Feature 1: Account Management

Pattern Chosen: Request/Response (REST API over HTTPS)

Reasoning:

Business requirement analysis: Customers expect **instant confirmation** when logging in or updating their profile, this insure that the user accesses the features that is allowed, by verifying who they are (Authentication).

Technical considerations:

- **Server load:** Login and profile updates are lightweight operations, well-suited to Request/Response.
- **Latency:** Users require immediate results, which synchronous APIs deliver.
- **Reliability:** Using secure HTTPS APIs with token-based authentication ensures security and data integrity (Json Web Token).
- **Approach:** Uses JWT to Authenticate and Authorize users before accessing app features, which the user uses to "tell the server whom communicating with me"

User experience impact:

- Immediate confirmation builds trust.
- Direct feedback when updating personal details improves usability.

Scalability factors: Request/Response (REST APIs) are stateless, so horizontal scaling is straightforward, since the server doesn't maintain any state between client's requests.

Alternatives considered:

- **Long Polling / WebSockets:**
  Rejected because this feature doesn't require real-time push updates. Users only need confirmation of their own actions immediately without adding the overhead of these communication patterns.

Trade-offs accepted: Nothing, synchronous calls are waited yes but in this case this what should happen.

Feature 2: Order Tracking for Customers

Pattern Chosen: Server-Sent Events (SSE)

Reasoning:

Business requirement analysis: Customers want to see their order status progress (Confirmed → Preparing → Ready → Picked up → Delivered). While updates don't need to be *instant*, they must feel **real-time**. The business gains happiness by making customers feel informed without overwhelming them with notifications, the change in order status reflect on it without refreshing or requesting multiple unnecessary times.

Technical considerations:

- **Server load:** with SSE the server pushes updates only when the status changes. With 1000+ concurrent users, this reduces unnecessary requests.
- **Latency:** SSE provides low-latency one-way updates directly from the server to clients, ensuring customers see status changes almost immediately.
- **Reliability:** SSE uses standard HTTP, can automatically reconnect if the connection drops.
- **Battery usage:** On mobile, SSE conserves battery compared to polling or other communication patterns because the client maintains a single lightweight open connection instead of repeatedly reconnecting or asking for it frequently which drains the mobile battery.

User experience impact:

- Customers perceive updates as "real-time" without draining their device battery.
- Showing status changes without the user needing to refresh or manually check.

Scalability factors:

- SSE scales well for thousands of concurrent users because it uses simple HTTP streaming.
- The server only sends data when necessary, keeping bandwidth usage low.
- Can be extended to hybrid approaches: SSE for live updates + caching for already pushed status to avoid missing, duplicating, or more server load.

Alternatives considered:

- **Short Polling (every 30–60s):**
  Rejected because it would significantly increase **server load** and waste **mobile battery**, especially at scale with 1000+ users and depending how many times the client requesting since the client could request a lot of requests and don't feel like "**real-time".**
- **Long Polling:**
  Rejected because it repeatedly opens and closes connections, and require multiple stages for the Order Status changing and this will increase the server load a lot.

Trade-offs accepted:

- SSE is **one-way only** (server → client) and the client need to be online.

Note: from what I understood from the project requirements is that the user could check the orders status any number of times which made my mind on SSE rather than Short Polling when checking the status is limited.

Feature 3: Driver Location Updates

Pattern Chosen: **WebSockets**

Reasoning:

Business requirement analysis:
Customers expect to **see their driver's location in real time** during delivery. Smooth, frequent updates (every 10–15 seconds).

Technical considerations:

- **Server load:** WebSockets establish a single persistent connection per customer.
- **Latency:** WebSockets provide near real-time updates with minimal latency, ensuring smooth marker movement on the map.
- **Reliability:** Modern WebSocket libraries support automatic reconnection, which is essential for mobile networks where connections may drop or have poor connection.

User experience impact:

- Customers see their driver's position update smoothly on the map, creating confidence and excitement.
- Bi-directional capability means drivers can send location updates, and the server can push them instantly to customers.

Scalability factors:

- Each customer-driver pair requires a dedicated WebSocket channel, which is scalable with proper connection management and load balancing.
- Since connections are short-lived (max 45 minutes), resource usage is manageable.

Alternatives considered:

- **Server-Sent Events (SSE):**
  Rejected because this feature requires **bi-directional communication** where drivers send updates and customers receive them, but SSE only supports server → client, which will add more complexity to the SSE, since the updates from driver will be separate for that of customer, that require more work and not as efficient as WebSockets.

Trade-offs accepted:

- WebSockets require more connection management and scaling compared to Request/Response (REST) or SSE.
- Connections consume memory and resources on servers.

Feature 4: Restaurant Order Notifications

Pattern Chosen: **Server-Sent Events (SSE)**

Reasoning:

Business requirement analysis:

Restaurants must be **notified immediately** when new orders arrive, since missed or delayed orders directly affect customer satisfaction and revenue. Staff should not need to

refresh their dashboard — the system must push updates automatically. Fast, reliable delivery is critical for the business.

Technical considerations:

- **Server load:** SSE uses a single persistent HTTP connection per client, which is efficient for one-way notifications like new orders. With 1–2 orders per minute per restaurant, the traffic is light and manageable.
- **Latency:** SSE delivers updates in near real time.
- **Reliability:** With multiple staff members logged in, updates can be broadcast to all connected clients for that restaurant.

User experience impact:

- Staff see new orders instantly on their dashboard without refreshing, improving workflow and responsiveness.

Scalability factors:

- SSE scales well for thousands of concurrent users because it uses simple HTTP streaming.
- The server only sends data when necessary, keeping bandwidth usage low.
- Can be extended to hybrid approaches to scale much more : SSE for live updates + Pub/Sub for already pushed status to avoid missing, duplicating, or more server load.

Alternatives considered:

- **WebSockets:**
  Introduce a lot of unnecessary complexity since the communication is from server to client (server to staff) and the scalability and managing of WebSockets connections overhead not worth it in this case.

Trade-offs accepted:

- **Nothing, but sometimes WebSockets have less delay but the difference is very small.**

Feature 5: Customer Support Chat

Pattern Chosen: **WebSockets**

Reasoning:

Business requirement analysis:
Customers and support agents need to exchange messages in **real time**, with features like typing indicators and delivery confirmations

Technical considerations:

- **Server load:** Each chat requires a persistent connection, but WebSockets scale well with thousands of concurrent lightweight connections.
- **Latency:** WebSockets provide low latency.
- **Reliability:** Messages can be stored in a database.

User experience impact:

- Customers and support agents see messages instantly, improving trust and engagement.
- Typing indicators and delivery confirmations are possible only with **bi-directional** connections like WebSockets.

Scalability factors:

- Horizontal scaling with message brokers (Pub/Sub pattern) ensures messages are routed efficiently across multiple servers.
- Database-backed storage maintains history without overloading WebSocket servers.
- WebSockets scales well for lightweight connecctions.

Alternatives considered:

- **Server-Sent Events (SSE):**
  Rejected because chat requires **bi-directional** communication. SSE only supports server → client, which makes typing indicators and message delivery confirmations impossible and much more complex than doing it using WebSockets.

Trade-offs accepted:

- WebSocket scaling requires careful connection management, monitoring, and horizontal distribution, but scaling in SSE much simpler
- **Resource overhead:** Persistent connections consume server memory.

Feature 6: Announcements

Pattern Chosen: Publish/Subscribe (Pub/Sub) + SSE

Reasoning:

Business requirement analysis:

The platform must deliver announcements to **all users simultaneously**. These announcements are important for communication but **not mission-critical** (a short delay is acceptable). The business value is in **keeping users informed** without impacting core operations.

Technical considerations:

- **Server load:** A Pub/Sub system allows the platform to **publish a message once**, and multiple subscribers (users/devices) receive it without overwhelming the core application servers.
- **Latency:** Real-time delivery is possible, but a small delay (seconds to minutes) is acceptable.
- **Reliability:** Messages can be persisted in queues/topics, ensuring late subscribers or offline users still receive the announcement when they reconnect (not implemented in the code, future planning …)

User experience impact:

- Users see announcements in-app while active, improving engagement.
- If the user is offline, the announcement can be delivered later or sent via email.

Scalability factors:

- Pub/Sub systems scale horizontally, which could scale to millions of users.

Alternatives considered:

- **WebSockets:**
  Rejected because keeping thousands of persistent connections open just for occasional announcements is wasteful. Pub/Sub is more efficient.

Trade-offs accepted:

- Announcements may arrive with slight delays depending on subscriber availability.
- Users offline may need special handling especially if the user is offline for a longtime.
- SSE sending when the Subscriber handle the announcement but separate functionality and have to deal with offline users.

Feature 7: Image Upload for Menu Items

Pattern Chosen: **Long Polling**

Reasoning:

Business requirement analysis:

Restaurants need to upload large images (2–10MB) for menu items. After upload, images undergo processing (resizing, compression, quality checks), which can take **30 seconds to 3 minutes**. Restaurant managers must know when processing is complete to publish items, without requiring constant server resources for tracking status.

Technical considerations:

- **Server load:** Long polling avoids over loading the server with very frequent requests, that isn't required.
- **Latency:** A slight delay is acceptable since processing is measured in minutes, not milliseconds.
- **Reliability:** The client can retry polling if the connection drops.

User experience impact:

- Once uploaded, the app switches to **long polling** to check the processing status.

Scalability factors:

- Long polling scales better than short polling for long-running tasks since fewer requests are made.
- Works with thousands of concurrent uploads without overwhelming the server.

Alternatives considered:

- **Short Polling:**
  Rejected because it creates excessive requests and unnecessary server load during multi-minute processing.

Trade-offs accepted:

- Managers may see the result a few seconds later than the exact completion time, depending on polling intervals.
- Long polling keeps connections open longer than regular polling, requiring efficient timeout handling.