CS301 : IT Solution Architecture
AY 2019-2020 Term 1

Final Report

The Runtime Terrors

Amos Lam Guo Lun	S9618653C
Gabriel Koh Zhe Ming	S9439850I
Ng Rui Qin	S9406141E
Lin Han Hui	S9372470D
Liu Zuo Lin	S9648544A
Truong Hai Bang	G1205746L

Instructor: ***Prof. Ouh Eng Lieh***

Background and Business Needs

Modern universities often use Student Management Systems (SMS) to automate previously manual tasks related to education and administration. The SMS concerns three main actors: students, teachers and administrators, but for this application, we will be focusing mainly on the functionalities for only students.

Students can use SMS for many different purposes - paying school fees, checking their enrolled classes or taking attendance etc. As SMS contains confidential information like payment details, users must be authenticated to use it. As SMS provides students, teachers and administrators with multiple key educational and administrative functionalities, its services need to be fast, well-performing and available 24/7. This holds especially true during peak periods such as checking of examination results, and periods nearing payment deadlines.

Stakeholders

Stakeholder	**Stakeholder Description**
Students	University students are the primary users of this application. They use this application frequently to manage their academic-related and administrative activities, ranging from checking enrolled courses, taking attendance, paying fees etc.
Teaching Staff	Teachers are also considered primary users of this application. They use SMS to disseminate announcements, as well as release the students' academic results.
System Administrators	System administrators, the secondary users of SMS, provide administrative support and configuration for all stakeholders, as well as creating semesters, courses and classes.

For the scope of this project, we will be focusing on only students and their use cases.

Key Use Cases

Log in	
Use Case ID	1
Description	Students need to log in to their own unique accounts (tied to their student_ids) to use SMS. This is to ensure confidentiality and integrity of their information
Actors	Students
Main Flow of events	Student logs in by keying in their student email and password. After authentication, they are automatically brought to the home page of SMS.
Alternative Flow of events	If a student entered invalid credentials, they will be unable to access the school system, and are redirected back into the login page that indicates an error message. The student can either choose to login again, or reset his/her password.
Pre-conditions	The student must be registered by a system administrator in the student portal.

Post-conditions	None

View Enrolled Classes	
Use Case ID	2
Description	Students can view the classes that they are enrolled in for the current semester. Each student typically enrolls in 4-5 courses before the start of the term. Students can hence use this feature to check what courses they are enrolled in.
Actors	Student
Main Flow of events	Having logged into the SMS, the student should be able to navigate to a tab which shows his list of enrolled courses for the semester.
Alternative Flow of events	Student does not have any enrolled classes, and the system will show the reason, such as vacation or leave of absence.
Pre-conditions	Student must be logged into SMS and previously enrolled into the classes
Post-conditions	None

Take Attendance	
Use Case ID	3
Description	In each class, students are expected to take their own attendance using SMS, so that other stakeholders can track their attendance in real time.
Actors	Students
Main Flow of events	Student navigates to his course enrollment, clicks on the class that he is currently in, and proceeds to click on the “mark attendance” to mark their attendance.
Alternative Flow of events	If the week that the student marks attendance is not currently ongoing, the student cannot mark attendance successfully, and an error message will show.
Pre-conditions	The student must be logged in to SMS. The class session the student is marking attendance for must be ongoing at the moment.
Post-conditions	The student should see a “attendance marked” message after marking their attendance successfully.

Pay School Fees	
Use Case ID	4
Description	Students can pay their school fees for the semester using SMS. The student portal integrates a third party API (Paypal) for this purpose.
Actors	Students
Main Flow of events	The student clicks on “pay school fees”, and is redirected to the payment page, after which they will complete the steps and pay through PayPal API.
Alternative Flow of events	If the student has insufficient funds in his bank, the payment will be rejected and an error message will be shown.
Pre-conditions	Student should be logged in to SMS and navigate to the payment page. The student should have sufficient funds in his account prior to payment.
Post-conditions	Students should see a success message stating that their payment was successful.

Key Architectural Decisions

Architectural Decision: Client/Server and Microservices architecture	
ID	1
Issue	SMS needs to be easily scalable and maintainable, being a school portal, as many more functions might need to be implemented according to growing business needs. ISO25010 Considerations: Maintainability
Architectural Decision	We implemented SMS using a microservice architecture - a frontend application using Angular communicating with 3 microservices - Module, Payment and Student - all implemented in Springboot. These microservices provide the frontend with the data, and the frontend displays it on the user’s browser.
Assumptions	We assume that the SMS’s user base is large enough to justify breaking its functionalities into separate deployments.
Alternatives	A monolithic application is another option that can be used if we wish to speed up development time. However, in the long run, such an architecture would be less maintainable and harder to manage compared to a microservice architecture.
Justification	Microservices are a collection of services that are highly maintainable and testable, loosely coupled, independently deployable and organized around business capabilities. We chose to implement our application using this as it allows us to work on the different parts independently - so there would be less of a hassle to coordinate between developers during the development process.

Architectural Decision: Implementing an API Gateway	
ID	2
Issue	As SMS scales and more microservices needs to be implemented, there eventually might be too many services, leading to a messy and hard-to-manage spaghetti architecture. Also, we might want to provide different access rights to different user privileges E.g. PUT student data only allowed by admin, GET allowed by student. Similarly, we might need to do the same on the service level. In addition, for security purposes, we want to hide the microservices' endpoints from the public. IS25010 Considerations: Security, Modifiability, Maintainability, Portability
Architectural Decision	Because of the aforementioned issues, we needed to hide certain HTTP methods/services, and to provide a high level and client facing interface that is unified, thus, connecting all the underlying microservices. This had to be done with the IS25010 considerations above. We decided to implement an API Gateway as a single entry point for our client, thus adopting elements of the Facade design pattern. With the API Gateway, security is ensured through HTTPS, as well as through the integration with AWS Cognito, a user management and authentication service. Furthermore, configuration could be done on the gateway layer to give/prevent user access on HTTP command layer. Microservices that should not be given access to clients can be hidden by not exposing them through the API gateway. With the API Gateway, complexities amongst the microservices are abstracted and decoupled, thus, enabling ease of modifiability, maintainability and portability of the underlying microservices.
Assumptions	Number of services will increase as features are being built on SMS
Alternatives	TIBCO - Enterprise Service Bus
Justification	An API Gateway was chosen over TIBCO due to two main reasons: - It has all the features of an ESB, and additional features such as user access restrictions and authentication - It is easier to integrate two systems from the same service (AWS) as compared to two different service providers (AWS and TIBCO)

Architectural Decision: Using S3 + CloudFront	
ID	3
Issue	As access to SMS might not be constrained to Singapore only, there might be cases where students use the application from overseas (Overseas Exchange, Business Study Mission, Vacation, etc). It was key to provide access to SMS at high speeds, from around the world. IS25010 Considerations: Performance Efficiency, Security, Availability Other Considerations: Caching, CDN services
Architectural Decision	The team decided to deploy the frontend assets to Amazon S3 and CloudFront. Our frontend application is built on Angular 7, and after provisioning the application, all assets are considered to be static. Since all assets are static, we could deploy them to S3 easily, which is simply an object storage service. Furthermore, S3 claims to provide 99.999999999% durability, which is also

	one of the main motivations behind adopting S3. Another reason behind adopting S3 would be its compatibility with CloudFront Content Delivery Network service by AWS. With CloudFront, we could easily secure our application with HTTPs and SSL encryption. Furthermore, CloudFront comes with AWS Shield, which protects our website against threats such as DDoS attacks.Concerning performance, It provides performant speeds as well due to its CDN services (E.g. Access from Malaysia hits Malaysia's data center instead of Singapore).
Alternatives	Deploying traditionally, either through a physical server/cloud server like EC2
Justification	The benefits of S3 + CloudFront outweighs traditional deployment. Traditional deployment would be cumbersome and unnecessary (Server setup, SSL encryption, dependency management, node.js server setup). Besides providing a whole suite of services, S3 + CloudFront also provide that sort of extensibility, if the team were to implement a new feature such as a Serverless function (AWS Lambda), it could be easily integrated with S3 + CloudFront.

Architectural Decision: Using AWS ECS Fargate - Containerization	
ID	4
Issue	As any IT application, application life-cycle management matters. To be able to roll out changes made by our developers smoothly from development to production, we needed to look at capabilities such as continuous integration and continuous delivery. Containerization is one such technology that supports the idea. Also, containerization fits a microservice architecture to ease deployment.
Architectural Decision	The team has decided to deploy SMS on AWS's cloud infrastructure. In particular, we chose AWS ECS fargate, which is a compute engine for Amazon ECS that allows us to run containers seamlessly. By providing the right configuration, we can provision and scale clusters of virtual machines to run containers. Hence, the general idea is that our CI-CD will build a docker container image within Amazon's Elastic Container Registry (AWS's version of Docker Hub) and we can easily pull these images to create microservices in Fargate.
Alternatives	Spawn our own EC2 instances and manage them individually to start up containers that run our microservices images from Docker Hub.
Justification	Using AWS' platform allows us to perform easier deployment, management, scaling, capacity provisioning and load balancing. This is also a perfect opportunity for us to explore the suite of services AWS has to offer and test out our IT solution architecture plans. Given a choice between setting up EC2 instances and using ECS Fargate, we decided that ECS fargate was more practical. With EC2 instances, we will have to scale, monitor, patch and secure the instances ourselves. These tasks each present their own complexities that our team would have to spend time learning. Whereas, with Fargate, most of the tasks are automated for us with the necessary configurations set on our end. Also, the team can package our application in containers, specify the CPU and memory requirements, define networking and IAM policies and launch the application. This makes deployment of containerized microservices easier in a clustered environment.

Architectural Decision: Autoscaling of Services using Fargate	
ID	5
Issue	Frequency of usage of SMS by the students differ across the day. SMS can expect most request hits to come in during the school timings and important timings such as when results are released. However, there are also times when requests are low such as at night into the wee hours. Then, is it practical and cost efficient to keep our services running at maximum capacity all the time? Also, how can we estimate the exact number of services we can scale up to account for the peak periods? IS25010 Considerations: **Availability**
Architectural Decision	The team has decided to configure our AWS fargate to use Auto Scaling to adjust our desired number of microservices count up or down automatically. This adjustment is based on the number of request counts that hits our Application Load Balancer (ALB) towards a particular microservice. Currently in our architecture, the services are set to scale up whenever they receive more than 10 requests from the ALB. This low request setting is done for easier demonstration during our presentation.
Alternatives	Manually shutdown additional redundant services during non-peak periods.
Justification	Auto-scaling is a preferred choice for the team as it is automated and can therefore give better guarantees on availability. If we were to manually shut down unused services ourselves, we might not be able to account for a large spike in request during unprecedented hours. Hence, Auto-scaling overall provides better availability and cost management as we are able to dynamically increase and decrease capacity as needed. This helps us, as students, to reduce our billing as well on AWS.

Architectural Decision: Implementing a Redis Cache	
ID	6
Issue	Every weekday in a semester, students go to the SMS main page for a variety of reasons. On the main page, the student's module information is initially retrieved from the Module service. This can be taxing for the database if a large number of students are accessing it at the same time. As such, we felt using a redis cache in this use case can improve the speed at which the main page loads and at the same time reduce the workload on the module microservice. IS25010 Considerations: Performance efficiency
Architectural Decision	We implemented a flask microservice connecting to a redis cache and deployed it to an ubuntu ec2 instance on aws. We implemented this only for use case 2 - view enrolled classes. On a normal school day, most students would have to view their class details for new content and updates. As such, we feel that high performance is needed in this area. Hence, when a student(client) requests for his classlist, the backend first checks if the student's key is in the redis cache. If it is, the student's classes are returned. Else, the backend makes an api call to the module service to retrieve the data, which is then inserted into the redis cache for faster future retrieval.
Assumptions	Our key assumption is that our redis server will have sufficient memory to accommodate the data of all the students using the student portal. Since redis is an in-memory database, the data is stored on RAM, and hence the total storage cannot be larger than the server's

	memory. Due to high costs, we have decided to use a free-tier aws ec2 ubuntu instance. Thus, the total storage might be insufficient, but given sufficient funds, we would have opted for a specialized redis server with sufficient memory.
Alternatives	Alternative to the redis cache includes memcached - another multi-threaded in-memory database for use on a smaller scale. Unlike memcached, redis has persistent data storage, meaning that the data stored on redis will remain there till it is removed, which is required for our use case, as we are expecting the module data to last across the entire semester unless there are changes to the student's enrolled modules. In addition, unlike memcached, redis supports replication. In the event that we wish to scale and add redundancy to our redis cache in the near future, we will be able to do so. Thus, we chose redis over memcached due to these advantages that redis presents over memcached.
Justification	Redis is an in-memory database - read and write speed is hence much faster than conventional databases, but the limitation is that total storage space taken by the data cannot be larger than the instance memory. We can safely assume that most students do not change their modules during the semester. As such, the module data that the student retrieves following the log-in procedure remains the same throughout. As such, using a redis cache to retrieve this mostly static data will improve performance and speed at which the student's module data loads following log-in.

Architectural Decision: Implementing a Dedicated User Management/Authentication Service	
ID	7
Issue	From a functional perspective, clients (educational institutions) of SMS prefer a one stop solution and fuss free experience when it comes to authentication and user management service. Because of this, we needed a system that is easily customizable, yet well integrated with the existing infrastructure. On a separate note, as SMS serves students, instructors and admin alike, proper user access control is also required. From a technical perspective, we would require this system to support oAuth2 and easy integration with commonly used applications such as Facebook and Google, as this Single-Sign-On feature is largely popular amongst applications in such enterprise/school context. Without saying, this module also has to be highly secure. IS25010 Considerations: Security, Compatibility, Maintainability
Architectural Decision	Based on the aforementioned requirements, we have decided to opt with Amazon Cognito. Cognito offers a great deal of customization, along with easy integration with our AWS based system infrastructure. It also offers integration with Facebook and Google with Oauth2 standards, thus, providing SSO that might potentially be used in the future. Cognito also supports Multi-Factor Authentication (MFA), which is a requirement for most systems today, thus, beefing up security on the system itself. With regard to role based access, Cognito also provides this feature as well, and can scale to millions of users.
Alternatives	Build your own user management/authentication module
Justification	There is minute value-add into writing a basic function from scratch, especially one that requires quite a number of non-functional requirements. Hence it is more practical in terms of effort to opt for an out-of-the-box solution and configure ourselves. Likelihood, the out-of-the-box solution is more robust and functional.

Architectural Decision: PayPal Integration for Payment Service	
ID	8
Issue	Implementing a payment system from scratch can be tedious, with many steps and protocols to adhere to. Implementing a third-party solution, in this case, is easier. ISO25010 Considerations: Usability, Security
Architectural Decision	We require a secure, end-to-end payment service provider that accepts different forms of payment in order to cater to the various modes of payment that students have. Furthermore, the provider should be able to integrate into the payment system. PayPal is able to satisfy these criteria. The PayPal API is invoked when the make payment web service is called, returning a URL for the user to direct to PayPal. The user is then redirected to PayPal's website to make the payment, and depending on whether it is completed or cancelled, the user will be redirected to the corresponding link in Student Management Service. If the user made a payment, the update payment web service is called, where the payment system will check the payment status with PayPal and update the database accordingly.
Assumptions	We have to assume that students at least have a PayPal account, a debit card or a credit card. This is important as payment through PayPal has to be done through one of these modes. Should the student decide to pay through other mediums, they will not be able to do so in Student Management System.
Alternatives	Other than PayPal, other payment services can be implemented for the student to pay, such as Stripe, Visa and MasterCard.
Justification	PayPal is an established third-party that can securely handle payments, and is able to provide developer tools to integrate with Student Management System. We created an email account to create a developer account in PayPal.

Architectural Decision: Vertical Scaling Along with Application Load Balancer	
ID	9
Issue	For the SMS, it is important that there is high availability and performance during school days. Without high availability, students may not be able to access important services that they need to attend classes. An example will be looking at their class venues. Performance-wise, students should be able to access their services quickly since their objective is to focus on learning and we should not let them be bogged down by under performing IT systems. Hence, there is a need for us to set up redundancies and the relevant failover mechanisms. IS25010 Considerations: Availability, Performance Efficiency
Architectural Decision	We will be horizontally scaling our microservices across separate machines offered by AWS. To perform failover on these redundancies, an application load balancer offered by AWS is used to direct traffic to our microservices. There are various reasons for why we chose to use a load balancer: 1) High availability can be achieved where in the event if one instance of a service fail, the load balancer is smart enough to route requests to other running instances. This helps to prevent downtime and essentially prevents single point of failure. 2) Also, each service would be less congested with requests at peak period allowing it to serve responses faster and thereby improve its performance.

	3) The load balancer can allow us to not expose our services to the Internet by sitting in front of our service tiers. This would enable our frontend website code to communicate with a single DNS name while our backend service would be free to elastically scale in-and-out based on demand or if failures occur and new containers need to be provisioned.
Alternatives	Vertical scaling and DNS failover
Justification	With vertical scaling, there is a higher risk in service interruptions due to a single point of failure which leads to lower availability as compared to horizontal scaling (one server vs many servers). Furthermore, scalability and flexibility is pretty limited for vertical scaling. When network requests increase (seasonal or increase in number of users), a particular service might become sluggish, and there is a need to increase the resources to support the influx of network requests. In such a situation, vertical-scaling is limited to the capacity of a single machine, scaling beyond that capacity can lead to downtime. Whereas for horizontal scaling, it is easier to scale dynamically by adding more machines into the existing cluster. DNS failover was considered an alternative to a load balancer as the failover mechanism. However, it is a less preferred approach simply because of its limitation. DNS mapping records is commonly stored in a local cache along the DNS resolution path, and users will continue to refer to the IP address until the Time to Live (TTL) for the host's DNS record expires. What this means is that the failover is not immediate and this reduces availability.

Architectural Decision: Database Replication (Master-Slave Configuration)	
ID	10
Issue	There is a requirement for 24/7 access to SMS, as students have to access the system for the aforementioned purposes. Thus, databases have to have high availability to provide data to the respective systems IS25010 Considerations: Availability
Architectural Decision	For each of our database, redundancy and seamless failover process has been incorporated, through a one-way database replication, following a master-slave configuration. The slave database is located in another geographical zone, With some sort of database replication, a failover management is done automatically, which means that the slave would take over the master database in the event of a failure that has occurred in the master. A one-way master-slave configuration is used to reduce
Assumptions	AWS is highly available and availability zones are not likely to all fail at one go
Alternatives	Two-way database replication, file transfer, caching
Justification	Two-way database replication requires databases to be sync at the start, and maintenance would be harder as compared to one-way database replication, in our case, the latter would help to achieve our requirements, thus, there is no need to implement the former. There is no legacy system/third-party involved, thus, there is no value-add for file transfer protocol if we can use database replication. Caching is not viable here, as we would need to persist the data stored in SMS.

Architectural Decision: Database as a Service (DBaaS) with AWS Relational Database Service	
ID	11
Issue	Setting up multiple databases for each microservices, across different universities might be very complicated to track, not to mention, costly. There needs to be a more efficient, as well as cost-effective way to provision a database.
Architectural Decision	The team decided to use AWS Relational Database Service (RDS), a DBaaS product, instead of a traditional database, for all of our backend microservices. As the team's architecture is mostly microservices oriented, we need an efficient and scalable way for setting up of our database. RDS DBaaS allows that sort of fuse free configuration and administration. Computational power and capacity can also be scaled dynamically, unlike a traditional database setup, where capacity has to be estimated upfront. In the latter, unnecessary procurement of capacity and computing power might bring costs up unnecessary. An example would be in the case of an increasing trend of student enrolment for SMS, this would mean more capacity is required, and since more database calls would be made, computing power requirements would increase too. With RDS, the administrator can easily scale up the requirements, and scale down in the opposite case. Setting up database availability and redundancy would also be easier when using a RDS, backups, master-slave, master-master configuration can be done easily as it is a feature that is provided out-of-the-box by RDS. Considering these concepts in a non-DBaaS service would be more cumbersome, as databases has so be mainly setup either at the university's premises or manually bootstrapped on a vanilla cloud server. IS25010 Considerations: Maintainability, Security, Reliability
Assumptions	It is required to store data across universities in separate databases, in other words, database information for University A's SMS should be isolated from database information for University B's SMS. This is to prevent contamination of data, and also for security reasons. Because of this, setting up of databases is a frequent task for the team, to be able to setup the database easily would be ideal. AWS services are generally highly available,
Alternatives	Using a traditional database setup for each of the universities using SMS.
Justification	For every university, a dedicated instance of SMS has to be setup. A dedicated instance would mean ***n*** number of database for each of the microservices. A traditional database setup would mean more resources needed for acquiring licenses and servers, as well as for hiring expertise in database management, which would be very costly. Opting for a DBaaS might lead to lesser control as compared to a traditional approach. However, in our case, there is no need for a lot of database customisation, thus, the benefits of DBaaS outweighs its limitations in our case.

Architectural Decision: Pre-fetch	
ID	12
Issue	Students are required to take their attendance every weekday when they have class. To do this, they need to click on their class on the main page, which redirects them to the attendance page

	and retrieves the student's attendance data from the Module service. If many students try this at the same time, this might overload the Module service, and students might need to wait for a slightly longer time before they are able to view their attendance data. As such, to prevent this and to enable the student to view their attendance data more quickly, we have decided to include a pre-fetch of the student's attendance data. IS25010 Considerations: Performance efficiency
Architectural Decision	When the user logs in and goes into the main page, we assume that the student will eventually want to view his attendance page and take his/her attendance. As such, we implemented a pre-fetch here - where the frontend fetches the student's attendance data along with the modules data when the student logs in and stores this data in the client's (student's) local storage. When the student wishes to view his attendance data, there is hence no need to call the corresponding microservice again (ModuleService), since this data is already on the student's local storage, This enables the attendance data to load much faster for the student also, as no calls are made to the Module Service.
Assumptions	We are assuming the student will navigate to the "take attendance" page from the "view class" to take their attendance. If this assumption fails, we would have made an api call to the module service for nothing.
Alternatives	Nil
Justification	When the student navigates from the "view class" page to the "take attendance" page, the data, being already pre-fetched, will hence load faster. As such, this pre-fetching improves performance when the student navigates to the "take attendance" page and reduces the loading time between the pages.

Architectural Decision: Payment Builder	
ID	13
Issue	As of now, in our Payment service, we make use of a PaymentRecord class to store data relating to payment records. This class has multiple constructors such as studentId and annualYear, as well as constructors that can be further broken down, such as semester. Using a raw PaymentRecords class might cause confusion for new developers reading the code due to the complicated nature of the class constructors. IS25010 considerations: Maintainability
Architectural Decision	As such, we have decided to implement the builder design pattern for the PaymentRecord class, in order to encapsulate the creation and assembly of the more complicated parts of the constructor to a separate builder object.
Alternatives	One alternative would be to not use this design pattern at all, and instead build the entire object on its own to improve runtime slightly. However, we felt that the improvement in code clarity and readability that comes with the builder design pattern was significantly more valuable than the slight improvement in runtime, so we still chose to make use of this builder pattern.
Justification	A builder class delegates object creation to a Builder object instead of creating the objects directly in order to avoid having the developer create a complicated Payment object all at once. This enables our code base to be more maintainable and readable, as newer developers will likely have an easier time understanding and implementing

Development View

As a team, we envisioned a development flow that is well organized and efficient. We decided to adopt several practices from Agile, Scrum and CI/CD, with a high level of focus on versioning, testing, automation and agility. Through this, we were able to ship quality code within a short time cycle, despite our distributed workflows.

Versioning:
The team employed code versioning on multiple levels. Being able to switch between different versions of our application allowed us to maintain our applications easily and deploy different versions of the application (hotfix, production, staging, development, country, organisation specific, etc). This also helped us track potential errors as we could easily revert our erroneous code to a working version.

On the source code level, we decided to adopt a branching model with three main branches; master, develop and features. Feature branching involves having a dedicated branch for each feature. Once a feature is ready to be merged, a pull request is to be made, following which a review would be done by other developers before being merged into our develop branch. After which, when ready, a repository merge can be made from develop to master, which reflects our UAT/Production environment. With multiple deployment environments, we were able to test our applications rigorously. Across all branches, we frequently committed to our GitHub repository, thus enabling ease of tracking, as well as frequent integration.

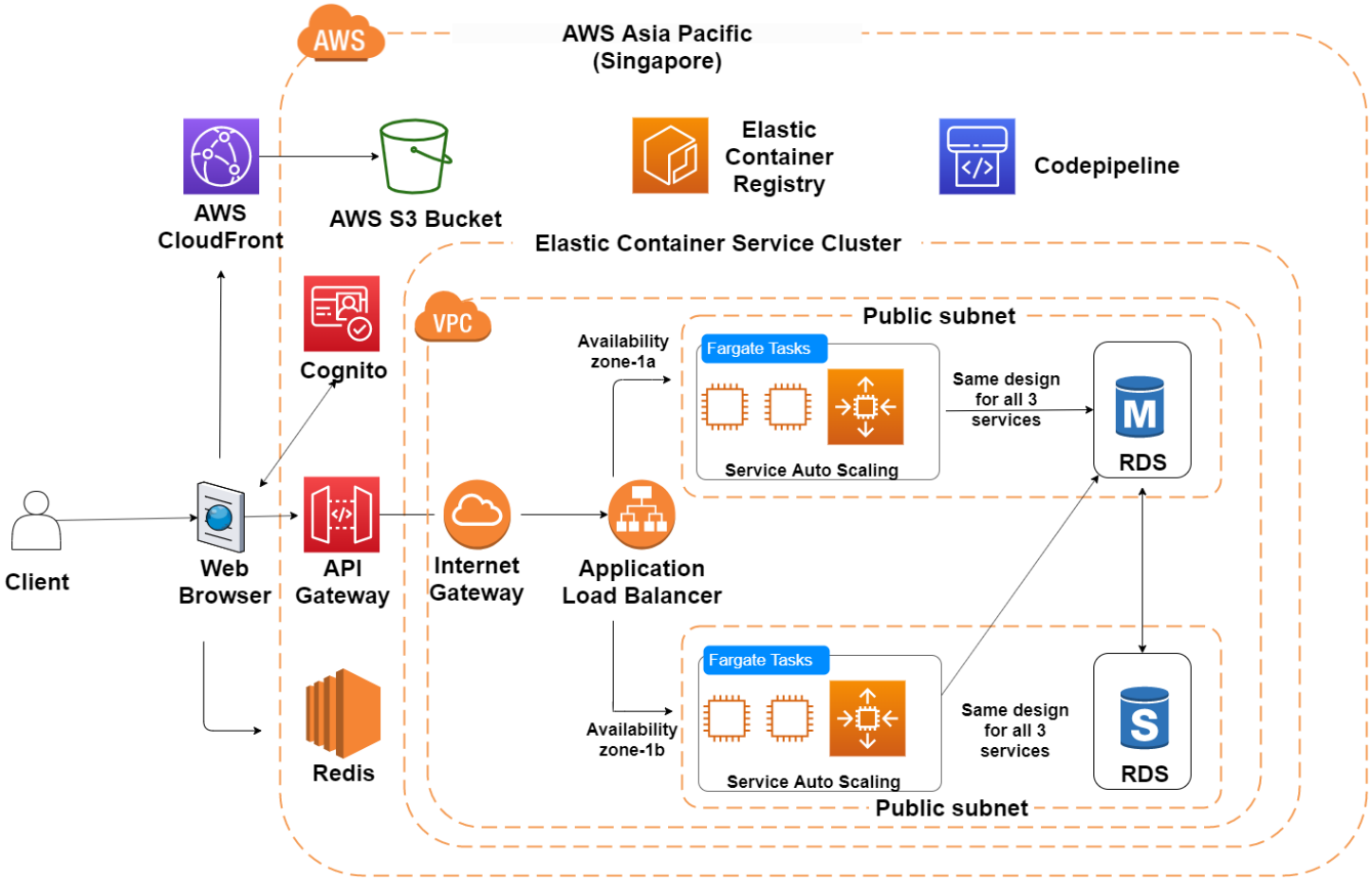
On the application image level, we were able to track and version all our microservices, as we were able to tag each of them This was made possible with the help of Docker and containerization.

Testing:
The importance of testing is manyfolds, testing not only helps us to validate the functional suitability of our application, but also acts as a safeguard to for detecting buggy code, preventing erroneous deployments to production. We employed three testing strategies. Unit tests were written for each microservice and for our frontend application, enabling testing in isolation. Integration testing was used to test how well the services integrated together. Finally, we used regression testing to validate the correctness and completeness of our software components as our code base changes over time to match changing business requirements.

Automation:
We wanted to focus more on development and architecture and less on the mundane tasks, and agreed that automation was key. Automation was engineered throughout the stages of our workflow, from development, versioning, provisioning, testing and finally to deployment. With the help of TravisCI, AWS CodePipeline and GitHub integration, a single git pull request would enable most of the aforementioned tasks to be automated. The clearer illustration is documented in appendix figure 1 and 2.

Agility:
Our development workflow, coupled with our frequent communication, kept all of us in sync despite our distributed working environment and different schedules.

Solution View



The infrastructure diagram above shows the cloud architecture we have on AWS (outsourced Private Cloud) that supports the SMS system. By out-sourcing our application deployment environment on AWS, we are able to free ourselves from the need to handle the underlying complexities of networking, storage, servers and virtualization. As a result, we can focus on doing the Proof-Of-Concept for our infrastructure architecture deployment.

As shown in the diagram above, our applications are deployed within AWS Singapore region where we are allocated a virtual network (VPC) specific to our team to deploy all of our services. Within the VPC, we are allocated to availability zones which are distinct locations that are engineered to be isolated from failures in other Availability Zones. Within each zone, we have one public subnet to host our services and corresponding redundancies.

For microservices deployment, we have chosen to use AWS's ECS Fargate for deployment. Reasons as to why we used fargate, refer to **Architectural Decision ID 4**. We define our microservice applications in ECS through ***task definitions***. They act as blueprints which describe what containers to use, ports to open and what memory and CPU requirements are needed. Then, we have the ***service*** component under fargate, which is responsible for using the task definitions to create and manage running processes in a cluster. The processes instantiated by the service are called ***tasks.*** As to why we use microservices, refer to **Architectural Decision ID 1** for justification. For RDS which acts as the database for the microservice, refer to **Architectural Decision ID 11** for justification.

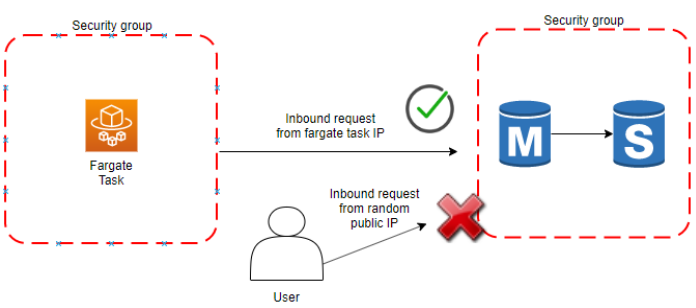
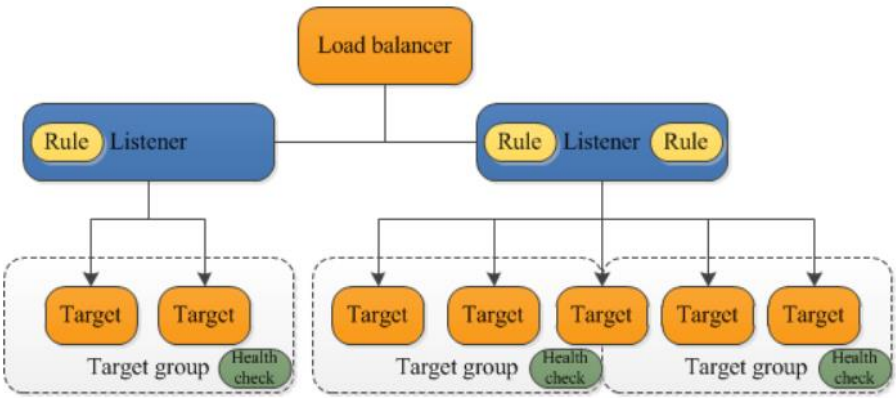


Figure 1

Figure 2

With regards to security access control, AWS allows the configuration for a security group which acts as a virtual firewall that controls the traffic for one or more instances. In Figure 1, we set up the security group for each of our service-specific RDS to only accept inbound requests from IP address of the corresponding microservice (fargate task). In Figure 2, we set up the security group for each microservice fargate task to only accept inbound request from the application load balancer (ALB)



Our ALB uses path-based routing to distribute load across our backend services with the help of rules. These rules are defined in paths such as when the ALB receives a path of /api/module*, it will know to route requests to a particular target group. A target group is a collection of endpoints or servers which the load balancer can route traffic to. Each target group in our case encompasses a microservice and its redundancies. When a target group receives a request from the ALB, it will route requests to one or more registered targets (servers running our application). Health checks can also be configured on a per target group basis which the load balancer will ping the targets to perform failover. This way, the ALB only sends requests to the healthy targets. The ALB functions at the application layer, the seventh layer of the OSI model. For reasons as to why we use a load balancer, refer to **Architectural Decision ID 9.**

The API gateway is found outside the VPC as shown in the solution view architecture diagram and the gateway environment acts as a demilitarized zone where the public would be sending their network requests into. The api gateway's justification is explained in **Architectural Decision ID 2**. As an add-on to the justification, the API gateway also acts as an air gap mechanism which helps to filter out malicious requests that tries to enter the VPC. The gateway helps to obscure the application load balancer to the public's eyes and in turn creates more security.

For the redis cache, refer to **Architectural Decision ID 6** for justification. The sequence diagram representing the redis workflow is detailed below (Figure 3).

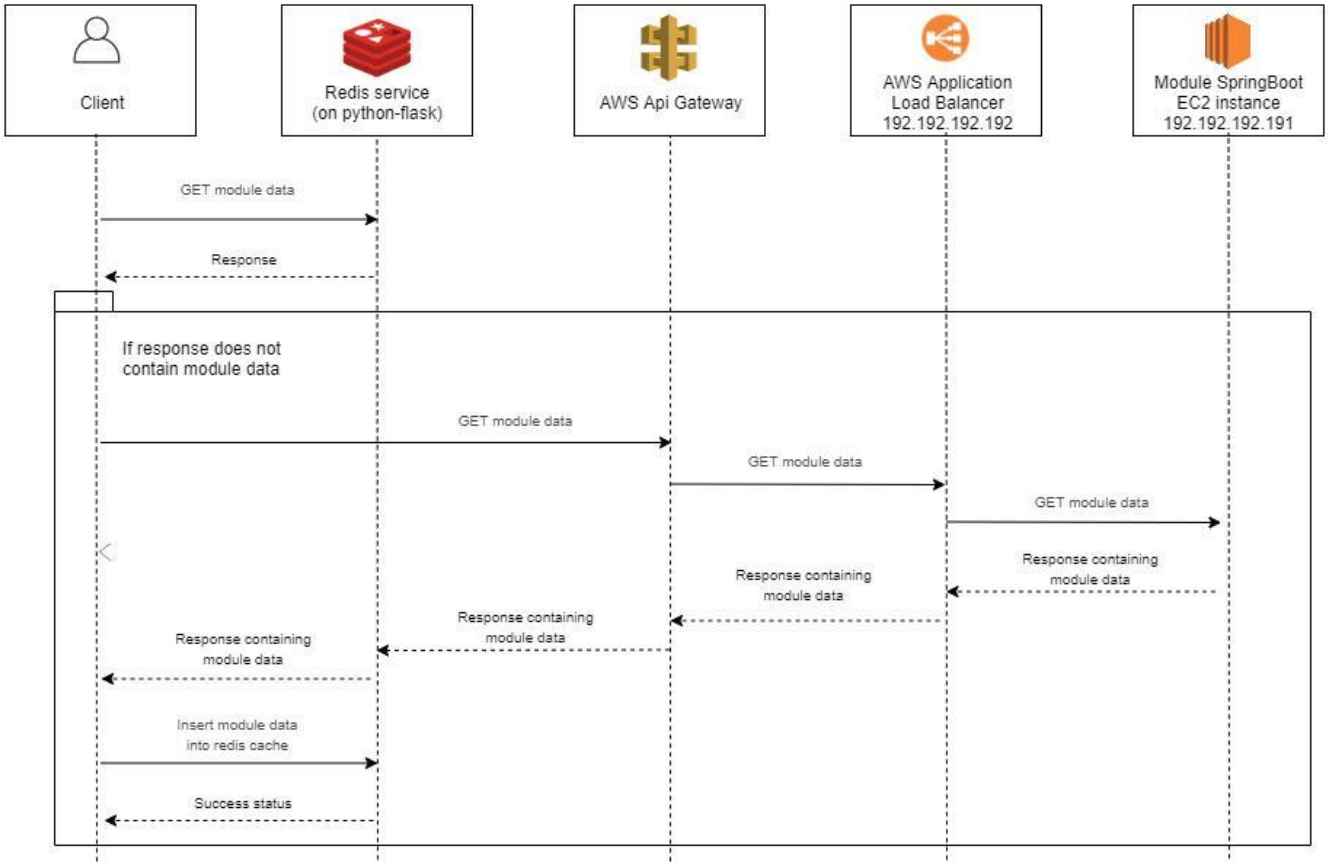


Figure 3

Integration Endpoints

Source System	Destination System	Protocol	Format	Communication Mode
Student Management System	API gateway	HTTPS	JSON, Token	Synchronous
API gateway	Application Load Balancer	HTTP	JSON	Synchronous
Application Load Balancer	Payment/Module/Student Microservice	HTTP	JSON	Synchronous
Payment/Module/Student Microservice	RDS database	MYSQL	SQL	Synchronous
Student Management System	Redis Cache	HTTPS	JSON	Synchronous

Hardware/Software/Services Required

No	Item	Quantity	License	Buy / Lease	Cost
1	AWS EC2 Instances	1	Open-sourced	Lease	$0
2	AWS RDS	6	Open-sourced	Lease	$0.034 per hour
3	AWS Fargate	1	Open-sourced	Lease	$0.05056 per vCPU hour $0.00553 per GB hour

4	AWS Application Load Balancer	1	Open-sourced	Lease	$0.0252 per hour
5	AWS ECR	1	Open-sourced	Lease	$0.12 per GB
6	AWS S3	3	Open-sourced	Lease	$0.025 per GB
7	AWS CodeDeploy	1	Open-sourced	Lease	$0.02 per instance
8	AWS CodeBuild	1	Open-sourced	Lease	$0.005 per min
9	Angular 7	1	Open-sourced	NA	$0
10	Ubuntu OS	9	Open-sourced	NA	$0
11	Spring Boot	6	Open-sourced	NA	$0
12	Apache Tomcat	6	Open-sourced	NA	$0
13	MySQL	1	Open-sourced	NA	$0
14	Travis CI	1	Open-sourced	Lease	$0
15	Jmeter	1	Open-sourced	NA	$0

Availability View

Node	Redundancy	Clustering			Replication (if applicable)			
		Node config.	Failure detection	Failover	Repl. Type	Session State Storage	DB Repl. Config.	Repl. Mode
Module Springboot Service (AWS ECS container instance) (Same for **Payment** and **Student** service)	Horizontal AUTO scaling (Scales up based on number of http requests)	Active-Active	Ping (Refer to figure 1 below for fail-over illustration)	Load balancer	NA	NA	NA	NA
AWS RDS (Master/slave, Sync/Async, Conflicts)	Horizontal Redundancy (back-up database in another availability zone)	Active-Passive	Ping (Refer to figure 2 below for fail-over illustration)	AWS Multi Availability zone automatic failover	DB (Multi Availability zone)	NA	Master-slave	Synchronous

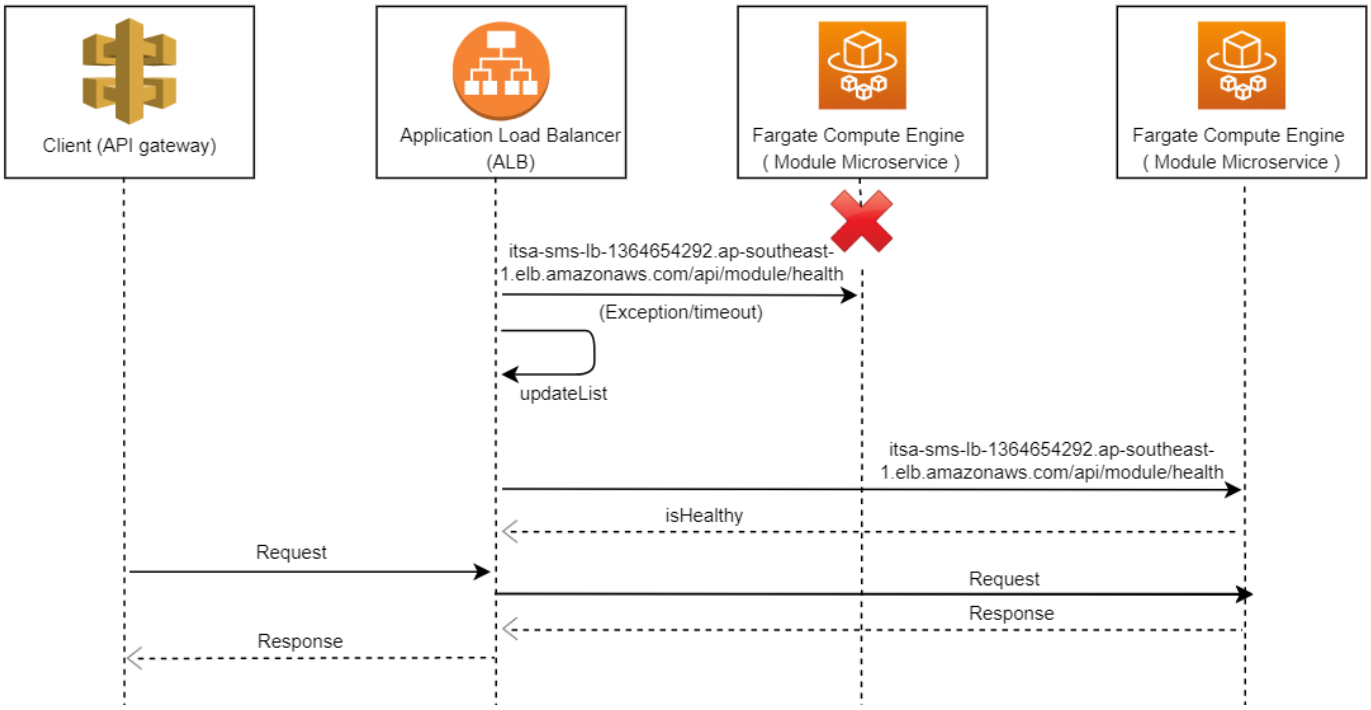


Figure 1. Load balancer' handle of loss in availability in microservices

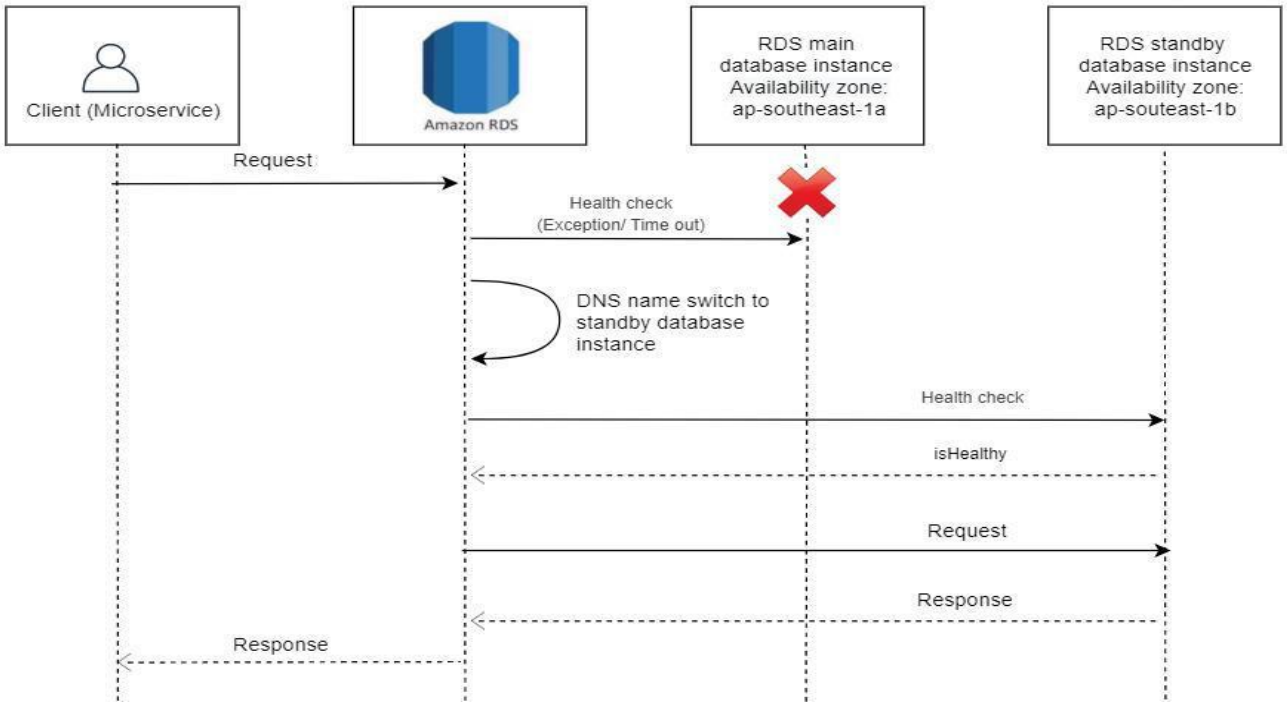


Figure 2. Microservice's handle of loss in availability in RDS database

Security View

For our microservices, Spring Security is implemented, that helps to protect the microservices from attacks such as session fixation, clickjacking and cross site request forgery. (Pivotal Software, 2019) For backend web applications, Spring Security is implemented as a servlet filter, which is part of a chain of filters that handles client requests.

No	Asset/Asset Group	Potential Threat/Vulnerability Pair	Possible Mitigation Controls
1	Login view	HTTP Parameter Tampering: an attacker	Implement validations in both client

		may bypass the validation for form inputs on the client side, hence sending unvalidated input directly to the server.	and server side.
2	UI portal	Unauthorized URL Access: when an unauthorized user trying to access a page even though the permission is not granted to them (e.g trying to access admin URL)	Implement AuthGuard in Angular 7 to allow only authorized user to access the different URL endpoints.
3	HTTP Response for web services	Attacks based on MIME type confusion when browser MIME-sniffs response other than the declared content-type.	Set X-Content-Type-Options header to “nosniff” to reject responses with incorrect MIME types.
4	Student and module data	Unauthenticated access to web services that can lead to data leakage to unintended people, compromising confidentiality	Set JWT (token returned by AWS Cognito after uses logs in) in headers for each HTTP Request. Microservices can only be called when token has been successfully validated by our API Gateway.
5	Payment data	Unauthorised update to payment database with the web service in payment system, affecting integrity	Validate payment with payment provider before updating database
6	Attendance data	Integrity of attendance can be compromised when database is attacked	Only allow attendance to be taken (updated) during class time
7	Database Systems	Confidentiality of data can be compromised with attacks such as SQL injection that can potentially retrieve information unintendedly	Utilise Hibernate that utilises a variant (object-oriented version) of SQL

Authentication is carried out via JSON Web Token (JWT) returned by AWS Cognito after the user’s email and password have been validated. JWT is then stored in session storage and passed in the header of each HTTP Request so backend webservices can validate before returning the data. Also, the expiry time of JWT limits the authentication time for users, hence preventing the threats of different attacks while users are still logged in.

Vulnerabilities are checked for the frontend and backend web services using OWASP ZAP. For backend services, an alert relating to X-Content-Type-Options header was given. To mitigate and fix the vulnerability, Spring Security is implemented in all backend web services. The framework provides an architecture revolving around authentication and authorisation, which provides access control and web security configurations, among others.

Refer to **solution view** for explanation on the access control design of our architecture. Figure 1 and 2 in solution view shows the access control we have set on the IP network level. The SpringBoot instances can only be accessed by our load balancer and our databases can only be accessed by their corresponding services.

Performance View

No.	Description of the strategy	Justification	Performance Testing (Optional)
1	**Application Load Balancer** Our application load balancer evenly distributes network traffic to prevent failure caused by overloading a particular resource, improving both performance and availability	Thousands of students use the student portal every weekday where there is class to take their attendance. As such, there is a need to ensure that the requests from the students are distributed so as to not overload a certain server, thus increasing both performance and availability	With the setting of 50 concurrent users accessing the service within 1 second, on average, the response time for multiple instances of module service managed by AWS application load balancer is 578.10 ms which is more than 7 times faster than single instance of module service on EC2 of 4167.24 ms shown in Appendix figure 3 and 4
2	**Redis Cache** When students log into the main page, the frontend attempts to retrieve the student's module data. It first checks our redis database if the student's module data exists. If it does, it retrieves the data from the redis cache. Else, it makes a GET call to the Module Service through the API gateway. It then stores the retrieved module data in the redis cache.	A student's module data is unlikely to change during the semester. As such, his module data can be placed into a redis cache - an in-memory storage database - for faster retrieval instead of having to make a GET request to the Module Service every time the student logs in. In the event that the semester ends or the student somehow changes his modules, the redis cache should clear the student's entry.	Using the in-built timer on the Chrome browser, the measured time for the frontend to retrieve data from the redis cache averaged 0.025 seconds over 10 tries, while the measure time for the frontend to retrieve the same data from the module service through the api gateway averaged 0.126 seconds over 10 tries - around 5 times that of the redis cache retrieval, shown in Appendix figure 5.
3	**Pre-fetch** When students log into the main page, we assume that they wish to navigate into the attendance page to mark their attendance. As such, to improve speed and performance, our frontend now makes a pre-fetch when the student logs in into the main page, and gets the student's attendance data and stores it in the student's local storage. When the student wants to view his attendance data, the data will be displayed from local storage without having to make an api call to the Module Service, hence improving speed and performance.	As there are classes every weekday, we can safely assume that students will log in onto the school portal in order to take attendance quite frequently. As such, in order to make the attendance-taking process faster and more seamless for the student.	Using the in-built timer on the Chrome browser, the measured time for the frontend to retrieve pre-fetched data from local storage averaged 0.008 seconds over 10 tries, while the measure time for the frontend to retrieve the same data from the module service through the api gateway averaged 0.055 seconds over 10 tries, around 7 times that of the pre-fetch retrieval, shown in Appendix figure 6.

Appendix

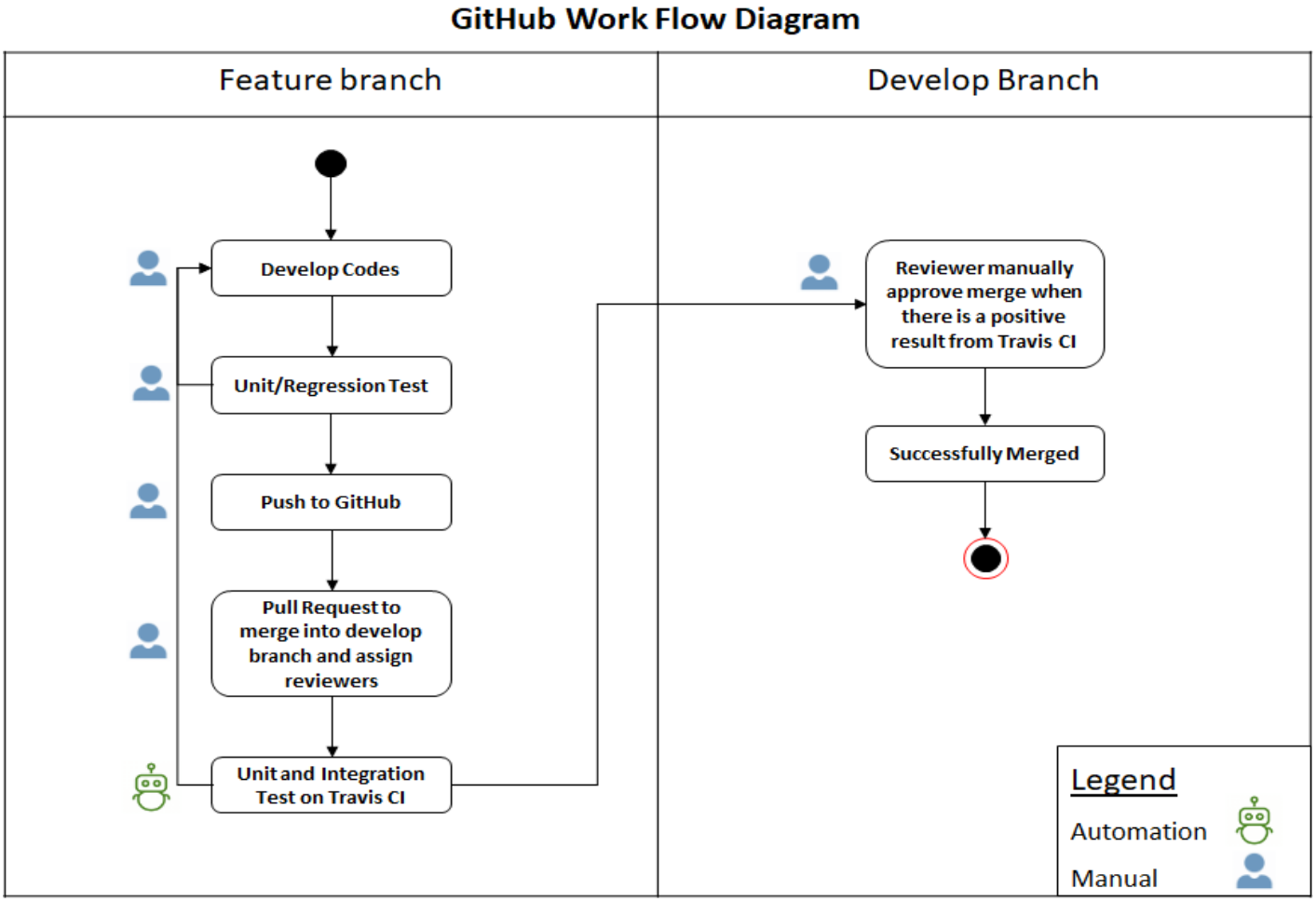


Figure 1

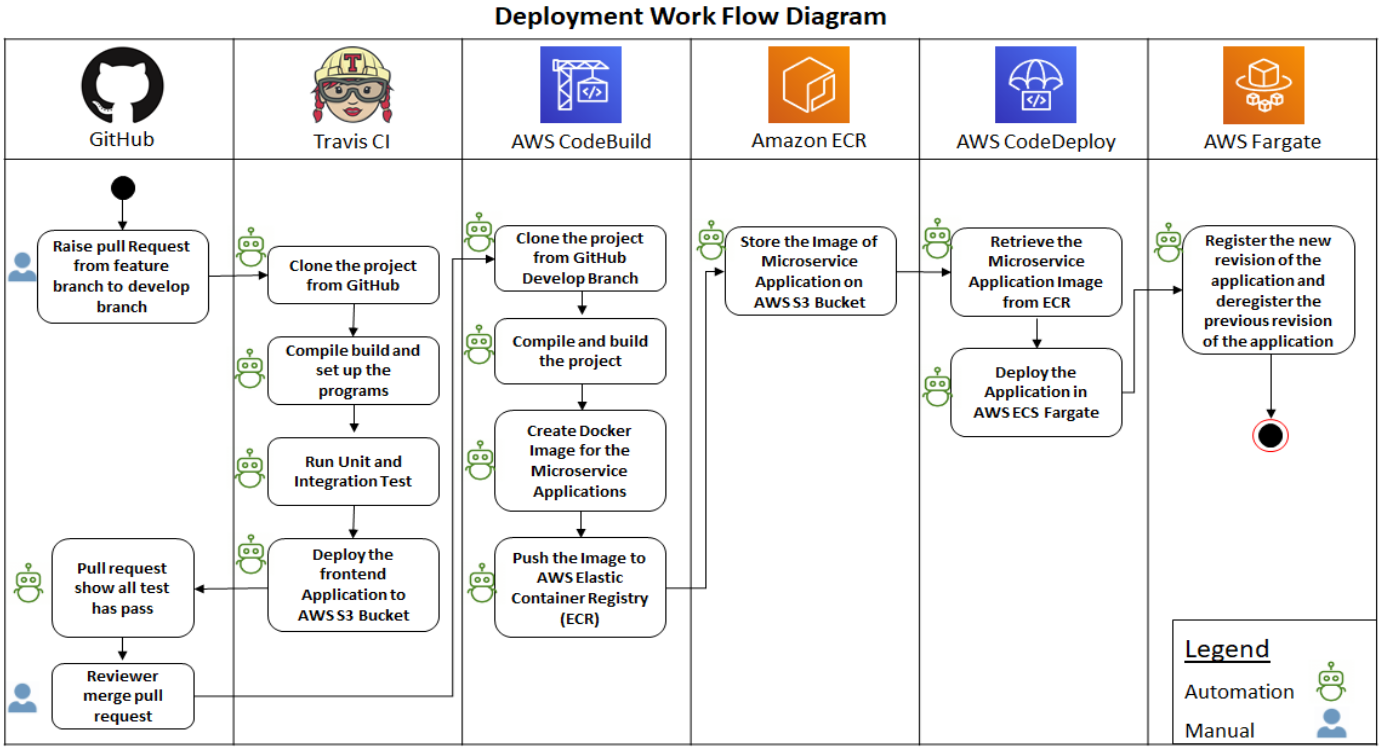


Figure 2

Requests	Executions			Response Times (ms)						Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	50	0	0.00%	578.10	35	1348	1066.00	1204.45	1348.00	29.53	10.73	6.06

Figure 3: Multiple ModuleService Managed by the Load Balancer

Requests	Executions			Response Times (ms)						Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	50	0	0.00%	4167.24	2303	5730	5437.00	5612.85	5730.00	8.31	2.82	1.38

Figure 4: Single ModuleService on EC2

	1	2	3	4	5	6	7	8	9	10	Average
Redis response time	0.022	0.023	0.027	0.028	0.023	0.024	0.025	0.029	0.035	0.036	0.025
Module service response time	0.126	0.084	0.267	0.103	0.204	0.033	0.063	0.140	0.153	0.087	0.126

Figure 5: Time taken to retrieve data from Redis cache vs Module service

	1	2	3	4	5	6	7	8	9	10	Average
Pre-fetched response time	0.013	0.006	0.017	0.002	0.003	0.003	0.005	0.011	0.012	0.008	0.008
Module service response time	0.064	0.055	0.040	0.053	0.064	0.063	0.048	0.050	0.053	0.060	0.055

Figure 6: Time taken to pre-fetch data from local storage vs Module service