needs to prepare login page. Port connection to user database established. Customer login input form displayed. Login data validation must be up.
Post-conditions	Customer will be redirected to the user summary page to view their booking details or make a new booking.

Use Case Title – Book Hotel	
Use Case ID	06
Description	Customer will be able to **make a hotel booking after providing their booking details**. Thereby generating revenue for the hotels utilising the system as an additional channel of customer acquisition.
Actors	Customer, System, Hotel API
Main Flow of events	1. Customer will check their booking information and submit, following which the customer will be redirected to make payment. 2. Upon successful purchase verified by the system, customers will be redirected to the booking summary page
Alternative Flow of events	Customer transaction failed. Will be sent back to the booking page with error message displayed on the UI.
Pre-conditions	System needs to prepare booking and summary page. Port connection to user database established. Booking page displayed. Payment system needs to be up. Hotel Booking API needs to be up.
Post-conditions	Customer will be redirected to the booking summary page.

Use Case Title – Checks Booking Information (Post-Purchase)	
Use Case ID	07
Description	Customers will be able to **view their past bookings upon login / successful purchase**. Thereby allowing them to check on various details associated with previous bookings such as date, amount spent, etc.
Actors	Customer, System
Main Flow of events	1. Upon logging in / making a successful purchase, customer will be able to view a summary of their recent booking details. 2. Users will also be able view relevant details of each of their previous hotel bookings upon clicking on the respective booking
Alternative Flow of events	Transaction / login error. Error message will be displayed on the UI.
Pre-conditions	System needs to prepare booking summary page. Port connection to user database established. Login data validation must be up.
Post-conditions	Customer will have the option to make a new purchase / print page / logout.

4. Quality Requirements

Quality Requirement	
Performance Efficiency	
Autocompleting of destination	Once cache is loaded to the local machine, autocomplete of destination is instantaneous. 1st load of cache under 500ms, subsequent loads take 0ms as cache is saved locally for 24 hours.
Caching	To ensure quick auto completion, hotel and destination id-name mapping is cached at the frontend (Local Storage) so the retrieval and of relevant results are brought to the customers with very little delay.
Load balancer	To ensure our system can main high capacity, load balance can be used as a reverse proxy and to handle more user requests especially when there is a spike in user load.
Reliability	
Availability	99.9% uptime
Datacenter in separate availability zones	Ensure that there are separate regions to failover to for high availability.
Active/passive standby	Ensure the web app instances on a separate availability zone for redundancy.
AWS Load Balancing	Application Load Balancing (ALB) will be used as a reverse proxy.

AWS Auto-scaling Group	Auto Scaling will create a new instance when CPU utilization reaches 70% and destroys an excess instance when CPU utilization is less than 20%
Maintainability	
CI/CD using GitHub action	CI/CD helps to rapidly integrate new features and help to achieve zero downtime during deployment to S3 Bucket and ECS respectively.
Terraform	Terraform will be used to set up, make changes and version our S3, CloudFront and Route53. This ensures that these services are simpler to manage, can be replicated or altered with greater accuracy. With version control, we will be able to keep track of the changes made to them as well.
Security	
General Web Security	Metric for security: Get a minimum of B grade for Mozilla Observatory Results
OWASP Secure Coding Practices	Overall, Secure Coding Practices have been followed when judged from a categorical perspective with the exception of Memory Management.
WAF Filter Rule Firewall	Implement firewall to prevent XSS attacks and SQL injections.
Separate Web and Database Servers in DMZ	Implement different security layers.
SSL Certification	Enable https connection to protect server-client connection.
Revalidate data on server side	Minimize the risks of HTTP parameter tampering.

5. Key Architectural Decisions

Architectural Decision – Multi-Availability Zones Configuration	
ID	01
Issue	Having a single AZ configuration results in a **single point of failure** and **decreases reliability** of the solution. A failure of the AZ would inevitably shut down the system and cause it to become unavailable.
Architectural Decision	**IS25010 Considerations: Reliability (Availability)** Create the public and private subnets in 2 separate availability zones. The traffic is routed via an Application Load Balancer **(ALB)** which performs health checks and routes to healthy instances in the event of AZ failure.
Assumptions	Assuming that a failure of AWS services is confined to 1 Availability Zone
Alternatives	Deploying a CloudFormation template in a separate region with Route53 that routes traffic in the event of an entire region failure
Justifications	Having a multi-AZ configuration means that there is another AZ to fall back to if any AZ goes down, hence eliminating the single point of failure. It is highly unlikely that the multiple AZs will be down at the same time as well. Hence, multi-AZ ensures that the system has high availability. Furthermore, the abovementioned alternative is much more expensive to having a multi-AZ.

Architectural Decision – Dockerisation of Booking, Login and Registration Services & Elastic Container Service (ECS)	
ID	02
Issue	Container management is manual and time consuming, affecting ease of maintenance for the services.
Architectural Decision	**IS25010 Considerations: Maintainability, Reliability, Usability** We have containerised our Booking, Login and registration services and are managing these containers though Amazon ECS which will help with provisioning of compute resources and auto-scaling.
Assumptions	We won't be needing/using Kubernetes in the future.
Alternatives	Deployment via EC2 instances with Auto-scaling groups & AWS Elastic Kubernetes Service (EKS)
Justifications	Amazon Elastic Container Service (Amazon ECS) was developed to easily run and scale Docker container-based applications on AWS. ECS is highly scalable, offers high availability and security, and is deeply integrated with a variety of AWS services, including Amazon ELB, Amazon VPC, AWS IAM, and many more. Our team also considered Elastic Kubernetes Service (EKS) but decided against it as ECS has a gentler learning curve versus EKS which required knowledge on Kubernetes.
Architectural Decision – Multi-AZ Configuration on RDS	
ID	03
Issue	Storing all our sensitive booking information on a single database puts our service at risk of catastrophic data loss while also risking poor read/write performance during traffic spikes.
Architectural Decision	**IS25010 Considerations: Reliability (Availability)** This is achieved through database replication, a failover management is done automatically by RDS, where the slave is on standby to takes over requests in the event of failure that result the master to be down (and automatically recovers). We have incorporated redundancy and seamless failover process for our databases, through a one-way database replication, following a master-slave configuration.
Assumptions	Multiple database failure is unlikely due to AWS high availability across multiple availability zones.
Alternatives	Relational Database Service database replication, 2-way database replication, file transfer
Justifications	Two-way database replication requires databases to be sync at the start, and maintenance would be harder as compared to one-way database replication. 1-way database replication is sufficient to help to solve our issue and ensure our system high availability.

Architectural Decision – Use of S3 and CloudFront	
ID	04
Issue	Global performance is consideration for the hotel booking application since it will be used all over the world and must be accounted to maintain consistent performance for different users.
Architectural Decision	**IS25010 Considerations: Performance Efficiency, Security, Availability** The team decided to deploy the frontend assets to Amazon S3 and CloudFront. Our frontend application is built on React, and after provisioning the application, all assets are **static**. As such, we could deploy them to S3 easily, which is simply an object storage service. The main motivation behind using S3 as our storage is because it provides 99.999999999% durability. Another reason would be its compatibility with CloudFront Content Delivery Network (CDN) service by AWS. We generated a certificate using AWS ACM for our domain and used CloudFront to secure our application with HTTPS and SSL encryption. CloudFront provides performant speeds due to its CDN services. It also securely delivers data, applications, and APIs to customers globally with low latency, high transfer speeds, all within a developer-friendly environment.
Assumptions	Our frontend will always be compatible with deployment via S3.
Alternatives	Deploying either through a physical server or a cloud server like EC2
Justifications	The advantages of using S3 and CloudFront outweighs the alternative traditional deployment. This is because S3 and CloudFront provides a whole suite of services and extensibility. If we plan to implement new feature, it could be easily integrated with using S3 and CloudFront. On the other hand, if we used traditional deployment, we would have a hard time to set up and integrate which is unnecessary if we have the option to use S3 and CloudFront.
Architectural Decision – Route53	
ID	05
Issue	Global performance is consideration for the hotel booking application since it will be used all over the world.
Architectural Decision	**IS25010 Considerations: Availability, scalability** We used “Namecheap” to provide us with the domain name and with Route 53, we associated the name server records that were created for the hosted zone. Route 53 was used to route domain traffic from our domain provider. Route 53 is connected to our Web Application Firewall and CloudFront, and since it is globally distributed, end users from all over the world will quickly be routed to the relevant DNS servers and to our services. Route53 provides a scalable Domain Name System (DNS) service intended to give business and developers a reliable way to direct users to applications. Furthermore, it can be used to combined with DNS failover

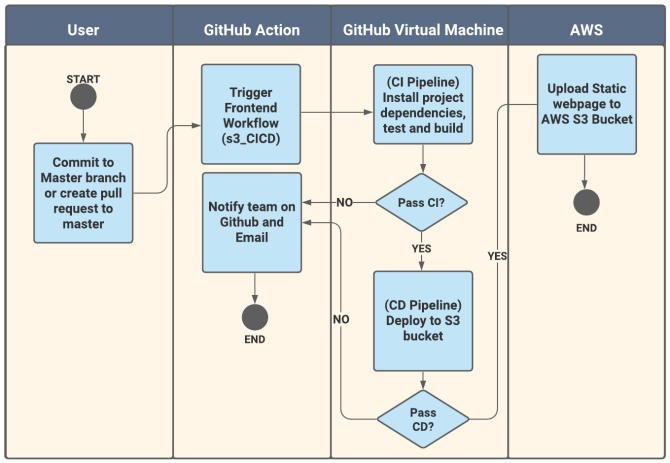
	which directs web traffic to a substitute location to prevent site outages, to enable a mixture of fault-tolerant, low latency architectures which are important requirements we need to fulfill. Route53 also has integration with AWS Elastic Load balancer (ELB) that allow us to map DNS directly to the existing ELB.
Assumptions	NIL
Alternatives	AWS CloudFlare DNS
Justifications	It is designed to give developers and businesses an extremely reliable and cost-effective way to route end users to Internet application. With AWS-hosted infrastructure, Route53 allows for a special set of alias records with extended privileges to simplify your DNS solution. Hence, as we are using AWS services, route53 would be a better option than CloudFlare.
Architectural Decision – Use Terraform for S3, CloudFront, API Gateway, Route53 and Lambda.	
ID	06
Issue	Our infrastructure requires manual configuration and set up (S3, CloudFront, Route53). This is not a feasible long-term solution since if there is a change in configuration, time and effort will be wasted to handle these changes meaning poor maintainability.
Architectural Decision	**IS25010 Considerations: Maintainability** Terraform is an infrastructure as code software tool that allows us to create, update and version our AWS infrastructure. It will be used to set up, make changes to our front-facing services - S3, CloudFront, API Gateway, Route53 and Lambda. Furthermore, it helps to version our S3, CloudFront and Lambda. This ensures that these services are simpler to manage, can be replicated or altered with greater accuracy. With version control, we will be able to keep track of the changes made to them as well.
Assumptions	Terraform will remain a suitable solution for the foreseeable future.
Alternatives	AWS CloudFormation
Justifications	The main motivation behind using Terraform is because it has a separate planning step which AWS CloudFormation does not provide. By running “terraform plan”, it generates an execution plan that will show us exactly what Terraform will do when we apply the template to our infrastructure and the orders of the execution steps as well. This makes it easier for us to reason about changes to our infrastructure. Compared to CloudFormation, we can rely on the Terraform visual graph and make better decisions to ensure that we will not inadvertently destroy critical infrastructure resources. Terraform is also an open-source project, there is a greater number of support regarding AWS services than compared to CloudFormation too. Furthermore, we do not have to be concern about the compatibility with AWS as terraform has sufficient support for the AWS services, as such we do not have to be locked-in to use CloudFormation.

Architectural Decision – Cached hotel information at frontend using Amazon ElastiCache for **Redis**	
ID	07
Issue	One of the key requirements was to provide the Destination Search feature that returns text-based auto-complete search that is near instantaneous (under 100ms) from a customer's perspective.
Architectural Decision	**IS25010 Considerations: Performance Efficiency** When the raw data is uploading into S3, an events notification will be sent to our lambda function, triggering it to process the data and then storing it on our ElastiCache (Redis). The data is then replicated into a replica node from the primary node in the second availability zone. In the event the cache data needs to be updated, the raw data (csv) uploaded to S3 will be processed again and uploaded into the primary node. The lambda function that accesses the data from Redis will only be accessing the data from the reader endpoint. We only used the reader endpoints to access the data to ensure that there will not be any instance in which when data is being changed, users will not be able to access the cached data. When the updating is done, the primary Redis will replicate the new cache data into the replica node to ensure data consistency. We also implemented local storage caching of the destination data on the client side with an expiration time of 1 day. Since we do not expect the data to change on daily basis, we felt that caching the data locally should not be an issue and will help to reduce the number of reading load on our Redis nodes.
Assumptions	Website will not be overloaded with traffic while the overwriting process is in progress.
Alternatives	Process data and store info as cache at the API gateway
Justifications	We chose to go with Redis as the main caching point as any user could pass a header of "Cache-Control: max-age=0" and it will cause our cache to be invalidated on the API gateway and they will need to execute the lambda function again to process data from s3 which causes timeout error. Thus, using Redis as our caching method ensures that data can be retrieved quickly even if the data is being overwritten with new incoming data.
Architectural Decision – API gateway (Facade) via AWS Lambda	
ID	08
Issue	The frontend would be directly calling our API endpoints and a change in the endpoint would require a change in the frontend code as well due to the tight coupling. Furthermore, there are security concerns of exposing our API endpoints due to multi points of entry which increases the attack surfaces. We are also facing CORS error when trying to hit the Ascenda API, hence we require a middleman to solve it.
Architectural Decision	**IS25010 Considerations: Security, Modifiability, Maintainability, Portability** API gateway acts as a facade and encapsulates all endpoints to only allow a single point of entry for clients to the API endpoint. Hence this ensure that

	there is a smaller attack surface, making the system more secure. The gateway via Lambda proxy adds the headers “Access-Control-Allowed": “*” which helps to solve our CORS error. After, the Lambda function containing the APIs will be called by the frontend.
Assumptions	API Gateway has access to all other endpoints through the internet or intranet.
Alternatives	Direct client-to-backend or microservice communication
Justifications	The API Gateway would serve as the single point of entry for clients to our APIs. This helps with enhancing security by preventing unauthorized usage of the API. The facade design pattern also hides the logic and calling order of APIs. Lastly, the API Gateway lets us decouple the frontend web application from the backend API. When changes to APIs and endpoints are made, the only change that needs to be made is on the API Gateway and the lambda function as opposed to changing every single call on the frontend if we were to directly have the frontend call the APIs.
Architectural Decision – CICD pipeline	
ID	09
Issue	Code contributed by multiple developers should not break the existing system. Code that breaks the solution and testcases should be identifiable through various stages of the pipeline and should not be deployed to the production environment in the S3 bucket.
Architectural Decision	**IS25010 Considerations: Maintainability** Created a workflow using GitHub Actions to set up CI/CD pipeline (Appendix A) which ensures that all test cases passed and there are no errors before deploying to S3 bucket. Continuous integration and development to ensure that the quality of code deployed is not compromised. Another workflow was created to deploy the docker file which contains our booking functions to docker hub and Elastic Container Service using the latest task definition (Appendix B). The AWS credentials needed for the GitHub actions to work will be encrypted in the Actions Secret (Appendix C). Lifecycle Policy is also added to remove untagged image 1 day after it is pushed into the repositories. (Appendix D)
Assumptions	Assumption that the AWS services required, mainly ECS Cluster containing the service and the S3 bucket, are up.
Alternatives	Only have 1 developer work on the project at a time or manually testing and deploy.
Justifications	Not feasible as the alternatives take too much time.

6. Development View

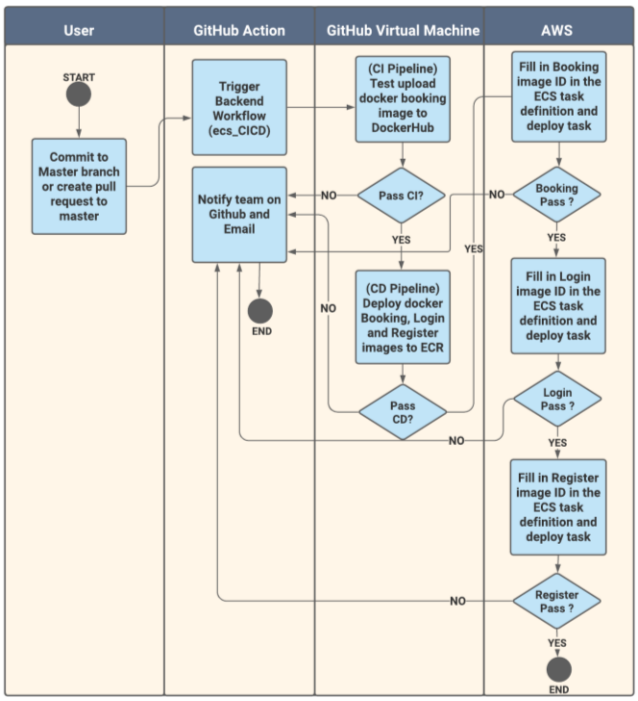
Frontend Activity Diagram



Workflow for Frontend GitHub Action

1. After user push or create request to master branch, it will trigger this workflow.
2. Install dependencies, build and test the frontend code. The test case can be found in app.test.js, it consists of a test case to check for syntax error and if the new snapshot of the app is the same as the stored snapshot.
3. If the CI step is successful, continue progressing the pipeline to deploy the code to S3 Bucket.
4. If the task fails at any step, the error log will be displayed on the GitHub workflow action and a notification would be sent to the team.

Backend Activity Diagram

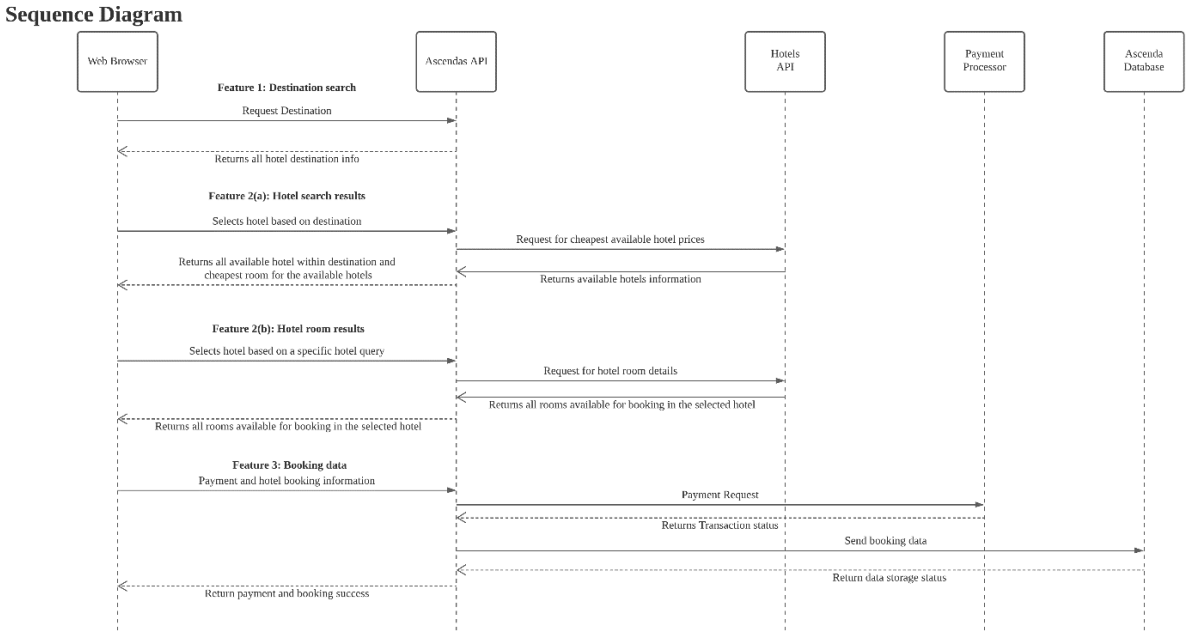


Workflow for Backend GitHub Action

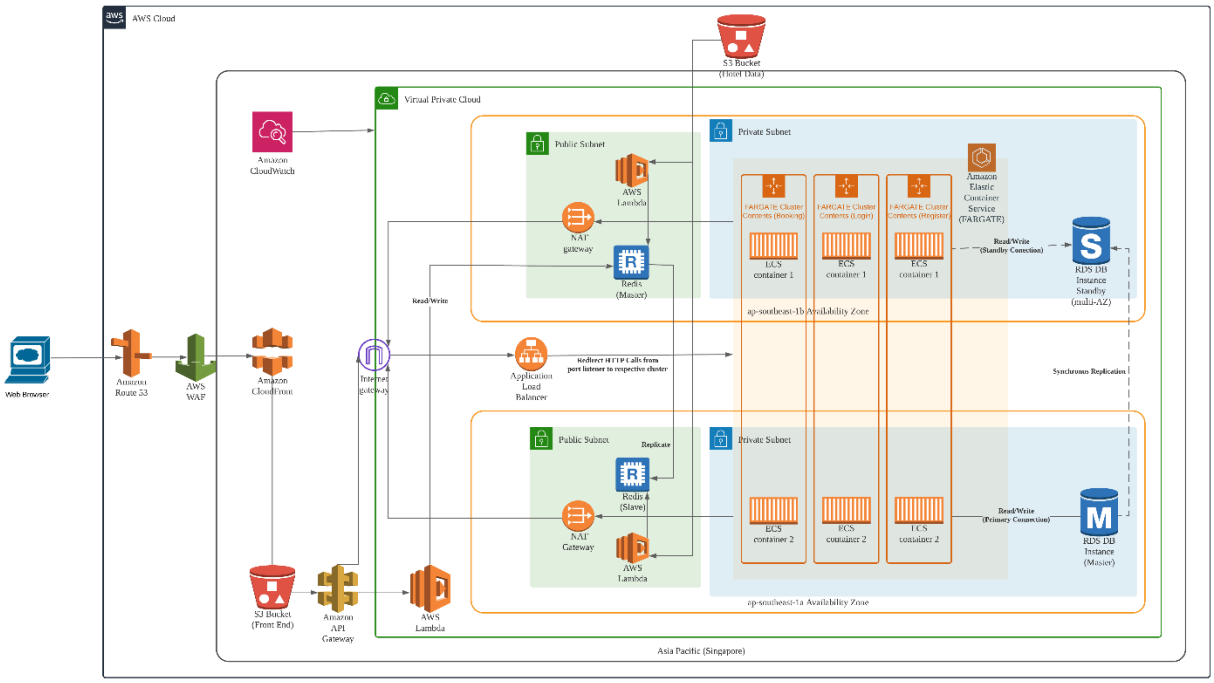
1. After user push or create request to master branch, it will trigger this workflow.
2. Test Docker image action by pushing the Booking Image to DockerHub. If successful, continue the pipeline to deploy the images.
3. In the CD step, the booking, registration and login task definitions are deployed to the ECS. If successful, the new task definitions of a newer version would be used for the images.
4. If the task fails at any step, the error log will be displayed on the GitHub workflow action and a notification would be sent to the team.

7. Solution view

7.1 Sequence Diagram



7.2 Architecture Diagram



The ease of maintainability designs is achieved in our architecture through the use of GitHub Actions, we implemented **Continuous Integration and Continuous Development** (CICD) into our project. Doing so, we are able to ensure that testing is done when integrating the incoming code with the current code base and then successfully deploy the new build file into S3 and then invalidating the previous version, serving the newly updated code to CloudFront without any down time. Furthermore, the backend codes are dockerised and the images and their respective task definitions are deployed to ECS through the CICD workflow. Lifecycle policy is also used to remove untagged images that are pushed after a day. **Terraform** are also used to set up, make changes and version our S3, CloudFront, API Gateway, Route53 and Lambda.

7.3 Integration Endpoints

Source System	Destination System	Protocol	Format	Communication Mode
AWS WAF	AWS CloudFront	HTTP/HTTPS	JSON	Asynchronous
AWS S3	Amazon API Gateway	HTTPS	JSON	Asynchronous
AWS API Gateway	AWS Lambda	HTTPS	JSON	Asynchronous
AWS ECS (Fargate)	AWS RDS	JDBC	MYSQL	Asynchronous
AWS S3	AWS Lambda	HTTPS	JSON	Asynchronous
AWS Lambda	Redis	HTTPS	JSON	Asynchronous
AWS API Gateway	AWS ECS (Fargate)	HTTPS	JSON	Asynchronous

8. Proposed Budgets

Development Budget

Activity/ Hardware/Software/ Service	Description	Cost ($/Month)
AWS EC2 Instances	Deployment of our backend services as containers in EC2 instances	$ 2.66
AWS ElastiCache	Used for caching our hotel and destination information using Redis for faster search results	$ 44.92
RDS MySQL	MySQL RDS used to store our booking information.	$ 11.54
Elastic Container Service (Fargate)	Deployment of our backend services as containers using Fargate	$ 7.69
Amazon Route 53	Maintains our registered DNS (ascendahotels.me) on a hosted zone	$ 3.35
AWS S3	Storage of static front-end webpages and other resources	$ 0.053
AWS API Gateway	Allow communication between VPC and internet	$ 0.059
AWS VPC (NAT Gateway)	Allow communication between VPC and internet	$ 57.00
AWS CloudWatch	We tried to implement some CloudWatch metrics but decided to use any	$ 0.059
AWS CloudFront	It is a fast content delivery network (CDN) service that allows us to quickly deliver hotel destination data and images to the frontend with low latency and high transfer speeds.	$ 0.058
AWS Application Load Balancer	Load Balancer will act as the reverse proxy and at the same time allow for high availability	$ 2.82

AWS WAF (Web Application Firewall)	It will be used to help to protect our system against common web exploits that may affect availability, compromise security, or consume excessive resources. It also gives us the control over how traffic reaches our applications by allowing us to create security rules that block common attack patterns.	$ 0.19
AWS EC2 Container Registry	Storage of docker images used to run backend services	$ 0.11
Total Monthly Cost		**$ 130.51**

Production Budget

Activity/Hardware/ Software/Service	**Description**	**Cost ($/Month)**
AWS Application Load Balancer	Load Balancer will act as the reverse proxy and at the same time allow for high availability **(1 ALB, 15LCUs)**	$ 25.30
Elastic Container Service (Fargate)	We are deploying our backend services on ECS Fargate as our services are dockerised in containers. **(6 task, 0.25 vCPU & 0.5 GB memory resource/task)**	$ 66.55
Elastic Container Registry	Storage of docker images to run backend services. **(75 MB / month)**	$ 0.013
RDS MySQL	MySQL RDS used as buffer cache. **(20GB, Multi-AZ)**	$ 58.29
AWS S3	Storage of Application files and other resources **(Standard S3, 1.5GB/Month)**	$ 0.05
AWS API Gateway	Allow communication between VPC and internet. **(40KB Average Request Size, 25 thousand request/month)**	$ 0.09
AWS VPC (NAT Gateway)	Allow communication between VPC and internet. **(2 NAT Gateway, 1 GB data processed/month)**	$ 116.99
AWS CloudFront	It is a fast content delivery network (CDN) service that allows us to quickly deliver hotel destination data and images to the frontend with low latency and high transfer speeds.	Not Applicable
AWS Lambda	A serverless compute function that runs the code on demand or in response to events. **(25 thousand requests/month)**	$ 0.23

AWS WAF (Web Application Firewall)	It will be used to help to protect our system against common web exploits that may affect availability, compromise security, or consume excessive resources. It also gives us the control over how traffic reaches our applications by allowing us to create security rules that block common attack patterns. **(1 Web ACL, 25 thousand Request/Month, 4 Rules)**	$ 12.08
Amazon Route 53	Maintains our registered DNS (ascendahotels.me) on a hosted zone. **(1 Hosted Zone)**	$ 0.51
AWS ElastiCache	We will be caching our hotel and destination information using Redis for faster search results. **(2 Standard Redis Nodes, t2.micro OnDemand)**	$ 43.06
Total Monthly Cost		**$ 323.27**

9. Availability View

Node	**Redundancy**	**Clustering**			**Replication (if applicable)**			
		Node Config	**Failure Detection**	**Failover**	**Repl. Type**	**Session State Storage**	**DB Repl. Config.**	**Repl. Mode**
Booking Service (Fargate)	Horizontal Scaling	Active-Active	Ping	Application Load Balancer				
AWS RDS	Horizontal Scaling / Multi-AZ deployment	Active-Passive	Ping	Managed by RDS	DB	Database	Master-slave	Synchronous
Redis	Multi-AZ	Active-Active	Ping	Managed by Redis cluster	Session	Memory Sessions	Master-Master	Asynchronous

10. Security View

No	Asset/Asset Group	Potential Threat/Vulnerability Pair	Possible Mitigation Controls
1	API Gateway	A Distributed Denial of Service (DDoS) attack could exploit our entry point (API Gateway), resulting in a single point of failure and affect the **availability** of our system.	Make use of AWS Web Application Firewall (WAF) to monitor HTTP/HTTPS requests forwarded to the API Gateway REST API (Use the following rules: Amazon IP reputation list, Core Rule Set, Linux Operating System & RDS Database)
2	RDS Database	An attack via malicious forms of information exposure could exploit the unencrypted RDS data/snapshots, affecting the **confidentiality** and **security** of the data.	The team has enabled encryption on the RDS storage data and placed it in the private subnet with only port 3306 opened.
3	Flask Microservices	SQL Injection which will affect the **confidentiality** in the stored data.	Ensured that all input validations are handled server-side and ensure we passed all the testcases in the SQL Injection Test (Appendix E).
4	Client/ Website	Cross-site Scripting (XSS) or XSS Injection might affect the **integrity** of the web application's Document Object Model (DOM) through injection of malicious scripts.	Ensure that all input validations are handled and ensured the passing of all testcases in the SQL XSS Test (Appendix F). Set content-security-policy headers implemented via Lambda@Edge.

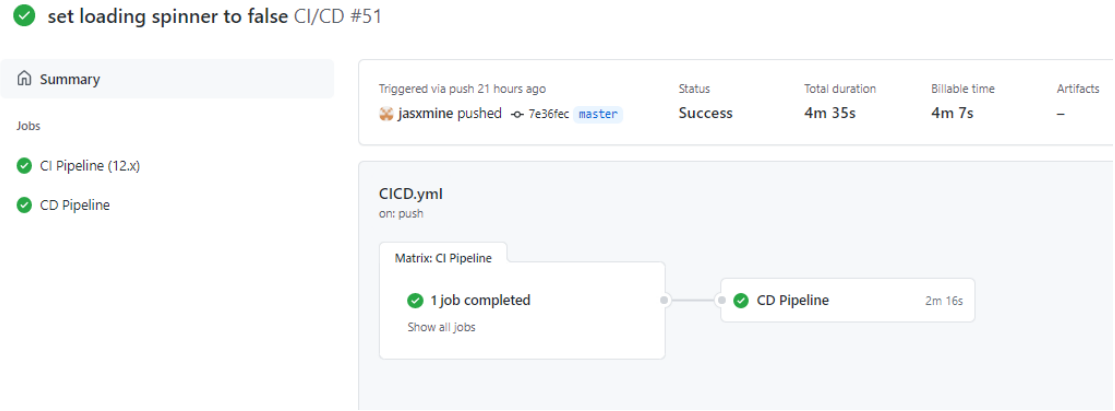
11. Performance View

No	Description of the Strategy	Justification	Performance Testing (optional)
1	Localized caching for destination search results	Our team's idea for how to achieve fast response for "Feature 1: Destination Search" was to provide a localized cache to provide for the autocomplete search feature. The rationale was that aside from the initial load time for the cache, subsequent searches would be able to autocomplete at near instantaneous speeds, thereby providing an optimal user experience.	To be demonstrated **Baseline model:** Show timer demonstration load time for first load. **Improved model**: show timer demonstration load time for instant response after local cache is loaded in.
2	API Gateway cache	By caching responses of requests on the API gateway, it further reduces the response time required to retrieve data for first time visitors.	
3	Stress testing on our backend	To simulate how the performance of the backend will be affected during high traffic (e.g., high number of booking requests) without any of the services failing.	Using the 'hey' test running on our API (/backend/booking) we can see the CloudWatch scaling up the number of tasks required. When the CPU more than 70% for 5mins (for demonstration purposes, we will put it at 1min)

Appendix

Appendix A: CI/CD Code for the workflow to Test and Deploy front-end to S3 Bucket.

```yaml
name: CI/CD

on:
  push:
    branches:
      - master
  pull_request:
    branches:
      - master

defaults:
  run:
    working-directory: ./initial-app/frontend

jobs:
  buildAndTest:
    name: CI Pipeline
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: ['12.x']

    steps:
      - uses: actions/checkout@v2

      # Initialize Node.js
      - name: Install Node.js ${{ matrix.node-version }}
        uses: actions/setup-node@v1
        with:
          node-version: ${{ matrix.node-version }}

      # Install project dependencies, test and build
      - name: Install dependencies
        run: npm install
      - name: Run build
        run: npm run build
      - name: Test
        run: npm test

  deploy:
    name: CD Pipeline
    runs-on: ubuntu-latest

    needs: buildAndTest
    steps:
      - uses: actions/checkout@v2

      # Initialize Node.js
      - name: Install Node.js ${{ matrix.node-version }}
        uses: actions/setup-node@v1
        with:
          node-version: ${{ matrix.node-version }}

      # Install project dependencies and build
      - name: Install dependencies
        run: npm install
      - name: Run build
        run: npm run build

      # Deploy the S3 bucket
      - name: deploy
        run: yarn deploy
        env:
          SOURCE_DIR: "build"
          AWS_REGION: 'ap-southeast-1'
          AWS_S3_BUCKET: ${{ secrets.AWS_S3_BUCKET }}
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY }}
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_KEY }}
```

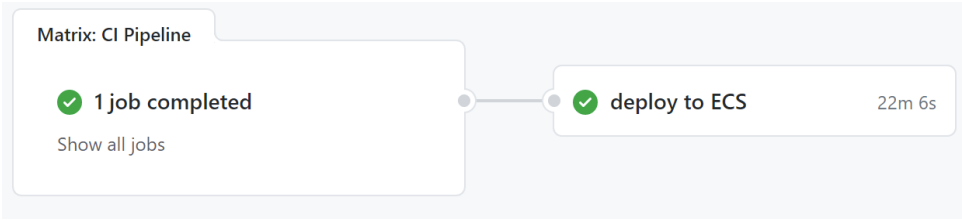


Appendix B: CI/CD Code for the workflow to Deploy DockerFile to Docker Hub and ECS.

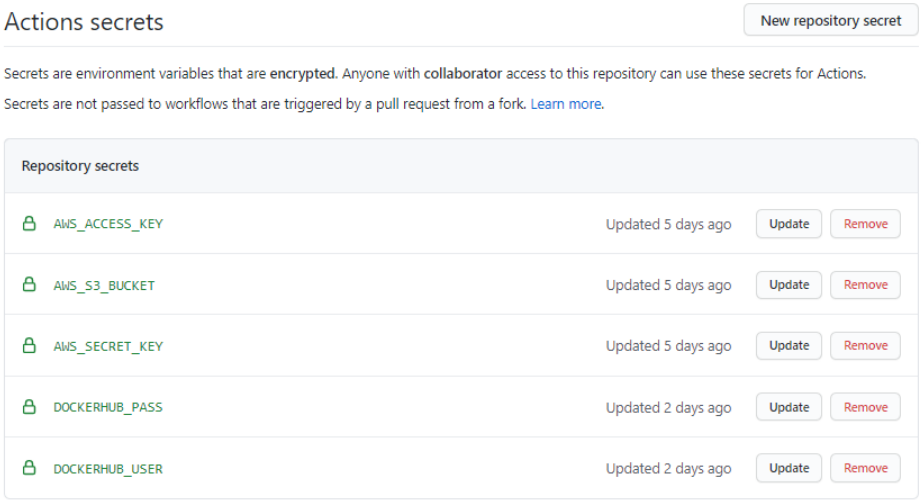
```yaml
name: Docker Image CI

on:
  push:
    branches:
      - master
  pull_request:
    branches:
      - master

env:
  AWS_REGION: ap-southeast-1
  ECS_TASK_DEFINITION_BOOKING: ./initial-app/backend/booking/task-definition.json
  ECS_TASK_DEFINITION_LOGIN:  ./initial-app/backend/login/task-definition.json
  ECS_TASK_DEFINITION_REGISTRATION: ./initial-app/backend/registration/task-definition.json
  CONTAINER_BOOKING_NAME: itsa-booking-container
  CONTAINER_REGISTER_NAME: itsa-register-container
  CONTAINER_LOGIN_NAME: itsa-login-container

defaults:
  run:
    shell: bash
    working-directory: ./initial-app/backend

jobs:
  CI:
    name: CI Pipeline
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: ['12.x']
    steps:
    - uses: actions/checkout@master
    - name: Publish Docker
      uses: elgohr/Publish-Docker-Github-Action@2.11
      with:
        # The name of the image you would like to push
        name: jasxmine/itsa
        # The login username for the registry
        username: ${{ secrets.DOCKERHUB_USER }}
        # The login password for the registry
```

```yaml
        # The login password for the registry
        password: ${{ secrets.DOCKERHUB_PASS }}
        workdir: ./initial-app/backend/booking

  deploy:
    name: deploy to ECS
    needs: CI
    runs-on: ubuntu-latest
    steps:
    - name: Checkout
      uses: actions/checkout@v2

    - name: Configure AWS credentials
      uses: aws-actions/configure-aws-credentials@v1
      with:
        aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY }}
        aws-secret-access-key: ${{ secrets.AWS_SECRET_KEY }}
        aws-region: ${{ env.AWS_REGION }}

    - name: Login to Amazon ECR
      id: login-ecr
      uses: aws-actions/amazon-ecr-login@v1

    - name: Build, tag, and push image to Amazon ECR
      id: build-image
      env:
        ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
        ECR_REPOSITORY: itsa_backend
        B_IMAGE_TAG: booking
        R_IMAGE_TAG: registration
        L_IMAGE_TAG: login
        DOCKER_REGISTRY: ${{ steps.login-ecr.outputs.registry }}/itsa_backend
      run: |
        # Build a docker container and
        # push it to ECR so that it can
        # be deployed to ECS.
        docker-compose build
        docker-compose push
        echo "::set-output name=bookingImage::$ECR_REGISTRY/$ECR_REPOSITORY:$B_IMAGE_TAG"
        echo "::set-output name=registerImage::$ECR_REGISTRY/$ECR_REPOSITORY:$R_IMAGE_TAG"
        echo "::set-output name=loginImage::$ECR_REGISTRY/$ECR_REPOSITORY:$L_IMAGE_TAG"
```

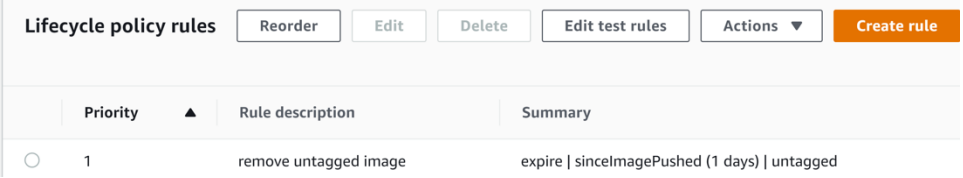
```yaml
    #booking
    - name: Fill in the new booking image ID in the Amazon ECS task definition
      id: task-def
      uses: aws-actions/amazon-ecs-render-task-definition@v1
      env:
         DOCKER_REGISTRY: ${{ steps.login-ecr.outputs.registry }}/itsa_backend
         ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
         ECR_REPOSITORY: itsa_backend
      with:
        task-definition: ${{ env.ECS_TASK_DEFINITION_BOOKING }}
        container-name: ${{ env.CONTAINER_BOOKING_NAME }}
        image: ${{ steps.build-image.outputs.bookingImage }}

    - name: Deploy Amazon ECS task definition for booking
      uses: aws-actions/amazon-ecs-deploy-task-definition@v1
      with:
        task-definition: ${{ steps.task-def.outputs.task-definition }}
        service: itsa-booking-service
        cluster: BookingCluster
        wait-for-service-stability: true

    #login
    - name: Fill in the new login image ID in the Amazon ECS task definition
      id: task-def-2
      uses: aws-actions/amazon-ecs-render-task-definition@v1
      env:
         DOCKER_REGISTRY: ${{ steps.login-ecr.outputs.registry }}/itsa_backend
         ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
         ECR_REPOSITORY: itsa_backend
      with:
        task-definition: ${{ env.ECS_TASK_DEFINITION_LOGIN }}
        container-name: ${{ env.CONTAINER_LOGIN_NAME }}
        image: ${{ steps.build-image.outputs.loginImage }}

    - name: Deploy Amazon ECS task definition for login
      uses: aws-actions/amazon-ecs-deploy-task-definition@v1
      with:
        task-definition: ${{ steps.task-def-2.outputs.task-definition }}
        service: itsa-login-service
        cluster: BookingCluster
        wait-for-service-stability: true

    #registration
    - name: Fill in the new registration image ID in the Amazon ECS task definition
      id: task-def-3
      uses: aws-actions/amazon-ecs-render-task-definition@v1
      env:
         DOCKER_REGISTRY: ${{ steps.login-ecr.outputs.registry }}/itsa_backend
         ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
         ECR_REPOSITORY: itsa_backend
      with:
        task-definition: ${{ env.ECS_TASK_DEFINITION_REGISTRATION }}
        container-name: ${{ env.CONTAINER_REGISTER_NAME }}
        image: ${{ steps.build-image.outputs.registerImage }}

    - name: Deploy Amazon ECS task definition for registration
      uses: aws-actions/amazon-ecs-deploy-task-definition@v1
      with:
        task-definition: ${{ steps.task-def-3.outputs.task-definition }}
        service: itsa-register-service
        cluster: BookingCluster
        wait-for-service-stability: true
```



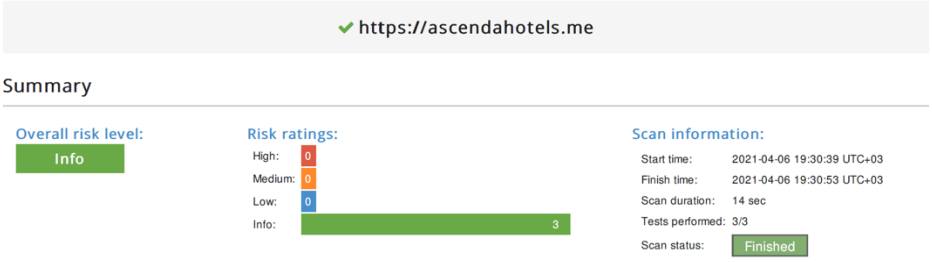
Appendix C: Action Secrets which Contains the Encrypted AWS Keys



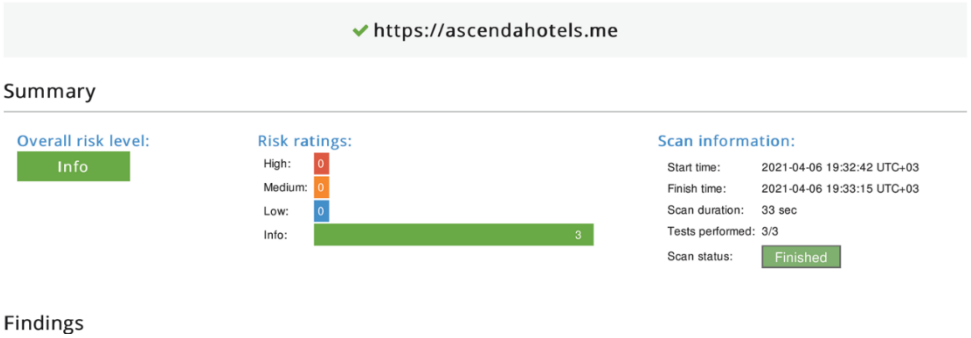
Appendix D – Lifecycle Policy of Removing untagged Image



Appendix E - SQL Injection Test Result on Pentest-Tool



Appendix F – XSS Injection Test Result on Pentest-Tool



Appendix G – Mozilla Observatory Grade