Campus CHOW: CS302 Final Report

Choo Yang Hwee, Chua Jia Jie Lennel, Doris Lim Lay Teng,
Ng Zhen Cong Matthew, Shiv Iyer, Xu Duo
School of Computing and Information Systems
Singapore Management University
Singapore

***Abstract*—Campus Chow is a microservices-based application designed to streamline food ordering and delivery within a campus environment. This report documents the problem context, solution design, key user scenarios, detailed microservice architecture (including Runner and Email services), DevOps practices, CI/CD pipeline, deployment approach, self-directed research, reflections, and work distribution.**

***Index Terms*—microservices, DevOps, CI/CD, food delivery, campus application**

I. Problem

When students have back-to-back classes, finding time to eat becomes a significant challenge. With little to no breaks between the classes, they often have to rush from one location to another to make it on time, leaving them with insufficient time to buy or eat food. This time pressure causes many students to end up skipping meals and relying on small snacks which impacts their concentration, energy levels and overall academic performance. The lack of convenient, accessible eating opportunities during tight class schedules creates a recurring problem that affects both student well-being and productivity.

II. Solution

Campus CHOW is a student-powered platform designed to make eating on busy school days easier and more efficient. It connects students who have limited time between classes with peers who are already heading to or from nearby food outlets. By intelligently matching users based on class schedules, locations, and food preferences, Campus CHOW streamlines group orders and ensures timely meal pickups. Additionally, this system not only helps students with short breaks get access to proper meals without rushing or skipping meals, but also creates flexible earning opportunities for students "food runners". These runners can take on orders between their own classes, earning extra income while helping schoolmates overcome the challenges of tight schedules and limited on-campus delivery options.

III. Key Scenarios

A. User Places a Successful Order

This scenario describes the process of a customer successfully placing an order through the Campus CHOW system.

1) The user begins by submitting an HTTP POST request via the frontend interface. This request is routed through the API Gateway (Kong), which forwards it to the Order microservice.
2) The Orders service then requests the latest menu items from the Menu microservice. The Menu service responds with the menu data, ensuring the order is validated with up-to-date information.
3) Once the user confirms their order through the frontend, the Orders service sends a request to the Payments microservice. The Payments service processes the payment and responds with a successful confirmation.
4) After the customer makes payment, the Payments service publishes a message via AMQP. This message notifies the system that the order is fully verified and paid for.
5) The Order microservice receives this message, updates the order status, and publishes a message to the Email service.
6) Upon receiving the message from Orders service, the Email microservice sends a confirmation email to the customer.

B. Assigning Orders to Runners

This scenario explains how runners are assigned to customer orders. The order creation and payment flow follow Scenario III-A.

1) The runner begins by setting their availability, indicating the runner is ready to take orders.
2) A customer then places and pays for an order as described in Scenario III-A. Once the payment is verified, the Order microservice publishes an order status event.
3) The Runners service listens for these order-verified events and, at the appropriate assignment time, selects available runners and assigns matching orders.
4) Once the assignment is successful, the Email microservice sends an email to the runner. This includes order details, pickup location, and timing.
5) Both the runner and the customer now see updated status information on the frontend reflecting the assignment.

C. Merchant Onboarding

This scenario details how a registered merchant creates a menu on the Campus CHOW platform. (A merchant account is created by the administrator.)

1) The merchant begins by logging into the system through the frontend.

2) The API Gateway forwards the request to the Merchant service, which verifies the provided credentials.
3) After authentication, the merchant submits a request to create or update their menu. The request is routed through the API Gateway and forwarded to the Menu microservice.
4) The Menu service processes the request and creates or updates the menu object.
5) The menu creation process ends with a 201 Created response, confirming the menu has been successfully created/edited.

IV. Microservice Architecture

The overall architecture of Campus CHOW is shown in Fig. 1 in the Appendix.

A. User

User microservice is an atomic microservice that handles everything related to user accounts in Campus CHOW. It stores key user details such as email, name, phone number, coins, profile picture and a hashed password in a dedicated PostgreSQL database. It manages user registration, login, profile updates, password changes and avatar uploads.

Authentication is implemented using JWT. Every protected route validates the token and extracts the user's ID and role, and admin only endpoints include an extra authorization check to ensure only authorised users can access management functions.

When a new user signs up, the service checks that the email is unique, hashes the password securely and creates the account. It also publishes a "user.welcome" event to RabbitMQ so the Email microservice can send a welcome email asynchronously. After logging in, users receive a JWT token which is required to access their personal data. Users can update their profile and manage their own account details as needed.

It also provides a basic management layer for admins. Admins can search for users, update their details and remove accounts when necessary.

B. Admin

Admin microservice is an atomic microservice that manages administrator accounts for Campus CHOW. It stores admin information such as email, password hash and role in its own PostgreSQL database, separate from the user database to keep credentials isolated. Authentication still uses JWT, but here it is enforced through Spring Security and a custom JWT filter that reads the token and sets the security context with the admin's role.

The service supports admin login and account management. Registration exists only as a backend route for internal setup and is not exposed to the public or to the admin frontend. After logging in, the service issues a JWT that the admin dashboard uses to access protected admin APIs. Admins can view their own profile, update their email and change their password, and these actions require the current password to ensure security.

The Admin microservice does not handle operational dashboard features such as managing users or verifying payment status directly. Those functions belong to other microservices, and the Admin service's role is simply to verify the admin's identity and role so those actions can be authorised correctly.

C. Order

The Order microservice is the system of record for all orders. Its main responsibilities are as follows.

1) creating new orders,
2) periodically checking for orders that require payment reminders or automatic cancellation, and
3) updating order status in response to requests and domain events from other services.

1) Synchronous order creation and orchestration: When a user places an order, the frontend sends an HTTP request to the Order microservice containing the customer identifier, target merchant, and list of selected items. The payload includes only menu item identifiers and quantities; the Order service never accepts client-supplied prices and instead retrieves the current prices from the Menu microservice before computing the total amount. In this flow, the Order service performs synchronous orchestration:

1) It calls the Menu microservice via HTTP to retrieve the current prices for the requested items.
2) After validating the items, it calculates the total amount and creates the order in its own database.
3) It then calls the Payment microservice via HTTP to create a corresponding payment record. The Payment service returns the payment details required by the customer:
    a) a payment reference,
    b) the PayNow recipient identifier (phone number),
    c) the total amount to pay, and
    d) a payment QR code.

The QR code can be scanned using the bank's "Scan and Pay" feature, which auto-fills and locks the payment fields, reducing the risk of user error. The explicit reference, recipient, and amount are still returned to support users who cannot scan the QR code and instead complete payment using the standard PayNow flow.

This flow is intentionally synchronous: the user is blocked on the order-creation screen waiting for payment details, so using asynchronous messaging here would only add complexity and latency without user benefit. In this scenario, the Order microservice acts as an orchestrator over the Menu and Payment microservices.

2) Scheduled reminders and automatic cancellations: Payment-related checks are not run continuously. The application supports four delivery time slots — 08:15, 12:00, 15:15, and 19:00 — aligned with SMU class timings. For each slot, scheduled jobs run at fixed offsets before the delivery time:

- **Payment reminder:** 90 minutes before the delivery time (e.g., at 10:30 for a 12:00 delivery), the Order microservice queries for unpaid orders in that slot and

publishes a message requesting an email reminder. The Email microservice subscribes to this message and sends the reminder email to the customer.

- **Automatic cancellation:** 60 minutes before the delivery time (e.g., at 11:00 for a 12:00 delivery), the Order microservice checks the same set of orders. Orders that remain unpaid are cancelled, and their status is updated in the Order database.

3) Domain events and integration with other services: Whenever an order is created or its status changes, the Order microservice publishes domain events for other services to consume. For example, when an order is cancelled, the service emits an `order.status.cancelled` event. The Payment and Email microservices both subscribe to this event:

- the Payment service marks the associated payment as `failed`, and
- the Email service sends a cancellation email to the customer.

No service calls the others directly in these cases; instead, each reacts independently to the published events, following the choreography pattern. Tables II and III in Appendix F summarise the events published and subscribed to by the Order microservice.

4) HTTP API and security: The Order service exposes HTTP endpoints for:

1) creating an order,
2) updating an order's status,
3) retrieving a user's order history, and
4) querying the status of a specific order.

All endpoints are protected using JWT-based authentication and authorization. For user-facing operations, the service ensures that the user identifier embedded in the token matches the owner of the order being accessed, preventing users from reading or mutating other users' orders.

Overall, the Order microservice is a composite service: it owns the order lifecycle while orchestrating calls to the Menu and Payment microservices and integrating with other components through domain events.

D. Payment

The Payment microservice is responsible for all payment-related functionality, including payment creation, QR code generation, evidence ingestion, automatic payment verification, and administrative workflows.

1) Responsibilities and HTTP API: The service exposes HTTP endpoints to:

- create and view payment records for new orders,
- allow administrators to view payments that are pending or pending refund, and
- let administrators update payment status (e.g., confirm payment, mark as refunded).

Whenever the status of a payment changes, the Payment microservice publishes a corresponding domain event, which other services (e.g., Orders and Email) may subscribe to. The full list of published and subscribed events is summarised in Tables IV and V.

2) QR code generation: To support PayNow across Singapore banks, the Payment microservice integrates with an external PayNow QR code generation API, which returns a compliant QR code that works across major banks. If the external API is unavailable, the service falls back to an in-house QR generator that is compatible with DBS. This fallback ensures that customers can still complete payments, albeit with a more limited bank set.

3) Payment evidence ingestion: After making a transfer, customers can upload a screenshot of their payment success page. The Payment microservice applies optical character recognition (OCR) with bank-specific parsing rules to extract the following fields:

1) transaction reference ID,
2) transaction amount,
3) transaction timestamp,
4) payee, and
5) payer.

Different banks format their confirmation screens differently, so the implementation relies on templates per bank rather than a generic free-form model. The extracted fields are stored in the database as part of the payment record.

Independently, when a transfer reaches the merchant's bank account, the bank sends a notification email to the email address linked to that account. A scheduled job within the Payment microservice periodically checks the mailbox using the Gmail API, detects new bank notification emails, and parses them to extract the same set of fields. These are persisted as *transactions* in the database.

4) Automatic matching and confirmation: Automatic payment confirmation is implemented as a matching process between payments (created by the system) and transactions (derived from bank emails and/or screenshots). Matching is triggered in both directions:

- when a screenshot is uploaded, the service looks for a corresponding transaction; and
- when a new bank notification email is processed, the service looks for an existing payment.

In practice, the bank email usually arrives before the customer uploads their screenshot, since the email is automated while the upload is manual.

A payment and a transaction are considered a match if they satisfy at least one of the following rules. The system first attempts a strict reference-based match; if that fails (for example, when the payer and payee use different banks and the transaction references differ), it falls back to an amount–time heuristic:

1) **Reference-based rule:** the transaction reference IDs match exactly, or the screenshot's reference ID is a truncated form of the reference in the bank email; or
2) **Amount–time rule:** the payment amount matches the transaction amount and the timestamps differ by at most ± 1 minute.

When a match is found, the payment is automatically marked as `payment_verified` and the Payment microser-

vice publishes a `payment.status.payment_verified` event. The Order microservice consumes this event, updates the corresponding order to `payment_verified`, and then publishes an `order.status.payment_verified` event. Downstream consumers such as the Email and Runner microservices listen only to `order.status.*` events, so they do not depend directly on the Payment service.

5) Manual verification and status model: Automatic verification may fail if the customer does not upload a screenshot, if OCR fails, or if the extracted data do not meet the matching criteria. In such cases, payments remain in a pending state and are surfaced in the administrative dashboard for manual review. Administrators can manually confirm or reject these payments based on additional evidence.

The Payment microservice maintains a simple status model with the following values:

- `pending`,
- `payment_verified`,
- `failed`,
- `pending_refund`, and
- `refunded`.

The Payment microservice subscribes to cancellation events published by the Order microservice. In particular, it listens for the `order.status.cancelled` routing key and marks the corresponding payment (identified by `orderId` in the payload) as `failed` (Table V). This ensures that payment and order lifecycles remain consistent even when orders are automatically cancelled.

E. Merchant

Merchant microservice is an atomic microservice that stores information about a merchant such as their name, email, contact information and picture. Every time the "Order" page is clicked, a GET request is called to display all the merchants. Merchant service is also important to the third sequence, as it dictates which menu items the user is able to edit.

F. Menu

Menu microservice is an atomic microservice that contains information about the menu items. Every Menu object has a merchant ID. Every time a merchant is clicked in the frontend, a GET request is made to display all AVAILABLE menu items to the user. For the third sequence, a GET request is made to display all menu items (available and unavailable). From here, the merchant user can edit the items (name, price, description, availability) as they please and add new items (POST request in menu).

G. Runner

The Runner microservice is a core component that ensures delivery operations run smoothly in the app. It manages two core functions: runner availability management and automated order assignment based on those availabilities.

Runners indicate their availability for the next day through the frontend interface. This triggers an API request that passes through the API Gateway and is routed to the Runner microservice. The submitted time slots are then stored in the database. These records form the foundation for order distribution later on.

When a customer successfully places and pays for an order, the Order microservice publishes an order-status event to RabbitMQ with the routing key `order.status.payment_verified`. This message is routed to the `order.inbox` queue, which the Runner microservice subscribes to. Upon receiving these events, the Runner microservice stores the corresponding orders as pending orders for later assignment.

Order assignment is handled automatically and scheduled to run at predefined times. For instance, if deliveries are scheduled for 12:00 PM, the assignment process runs at 11:00 AM. At that point, the system checks all runners who have indicated availability in the relevant time slot and distributes pending orders evenly among them. This ensures that every available runner receives a fair workload and that deliveries can proceed efficiently without manual intervention. Assigned orders would be shown in the frontend and sent to the runners' email.

When merchants mark an order as "ready for collection", the Order microservice emits another event `order.status.ready_for_collection`, that is received by the Runner microservice. The Runner microservice then looks up which runner was assigned to that order and sends a "ready for collection" email to the corresponding runner. This real-time update ensures that runners can pick up meals immediately once they are ready, reducing delays and improving operational flow.

H. Email

The Email microservice is responsible for delivering timely and event-driven notifications to both customers and runners. The email service operates primarily through AMQP, allowing it to respond asynchronously to events emitted by other microservices such as the Order Service, Payment Service, and Runner Service.

Because of this event-driven architecture, the Email microservice is an atomic service that focuses solely on receiving messages and sending the appropriate emails and acts on the payloads provided by other services. This design makes it highly reusable, scalable, and independent, enabling multiple microservices to invoke it repeatedly without creating tight coupling.

The Email microservice subscribes to a set of RabbitMQ queues. Each queue corresponds to a specific workflow trigger, such as payment reminders, runner assignments, and many more. When a message arrives on any of these queues, the Email microservice extracts the relevant data and generates the appropriate email using a corresponding HTML template.

V. DevOps Practice

To ensure reliability, consistency, and efficient collaboration across our microservices architecture, several DevOps practices were applied throughout the development lifecycle. These

practices enabled smooth integration between team members and allowed automated workflows. The team used GitLab as our central version control platform.

All microservices and databases were containerised using Docker. Containerization ensured consistent environments across development, testing, and deployment by packaging each service with its required dependencies.

We employed Docker Compose to orchestrate and manage multiple containers simultaneously. This enabled us to define and start up the system, such as services, databases, RabbitMQ, and API Gateway.

To ensure code correctness and maintain system stability, the team implemented both unit tests and integration tests. Unit tests validated individual functions and components, while integration tests verified communication between microservices. Automated testing pipelines were triggered upon every push to GitLab, enabling early bug detection and reinforcing continuous integration best practices.

One such instance where the unit tests helped us was when Menu microservice changed a response field from priceCents to price_cents. When I pulled the changes and ran tests locally, menus.client.test.js failed. The test caught that our mock data still used the old format while the client code expected the new one. We fixed it in 5 minutes. Without this test, we would have discovered the issue during our team integration session, wasting 30+ minutes of debugging time.

VI. CI/CD Pipeline

For this project, we implemented a fully automated CI/CD pipeline. CI ensures that every commit is built and tested before being merged into the main branch. Our pipeline consists of four stages: build, static analysis, testing, and Docker-based packaging. Continuous Delivery (CD) treats every successful pipeline run as deployable; in our setup, we still require a manual decision to deploy, whereas full CD would push the changes automatically.

The GitLab pipeline is integrated with our local Minikube Kubernetes cluster via a GitLab Runner. The pipeline automatically executes on every commit and push to the repository, ensuring code correctness before changes are integrated into the main codebase.

To make deployments observable, we modified the `/user/health` endpoint by adding a `"version"` field to the response. This allowed us to verify deployments end-to-end. When a developer pushes a change, the following sequence occurs:

1) **Build stage:** the pipeline compiles the application and prepares it for testing.
2) **Static analysis and testing:** code quality checks and automated tests run to verify the correctness of the changes.
3) **Release stage:** the Docker image is built and pushed to the GitLab Container Registry.
4) **Delivery stage:** the GitLab Runner connected to our Minikube cluster updates the Kubernetes Deployment, triggering a rolling update that replaces the existing pod with a new one running the updated image.

After the pipeline completes, refreshing the `/user/health` endpoint shows the updated `"version": "v2"` field, confirming a successful rollout.

Through this implementation, we experienced several key advantages of a CI/CD pipeline such as early bug detection, consistent build processes, and seperation of concern.

VII. Deployment

For deployment, our initial plan was to host Campus Chow on AWS using ECS. We set up the environment and attempted to deploy our microservices there, but we ran into several configuration issues, particularly around networking and service discovery across multiple microservices. Due to the limited time left in the project, we were unable to complete the full ECS deployment.

To still demonstrate a realistic deployment setup, we switched to Minikube as our deployment target. Minikube allowed us to run a full Kubernetes environment locally while following the same deployment concepts we would have used in the cloud.

We first deployed the User microservice by building and loading a local Docker image directly into Minikube to verify that it worked correctly in a Kubernetes environment. We then formalised this deployment into Kubernetes manifests, configuring the core components:

- a **Deployment** for the User microservice pods,
- a **Service** to expose the Deployment inside the cluster,
- a **StatefulSet** for the PostgreSQL database,
- **ConfigMaps** and **Secrets** for configuration and credentials, and
- an **NGINX Ingress** to route external HTTP traffic into the cluster.

Finally, we wired this Kubernetes configuration into the GitLab CI/CD pipeline described in Section VI (if you add a label), so that successful pipeline runs automatically update the Kubernetes Deployment via the GitLab Runner.

In the future, we would like to revisit deployment on AWS ECS and reuse the same CI/CD flow for a true cloud environment.

VIII. Self-directed Research

A. DevSecOps

DevSecOps is a framework that integrates security into every phase of the software development lifecycle. We applied this by using security checks as part of our CI/CD pipeline. Making it part of our pipeline helped automate and incorporate security practices early and continuously, rather than as an afterthought. We used the following security tests:

- **OWASP Dependency-Check** → Scans libraries for known vulnerabilities.
- **SpotBugs + FindSecBugs** → Finds security bugs in your code (SQL injection, XSS, etc.).

- **Secret Detection** → Finds hardcoded passwords/API keys.
- **Trivy Container Scan** → Scans Docker images for vulnerabilities.
- **Security Config Tests** → Validates secure settings (passwords, ports, etc.).
- **JaCoCo Coverage** → Ensures tests actually run.

One example where we felt the value of DevSecOps was during development, one member had hardcoded a JWT secret token instead of using an environment variable. Our Secret Detection scan immediately flagged this. Without DevSecOps, if the code had reached production it would have allowed anyone with repository access to forge authentication and would pose a serious security risk.

B. Kong API Gateway

Kong serves as our central API Gateway, acting as a single entry point for all client requests to our microservices architecture. This centralization delivers two key benefits:

1) **Ease of frontend development:** Instead of managing seven different service URLs, the frontend only needs to communicate with a single Kong endpoint. We can also implement a unified CORS policy and centralised JWT authentication at the gateway.
2) **Protection against API abuse and DDoS attacks:** We configured global rate limiting at 100 requests per minute per client, which helps protect the platform from malicious traffic.

C. Outbox Pattern

The Order and Payment microservices implement the outbox and inbox patterns to remain robust when the RabbitMQ broker is temporarily unavailable. Without these patterns, the system could end up in an inconsistent state if the broker is down or the process crashes between the database write and message send operations.

In both microservices, the outbox pattern is implemented by using an *outbox* table in the database. Whenever a domain event (such as `order.status.cancelled`) needs to be published, the service does not publish directly to RabbitMQ from the request handler. Instead, it performs a single atomic database transaction that:

1) updates the domain entity (for example, changing the order or payment status), and
2) inserts a row into the outbox table containing the target routing key, serialised payload (JSON), and metadata (such as `messageId`, `sourceService`, and `sentAt`).

If the transaction commits successfully, the state change and the intent to publish the event are durably stored together. A separate background worker in each microservice periodically polls the outbox table for unsent records, publishes the corresponding AMQP messages to RabbitMQ, and then marks those rows as sent. If the broker is down at the time of publishing, the records simply remain in the outbox and will be retried when the broker becomes available again. This design ensures that no domain event is lost, even across process restarts or temporary broker outages.

On the consumer side, the inbox pattern is used to achieve idempotent message handling. Each microservice maintains an *inbox* table keyed by the globally unique `messageId` carried in every AMQP message. An *inbox worker* consumes messages from RabbitMQ and, for each delivery:

1) checks whether the `messageId` already exists in the inbox table, and
2) if it has not been seen before, inserts a new inbox row containing the `messageId`, routing key, payload, and metadata.

Duplicate deliveries with the same `messageId` are detected at this stage and discarded, so the database contains at most one inbox record per message.

A separate *inbox processing worker* then polls the inbox table for unprocessed entries and dispatches them to the appropriate domain-specific handler. The handler logic and the update of any local domain entities are executed within a single database transaction that also marks the inbox row as processed. By splitting ingestion/deduplication (inbox worker) from domain logic (processing worker), the system can safely retry processing on failure while still guaranteeing that each message is handled at most once from the perspective of the business logic.

Together, the outbox and inbox patterns allow the Order and Payment microservices to publish and consume AMQP events reliably without sacrificing consistency, even in the presence of broker downtime or message redelivery.

IX. Reflections

Overall, the team found the project technically challenging, but valuable for learning unfamiliar tools we hadn't worked with before: microservices, AMQP, and Vue.js. A recurring theme across our team members' thoughts is that the frontend was developed too late and should have been built in parallel with the backend using mock data. Poor planning and task management caused significant issues, with some team members missing deadlines or failing to communicate delays effectively. The late integration of frontend to backend components created a major bottleneck that led to unnecessary stress during the final deployment period.

In other considerations, several members felt the initial planning phase was too weak, resulting in unfocused experimentation without clear direction. Deployment persented persistent challenges, particularly with AWS and the Kong API Gateway. Yang Hwee felt that the Ci/CD overhead was excessive for the project's small scale, although he acknowledged its theoretical benefits. Overall, the consensus is that better upfront planning, clearer task assignments, earlier frontend development, and more realistic timeline management would have significantly improved the project experience.

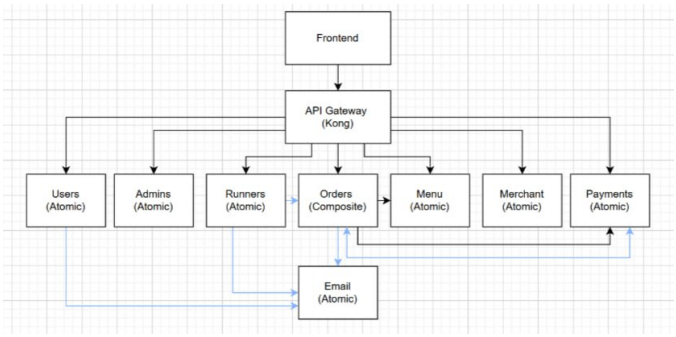

Fig. 1. Overall Campus CHOW Microservice architecture.

APPENDIX A
SYSTEM ARCHITECTURE DIAGRAM

APPENDIX B
KEY SCENARIOS DIAGRAM

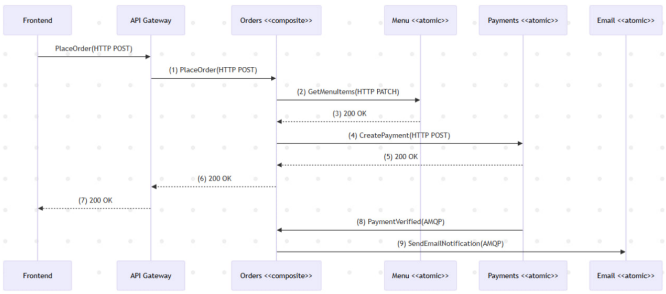

Fig. 2. Place Order Flow Diagram.

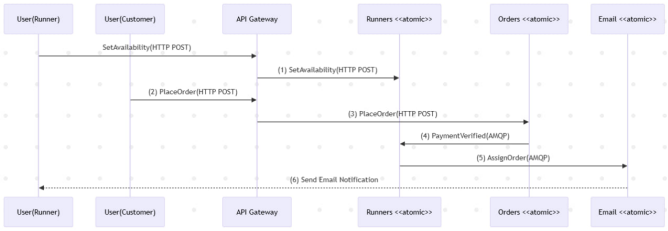

Fig. 3. Assign Orders Flow Diagram.

APPENDIX C
DEVSECOPS PIPELINE

APPENDIX D
INDIVIDUAL REFLECTIONS

A. Matthew

Given more time, it would have been great to get the AWS deployment working, the only thing I couldn't get working was to get the Kong API gateway to correctly route the requests. Also could have tried terraform as I heard it was very useful. Should have had more in depth discussions at the start to clarify understanding on the entire system so there would have been less confusion / surprises.

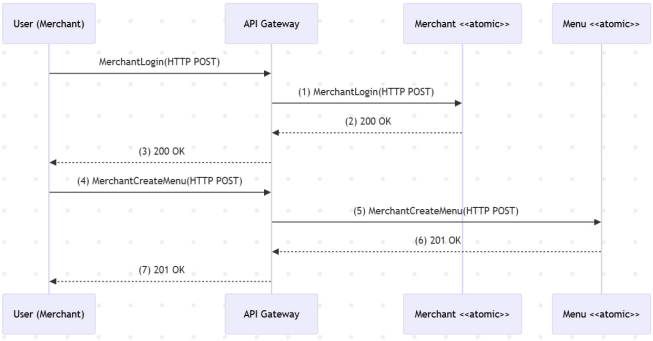

Fig. 4. Merchant Onboarding Flow Diagram.

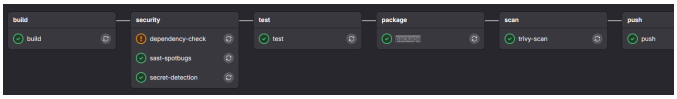

Fig. 5. DevSecOps Pipeline

B. Yang Hwee

I feel that because I was working on a smaller scale project, I could not really appreciate the value of CI/CD. The overhead of setting up pipelines, configuring automated tests, and maintaining the CI/CD infrastructure took time away from actually building features and getting a MVP up. While I do understand the theoretical benefits from class, I feel like it would be more useful for bigger projects with existing users already. I also feel that the front-end should have been developed earlier; in parallel with the backend (with mock values). That way, we would have had more time to deploy everything properly.

C. Xu Duo

Looking back, I wish we had managed our tasks and deadlines better. Next time, I hope everyone can finish their assigned work by the agreed dates, or at least tell the team early if they cannot. I also feel our initial planning was not strong, so we spent a lot of time trying new things without a clear direction. In future projects, I would want us to plan more carefully upfront so we can use our time more efficiently.

D. Doris

This project was definitely challenging, but it taught me a lot, especially when working with microservices and AMQP. One key improvement for future iterations would be to set up the frontend much earlier and start exploring deployment options sooner. Our biggest setback came from connecting the frontend to the backend too late, which left us with limited time to resolve issues.

E. Shiv

I feel as though this project was a complex project involving many tools most of us were unfamiliar with, which ultimately ended up providing a lot of learning. In hindsight, as someone who was responsible for the frontend, I should have managed

my deadlines better and communicated with my team. I was away on a work trip during the project, and I ended up contributing later than initially planned. I believe the delays with linking the frontend and the backend contributed to extra, unnecessary stress during the final deployment period.

F. Lennel

Having worked primarily on the frontend, I gained valuable experience strengthening my proficiency in Vue.js, a framework I've come to appreciate for its lightweight nature and flexibility across different use cases. One practical approach I implemented was the use of mocked data, which I found to be helpful in the integration process. By developing components with mock responses early, the integration process became easier, as it required switching from static placeholder values to dynamic data retrieved from the database. My biggest takeaway, however, is the importance of strong coordination between the frontend and backend teams. The time needed for integration and deployment must be carefully planned and incorporated into our progressive deadlines to avoid bottlenecks. I also believe that clearer task assignments and more well-defined internal deadlines within the frontend team would have helped streamline our workflow and accelerated the overall development process.

APPENDIX E
WORK DISTRIBUTIONS

TABLE I
WORK DISTRIBUTION AMONG TEAM MEMBERS

Name	Contributions
Matthew	Project Idea; Coding, CI/CD, and integration of the Order and Payment microservices with the frontend and Kong API gateway; attempted deployment on AWS.
Yang Hwee	Implemented Merchant and Menu microservice and connected them to the frontend and kong api gateway. Set up CICD pipeline for both services. Helped implement automatic transaction detection from email for payment microservice. Did the frontend pages for the third scenario. Deployed Merchant Menu service using AWS copilot but could not get it to work as part of CD pipeline.
Lennel	Implemented majority of the frontend and required pages: Landing (Home) page, Payment page,Login page, Register page, Runner page, Merchants page, Menu page, Order pages (add to cart, order details), Checkout pages (order summary, payment)
Doris	Implemented Runner and Email microservice and connected them to frontend and kong API gateway. Set up CICD pipeline for both services. Attempted deployment using Kubernetes (Minikube) for the runner microservice.
Xu Duo	Implemented the user and admin microservices and connected it to the frontend; built the admin section frontend; set up GitLab CICD for both services and Kubernetes (Minikube) deployment for the user microservice.
Shiv	Implemented remaining frontend requirements: Profile page, Active Orders page, Past Orders page, FAQ page, Contact page, Play page (Spin the Wheel game, Guess the Merchant game)

APPENDIX F
AMQP MESSAGE CONTRACTS

All AMQP events carry a unique `messageId` (UUID v4), a `sourceService` header, and a `sentAt` timestamp.

TABLE II
AMQP MESSAGES PUBLISHED BY THE ORDER MICROSERVICE

Routing key / pattern	Target service(s)	Purpose / payload summary
order.status. {new_status}	Any interested consumer (e.g., Payment, Email, Runner)	Emitted whenever an order's status changes. Payload contains an order snapshot (including `order_id`, status, delivery time, payment deadline, location, items, and amounts). Consumers react to specific statuses such as `ready_for_collection` or `cancelled`.
order.created	Payment microservice (and other interested services)	Emitted when a new order is created and is awaiting payment. Payload is the same order snapshot. The Payment microservice uses it to create a corresponding payment record, and other services may use it for confirmation or logging.
email.command. send_payment _reminder	Email microservice	Emitted by a scheduled reminder job when an order remains unpaid close to its payment deadline. Payload again contains the order snapshot so the Email service can compose and send a payment reminder email.

TABLE III
AMQP MESSAGES SUBSCRIBED BY THE ORDER MICROSERVICE

Routing key	Source service(s)	Purpose / payload summary
order.command. status_update	Runner, Merchant	Carries a status-update command for an existing order. Payload has the shape {`"orderId": <id>, "newStatus": "<status>"`}. Runners send updates such as `delivering` or `delivered`, and merchants send `preparing` or `ready_for_collection`. The Order microservice validates and applies the new status and then emits the corresponding `order.status.{new_status}` event.
payment.status. payment_verified	Payment microservice	Event emitted when a payment has been successfully verified. Payload has the shape {`"orderId": <id>, "newStatus": "payment_verified"`}. The Order microservice uses `orderId` to locate the order and update its status accordingly.

TABLE IV
AMQP MESSAGES PUBLISHED BY THE PAYMENT MICROSERVICE

Routing key / pattern	Target service(s)	Purpose / payload summary
payment.status. {new_status}	Order microservice (and other interested consumers)	Emitted whenever a payment's status changes. The payload has the shape {`"orderId": <id>, "newStatus": "<status>"`}.

TABLE V
AMQP MESSAGES SUBSCRIBED BY THE PAYMENT MICROSERVICE

Routing key	**Source service**	**Purpose / payload summary**
order.status. cancelled	Order microservice	Emitted whenever an order's status changes to `cancelled`. The payload contains an order snapshot including `order_id`. The Payment microservice uses this to locate the corresponding payment and mark its status as `failed`.

TABLE VI
EMAIL MICROSERVICE QUEUES

Queue	**Purpose**
email.command. send_payment_reminder	Sends customers a payment reminder before their payment deadline.
runner.assignment.queue	Sends runners a summary of assigned orders, including delivery info and items.
runner.order.ready.queue	Notifies a runner that their assigned order is ready for collection.
order.status.cancelled.queue	Sends customer an email stating their order has been cancelled.
order.status.payment_verified	Sends a confirmation email after a successful payment.
account.welcome.queue	Sends a welcome email when a user signs up.
internal.test.queue	Allows developers to test email sending internally without triggering real flows.