iFreelance

Graciella Antonius Putri
School of Computing and Information Systems
Singapore Management University
Singapore
gaputri.2019@scis.smu.edu.sg

Hartanto Ario Widjaya
School of Computing and Information Systems
Singapore Management University
Singapore
hawidjaya.2019@scis.smu.edu.sg

Lindy Lim Li Wen
School of Computing and Information Systems
Singapore Management University
Singapore
lindy.lim.2019@scis.smu.edu.sg

Seah Li Ping Megan
School of Computing and Information Systems
Singapore Management University
Singapore
megan.seah.2019@scis.smu.edu.sg

Thet Thet Wai
School of Computing and Information Systems
Singapore Management University
Singapore
twthet.2019@scis.smu.edu.sg

Xue, Wenyu
School of Computing and Information Systems
Singapore Management University
Singapore
wenyu.xue.2019@scis.smu.edu.sg

***Abstract*—iFreelance is a website that provides e-commerce features for freelance services. The website is created using a microservices architecture and while adopting DevOps practices.**

Keywords—microservices, DevOps, freelance, API

I. Problem

Freelancers have few centralized platform options to promote their services. Most job platforms cater to professional services, while for non-professional services that are needed (dog walker, freelance babysitters, homework tutor, etc.) customers need to find manually through their contacts/network which is inefficient and time consuming. There are some international platforms such as Fiverr and Upwork where freelancers need to bid for projects and have to be specialised (i.e. offering a smaller range of services per individual). Customers may not know if they can trust a freelancer, and vice versa.

II. Solution

To solve this problem, we introduce iFreelance; a website that provides e-commerce features for freelance services. Freelancers can offer a wide range of services from non-professional to professional services, and with a dual confirmation and a centralized payment system, freelancers and their clients are able to safely book and complete tasks/requests.

III. Key Scenarios

A. Book A Service

Buyers are able to book a service that is offered by the seller (freelancer). For the order to be active, the request must be confirmed by the seller and the buyer must pay a deposit which will be transferred to the seller after service completion.

The detailed steps are as follows:

1. Buyer browses the available services on the UI
2. Buyer clicks the *'Get now'* button on the specific service.
3. Request is sent to API Gateway and routed to the place-order and order-service.
4. Buyer's order is placed, and a notification email is sent to the seller.
5. After the seller confirms (see Key Scenario B), the buyer clicks the *'Pay'* button to pay a deposit to the seller. This will be done before the seller has completed the request

B. Seller Accepts Booking Request

Sellers are able to accept the booking request. When the seller confirms, the order will be active and the seller is allowed to complete the service. After the seller completes the service, the seller indicates on the website and the buyer will be notified through email. The buyer then confirms whether the service is actually completed.

The detailed steps are as follows:

1. Seller views the service booking requests from UI (on their dashboard).
2. Seller clicks the *'Confirm Request'* button on the request they want to undertake.
3. The confirmation is sent to the API Gateway and routed to the place-order and order-service to update status.
4. Buyer receives the notification that request is successfully accepted.

C. Process Payment

Buyers are able to process payment for the selected service. When the buyer books a request, he/she needs to pay a deposit. The deposit will be kept by the payment server until the service is completed (confirmed by both the seller and buyer). The deposit will be sent to the seller and marks the end of the workflow.

The detailed steps are as follows:

1. Buyer reviews the details of the service to purchase and clicks on *"Make Payment"*.
2. The request is sent to the API Gateway and routed to the payment service which calls the Stripe API.
3. The buyer is brought to the checkout page, where he/she selects the payment method and enters the relevant details.
4. Stripe process the payment and the status of the payment is displayed to the buyer.
5. After the buyer confirms the completion of the services on the UI, the buyer is brought to the checkout page to

Funding was received from Chris Poskitt, Ph.D., Assistant Professor of Computer Science (Education) at Singapore Management University.

pay the remaining amount to the seller. After successful payment, the payment will be transferred to the seller's account and the whole transaction will be completed. The order will be closed.

IV. Microservices Architecture

An orchestration pattern was used for communication between the microservices. Each microservice would send a synchronous request, and await a synchronous reply. We chose this over other patterns, e.g. choreography, as we deemed it a better fit for our product and use cases. For example, there is a clear flow from the web UI all the way to the payment gateway. When a user clicks on a button on the UI, a response would be expected in the form of a synchronous UI update. Hence, another pattern such as choreography did not feel as intuitive, as it is largely asynchronous and low on process visibility.

For communication style, all microservices contact each other through synchronous one-to-one calls, except for the communication between place-order, RabbitMQ and notifications. The request sent from place-order to the message broker to be consumed by notifications is an asynchronous fire-and-forget.

In terms of modularity, we implemented a RabbitMQ broker, with a queue for order-related messages. Hence, if we needed to add another service that also sent notifications, we would only need to create a new queue for the service's routing key and an exchange for the new topic. Furthermore, we had a separate container for Stripe, to make it easier to integrate with another payment system or API should we no longer use Stripe. The new system would only need to receive information from Payment, and call the webhook method once it is complete.

A downside of this architecture was the tight coupling. For example, both the Payment and Stripe containers would need to be live before a payment can successfully go through. Else, the outgoing request from Payment is never processed by Stripe, and the order cannot be updated as there is no call to the webhook.

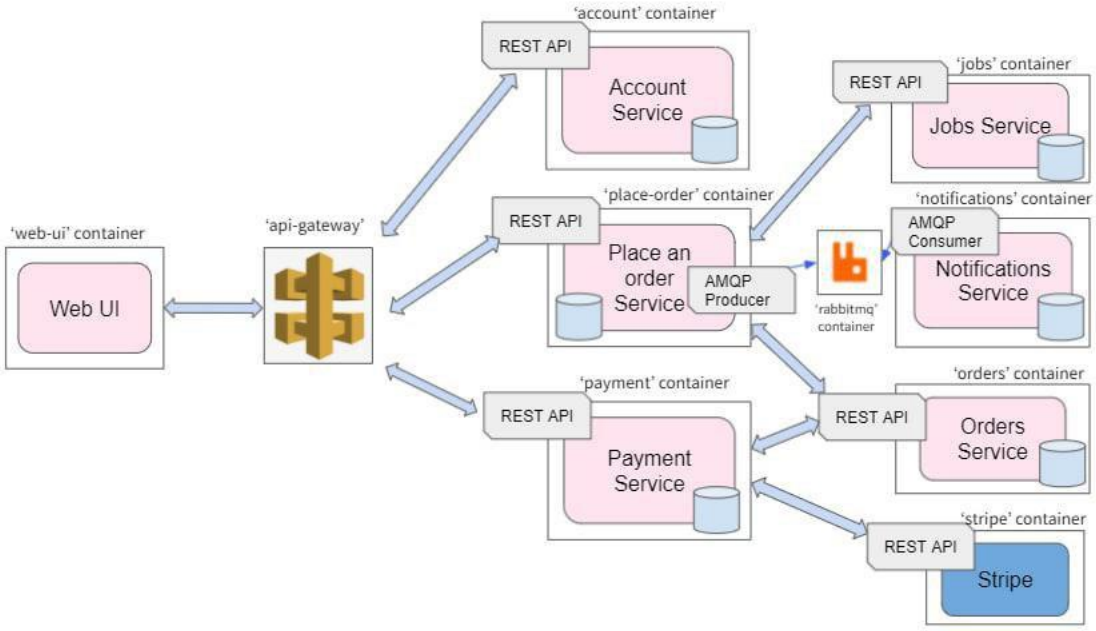
V. DevOps Practices

For our DevOps practices, we made sure that our approach adheres to the CALMS principle: collaborative, automation, lean, measurement, and sharing. We also made sure to follow Gene Kim's three way principles: first way (flow), second way (feedback), third way (continual learning).

A. *Team Practices*

Throughout the project, we adopted some practices in the Scrum methodology, which is an Agile framework commonly used in software development projects. Firstly, before the beginning of development, we had a meeting to create the product backlog together and delegate responsibilities. This allowed us to have a shared understanding about the various components of the project. We also had weekly meetings to update on each of our progress and set new milestones for the following week. With this, we established an environment for regular feedback and greater responsiveness to changing needs and requirements. In addition, we documented our progress using a Kanban board through Google Spreadsheet where we listed out each of the team members' responsibilities and each of us marked down our progress throughout the development cycle. This enabled us to visualize how work is flowing and quickly determine and eliminate the bottlenecks.

For our support system, we applied the swarming approach where we collaborate together when there are errors whether in our pipeline, code, AWS, etc.. As soon as there is an error, we notify each other in the Telegram group to collaboratively solve the problem, although the team member who is in charge of the part where the error comes from is the one who is mainly in charge of the solution. This in turn fulfills the group's continual learning where we are notified of what is the reason for the current problem and the solution. From there, each group member is able to identify, reflect, and learn from theirs and others mistakes, which provides a good

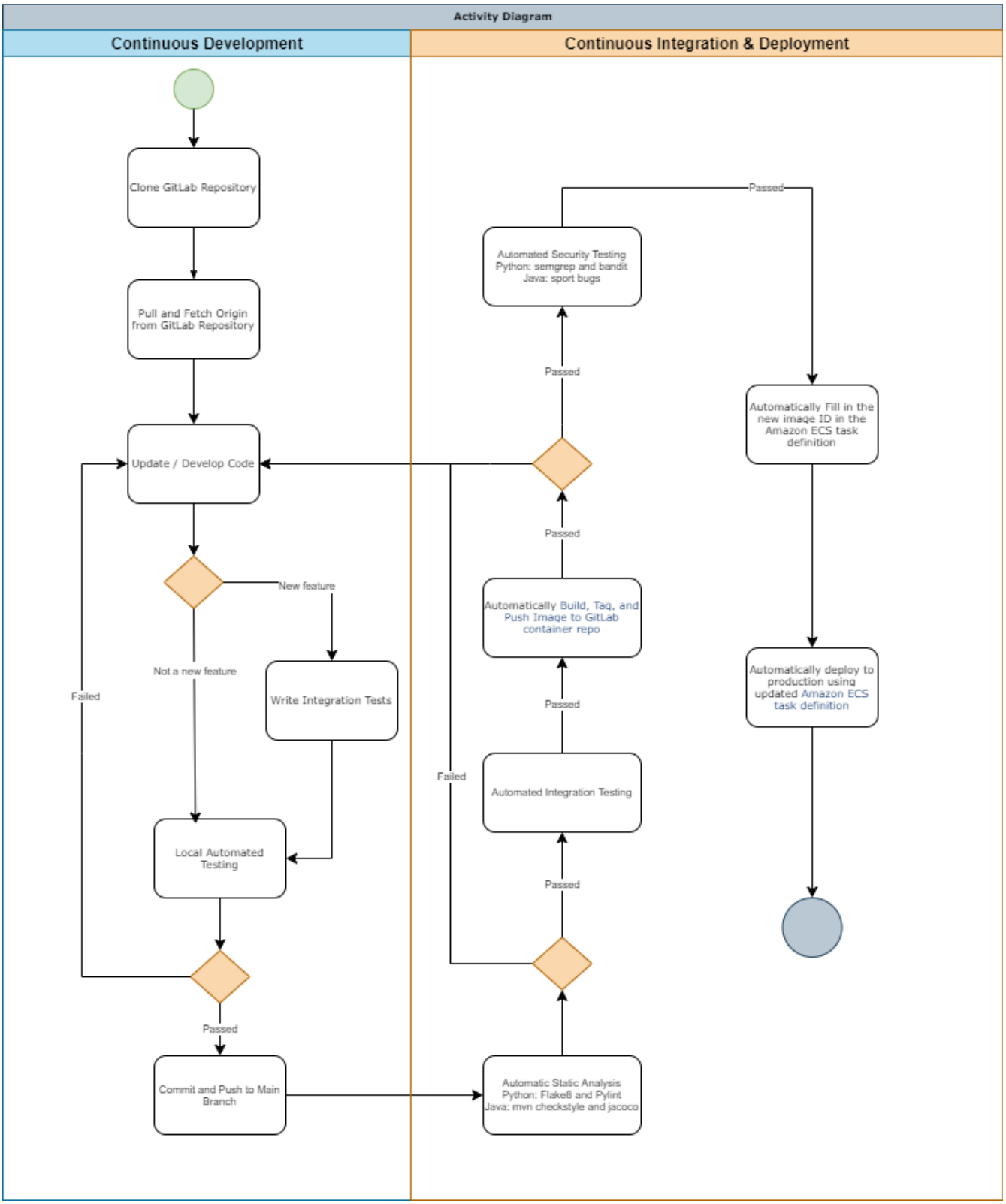


.Figure 1. Microservices Architecture Diagram.

environment and provides an open and blameless working culture. In reducing batch sizes and handoffs, we regularly commit and push to our GitLab repository in order to have small changes rather than large changes in one go to limit work in progress. As a result, we must commit using clear descriptive commit messages for clear documentation and navigation.

B. Continous Development

We started by cloning the GitLab Repository if we have yet to do so. Following that, we pulled the changes and fetched the origin from the GitLab Repository. We then proceed to update and develop the code. If the code written was for a new feature, we will include additional tests in the test file. Afterwards, we will run tests locally and make some calls through Postman to ascertain that features perform as



.Figure 2. Activity diagram illustrating the overall development process.

expected. For example, for the notifications service, we had some environment variables (SES_REGION, SES_EMAIL_SOURCE, AWS_ACCESS_KEY_ID_SES, AWS_SECRET_ACCESS_KEY_SES) which are required for Boto3 to interact and automate SES to send the email. In this case, doing local testing was particularly helpful because we were not allowed to commit the credentials directly and had to do so by setting variables in GitLab. Hence, if any problems occur even though local testing is successful, it means that the problem most likely lies in the retrieval of the environment variables in GitLab, enabling us to quickly isolate and fix the problem. If the tests are successful, we will commit and push these changes to the main branch in GitLab. If not, we will revert back to updating the code to fix the problems causing these test failures.

Overall, by practising continuous development, we are able to detect and fix errors and vulnerabilities in the code during early stages of development. This is much more efficient as compared to detecting and fixing these errors late in the development cycle, where high technical debts would have been incurred and it becomes difficult to isolate the source of the problem. Continuous development also helps with eliminating code incompatibilities and code conflicts early. Overall, the continuous feedback from doing continuous development reduces project risks and increases productivity [1].

C. Continuous Integration and Deployment

We have configured the stages involved to run automatically for every commit pushed to GitLab by specifying the details in a .yml file uploaded on each GitLab repository.

The first stage is static analysis, which we implemented using Flake8 and Pylint for Python code, and Maven Checkstyle and JaCoCo for Java code. This stage helps to enforce coding standards and prevents errors in the program syntax and bad formatting. This would ensure that developers in the team write better code and makes it easier to review the code [2]. Some checkstyle rules were omitted for brevity, such as the requirement to write javadoc comments, as we were not planning on automatically generating the documentation. If static analysis tests are not passed, the pipeline fails and the developer will need to update the code to fix these program syntax issues which can be found in the logs on GitLab.

Once the static analysis tests pass, the pipeline would proceed to do automated integration testing for unit services and component testing for composite services (place-order). This serves to verify whether the service works with other services correctly. For Python code, this is done using Pytest whereas for Java code, this is done using Spring's WebMVC framework. By doing automated integration testing continuously, it provides assurance that the features work as expected and allows problems to be identified early [3]. Similar to the static analysis stage, failure of the integration tests would result in failure of the pipeline. The developer will have to update the code to fix the errors causing the failure. Once the integration tests pass, the pipeline will proceed to the stages involved in Continuous Deployment.

During the release stage, a Docker image is automatically built, tagged and pushed to the GitLab Container Registry. Following on, the AWS Command Line Interface requests ECS to pull that latest image that was pushed to the GitLab Container Registry during release.

Static Application Security Testing (SAST) is then used to check the source code for any vulnerabilities and is done using Bandit and Semgrep for Python code and SpotBugs for Java code. The GitLab Container Scanning feature is also used to further check the Docker image for any known vulnerabilities. This feature is part of the GitLab Ultimate Plan and Trivy is used as the default static scanner. With this extra security layer in our CI/CD pipeline, we are able to detect and address security vulnerabilities in the code early in the development process. Otherwise, it can be costly and difficult to fix. Security is built into the product rather than applied to the end product, so the probability of discovering unexpected issues late into the development cycle is much lower [4]. These security checks, the GitLab Container Scanning feature in particular, requires the image, so they can only be done after the release stage.

ECS then updates the corresponding task definition and re-deploys the microservice to the production environment using this updated task definition. In Amazon ECS, we used rolling deployment with desired count as 2, minimum healthy percent of 50% and maximum healthy percent of 100%. During rolling deployment, the containers that are running the previous versions of the application will be replaced in succession with containers running the new versions. As compared to blue/green deployment, rolling deployment is generally faster [5]. With continuous deployment, quality of releases and development speed increases as changes are small [6]. Furthermore, teams can respond to customer feedback immediately and ensure faster time to market [7].

D. Continuous Monitoring

We implemented continuous monitoring using Amazon CloudWatch. Since our application uses microservices, there will be large volumes of data generated in the form of metrics, logs and events. With CloudWatch, we are able to obtain system-wide visibility on a single platform due to its native integration with many AWS services, enabling problems to be resolved quickly and breaking down data silos. We will also be able to obtain utilization insights, which provides some indication on whether the resource utilization is optimised and would require further improvements [8]. Overall, practising continuous monitoring would provide real-time feedback on our application, which can be used to optimize performance and resource utilization, especially if the application were to be scaled up.

VI. Self-Directed Research

A. AWS Infrastructure and Deployment

One of our self-directed research components is with regards to the AWS infrastructure. For our deployment, we utilized Amazon Elastic Container Services (ECS) to host our Docker containers.

The ECS cluster is hosted on two separate Amazon Elastic Compute Cloud (EC2) instances on the us-east-2 region. One of the instances is located on the availability zone us-east-2a, while the other is on us-east-2b.

To enable our services to store data, we created an Amazon Relational Database Service (RDS) running a MySQL server. Additionally, we have an Amazon Simple Storage Service (S3) bucket which we used to store our environment variables.

Furthermore, to enable our messaging protocol to work, we use the Amazon MQ managed message broker which host a RabbitMQ instance.

For each of the services we provided, two tasks are set up with the minimum health of 50% and a maximum of 100%. The tasks are distributed among the two EC2 instances to allow for any contingency planning in the event that one availability zone went down.

The only exception to this is our notification service which act as an AMQP consumer. The service will then add the notification to our RDS and send an email via Amazon Simple Email Service. Due to the possible dual entry into the RDS, we decided to only host one task of this service.

The two tasks in each service are then load balanced via an Application Load Balancer (ALB) which allows for dynamic port mapping.

Each service will also pull the Docker image link from the GitLab Container Registry through the use of a GitLab API Key stored on the AWS Secrets Manager.

Additionally, we utilized the Amazon API Gateway to act as a proxy to our load balancers. The API Gateway also provides the ability for us to implement API keys and TLS connections. However, one thing that we note is that a user will be able to bypass our API Gateway if they know the HTTP endpoint of ALB.

We undertook research to determine how we can solve this issue. In particular, we want to have private integration to allow the routing of traffic from the API Gateway to a private subnet in our Virtual Private Cloud (VPC). This, however, relied on a VPC endpoint service called AWS PrivateLink that is tied to Network Load Balancers (NLBs). Therefore, we decided to create a NLB in a private subnet to be put in front of our ALB, which we shift to the private subnet as well. This would introduce additional infrastructure cost and additional network hop, but it wouldn't require us to redesign our existing infrastructure to route traffic via NLB instead of ALB [9].

So, a request that hits our API Gateway would be directed via VPC link to the NLB, before being forwarded to the ALB, and then to one of the two EC2 instances which host our Docker container as part of the ECS Cluster.

B. *Production WSGI Server*

By default, Flask wraps around Werkzeug, a WSGI web application library. For ease of development, Werkzeug comes with a builtin development server. According to the Werkzeug developers, the development server is not intended for production and performs poorly under high load [10]. To enable our application to perform better under load, we decided to use Gunicorn as the production WSGI server.

C. *Email Notification through Amazon SES*

We also explored the possibility of sending email notifications to our buyers and sellers with the use of Amazon Simple Email Service. After our notification service receives the MQ message, it will execute a script which will send out an email to the corresponding users.

D. *DevSecOps – Adding Security Checks into the CI/CD Pipeline*

Lastly, we decided to add security checks into our CI/CD pipeline through the use of GitLab services. In particular, we make full use of the GitLab Static Application Security Testing (SAST) which checks our code statically for any known vulnerabilities, and the GitLab Container Scanning which statically checks our Docker container. The results of both will be delivered to the inbuilt Security Centre where we can review them, and act upon them if necessary.

VII. Reflections

A. *What Went Well*

This project was a good extension of the lab exercises which enabled us to obtain a much better understanding of developing a full stack application using a microservices architecture while exploring additional features. In addition, we were able to apply firsthand some of the DevOps practices that we had learnt in class, such as swarming, to enhance collaboration, flow of work and quality of work. We managed to fulfil the expectations that we had set as a group before starting the development process, or even exceed these expectations as we had no experience in creating UI but were able to create one for this project. Additionally we also learned how to integrate two different languages into one system and integrate with an external payment service which is completely new to us. Most importantly, this was a fun and fruitful experience for all of us. We gained a lot of exposure and were inspired to research more and try applying our findings to our future projects.

B. *Potential Improvements*

For web UI, although it's not the main focus of this project, we needed to carefully consider the consequences and impact of each user request to ensure the trading can proceed smoothly so as to clearly demonstrate the key scenarios from the user (freelancer & customer). If we had more time, one thing we can further explore from a UI perspective is the matching between a buyer's requirements and a seller's offers. As the current approach, buyers only book a service based on service description but have limited knowledge about the seller's information. We can provide the seller's profile and customer reviews related to the service to the buyer as another purchase criterion.

Moreover, as we have a significant amount of user interactions that need to be initiated from the GUI, we need to call different microservices to retrieve and consolidate the required data. If we have time we could use GraphQL as an API gateway solution to query and mutate the exact data in a single request.

For Account login feature, the security of the password currently is only encryption using werkzeug.security's generate_password_hash and check_password_hash. Although this should provide basic security as it encrypts the password with the indicated hash function and random added salt for each password, we feel that the whole process could be more secure such as adding features: validating user email, phone number, and other supposedly unique credentials, check the user password's strength, and other additional features regarding account such as recovering forgotten passwords. The login feature could also be incorporated using AWS Cognito.

For Payment and Stripe, the communication between these services went well when both services were live, thanks to Stripe's robust developer tools. The request returning from Stripe API was well-formatted and easy to process. As we needed some custom information such as Order ID to be

present in the requests, we were able to set it in the description field of the Stripe PaymentIntent, and then extract it to identify the order to be updated. If we had more time, we would have looked into Stripe's Connect API to collect and send out payments to the sellers, as the original plan included a centralised credit repository (i.e.. Stripe), where we would only deliver the payment to the seller if the buyer confirms the deliverables were up to standards. Also, we would have included webhook signatures, which check that a request to a webhook is indeed coming from Stripe, to protect against man-in-the-middle attacks. In terms of CI/CD, due to the tightly coupled nature of Payment and Stripe, and Stripe API's restrictions we would have tried to write better integration tests, as the current ones only test if the service itself and its connection to the other are working. For example, Stripe API expects requests in a certain format to be passed in to their PaymentIntentCreateParams builder, with the successful response being a client secret. It is difficult to test this accurately since the client secret would vary.

Lastly, for deployment via AWS, we currently route the frontend website through the API Gateway. Hence, any request without TLS connection would fails without any possible redirection. Furthermore, with the way API Gateway is set, we are unable to redirect from the www subdomain to the root domain, or vice versa. We can explore AWS Amplify or hosting the API Gateway on our subdomain as an alternative.

ACKNOWLEDGMENT

We would like to acknowledge Chris Poskitt, Ph.D., Assistant Professor of Computer Science (Education) at Singapore Management University for his assistance in this project.

REFERENCES

[1] Foster, S. (2021, April 23). *What is continuous development?* Perforce Software. Retrieved November 10, 2021, from https://www.perforce.com/blog/kw/what-is-continuous-development.

[2] Null, D. (2017, January 31). *What is Flake8 and why we should use it?* Medium. Retrieved November 10, 2021, from https://medium.com/python-pandemonium/what-is-flake8-and-why-we-should-use-it-b89bd78073f2.

[3] Liu, Z. (2018, November 27). *Automated Integration Testing*. Medium. Retrieved November 10, 2021, from https://medium.com/@allenliuzihao/automated-integration-testing-a295d21e513a.

[4] Mukherjee, J. (2021). *DevSecOps: Injecting Security into CD pipelines.* Atlassian. Retrieved November 10, 2021, from https://www.atlassian.com/continuous-delivery/principles/devsecops

[5] Amazon Web Services, Inc. (2021). *Rolling Deployments*. Amazon Web Services. Retrieved November 10, 2021, from https://docs.aws.amazon.com/whitepapers/latest/overview-deployment-options/rolling-deployments.html.

[6] Quine, D. (2016, January 12). *Why continuous deployment matters to business.* Continuous Delivery. Retrieved November 10, 2021, from https://medium.com/continuous-delivery/why-continuous-deployment-matters-to-business-6a79b5602145.

[7] Pittet, S. (2021). *Continuous Deployment*. Atlassian. Retrieved November 10, 2021, from https://www.atlassian.com/continuous-delivery/continuous-deployment.

[8] Amazon Web Services, Inc. (2021). *Amazon CloudWatch*. Amazon Web Services. Retrieved November 10, 2021, from https://aws.amazon.com/cloudwatch/.

[9] Amazon Web Services, Inc. (2021). *Best Practices for Designing Amazon API Gateway Private APIs and Private Integration. Amazon Web Services*. Retrieved November 12, 2021, from https://docs.aws.amazon.com/whitepapers/latest/best-practices-api-gateway-private-apis-integration/best-practices-api-gateway-private-apis-integration.html.

[10] Pallets. (2007). *Serving WSGI Applications - Werkzeug Documentation (2.0.x)*. Werkzeug Documentation. Retrieved November 15, 2021, from https://werkzeug.palletsprojects.com/en/2.0.x/serving/.