## UI Flows:

### Student (Customer) Placing Order:

Customer create account & adds payment method via Profile UI

**Customer places an immediate order via Customer UI**

**~~Customer places a scheduled order via Customer UI~~**

Customer views order status

***Customer is prompted to retry or cancel order (Scenario 1)***

***Customer cancels order via Customer UI***

Customer reviews order

### Student (Runner) Delivering Order:

Runner views available orders for request via Delivery UI

**Runner accepts order request via Delivery UI**

***Runner cancels acceptance on order via Delivery UI***

Runner updates that order is on picked-up via Delivery UI

Runner updates that order has been delivered via Delivery UI

*Legend:*

*Bold: Key scenarios*

*Italics: Hard scenarios*

#### status

orderStatus:

pendingPayment → [if paymentStatus = authorized] created [else, break] failed → if timeout: ‘timeoutCancelled’, else ‘accepted’ → onSite → purchased → collected → onTheWay → delivered → completed (after paymentStatus = released)

paymentStatus:  
initiating → authorized/failed → [if paymentStatus = authorized] inEscrow → released

INITIATING

AUTHORIZED

INESCROW

FAILED

escrowStatus:

initiating → held → released/refunded

initiating → failed

initiating → held → failed

failed

| **Service** | **Endpoints** | **Kafka Binding Keys** |
| --- | --- | --- |
| **User Service** | - **GET/getPaymentInfo** – Retrieves payment information (customer in Create Order Saga; runner in Complete Order Saga).  *Rollback:* Not needed (saga stops on failure).  - **POST/updateRatingCustomer** - Updates the rating of the customer based on their order cancellation rate  - **POST/updateRatingRunner** - Updates the rating of the runner based on the user’s input & their delivery order cancellation rate  *- GET/getRunnerInfo - For front end service*  *- GET/getCustomerInfo - For front end service* | - **user.paymentInfoRetrieved**  **- user.updateRatingCustomer**  **- user.updateRatingRunner** |
| **Order Service** | - **POST /createOrder** – Creates a new order and initializes status.  - **POST /updateOrderStatus** – Updates the order status.  - **POST /cancelOrder** – Cancels the order and marks it as "Cancelled" (*rollback*).  - **POST /verifyAndAcceptOrder** – Verifies order availability and accepts order.  - **POST /cancelAcceptance** – Reverts acceptance (*rollback*)  - **POST /completeOrder** – Finalizes the order to "Completed".  *- GET/getOrderDetails - For front end service* | - **order.created**  - **order.updated**  - **order.cancelled** (*rollback*)  - **order.accepted**  - **order.acceptanceCancelled** (*rollback*)  - **order.statusUpdated**  - **order.completed** |
| **Notification Service** | - **POST /sendNotification** – Sends notifications (via Kafka in Create Order Saga; via Twilio for order acceptance in Accept Order Saga).  - **POST /revertNotification** – Retracts or updates notifications (*rollback*). | - **notification.sent**  - **notification.orderAccepted**  - **notification.reverted** (*rollback*) |
| **Payment Service** | - **POST /authorizePayment** – Initiates payment authorization via Stripe.  - **POST /revertPayment** – Reverts payment authorization or initiates a refund (*rollback*). | - **payment.authorized**  - **payment.reverted** (*rollback*) |
| **Escrow Service** | - **POST /holdFunds** – Holds funds in escrow.  - **POST /releaseFunds** – Handles fund release to the runner on order completion and refunds to the customer on rollback. | - **escrow.fundsHeld**  - **escrow.fundsReleased** (funds sent to runner)  - **escrow.fundsRefunded** (*rollback*: funds refunded to customer) |
| **Timer Service** | - **POST /startOrderTimer** – Schedules the order for execution.  - **POST /cancelTimer** – Cancels the scheduled task (*rollback*). | - **timer.started**  - **timer.cancelled** (*rollback*) |

### Order Service Schema

| **Field** | **Updated Schema** | **Comments** |
| --- | --- | --- |
| **order\_id** | String (UUID) | Primary key. |
| **cust\_id** | String | References the user (customer); kept as is. |
| **runner\_id** | String (nullable) | Nullable since a runner may not be attached initially. |
| **order\_description** | Text | As required. |
| **food\_fee** | Decimal (5,2) *10 digits total, 2 digits after decimal* | As required. |
| **delivery\_fee** | Decimal (5,2) | As required. |
| **delivery\_location** | String (255) *input from user* | As required. |
| **order\_status** | Enum (‘CREATED’, ‘ACCEPTED’, ‘PLACED’, ‘ON\_THE\_WAY’, ‘DELIVERED’, ‘COMPLETED’, ‘CANCELLED’) | Reflects the saga flow through order creation, acceptance, placement, and completion. |
| **created\_at** | DateTime (default = now) | For audit. |
| **updated\_at** | DateTime (auto-updated) | For audit. |
| **completed\_at** | DateTime (nullable) | As required. |

### User Service Schema

| **Field** | **Updated Schema** | **Comments** |
| --- | --- | --- |
| **user\_id** | String (UUID) | Primary key. |
| **email** | String | As required. |
| **first\_name** | String | As required. |
| **last\_name** | String | As required. |
| **phone\_number** | String | New field; retained for contact purposes. |
| **user\_stripe\_card** | JSON | As required for payment info. |
| **customer\_rating** | Decimal | New field; kept for performance evaluation. |
| **runner\_rating** | Decimal | New field; kept for performance evaluation. |
| **created\_at** | DateTime (default = now) | As required. |
| **updated\_at** | DateTime (auto-updated) | As required. |

### Payment Service Schema

| **Field** | **Updated Schema** | **Comments** |
| --- | --- | --- |
| **payment\_id** | String (UUID) | Primary key. |
| **order\_id** | String | As required. |
| **customer\_id** | String | As required. |
| **amount** | Decimal (5,2) | As required. |
| **status** | Enum (‘INITIATING’, ‘AUTHORIZED’, ‘INESCROW’, ‘FAILED’) | . |
| **customer\_stripe\_card** | String *(Pending API integration)* | Placeholder; update when API integration is complete. |
| **stripe\_payment\_id** | String *(Pending API integration)* | Placeholder; update when API integration is complete. |
| **created\_at** | DateTime (default = now) | As required. |
| **updated\_at** | DateTime (auto-updated) | As required. |

### Escrow Service Schema

| **Field** | **Updated Schema** | **Comments** |
| --- | --- | --- |
| **escrow\_id** | String (UUID) | Primary key. |
| **order\_id** | String | As required; |
| **customer\_id** | String | As required. |
| **runner\_id** | String (nullable) | Remains nullable since a runner may not be attached initially. |
| **amount** | Decimal (10,2) | As required. |
| **food\_fee** | Decimal (10,2) | As required. |
| **delivery\_fee** | Decimal (10,2) | As required. |
| **status** | Enum (‘PENDING’, ‘HELD’, ‘RELEASED’, ‘REFUNDED’, ‘FAILED’) | Updated per requirements. |
| **created\_at** | DateTime (default = now) | As required. |
| **updated\_at** | DateTime (auto-updated) | As required. |

### Timer Service Schema

| **Field** | **Updated Schema** | **Comments** |
| --- | --- | --- |
| **timer\_id** | String (UUID) | Primary key. |
| **customer\_id** | String | As required |
| **runner\_id** | String | As required |
| **order\_id** | String | As required |
| **runner\_accepted** | Boolean | Checks if runner has accepted request |
| **created\_at** | DateTime (default = now) | As required. |
| **~~event\_type~~** | ~~Enum (‘ORDER\_TIMEOUT’, ‘PAYMENT\_AUTHORIZATION\_EXPIRY’, ‘DELIVERY\_REMINDER’, ‘ESCROW\_TIMEOUT’)~~ | ~~Use enum for consistency as in the original schema.~~ |
| **~~scheduled\_time~~** | ~~DateTime~~ | ~~As required.~~ |
| **~~processed~~** | ~~Boolean~~ | ~~Simple indicator for whether the event was processed (replaces retryCount/lastProcessed).~~ |
| **~~payload~~** | ~~JSON~~ | ~~As required for event-specific data.~~ |
| **~~updated\_at~~** | ~~DateTime (auto-updated)~~ | ~~As required.~~ |

### Notification Service Schema

| **Field** | **Updated Schema** | **Comments** |
| --- | --- | --- |
| **notification\_id** | String (UUID) | Primary key. |
| **customer\_id** | String | As required |
| **runner\_id** | String | As required |
| **order\_id** | String | As required |
| **event** | String | message body |
| **status** | Enum (‘CREATED’, ‘SENT’) | to keep track of when the notif is sent via Twilio (updated to SENT when twilio gives a response) |
| **sent\_at** | DateTime | As required. |
| **created\_at** | DateTime (default = now) | Sufficient for audit; updated\_at can be omitted if not required. |
| **~~recipient\_id~~** | ~~String~~ | ~~As required.~~ |
| **~~recipient\_type~~** | ~~Enum (CUSTOMER, RUNNER)~~ | ~~Limited to CUSTOMER and RUNNER; avoids confusion.~~ |
| **~~type~~** | ~~Enum (e.g., ORDER\_CREATED, ORDER\_ACCEPTED, etc.)~~ | ~~Use enum for fixed notification types.~~ |
| **~~title~~** | ~~String~~ | ~~As required.~~ |
| **~~message~~** | ~~String~~ | ~~As required.~~ |
| **~~data~~** | ~~JSON~~ | ~~As required for extra details.~~ |
| **~~read~~** | ~~Boolean~~ | ~~Renamed from isRead; indicates if notification has been read.~~ |

### Kafka Event Structure

For each Kafka message, I recommend the following structure:

{

"type": "EVENT\_TYPE",

"timestamp": "ISO-8601 timestamp",

"payload": {

// Event-specific data

}

}

### Python/Flask Services (using SQLAlchemy)

1. Configure database connection in an environment variable:

# app/config/config.py

import os

DATABASE\_URL = os.getenv('DATABASE\_URL', 'postgresql://postgres:postgres@postgres:5432/order\_db')

1. Setup SQLAlchemy in your Flask app:

# app/\_\_init\_\_.py

from flask import Flask

from flask\_sqlalchemy import SQLAlchemy

from flask\_migrate import Migrate

db = SQLAlchemy()

migrate = Migrate()

def create\_app():

app = Flask(\_\_name\_\_)

app.config['SQLALCHEMY\_DATABASE\_URI'] = DATABASE\_URL

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False

db.init\_app(app)

migrate.init\_app(app, db)

# Register blueprints and routes

return app

1. Run migrations:

flask db init

flask db migrate -m "Initial migration"

flask db upgrade

### Local Testing Approach

I recommend a multi-stage testing approach:

1. **Unit Testing**: Test individual components in isolation
2. **Integration Testing**: Test interaction between components
3. **End-to-End Testing**: Test complete workflows

For local testing, I suggest:

1. **Docker Compose for local environment**:
   * Your docker-compose.yml is already well-configured
   * Use it to run all services and dependencies locally
2. **Service-level testing**:
   * Each service should have its own unit tests
   * Mock external service dependencies
   * Use in-memory databases for fast tests
3. **Integration testing with Kafka**:
   * Test message production and consumption
   * Verify saga patterns work correctly

### Kubernetes Implementation

Your project already has a Kubernetes setup in the kubernetes/ directory. To enhance it:

1. **Create Helm charts** for easier deployment management
2. **Configure proper logging** with:
   * EFK (Elasticsearch, Fluentd, Kibana) or
   * PLG (Prometheus, Loki, Grafana) stack
3. **Setup proper health checks** for each service:
   * Add health check endpoints to each service
   * Configure Kubernetes liveness and readiness probes

### Service Logs Management

For viewing individual service logs:

1. **Centralized Logging with Elasticsearch + Kibana**:
   * Deploy EFK stack in Kubernetes
   * Configure each service to send logs to Fluentd
   * Create Kibana dashboards for each service
2. **Distributed Tracing**:
   * Implement OpenTelemetry in each service
   * Use Jaeger or Zipkin for tracing visualization
3. **Kafka-based Log Aggregation**:
   * Create a dedicated Kafka topic for logs
   * Have each service publish logs to this topic
   * Implement a log consumer service that stores logs

### Monitoring and Metrics

1. **Service Metrics**:
   * Implement Prometheus metrics in each service
   * Set up Grafana dashboards for visualization
2. **Kafka Metrics**:
   * Monitor Kafka performance and message throughput
   * Create alerts for message processing delays

## Services Endpoints:

## Create Order Saga

### User Service

**Endpoints:**

* **GET** /getPaymentInfo – Retrieves the payment information of the customer.
* **Rollback:** Not needed (if user retrieval fails, the saga stops).

**Kafka Binding Keys:**

* **Event:** user.paymentInfoRetrieved

### Order Service

**Endpoints:**

* **POST** /createOrder – Creates a new order and initializes status.
* **POST** /updateOrderStatus – Updates the order status.
* **POST** /cancelOrder – Cancels the order and marks it as **"Cancelled"** (rollback).

**Kafka Binding Keys:**

* **Event:** order.created
* **Status Update:** order.updated
* **Rollback:** order.cancelled

### Notification Service

**Endpoints:**

* **POST** /sendNotification – Sends notifications to users via Kafka.
* **~~POST~~** ~~/revertNotification – Retracts or updates notifications (rollback).~~

**Kafka Binding Keys:**

* **Event:** notification.sent
* **~~Rollback:~~** ~~notification.reverted~~

### Payment Service

**Endpoints:**

* **POST** /authorizePayment – Initiates payment authorization via Stripe.
* **POST** /revertPayment – Reverts payment authorization or initiates a refund (rollback).

**Kafka Binding Keys:**

* **Event:** payment.authorized
* **Rollback:** payment.reverted

### Escrow Service

**Endpoints:**

* **POST** /holdFunds – Holds funds in escrow.
* **POST** /releaseFunds – Handles both **fund release to the runner (on order completion)** and **refunds to the customer (on failure/rollback)**.

**Kafka Binding Keys:**

* **Event:** escrow.fundsHeld
* **Event:** escrow.fundsReleased (funds sent to runner)
* **Rollback:** escrow.fundsRefunded (funds refunded to customer)

### Timer Service

**Endpoints:**

* **POST** /startRequestTimer – Starts the timer for the initial creation of request
* **POST** /stopRequestTimer – Stops the timer when the request has been accepted.
* **POST** /cancelTimer – Cancels the running timer (rollback).

**Kafka Binding Keys:**

* **Event:** timer.started
* **Event:** timer.stopped
* **Rollback:** timer.cancelled

## Accept Order Saga

### Order Service

**Endpoints:**

* **POST** /verifyAndAcceptOrder – Verifies order availability and updates the order status to **"Accepted"** if successful.
* **POST** /cancelAcceptance – Reverts the acceptance by unbinding the runner and resetting the order status (rollback).

**Kafka Binding Keys:**

* **Event:** order.accepted
* **Rollback:** order.acceptanceCancelled

### Notification Service

**Endpoints:**

* **POST** /sendNotification – Sends an order acceptance notification via Twilio.
* **~~POST~~** ~~/revertNotification – Retracts or updates the notification if order acceptance is rolled back.~~

**Kafka Binding Keys:**

* **Event:** notification.orderAccepted
* **~~Rollback:~~** ~~notification.reverted~~

## Complete Order Saga

### Order Service

**Endpoints:**

* **POST** /updateOrderStatus – Updates the order status at different stages (**On-Site, Order Purchased, Collected, On the Way, Delivered, Completed**).
* **POST** /completeOrder – Finalizes the order status to **"Completed"**. *(not triggered from runner UI but internally)*

**Kafka Binding Keys:**

* **Event:** order.statusUpdated
* **Event:** order.completed

### User Service

**Endpoints:**

* **GET** /getPaymentInfo – Retrieves the payment information for the runner.

**Kafka Binding Keys:**

* **Event:** user.paymentInfoRetrieved

### Escrow Service

**Endpoints:**

* **POST** /releaseFunds – Handles both **fund release to the runner** (on successful order completion) and **fund refunds to the customer** (on rollback).

**Kafka Binding Keys:**

* **Event:** escrow.fundsReleased (funds sent to runner)
* **Rollback:** escrow.fundsRefunded (funds refunded to customer)

## A. Standard Order Lifecycle

*Design Principle:  
Every core action is implemented with built-in retry mechanisms (with exponential backoff) to handle transient errors. Only key business actions have specific compensation steps described below. All events and state changes are recorded via the Log Service as individual atomic steps.*

### 1. Create Order Saga (Order Creation & Payment)

| **Step** | **Task** | **Microservice Breakdown** | **Rollback Task** |
| --- | --- | --- | --- |
| **1** | **Generate Order ID & Create Order Record:** Create an order record with status "pendingPayment." | Order Service | If creation fails after retries, log it (Failed) and exit, telling the user to try again. |
| **2** | **Log Order Status:** Record the order creation with status “pendingPayment” | Log Service |  |
| **3** | **Process Payment Authorization:** Transition payment status from Initiating → Authorized/Failed. | Payment Service | If payment authorization ultimately fails, cancel the order and ask the user to check their payment methods. |
| **3.1** | **Log Payment Authorization:** Record the outcome along with order details and the latest payment status. | Log Service | – |
| **4** | **Place Funds in Escrow:** Move authorized funds to "inEscrow." | Escrow Service | If escrow transfer fails after retries, trigger a refund and cancel the order and tell the user to try again later. |
| **4.1** | **Log Escrow Placement:** Record the escrow update with order and payment details. | Log Service | – |
| **5** | **Update Order Status to Created:** Mark the order as "Created." and alert scheduler service. | Order Service, Scheduler Service | If updating fails after retries, reverse the payment capture and escrow actions, informing the user to try again later. |
| **5.1** | **Log Created Status:** Record the status change (order marked as Created). | Log Service | – |
| **6** | **Trigger UI**: Show the updated status | Log Service | – |
| **7** | **Timeout: Hard Scenario 1** | | |

### **2. Accept Order Saga (Order Acceptance)**

| **Step** | **Task** | **Microservice Breakdown** | **Compensation/Retry Action (Key Actions Only)** |
| --- | --- | --- | --- |
| **1** | Verify that the order is available and bind runner | Order Service | If verification fails after retries, return an error to the runner, unable to take up order.  If binding fails after retries, revert the order status to **"**Created.**" -** Place back into order list (PQ) |
| **2** | Log Order as Accepted: Record the order assignment (or reversion) event with the updated details. | Log Service | – |

### **3.** Complete **Order Saga (Order Delivery & Completion)**

| **Step** | **Task** | **Microservice Breakdown** | **Compensation/Retry Action** |
| --- | --- | --- | --- |
| **1** | **Runner Arrives at Store:** Update order status to "Onsite” | Order Service, User Service, Location Service, Notification Service | If the pickup tracking fails after retries, cancel the pickup and revert the order status. |
| **2** | **Log Arrival at Store:** Record the updated status ("Onsite"). | Log Service | – |
| **3** | **Runner Places Order:** Uploads photo of receipt, triggering order status to be updated to "Order Purchased" | User Service, Order Service, Notification Service |  |
| **4** | **Log Order Placed:** Record the "Order Purchased" status along with order details. | Log Service | – |
| **5** | **Runner Collects Order: “Collected” Status** | User Service, Order Service, Notification Service |  |
| **6** | **Log Collection of Order** |  |  |
| **7** | **Runner is On the Way:** Update order status to "On The Way" | User Service, Order Service, Notification Service, Location Service | – |
| **8** | **Log On The Way Status:** Record the "On The Way" status update. | Log Service | – |
| **9** | **Process Delivery Confirmation:** Runner submits a confirmation picture with a timestamp, and the system updates the order status to "Delivered." | User Service, Order Service | – |
| **10** | **Log Delivery Confirmation:** Record the confirmation details and updated status. | Log Service | – |
| **11** | **Release Payment from Escrow:** Instruct the Escrow Service to release funds to the runner, updating payment status to "Released." | Escrow Service, Payment Service | - |
| **12** | **Log Payment Release:** Record the payment release event with associated order details. | Log Service | – |
| **7** | **Finalize Order:** Update the order status to "Completed," generate receipts, and prompt for ratings/feedback. | Order Service |  |
| **7.1** | **Log Finalization:** Record the final order state as "Completed." | Log Service | – |

## **B. Hard Sce**narios Detailed Steps

### Scenario 1: Order Timeout & Refund Process

**User Story:** A student places an order during a slow period, but no delivery student accepts it within the 30-minute window.

Following the last step after order creation:

| **Step** | **Task** | **Microservice Breakdown** |
| --- | --- | --- |
| **1** | **Timeout Detection:** Scheduler Service sets a 30-minute timer and, if no acceptance occurs, publishes an **OrderTimeout** event. | Scheduler Service |
| **1.1** | **Log Timeout Event:** Record the timeout event. | Log Service |
| **2** | **Order Cancellation:** Order Service receives the **OrderTimeout** event and **cancels the order**, publishing an **OrderCancelled** event. | Order Service |
| **2.1** | **Log Order Cancellation:** Record the cancellation event. | Log Service |
| **3** | **Refund Initiation:** Escrow Service receives the **OrderCancelled** event and initiates a refund, updating status to **Refunded** and publishing a **RefundInitiated** event. | Escrow Service, Payment Service |
| **3.1** | **Log Refund Initiation:** Record the refund event. | Log Service |
| **4** | **Customer Notification:** Notification Service sends notifications regarding the order cancellation and refund. | Notification Service |
| **4.1** | **Log Notification Event:** Record the notification event. | Log Service |