

WineShop: E-Commerce Platform for Wine Sales with Microservices Architecture and Distributed Transactions

Final Report for the Distributed Systems Course [2025/2026]

Giovanni Maria Savoca
`giovannimaria.savoca@studio.unibo.it`

Enrico Valentini
`enrico.valentini7@studio.unibo.it`

April 1, 2026

This report describes the design and implementation of an e-commerce platform for wine sales built following a microservices architecture. The system is composed of seven independent services — users, catalog, inventory, cart, orders, and payments — coordinated by an API gateway and communicating via RabbitMQ as a message broker. The main objective of the project is to address two fundamental challenges of distributed systems: managing transactions that span multiple services without global ACID transactions, and maintaining the consistency of a distributed cache. The first challenge is solved through the implementation of the Saga Pattern in choreographed style, where services react to events published on a RabbitMQ topic exchange without a centralized orchestrator. The second is addressed through Redis, used both as a cache layer for the catalog and for temporary persistence of guest user carts, with asynchronous key invalidation when the underlying data changes. User authentication is handled via JSON Web Tokens, while payments are delegated to Stripe with cryptographic verification of incoming webhooks. The entire system is containerized with Docker Compose, which manages the lifecycle of all infrastructure components including six PostgreSQL databases, RabbitMQ, and Redis.

1 Concept

The project consists of an e-commerce web application specialized in wine sales, developed with a microservices architecture. The system exposes a graphical interface accessible via browser on desktop or laptop devices, through which users can browse a product catalog, manage a shopping cart, and complete purchases with credit card payment.

The end users of the platform are private consumers who make online purchases on an occasional basis. They interact with the system via browser, in sessions that typically include catalog exploration, adding items to the cart, and finally checkout. The system needs to store user personal data, their cart sessions, product stock availability, and the history of placed orders.

Distribution is necessary for several reasons. First, the different subdomains of the system — user management, catalog, cart, inventory, orders, payments — have very different load, update, and persistence requirements: a microservices architecture allows scaling independently only

the components that require it. Second, managing a purchase requires coordinating multiple critical operations that must result in a globally consistent state while involving autonomous services with their own databases: this makes the problem of distributed transactions a real and fundamental challenge of the project. Finally, delegating payment to an external provider (Stripe) inherently imposes distributed communication with heterogeneous systems.

2 Requirements Elicitation and Analysis

Functional Requirements

- The system must allow user registration and authentication. Acceptance criterion: A user can register with email and password, log in, and receive a valid JWT token for subsequent operations.
- The system must expose a wine catalog browsable with filters by type (red, white, rosé, sparkling, box) and text search by name. Acceptance criterion: to avoid overloading the service with excessive requests, Redis cache is used to store recurring catalog requests.
- The system must allow management of a persistent cart, with the ability to add, remove, and modify item quantities. Acceptance criterion: an authenticated user who logs out and logs back in finds their cart unchanged.
- The system must support unauthenticated users: their cart is maintained in Redis and migrated to the database when they log in. Acceptance criterion: after login, the guest cart is visible as the user's cart.
- The system must allow completion of a purchase through checkout: order creation, stock availability verification, payment authorization via Stripe, and final confirmation. Acceptance criterion: at the end of the flow, the order is in **processing** status, the reserved inventory quantity matches the purchased quantity, and the payment has been captured by Stripe.
- In case of payment failure or order cancellation, the system must restore the inventory to its previous quantities. Acceptance criterion: after a failed payment, the available product quantity is unchanged compared to the pre-checkout situation.
- An authenticated user must be able to view the history of their orders. Acceptance criterion: the order list shows the status and details of each previous purchase.

Non-Functional Requirements

- **Eventual consistency:** the system does not guarantee global atomicity across services, but guarantees that, in the absence of further modifications, the overall state converges to a consistent state.
- **Graceful degradation:** if the order service is temporarily unavailable, users must still be able to browse the catalog and manage the cart.
- **Payment security:** the system does not directly handle credit card data; payments are delegated to Stripe with cryptographic webhook verification.
- **Message durability:** events published on RabbitMQ must be persistent, so that a broker restart does not cause loss of messages not yet processed.

Implementation Requirements

- All backend services must be implemented in TypeScript on Node.js, for codebase uniformity and type safety.
- Asynchronous communication between services must occur via RabbitMQ, which enables message exchange between services and therefore their orchestration.
- The entire system must be containerized with Docker and orchestrated via Docker Compose to simplify deployment and ensure environment reproducibility.
- The frontend must be a Dynamic Single Page Application (SPA) developed with React and TypeScript, using React Routes.

2.1 Relevant Distributed System Features

Fault tolerance and dependability. Fault tolerance is a fundamental property of the system. The Saga Pattern ensures that a temporary failure of a service (for example, the payment service) does not leave the system in an inconsistent state: RabbitMQ messages persist in the queue until the service recovers, which reprocesses them as soon as it is available. Graceful degradation is also implemented in the cart service, which connects to Redis in non-blocking mode and operates correctly even if Redis is unreachable.

Scalability. Each service is independent and can be horizontally scaled in response to specific loads. The catalog, which has an access pattern with many reads and few writes, particularly benefits from Redis caching. The use of a message broker decouples producers from consumers, allowing instances to be added without modifications to existing services.

Security and trust. The system handles user personal data and financial transactions, so security and authentication are primary requirements. Authentication is based on JWT, verified by each service that exposes protected endpoints. Payments are delegated to Stripe; incoming webhooks from Stripe are authenticated via HMAC-SHA256 cryptographic signature.

Transparency. Distribution is hidden from the end user, who interacts with a single entry point (the API gateway). Internal Docker service names are never exposed to the frontend. Service discovery occurs through Docker's internal DNS, transparent to the services themselves.

Resource sharing. RabbitMQ and Redis are shared resources among multiple services. Concurrency on RabbitMQ queues is managed by the broker, which guarantees exactly-once delivery per consumer. Concurrency on PostgreSQL writes is managed through SQL transactions with row-level locks.

Interoperability and heterogeneity. The system integrates with Stripe, a heterogeneous external service, via REST API over HTTPS and signed webhook verification. Internal communication between services occurs via REST+JSON and AMQP, standard and widely supported protocols.

Performance and concurrency. The catalog is the component with the highest read load. Redis drastically reduces average latency, eliminating the round-trip to the PostgreSQL database for the most frequent queries. Distributed transactions via saga occur asynchronously, without blocking the client response thread.

Evolvability and maintainability. Database isolation (a dedicated PostgreSQL database per service) allows modifying a service’s schema without impact on others. The event-driven architecture enables adding new consumer services for existing events without modifying the producers.

3 Design

The design choices directly follow the requirements identified in section 2 and the distributed characteristics discussed in section 2.1. The guiding principle is the separation of business domains into autonomous services, coordinated asynchronously through events.

3.1 Architecture

The adopted architecture is **microservices**, with hybrid communication: synchronous (REST over HTTP) for operations requiring an immediate response to the client (catalog browsing, cart management), and asynchronous (AMQP over RabbitMQ) for workflows involving multiple services and requiring failure handling and compensations (checkout, order cancellation).

The choice of microservices is motivated by the need to independently scale components with different loads, to isolate failures, and to allow independent evolution of individual domains. The catalog, for example, is read-heavy and benefits from aggressive caching; the payment service has stricter security requirements and an external dependency (Stripe) that it does not share with others.

Within the checkout flow, the **choreographed Saga Pattern** is implemented. Unlike an orchestrated saga, where a central coordinator issues commands to services, in the choreographed saga each service is autonomous: it publishes events when it completes its local transaction, and reacts to events published by others. This approach maximizes decoupling and resilience, at the cost of greater complexity in tracking the global state.

3.2 Infrastructure

The infrastructure components of the system are the following.

An **API gateway** (Node.js/Express) acts as a reverse proxy and single entry point for the frontend. It handles routing to internal services, forwarding of authorization headers, and CORS configuration.

Seven **backend services** (Node.js/TypeScript/Express): user-service, catalog-service, inventory-service, cart-service, order-service, payment-service, and the api-gateway itself. Each exposes a REST API on a dedicated port.

Six **PostgreSQL 15 databases**, one per service (except the gateway). Each service has exclusive access to its own database, ensuring schema isolation and impossibility of direct coupling between services via cross-database joins.

A **RabbitMQ 3 message broker** exposing the AMQP protocol on port 5672 and a management UI on 15672. It serves as the backbone for asynchronous communication between services.

A **Redis** instance used by the catalog-service as a cache and by the cart-service for guest cart persistence.

The **frontend** is a React SPA served by a Vite development server on port 3030.

All components run on the same Docker host (local development), on the same Docker bridge network, and are therefore on the same virtual network segment. Service discovery occurs through Docker’s internal DNS: each container is reachable from others via the service name defined in `docker-compose.yml` (for example, `inventory-service:3003`). Load balancing is not present in the current configuration, but the architecture would support it by adding multiple instances of the same service.

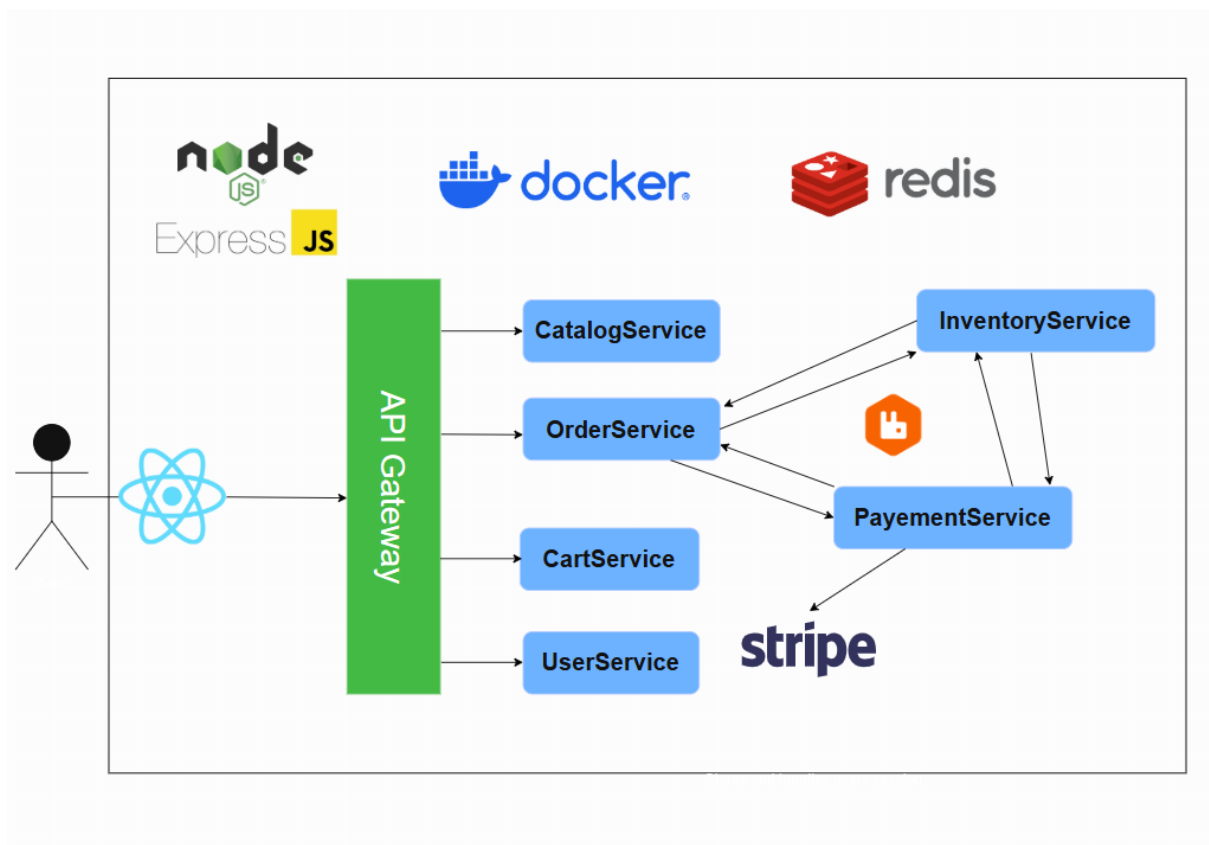


Figure 1: Microservices topology and main event flow

3.3 Modelling

The main **domain entities** of the system are:

- **User** (user-service): email, password, personal data, shipping addresses.
- **Wine** (catalog-service): SKU, name, description, price, type, winery, vintage, region, alcohol content, image URL.
- **Inventory** (inventory-service): available quantity, reserved quantity, minimum stock threshold for each wine.
- **Cart** and **CartItem** (cart-service): association between user/guest session and products with quantity and unit price at the time of addition.
- **Order** and **OrderItem** (order-service): order with status (pending/processing/failed), total, shipping address, and detail of purchased items.
- **Transaction** (payment-service): amount, currency, payment status (pending/captured/failed), order reference.

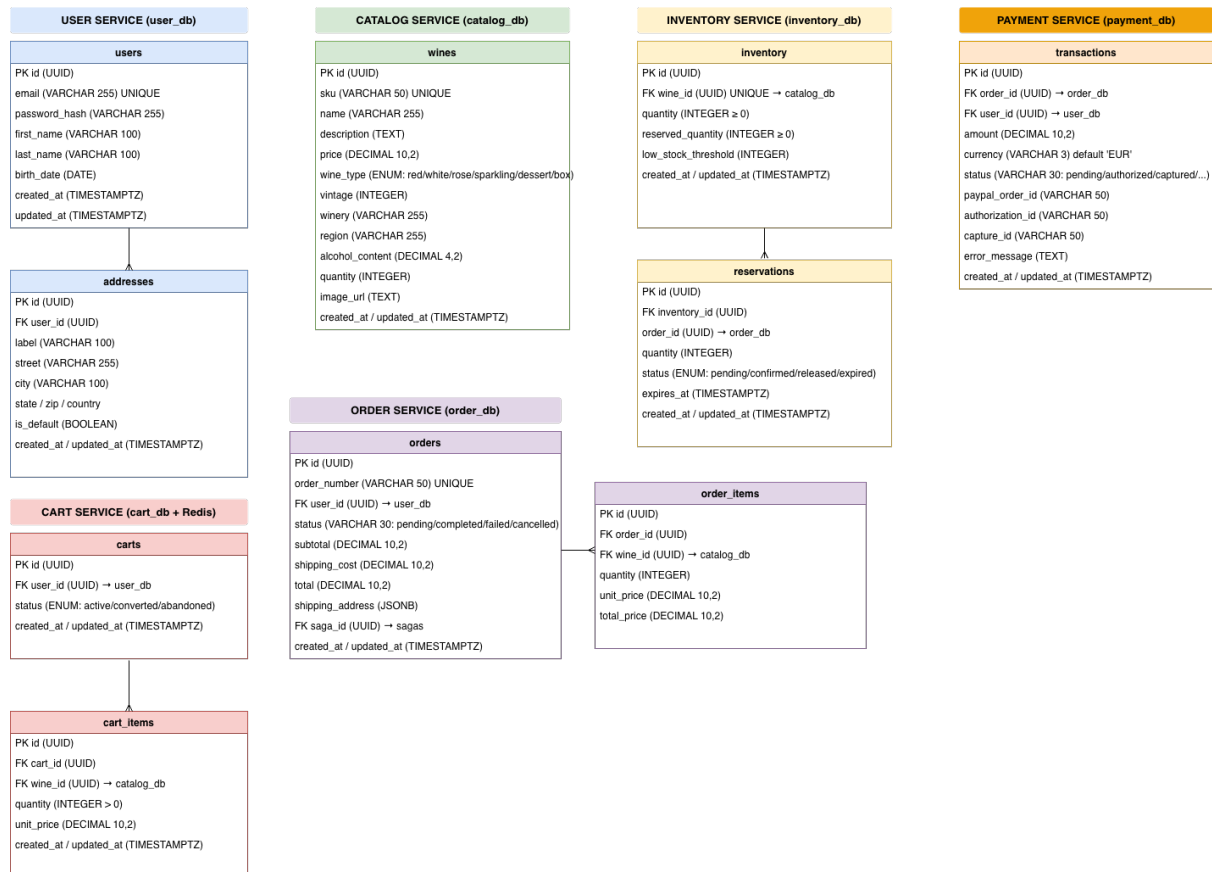


Figure 2: Database ER model over the services

The entities are distributed among services according to the *bounded context* principle: each service holds its own domain data and does not directly access others' data. When the order service needs the list of items in an order to publish a cancellation event, it reads them from its own database, not by querying the inventory.

The main **domain events** transiting on RabbitMQ are:

- `order.created` — published by the order service when a new order is created in pending status.
- `order.cancelled` — published when an order is cancelled (payment failure or user action).
- `inventory.reserved` — published by the inventory service when the stock reservation succeeds.
- `inventory.failed` — published when there is insufficient availability.
- `payment.succeeded` — published by the payment service after confirmation from Stripe.
- `payment.failed` — published in case of payment rejection.

The **system state** is distributed across the six relational databases: the order status, updated in response to events, is the primary representation of the progress of a distributed transaction.

3.4 Interaction

The main interactions are two: navigation/purchase (synchronous) and the checkout workflow (asynchronous).

For read operations (catalog, cart), the frontend communicates with the API gateway via HTTP REST, which forwards the request to the appropriate service. The response returns synchronously to the client.

The checkout workflow instead follows an asynchronous event-based sequence, as shown in fig. 3.

1. The frontend creates the order via `POST /api/orders`. The order service creates the record in the DB with status `pending` and publishes `order.created` on RabbitMQ.
2. The inventory service, subscribed to `order.created`, attempts to reserve the requested quantities in an atomic transaction. On success it publishes `inventory.reserved`; on insufficient stock it publishes `inventory.failed`.
3. If the inventory fails, the order service (subscribed to `inventory.failed`) updates the status to `failed`. The flow ends.
4. If the reservation succeeds, the frontend creates a Stripe `PaymentIntent` and collects card data via Stripe.js, which handles communication with Stripe directly on the browser side. The Stripe webhook informs the payment service of the outcome.
5. The payment service publishes `payment.succeeded` or `payment.failed`.
6. The order service updates the status to `processing` (success) or initiates compensation: updates the status to `failed` and publishes `order.cancelled`.
7. The inventory service, subscribed to `order.cancelled`, releases the previously reserved quantities.

3.5 Behaviour

Each service is **stateful** with respect to its own database, but **stateless** with respect to HTTP requests: it does not maintain state in memory between one request and the next. This enables horizontal replication without synchronization between instances.

The order service is the component that manages the order **state machine**. A single order transitions through the states `pending` → `processing` (in case of success) or `pending` → `failed`

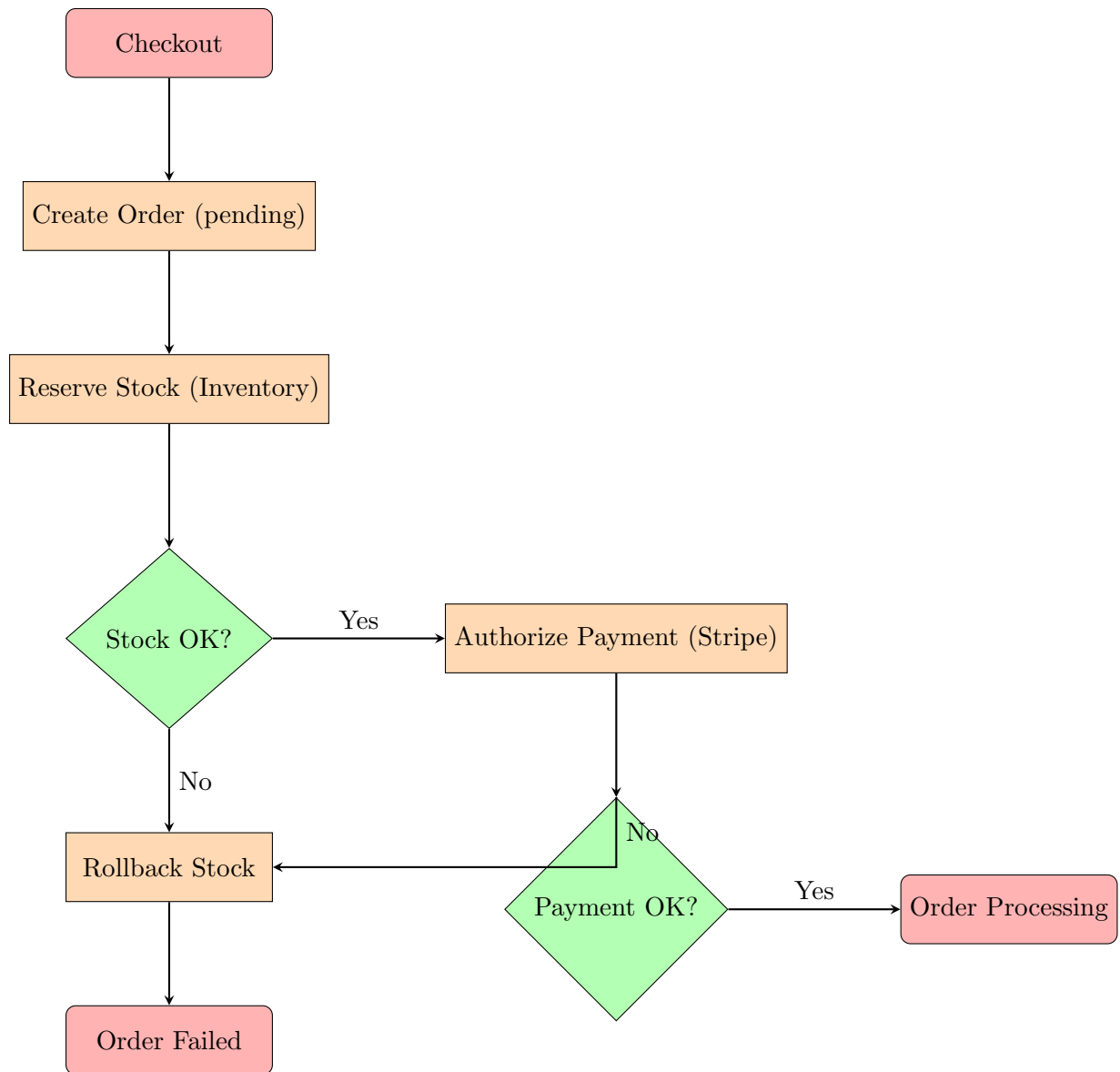


Figure 3: Checkout saga flow diagram

(in case of inventory or payment failure). Transitions are triggered by receiving RabbitMQ events. To prevent spurious transitions due to duplicate messages, update queries include conditions on the current state value (e.g. `WHERE status = 'pending'`).

The inventory service is stateful with respect to the `reserved_quantity` column: each reservation atomically increments this value, and each release decrements it. The behavior in response to `order.created` is transactional: if any of the items does not have sufficient availability, the transaction is rolled back (`ROLLBACK`) and no reservation is confirmed.

3.6 Data and Consistency Issues

Each service owns a dedicated PostgreSQL database, chosen for its guarantee of ACID properties in local transactions and for familiarity with the relational model. Data is stored in tables with intra-service foreign keys; no foreign keys exist between databases of different services.

The main consistency issue concerns the inventory state during checkout. The adopted solution leverages PostgreSQL's transactional semantics: the reservation query is an `UPDATE` with an atomic condition (`WHERE quantity - reserved_quantity >= requested`), which returns zero rows if the condition is not satisfied. This avoids race conditions between concurrent requests without explicit application-level locks.

Data shared between services is transmitted exclusively through RabbitMQ events or REST calls. The event payload includes only the information strictly necessary for the receiving service (for example, `orderId` and a list of `wine_id/quantity`).

The cart service uses Redis as storage for guest user carts. Redis is configured in non-persistent mode (in-memory), so guest carts are lost in case of restart. This is an acceptable trade-off given their short lifecycle (one-hour TTL).

3.7 Fault Tolerance

The main fault tolerance measure is the **durability of RabbitMQ queues**. Both the `wine.events` exchange and all queues declared by services are configured as `durable: true`. Messages are persisted to disk by the broker and survive restarts. If the inventory service is momentarily offline, `order.created` messages remain in the queue and are processed as soon as the service reconnects.

Message acknowledgment is manual (*manual ACK*): the service sends the ACK only after successfully processing the message and updating the database. If the service crashes before sending the ACK, RabbitMQ requeues the message for reprocessing.

The cart service implements a **non-blocking Redis connection**: if Redis is unreachable at startup, the service starts anyway and operates exclusively with the PostgreSQL database for authenticated user carts.

All services implement healthchecks declared in `docker-compose.yml`, which Docker uses to verify availability before starting dependent services. The inventory service, for example, is not started until RabbitMQ responds correctly on port 5672.

Handling of *dead-lettered* messages (messages reprocessed too many times due to persistent errors) is not currently implemented: this is identified as future work in section 9.

3.8 Availability

The wine catalog is the component with the highest read load. To meet requirement FR2 (latency below 200 ms), the catalog-service implements a Redis cache layer with a one-hour TTL. Cache keys include the query parameters (type, prices, search text), enabling granular caching for different filter combinations.

Cache invalidation occurs through natural expiration (TTL). Active invalidation based on RabbitMQ events is part of the original design but not yet fully implemented in the catalog

service (discussed in section 9).

There is currently no load balancing mechanism: each service runs as a single Docker instance. However, the architecture is prepared for balancing: services are stateless with respect to HTTP requests, and adding multiple instances would only require configuring a load balancer (for example Nginx) in front of the services.

In case of a network partition between the frontend and API gateway, or between the gateway and a backend service, the user receives an HTTP error (502 Bad Gateway). The gateway does not implement automatic retries, which are therefore the client's responsibility.

3.9 Security

User authentication is based on **JSON Web Token (JWT)**. At the time of login or registration, the user service generates a JWT token signed with a shared secret (`JWT_SECRET`), with a 24-hour expiration. The token is included by the client in the `Authorization: Bearer <token>` header of every request to protected endpoints. Each service that exposes authenticated endpoints verifies the token signature autonomously, without needing to query the user service for each request.

Authorization follows a simplified **RBAC** model: there is a user role (access to one's own resources) and implicitly an administrator role (not yet implemented with dedicated endpoints). Profile, address, and order endpoints are accessible only with a valid token; the catalog and cart creation are public.

Payments never transit through the project's servers: the card number is entered by the user directly into the Stripe.js interface, which transmits it to Stripe via a TLS connection. The backend receives exclusively the `PaymentIntentId`. Incoming webhooks from Stripe are authenticated via **HMAC-SHA256 signature verification**: the payment service extracts the signature from the `Stripe-Signature` header, reconstructs the payload, and verifies that the signature matches the one computed with the `STRIPE_WEBHOOK_SECRET`. Only events with a valid signature are processed.

All SQL queries are parameterized with prepared statements via the `pg` library, protecting against SQL injection. User passwords are stored as bcrypt hashes with 10 salt rounds.

4 Implementation

The implementation choices follow the designs defined in section 3 and are constrained by implementation requirements IR1–IR4.

Client-server communication occurs via **HTTP/1.1 with REST** for synchronous operations. Data in transit is serialized in **JSON**. Asynchronous inter-service communication occurs via **AMQP** (Advanced Message Queuing Protocol), RabbitMQ's native protocol. Stripe communicates with the backend via webhooks over **HTTPS**, while the browser communicates with Stripe directly via the Stripe.js library over **HTTPS**.

Databases are queried via **SQL** with prepared statements. No ORM is used, in order to maintain full control over queries and leverage PostgreSQL-specific features (for example, `RETURNING`, `GREATEST()`, and explicit transactions).

Authentication is implemented via **JWT** (IR4). Authorization follows a simplified **RBAC** at the service level.

4.1 Technological Details

Backend. All services are implemented in **TypeScript 5.3** on **Node.js 24**, with the **Express.js 4.18** framework. The `pg` library provides access to PostgreSQL with connection pooling. The `amqplib` library provides the AMQP client for RabbitMQ. The `redis` library is

used in the catalog-service and cart-service. The payment service uses the official **stripe** library (v14) for PaymentIntent creation and webhook verification.

Frontend. The frontend is a **React 18** SPA with **TypeScript**, compiled with **Vite 5**. HTTP calls to the gateway use **axios**. Browser-side Stripe integration is achieved through **@stripe/react-stripe-js** and **@stripe/stripe-js**: the **CardElement** component collects card data without ever exposing it to the application code. The **PaymentGateway** component is loaded with **React.lazy()** and **Suspense**, so the Stripe library is downloaded only when the user navigates to the payment page.

Infrastructure. The deployment is defined in a single **docker-compose.yml** file that defines 14 services: 6 PostgreSQL databases, 7 application services, RabbitMQ, Redis, and the frontend. Service images are built from multi-stage Dockerfiles. Database data is persisted on named Docker volumes.

Implementation challenges. One of the most significant challenges was managing PostgreSQL *pool exhaustion* in the inventory service: the inventory check function acquired a connection from the pool via `pool.connect()` but never released it in case of error, exhausting the pool and blocking all subsequent requests. The solution was to use a `try/finally` block with `client.release()` in the `finally` clause, ensuring release regardless of the outcome.

Another challenge involved inter-container communication: the order service originally called the inventory service via `localhost:3003`, which is correct in the host context but not within the Docker network. The fix was to set the `INVENTORY_SERVICE_URL` variable to `http://inventory-service:3003` in the `environment` block of `docker-compose.yml` (which takes precedence over the `.env` file).

The double firing of React effects in StrictMode required attention during checkout implementation: the order was created twice due to the double invocation of `useEffect` in development. The solution was to introduce a cleanup ref that cancels the created order upon component unmount if the user has not proceeded to payment.

5 Validation

5.1 Automatic Testing

Given the primarily practical nature of the project and the limited time, no formal automated unit tests were written. Individual controller functions were verified manually through direct HTTP requests to the services (using tools such as Postman and `curl`) during development.

Integration validation between services was performed through **manual end-to-end tests** simulating the entire purchase flow:

- User registration and login with verification of the returned JWT token.
- Catalog browsing with and without filters, verification of response times with and without Redis cache.
- Adding products to the cart as a guest user, logging in, and verifying cart migration.
- Complete checkout: order creation, inventory reservation verification (via direct database query), payment with Stripe test card (4242 4242 4242 4242), and verification that the order status updated to **processing**.
- Failed payment scenario: use of the Stripe test card that simulates rejection (4000 0000 0000 9995), with verification of reserved inventory quantity restoration and order status updated to **failed**.

- Navigating away from checkout without completing payment: verification that the order is cancelled and inventory is restored.

Stripe provides a dedicated sandbox environment for testing, with predefined test cards that simulate various scenarios (success, rejection, 3D Secure authentication, etc.), which greatly simplified validation of the payment flow without the need for real transactions.

5.2 Acceptance Testing

Acceptance tests were conducted manually by verifying that each functional requirement was satisfied.

- **Functional Requirement 1:** registration and login verified with a valid JWT token returned and usable for subsequent protected requests.
- **Functional Requirement 2:** name search and type filtering verified; the response with active Redis cache was measured as significantly faster than the first request (cache miss).
- **Functional Requirement 3** and **Functional Requirement 4:** cart persistence between sessions and guest migration verified.
- **Functional Requirement 5:** complete checkout flow verified with Stripe sandbox; order found in `processing` status after payment.
- **Functional Requirement 6:** inventory rollback verified both in case of failed payment and navigation away from checkout.
- **Functional Requirement 7:** order history displayed correctly for a user with multiple orders.

Load tests were not formally executed; cache verification was performed by manually observing response times.

6 Deployment

The entire system is deployed via Docker Compose. The prerequisites are **Docker Engine** (version 24+) and **Docker Compose** (version 2.20+).

To start the system from scratch:

1. Clone the repository and navigate to the project root directory.
2. Create a `.env` file in the root directory with the following variables:
 - `STRIPE_SECRET_KEY` — Stripe secret key (test key: starts with `sk_test_...`)
 - `STRIPE_PUBLISHABLE_KEY` — Stripe publishable key (`pk_test_...`)
 - `STRIPE_WEBHOOK_SECRET` — Stripe webhook secret (`whsec_...`)
 - `JWT_SECRET` — arbitrary string for JWT token signing
3. Run the command: `docker compose up --build`

Docker Compose builds the images for all services, starts the containers in the correct order respecting declared dependencies (application services wait for their respective databases and RabbitMQ to be healthy), and configures the internal network and volumes for persistence.

Once startup is complete, the frontend is accessible at `http://localhost:3030` and the RabbitMQ management UI at `http://localhost:15672` (default credentials: `guest/guest`).

No manual migration is required: each service automatically executes the initialization and seeding SQL scripts defined in `src/db/init.sql` and `src/db/seed.sql` at startup. The catalog is populated with 12 wines from the Gigliotto Tenute winery and the inventory is initialized with 50 units per wine.

To update a single service without restarting the entire system: `docker compose up -d --build <service-name>`.

7 User Guide

Upon opening the browser at `http://localhost:3030`, the user accesses the homepage showing the complete catalog of available wines. Wines can be filtered by type using the buttons at the top (Red, White, Rosé, Sparkling, Sweet) or searched by name or producer via the search bar.

Clicking on a wine opens the detail page, from which it is possible to add the product to the cart by specifying the quantity. Unauthenticated users can add products to the cart: the cart is maintained temporarily until the browser is closed (1-hour TTL on Redis).

To proceed with the purchase, authentication is required. From the login page, it is possible to register or log in with existing credentials. After login, the guest cart is automatically migrated to the user's account.

From the cart, clicking “Proceed to Checkout” opens the checkout page where the shipping address can be entered and the order confirmed. After confirmation, the system verifies stock availability and, if available, redirects to the payment page.

On the payment page, card details must be entered in the Stripe form and “Pay” must be clicked. The data is transmitted directly to Stripe securely. On success, the user is redirected to a confirmation page. On error (for example, card declined), an error message is shown and the order is automatically cancelled.

From the order history (accessible from the profile menu), all previous orders can be viewed along with their status.

8 Self-Evaluation

Giovanni Maria Savoca

My main contribution to the project focused on developing the user interface in React and its integration with the backend infrastructure, with particular attention to the visual and functional aspects related to my family's winery business. I worked on the frontend implementation and the integration of the payment system through Stripe, managing secure communication with webhooks for order confirmation. On the infrastructure side, I handled environment configuration via Docker and collaborated with Enrico on defining the RabbitMQ messaging architecture.

Enrico Valentini

Within the team, I primarily handled the persistence logic and the robustness of the distributed system. I implemented cart and session management through Redis, ensuring high performance and consistency of temporary data. Together with Giovanni, I designed and configured the RabbitMQ broker for asynchronous communication between services, defining the queues and routing strategies necessary for the system's operation. I also dedicated a significant part of the work to the testing phase, verifying the resilience of the microservices and the correct propagation of messages.

Joint Work

The project is the result of close collaboration that saw us working synergistically during the crucial phases of development. We worked together on the initial definition of the system architecture, analyzing requirements and choosing the most suitable technology stack to address the challenges of distribution. During the final phases, we jointly managed delivery preparation and final validation of the entire workflow, ensuring that the integration between modules met the established standards and that the transactional flow was consistent in every aspect.

9 Future Works

Several natural extensions of the system have been identified but not implemented due to time constraints.

Active cache invalidation in the catalog service is designed but not fully implemented. When an administrator updates a wine, an event should be published that causes the deletion of related Redis keys, rather than waiting for TTL expiration.

An **administrative panel** would allow catalog and inventory management through a graphical interface, with RBAC protection for the administrator role.

The integration of a **distributed tracing** system (for example OpenTelemetry with Jaeger) would make debugging multi-service flows significantly easier, providing a unified view of latencies and errors across all microservices involved in a single operation.

Finally, a natural and ambitious extension concerns **deployment in cloud environments or on Kubernetes clusters**. The current Docker Compose configuration is adequate for local development but is not designed for high-availability production scenarios. A migration to **Kubernetes** would enable leveraging features such as automatic pod self-healing, reactive horizontal scaling via HorizontalPodAutoscaler, and rolling updates without downtime. Each microservice would become a separate Deployment with an associated Service for internal discovery; RabbitMQ and PostgreSQL could be deployed as StatefulSets with persistent volumes. Alternatively, deployment on **AWS** via managed services would eliminate part of the operational complexity: Amazon ECS or EKS for containers, Amazon RDS for PostgreSQL databases, Amazon MQ or Amazon SQS for asynchronous messaging, and Amazon ElastiCache for Redis. Both solutions would require defining secure networking strategies (VPC, security groups), configuring an ingress controller or cloud-native API gateway, and managing secrets through dedicated tools such as AWS Secrets Manager or Kubernetes Secrets.

Eventually, in a production environment, it is possible (and recommended) to use **Cloudflare** for CDN to make the site faster, with its caching mechanism for static content files and to prevent DDoS attacks or adopt countermeasures against them. It also allows you to handle the DNS configuration for your site and SSL certificates.