SMU
SINGAPORE MANAGEMENT UNIVERSITY

CS301: IT Solution Architecture
AY 2019-2020 Term 2

Final Report

Team Name:
ITSA GOURDS

Wayne Loh	01351009
Seow Jian Liang Job	01334202
Lee Wei Han Sean	01314016
Li KangZheng	01309534
Wong Xian Rui Abel	01316686

Instructor: Dr. Ouh Eng Lieh

Contents

Background and Business Needs

In light of the recent Coronavirus outbreak, the shortage of masks available at physical stores, and the close physical proximity when queueing for them has led to the need of an eCommerce store selling masks. The eCommerce store will feature a wide array of masks that will cater to the different levels of coverage and filters needed for different strains of viruses. Since prices sold on the site will be standardized, the site aims to eliminate the existence of potential black markets overcharging for the masks. Customers' information will also be processed to eliminate the scenario of hoarding.

The functionality of the store will be focused around customer-side interactions present in an eCommerce store. Shoppers will be able to find a specific mask based on the ID, name, price, view the masks, and checkout their orders. The site will provide a convenient platform for citizens to purchase mask at the comfort and safety of their home.

Stakeholders

Stakeholder	Stakeholder Description
Customer	• Checks out mask(s) from the website • Views masks • Login into his/her account to view orders
Business Owner	• Conceive objectives for greater profit • Design and implement plans and strategies • Ensure that the company has the adequate and suitable resources to complete its daily routines • Organize and coordinate operations for optimal productivity • Supervise the work of employees • Liaise with external partners/vendors/suppliers • Assess overall company performance periodically against objectives
Project Manager	• Define project scope and objectives • Planning of resources (manpower, time, money, hardware) needed to reach objectives • Manage the utilization of resources effectively and efficiently
Security Manager	• Monitor networks for security breaches and investigate violations • Design, implement, and maintain the organization's cyber-security plan. • Direct the installation and use of security tools (e.g., firewalls, data encryption), to protect sensitive information. • Recommend and implement security standards and best practices • Ensure that IT security audits are conducted periodically or as needed (e.g., when a security breach occurs).
Infrastructure Manager	• Overviews IT operations • Maximise system uptime to meet service level agreements (SLAs) • Ensure maintainability in system architecture • Promote consistent standards and reusability in codes • Maintain logs, documentation and reporting of network irregularities

Key Use Cases

Login	
Use Case ID	1
Description	Customer logs into his account to make a purchase or view history of products
Actors	Customer
Main Flow of events	1. Customer enters login page 2. Customer is prompted to enter username and password 3. Customer enters username and password 4. Customer is logged in and redirected back to homepage
Alternative Flow of events	*Customer logins without a valid account* 1. Customer enters login page 2. Customer is prompted to enter username and password 3. Customer enters invalid username and password 4. Error is shown and customer is not logged in
Pre-conditions	Customer is not logged in
Post-conditions	Customer is logged in

Purchase Mask (Add to Cart)	
Use Case ID	2
Description	Customer views the mask and specifies the quantity that he would like to purchase and adds it to the cart
Actors	Customer
Main Flow of events	1. Customer enters the mask details page 2. The system displays the mask information and quantity information on the page 3. The customer selects the desired quality and clicks the “add to cart” button 4. The system adds the order into the cart and updates the chart icon on the page
Alternative Flow of events	*Mask is out of stock* 1. Customer enters the mask details page 2. System displays the mask information but disable the field to select quantity 3. Customer exits the page and the system does not update *Customer changes his mind* 1. Customer enters the mask details page 2. System displays the mask information and quantity information on the page 3. Customer decides to not proceed with the order 4. Customer exits the page and the system does not update
Pre-conditions	• Customer should be logged in • Mask information should be in the database
Post-conditions	Cart information is updated

View Masks	
Use Case ID	3
Description	Customer has a bird's eye view of all the products that the business is currently selling
Actors	Customer
Main Flow of events	1. Customer enters the view masks page 2. System displays hyperlinks that redirect to mask details page. Hyperlinks represented by images, names, and price of masks 3. Customer clicks on mask that he wants to know more about or potentially purchase, is redirected to mask details page of a particular mask id
Alternative Flow of events	1. Customer decides to not to view 2. Customer exits the page
Pre-conditions	• Customer should be logged in • Mask information should be in the database
Post-conditions	NULL

Checkout	
Use Case ID	4
Description	Customer checks out their cart once they have their desired item(s) placed in the cart, confirming their purchase of mask(s)
Actors	Customer
Main Flow of events	1. Customer is already viewing their cart 2. Customer clicks the "check out" button to confirm their purchase 3. System processes the checkout 4. Confirmation email is sent to customer
Alternative Flow of events	*Customer logs out without checking out* 1. Customer is already viewing their cart 2. Customer then logs out of the web page without checking out 3. System saves cart information into database
Pre-conditions	• Customer should be logged in • Mask information should be in the database
Post-conditions	• Confirmation of purchase is sent to customer

Quality Attributes

Quality Requirement Title - Maintainability	
Quality Requirement ID	1
Description	• Conduct and run test cases for all classes • Ensure CI/CD • Ensure loose coupling, high cohesion • Ensure API naming convention follows best practices • Singleton design pattern for database connection • Façade design pattern to hide complexity • Clear deployment instructions for future reference (README.md)
Significance	• Test cases decreases chances of introducing new bugs or breaking new builds while fixing bugs and adding new features. • It also provides ease of bringing new developers on board and reduces technical debt. • Ensures that the application is understandable and can be easily handed over to future developers

Quality Requirement Title - Availability	
Quality Requirement ID	2
Description	• 99.99% uptime between 8am – 12pm daily • Ensure that user is able to maintain access to the website in the event of system failure(s) • Ensure that user maintains his state in the event of system failure(s)
Significance	• It is important to ensure that customers are able to swiftly and efficiently place their orders during peak hours. This is especially pertinent in times of a pandemic. • Ensuring that redundancy is in place ensures that users' experience will not be affected in the event of the system failure. • Maintaining the user state allows him to quickly continue from when he last left off in the event of a system failure.

Quality Requirement Title - Security	
Quality Requirement ID	3
Description	• No unauthorised access to protected pages and backend APIs • No SQL injection • No breach of data confidentiality during data transmission • No breach of data integrity • DDoS Protection
Significance	• Keeping sensitive information secure is key to building trust with

	customers. Other areas of security such as data tampering are even more relevant as eCommerce sites are susceptible to payment fraud(s).

Quality Requirement Title - Performance	
Quality Requirement ID	4
Description	• 1 second initial web page load time with 10,000 requests • Handle 100 concurrent users
Significance	• Performance is pertinent in times of crisis due to the global rush for protective masks. Estimating the sum of physical customers and translating them to online ones, it is imperative that our system is able to handle a large number of users at any point in time. • A fast response time is crucial in ensuring a smooth user experience when using our website

Key Architectural Decisions

Architectural Decision – Cloudflare as Proxy Server	
ID	1.0
Issue	• Slow response time • DDoS attack • Unsecure connection
Architectural Decision	The Cloudflare will act as the proxy server and expose the entry point to our website via https://itsamasks.tk. Requests that are sent to the domain name is then redirected to our frontend server. As the proxy server, Cloudflare offers caching capabilities, DDoS protection and SSL Certificates to secure our connections.
Assumptions	Cloudflare has its own redundancy in place and there is no need for us to introduce more redundancy at the proxy level
Alternatives	• Using Amazon Certificate Manager to obtain a certificate to secure our HTTPs connections • Using Amazon Elastic Beanstalk to handle caching • Using Amazon Shield/Amazon WAF for DDoS Protection
Justification	Cloudflare services are free and does not require us to handle multiple services. Using of AWS services would require us to pay for them and would exceed our project budget.

Architectural Decision – Use of Elastic Beanstalk	
ID	2.0
Issue	Traditional means of setting up and maintaining these infrastructures is time-consuming: • Auto-Scaling • Load Balancing • Easy deployment

	• Health and traffic monitoring tools
Architectural Decision	We will be using AWS Elastic Beanstalk platform as our deployment, scaling, and monitoring service.
Assumptions	Merely using Elastic Beanstalk will remain sufficient to cater to all our needs in the future, and that we should not require the added flexibility of setting up ourselves
Alternatives	Manually setting up the above components ourselves
Justification	• Elastic Beanstalk comes with the above tools and are already automated. These are consolidated into a highly abstracted console with a simple UI that new software developers within the firm could easily adopt, learn and monitor. • Setting up from scratch by ourselves would incur substantial time and money costs, along with the additional costs of maintenance. • By offloading these functionalities to AWS, we will be able to devote more resources to focus on developing more new functional features.

Architectural Decision – Use of Elastic Load Balancer [Application Load Balancer]	
ID	3.0
Issue	• Single point of failure on EC2 instances • Lack of ability to remain fault-tolerant for varying load of incoming application traffic • Minimise downtime when switching between targets
Architectural Decision	We will be using Elastic Load Balancer (Application Load Balancer) to automatically distribute incoming traffics to healthy targets in different Availability Zones, ensuring 99.99% availability.
Assumptions	• Elastic Load Balancer will not be unavailable as a service itself • All targets will not be unavailable over the same period
Alternatives	• Other load balancing services such as NGINX Load Balancer
Justification	Elastic Load Balancer (Application Load Balancer) has an array of features and built-in configurations that makes it simple to set up and point to multiple targets. Beyond ensuring high-availability, ELB also has deep integration with Auto Scaling to meet varying levels of application load without requiring manual intervention (as explained below). Furthermore, ELB allows round-the-clock monitoring with Amazon CloudWatch metrics and this gives us clear and single-pane visibility into the behavior of our applications, identifying any anomalies or getting to the source of the problem efficiently.

Architectural Decision – Use of autoscaling, active-active instances	
ID	4.0
Issue	• Process of catering for more users (and requests) to the server is not automated • Downtime when switching from master to slave instance
Architectural Decision	As mentioned above, we will be using the Elastic Beanstalk suite of services, which includes automatic horizontal scaling of active (not passive) EC2 instances based on monitored metrics (e.g. CPU Utilization, Network Output)

	and Elastic Load Balancer, that distributes the requests to the spawned instances accordingly.
Assumptions	• Demand is significant enough or fluctuates to the extent that the additional costs incurred by autoscaling is justified • Any potential downtime incurred (if we use active-passive) could jeopardize the business, thus justifying the use of active-active instances
Alternatives	• Continue using active-passive configuration • Manually spawn or kill instances • Manually configure load balancer to distributes depending on the number of active instances
Justification	• Our business case necessitates that our application needs to be served with as little downtime as possible in times of a pandemic (like COVID-19) • Elastic Beanstalk allows us to easily manage, distribute requests, and automatically scale between 1-4 active instances • No downtime when an instance shuts down or terminates if there are one or more active instances available • Automatically spins up another instance to compensate for the downed one • Little downtime taken to re-spin another instance when an instance shuts down or terminates and if there is no other active instance currently available

Architectural Decision – Multiple Availability Zones (Database, Front End and Back End)	
ID	5.0
Issue	Database • Failures that affect the availability of database instances spun up in the same location • To facilitate active-passive approach • Minimise time for manual intervention and initiation for database failover Front End and Back End • Failures that affect the availability of front end and back end instances spun up in the same location • Facilitates active-active approach • Minimise time for manual intervention and initiation for front and back end failover
Architectural Decision	Using RDS, we will have Multi A-Z deployments for our database instances and respective read-replicas to ensure high availability with automatic failover procedures. Using Elastic Bean's Multi A-Z deployments for our front and back end instances to ensure high availability with automatic failover procedures.

Assumptions	N/A
Alternatives	• Amazon Aurora • Traditional Master and Slave Approach • Route 53 for manual routing to specific availability zones under specific conditions
Justification	AWS Relational Database Service (RDS) and Elastic Bean automatically performs a failover in the event any of the following happens: • Loss of availability in primary Availability Zone • Loss of network connectivity to primary • Compute unit failure on primary DB • Storage failure on primary DB The whole failover process is entirely automatic, requires no manual intervention and the endpoint for our database, front end and back end instances remains the same after a failover. Moreover, planned maintenances and backups are applied first to the standby, prior to any automatic failover for the database.

Architectural Decision – Read Replica of database	
ID	6.0
Issue	• Lack of scalability • Inability to handle concurrent requests hitting the database for read purposes
Architectural Decision	As we are using RDS, we are also creating read replica(s) for our database instance(s) to serve high-volume application read traffic from multiple copies of our data, ultimately increasing aggregate read throughput.
Assumptions	• High-volume traffic expected to access database
Alternatives	• Amazon Aurora
Justification	AWS RDS Read Replicas provide enhanced performance and durability of database instances through the use of asynchronous replication. Although only allowing read-only connections, read replicas can also be manually promoted to a standalone database instance, whenever necessary.

Architectural Decision – Microservices and containerization of services	
ID	7.0
Issue	• Complexity in horizontal scaling (Functional-wise) of services and product features • Difficulty in fault isolation • Difficult to develop and debug • Ensuring cloud native development and deployment
Architectural Decision	• Decouple and package the frontend and backend applications into two separate applications • Separation of concerns into different microservices for all future functional requirements that are functionally or technically different from the current ones (e.g. Python-based data analytics backend

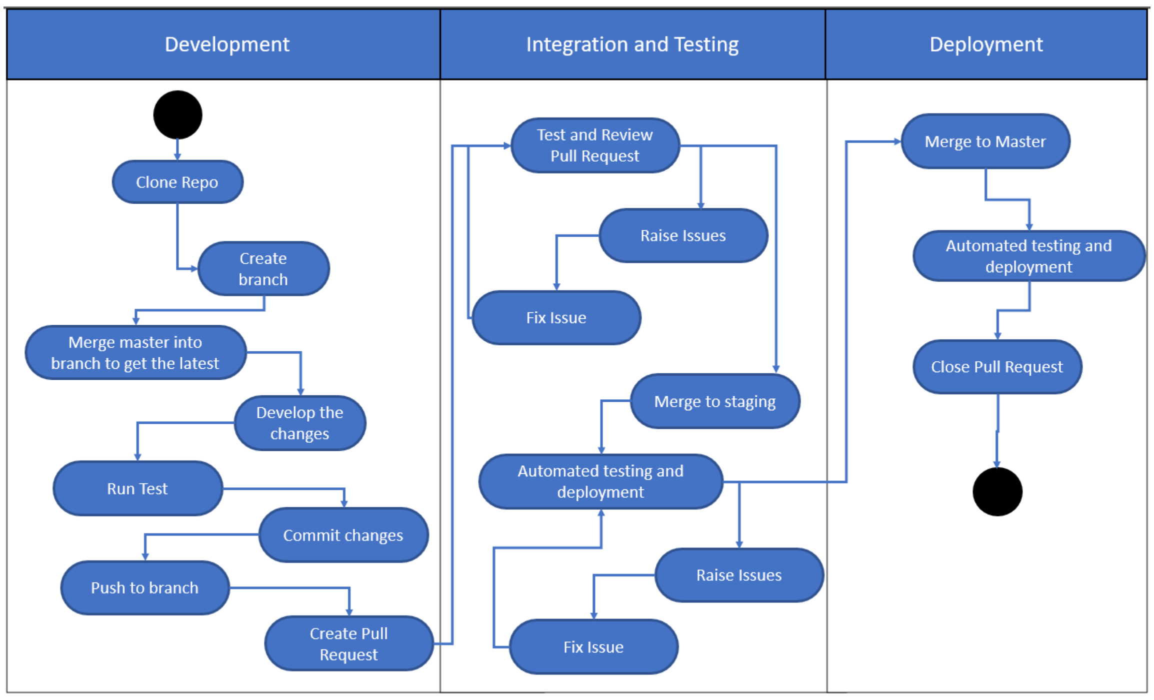
	microservice, Svelte-based admin frontend interface) • Multiple thin clients and thin servers, communicating by APIs • Containerize all microservices using Docker
Assumptions	• There is currently an intention to introduce a substantial number of microservices in the future to justify setting up a microservice architecture rather than a monolithic one
Alternatives	• Combine frontend and backend into one monolith (by serving the frontend static files via the backend) • Incorporate subsequent functional features within preexisting code • Use VMs instead of containerization
Justification	• If we were to introduce more features in future, each new microservice will be its own discrete entity - Scalability through modularity • Fault isolation present since if one microservice fails, the others will likely continue to work, unless there is tight data or control coupling between two microservices • Since complexity is localized within each microservice, development and debugging of large applications is optimized • With containerization of each microservice, employers can focus more on functional development and satisfying business cases, rather than spending resources on lower level concerns

Architectural Decision – Continuous Integration/Delivery	
ID	8.0
Issue	• Process of building and testing applications is not automated • Process of deploying application across environments and cloud providers is not automated • Time-to-market or rollout of new features is slow and unstructured • Susceptible to human error
Architectural Decision	• Use integrated Travis CI service to: o Automate building and testing processes o Automate deployment to different environments (e.g. staging, production) hosted on AWS
Assumptions	N/A
Alternatives	• Other CI/CD services such as CircleCI, Jenkins • Conduct building, testing, and deployment manually
Justification	• Comparing between alternatives, Travis CI is stated to take lesser time to get started, and development team is already acquainted with it • Automated and standardized building, testing, and deployment across different environments and cloud providers • Allows for rapid development and rollout of new features

Architectural Decision – Gradle Build Tool	
ID	9.0
Issue	• It is difficult to manage the different libraries and it's their different versions
Architectural Decision	By utilizing a build tool, it facilitates the deployment process and simplifies the management of external libraries
Assumptions	N/A
Alternatives	Maven, Apache Ant
Justification	Gradle offers faster performance as compared to other build tools available

Architectural Decision – Single Page Application (SPA)	
ID	10.0
Issue	• Slow web page response time when there are too many connections to the frontend server
Architectural Decision	• Use Vue.JS to serve Single Page Applications. While the initial page load might take a little longer, subsequent redirects will be much smoother.
Assumptions	User does not refresh the page and rely on Vue.JS routing capabilities to navigate through the pages
Alternatives	• Serve a single page each time a user requests for one (server-side rendered, multi-page application)
Justification	• The SPA design only requires 1 load from our backend server, greatly reducing the workload

Development View



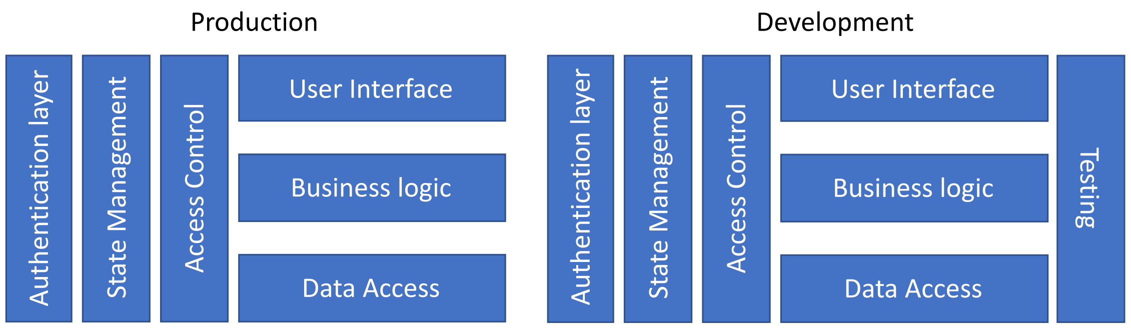
Our deployment strategy is split into 3 main phases: Development, Integration and Testing, and Deployment. Our development phase focuses on the developing of the new feature/bug fix, and first involves the user having to create a new branch after cloning the repository. To ensure that the user is working with the latest code, he will have to first merge the *master* branch with his working branch. After developing the changes and testing it locally, he will then commit his changes and push it to his branch. After which, he will then create a pull request and assign a reviewer to check the code.

In the Integration and Testing phase, the reviewer will then test the pull request that he is assigned to. If there are any issues, he will raise them to the developer who would then issue a fix. Once the review is approved, the branch is then merged to our staging branch. Our staging branch runs on our staging environment, and the codes are automatically tested and deployed using TravisCI. If there are any issue spotted, the reviewer and developer will both be notified so that a fix can be made.

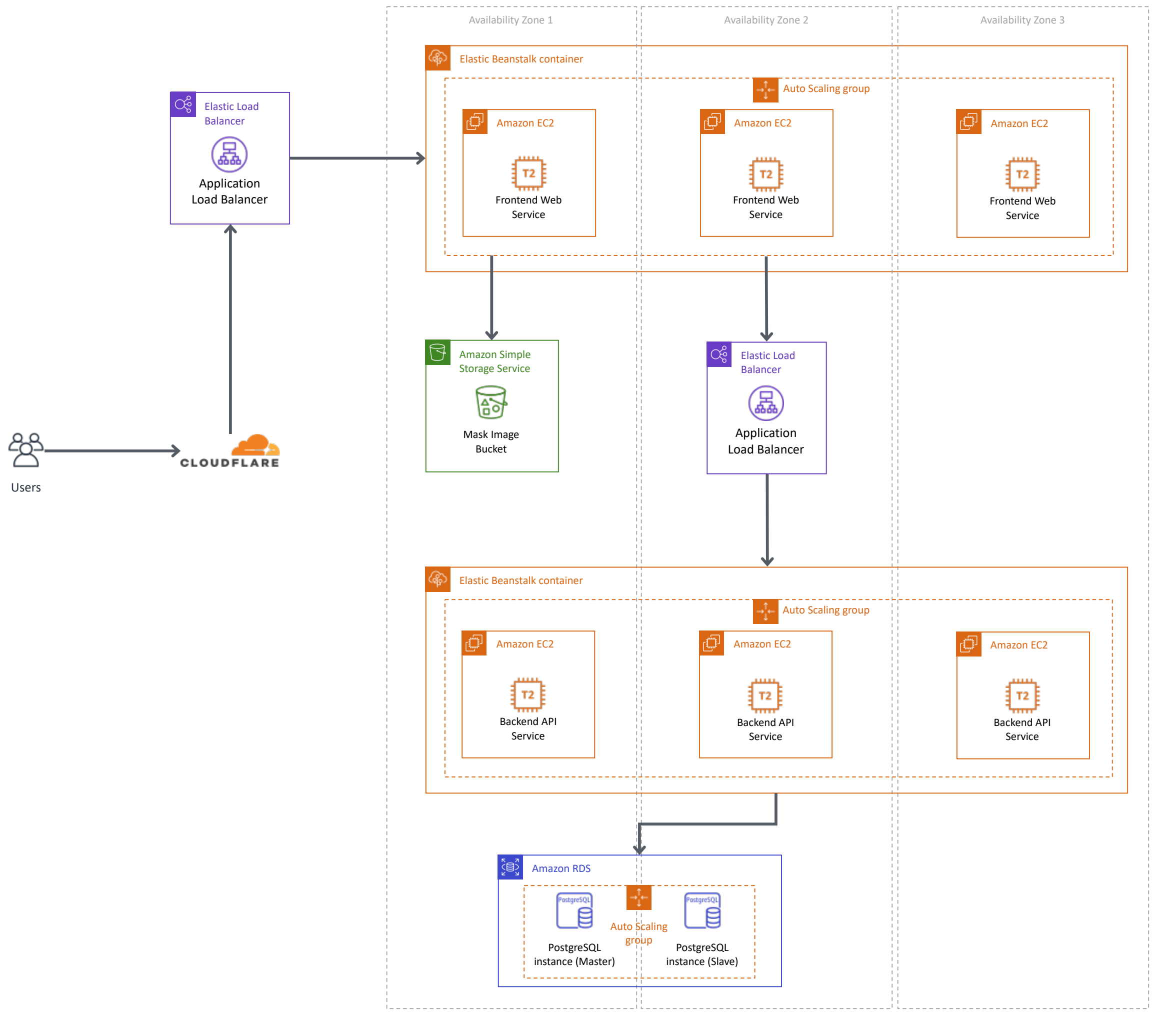
The final phase is our Deployment phase, where we will then merge our changes to the master branch. The production is similar to the staging environment, where TravisCI will run another set of test and automatically deploy the changes to the production environment. Once the pull request has been merged, it is then closed.

Solution View

Layered Diagram



AWS Deployment Diagram



Ease of Maintainability

Architectural Styles

Deploying our applications on AWS removes the need for us to maintain and handle our own physical infrastructure.

We separate our architecture into 3 logical layers, the User Interface, Business Logic and Data layer. Each higher layer relies on the services that the lower layers provide. Within each of the layers, we also have the authentication layer to verify the identity of the user, state management to ensure that the state is persistent and access control to ensure that only authorized users are allowed access. For our development environment, we also have the testing layer to ensure that each service is properly tested before they are deployed.

We use a tiered architecture style, between our backend systems and the client, we deployed Cloudflare to act as the proxy, providing protection and caching for our systems. In terms of security, we do not have to expose our endpoint directly to the end users, Cloudflare also offers service such as DDoS Protection in the event of an attack. The caching function of Cloudflare helps to reduce the workload our frontend servers have to handle, helping us serve cached pages instead of having to generate new web pages.

Microservices

Our architecture employs a micro-service architecture, decoupling our services from one another. This strategy allows us to make changes easily without affecting other services and facilitate testing and deployment efforts when we make changes to a service.

Design patterns

We use the Singleton design pattern for our database classes, allowing us to rely on a single instance to connect and do operations on the database. This pattern ensures efficient database operations, removing the time needed to re-establish a new database connection whenever there is a new request.

We also utilize the Façade design pattern, where we hide the complexities of a class from its caller. For instance, we create a controller class to handle the creation of the database connection. For the caller, they similar have to call the `getConnection()` method and do not have to worry about handling any of the credentials or error handling.

Multi-tenancy

Each of our services share the same database, and in the event of a database failure, we have an auto recovery process that will spin up a new database to replace the failed instance.

Integration Endpoints

Source System	Destination System	Protocol	Format	Communication Mode
Cloudflare	Elastic Load Balancer	HTTPS	HTML	Synchronous
Frontend Elastic Load Balancer	Frontend Web Service	HTTPS	HTML	Synchronous
Frontend Web Service	Backend Elastic Load Balancer	HTTPS	JSON	Synchronous
Backend Elastic Load Balancer	Backend API Service	HTTPS	JSON	Synchronous
Backend API Service	PostgreSQL	JDBC	SQL	Synchronous
Frontend Web Service	S3 Mask Image Bucket	HTTP	PNG	Synchronous

Hardware, Software and Services

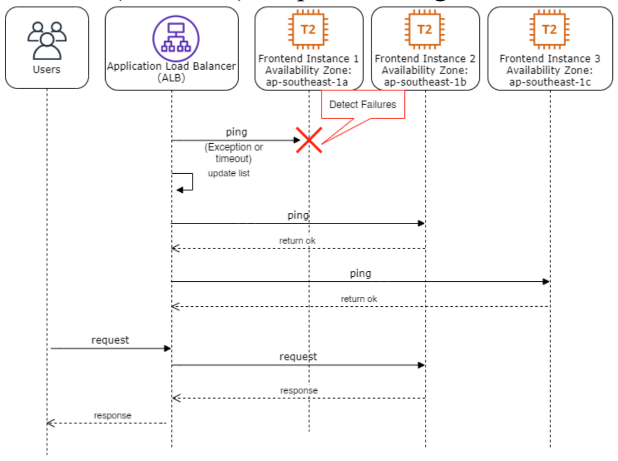
No.	Item	Quality	License	Buy / Lease	Cost (Optional)
1.	Cloudflare	N/A	Proprietary	Lease	Free
2.	AWS Elastic Load Balancer (ALB)	2	Proprietary	Lease	$0.0252 per ALB-hour (link)
3.	AWS Elastic Beanstalk	2	Proprietary	Lease	No additional charge
4.	AWS EC2 Instance (t2.micro)	6	Proprietary	Lease	$0.0192 per Hour (link)
5.	AWS S3 Bucket (Standard)	1	Proprietary	Lease	$0.025 per GB (link)
6.	AWS RDS for PostgreSQL (Multi-AZ, t2.mirco)	2	Proprietary	Lease	$0.056 per hour (link)
7.	Travis CI	N/A	Proprietary	Open Source – MIT License	N/A
8.	Apache Tomcat	N/A	N/A	Open Source – Apache License 2.0	N/A

9.	Gradle	N/A	N/A	Open Source – Apache License 2.0	N/A
10.	Spring	N/A	N/A	Open Source – Apache License 2.0	N/A
11.	Nuxt.js	N/A	N/A	MIT License	N/A
12.	Vue.js	N/A	N/A	MIT License	N/A
13.	Npm	N/A	N/A	Artistic License 2.0	N/A

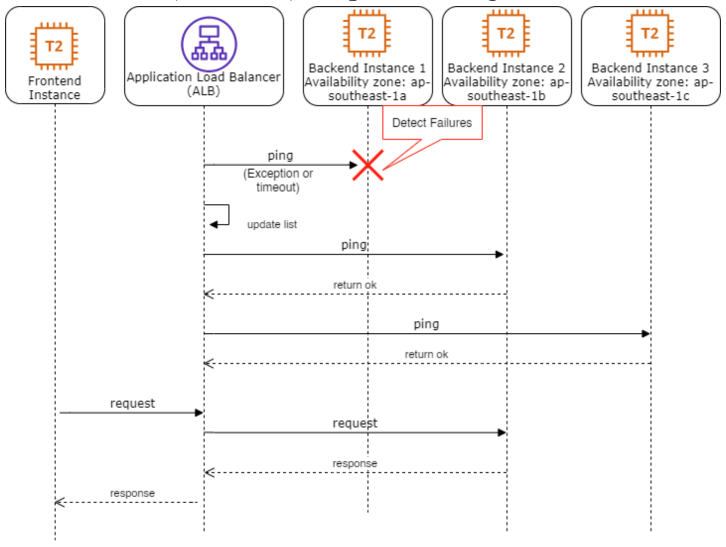
Availability View

Node	Redundancy	Clustering			Replication			
		Node Config.	Failure Detection	Fail Over	Replication Type	Session State Storage	DB Replication Configuration	Replication Mode
Frontend Web Service (AWS EC2)	Horizontal Scaling & Vertical Scaling	Active-Active & Active-Passive	Ping	Load-Balancer	N/A	N/A	N/A	N/A
Backend API Service (AWS EC2)	Horizontal Scaling & Vertical Scaling	Active-Active & Active-Passive	Ping	Load-Balancer	N/A	N/A	N/A	N/A
AWS PostgreSQL	Vertical Scaling	Active-Passive	Ping	Load-Balancer	Database	Database	Master-Slave	Synchronous

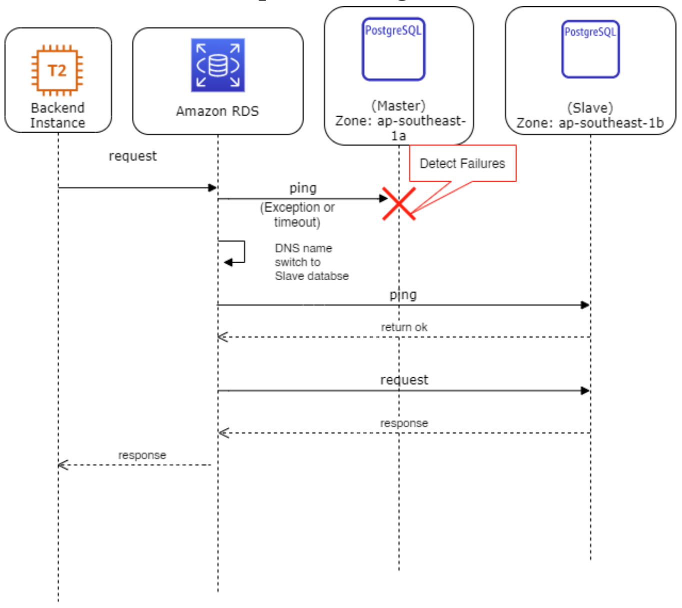
AWS EC2 (Frontend) Sequence Diagram for failover


Users
Application Load Balancer (ALB)
T2
Frontend Instance 1 Availability Zone: ap-southeast-1a
T2
Frontend Instance 2 Availability Zone: ap-southeast-1b
T2
Frontend Instance 3 Availability Zone: ap-southeast-1c
Detect Failures
ping
(Exception or timeout)
update list
ping
return ok
ping
return ok
request
request
response
response

AWS EC2 (Backend) Sequence Diagram for failover


T2
Frontend Instance
Application Load Balancer (ALB)
T2
Backend Instance 1 Availability zone: ap-southeast-1a
T2
Backend Instance 2 Availability zone: ap-southeast-1b
T2
Backend Instance 3 Availability zone: ap-southeast-1c
Detect Failures
ping
(Exception or timeout)
update list
ping
return ok
ping
return ok
request
request
response
response

AWS RDS Sequence Diagram for failover



Security View

No	Asset / Asset Group	Potential Threat / Vulnerability Pair	Possible Mitigation Controls
1.	Servers (Hardware)	Denial of Service Attack (Availability)	Enable DDoS protection in Cloudflare (Implemented)
2.	Data in transit through HTTP (Data)	Man-in-the-middle attack (Confidentiality, Integrity)	Use HTTPS to encrypt data to ensure that it is not compromised during transit (Implemented)
3.	Backend APIs (Service)	Unauthorised access to backend API (Confidentiality)	JSON Web Token (JWT) to protect ensure only authorized user can access the endpoints (Implemented) CORS protection to only allow access from trusted source (Implemented)
4.	Data storage in database (Database)	SQL Injection (Integrity, Availability, Confidentiality)	Prepared statement on the backend to encode potentially malicious queries (Implemented) Hash passwords (Implemented)
5.	Frontend Web Service (Client)	Cross-site scripting (Integrity)	Enable Content-Security Policy (Implemented)
6.	State stored in local storage (Client)	Unauthorised access to state data stored in browser local storage (Confidentiality)	Encrypt the data stored (Implemented)
7.	Frontend Web Service (Client)	Cross-site request forgery (Integrity)	CSRF token for all requests (not implemented)

Performance View

No.	Description of The Strategy	Justification	Performance Testing (Optional)
1.	Static website caching (Frontend)	When there are too many requests, it is very resource intensive to serve a new page for every request. By caching it on our proxy server, the cached webpage can be served instead, saving up previous resource.	Before (Without proxy): 2829 requests, **0.29 secs** After (With proxy): 2666 requests, **0.16 secs**
2.	Two frontend instances with load balancing	When there are too many requests to our frontend server, one server may not be able to handle the load and performance may degrade as a result.	Before (1 instance): 2829 requests, **0.29 secs** After (2 instances): 4267 requests, **0.22 secs**
3.	Two Backend instances with load balancing	When there are too many requests to our backend server, one server may not be able to handle the load and performance may degrade as a result. For this test, although the 2 instances performed poorer, we suspect it might be because the 2nd server may be situated further away. And since the initial response time is already very fast, any improvement from the additional instance would be less obvious in this case.	Before (1 instance): 2354 requests, **0.01 secs** After (2 instances): 2666 requests, **0.02 secs**
4	CPU Capacity Threshold before significant performance degradation	This test allows us to identify the CPU utilization rate when performance starts to degrade. This would allow us to set the correct threshold to spin up new instances when required to offload the demand and improve performance. For our baseline, we determine performance degradation when the response time takes more than 1 second as defined in our performance quality attribute.	**1.14 secs** response time at 26.925% CPU utilization. Based on this result, we set a threshold to spin up a new instance when CPU utilisation >25%.

5	Full infrastructure setup	This setup allows us to determine if our design is able to meet the quality attributes that we have defined. Using Cloudflare as our proxy server, 2 active instance and 1 passive instance, we then simulated a worst case scenario on our system to see if our system can handle it.	**1.06 secs** response time for 19670 requests from 100 users within 215 seconds.