Feature 1 : Customer Account management
Pattern chosen : Request/Response (synchronous)
Reasoning :

- Business requirement analysis :
    1. Customers expect immediate confirmation when registering or logging in.
    2. Profile updates (e.g., changing name, phone number) should be reflected right away across devices.
    3. Payment methods need secure, reliable transactions with clear success/failure responses.
- Technical considerations :
    1. Request/Response is the simplest, most reliable for CRUD-style operations (create account, update profile, manage payments).
    2. HTTPS provides security (TLS encryption) and easy integration with authentication (JWT, sessions).
    3. Idempotent operations (e.g., updating profile info, saving payment tokens) handle retries gracefully if connections drop.
- User experience impact :
    1. Users get immediate feedback (success/failure messages) when they log in or change info.
    2. No unnecessary persistent connections, which conserves mobile battery.
    3. Clear error handling when something fails (invalid credentials, network timeout).
- Scalability factors:
    1. Easy to scale horizontally behind a load balancer.
    2. REST endpoints can be cached (e.g., for profile retrieval).
- Alternatives : There are no suitable alternatives for this scenario.
- Trade-offs : There are no trade-offs, since Request/Response is the most natural and only appropriate pattern to meet the requirements.

Feature 2 : **Order Tracking for Customers**
Pattern chosen : Server-Sent Events (SSE)
Reasoning :

- Business requirement analysis : Customers want to track their order progress (Confirmed → Preparing → Ready → Picked Up → Delivered). The updates must feel real-time but do not require millisecond precision.
- Technical considerations : SSE provides a single, lightweight, one-way connection from server to client. This minimizes server load compared to long polling and is more battery-efficient for mobile devices. With SSE, updates are pushed immediately when the status changes, without the client constantly asking.

- User experience impact : Customers get near real-time status updates without constantly refreshing. Automatic reconnect ensures smoother experience if the connection drops.
- Alternatives :
    - Long polling**:** Increases server load and drains battery with repeated requests.
- Trade-offs :
    - Client must stay online: If the customer loses internet connection, they stop receiving updates until they reconnect.
    - One-way only: SSE can't send data from client → server; a separate REST call is required if the client needs to act.
    - Server resources: Each open SSE connection consumes some memory, though manageable with scaling.

Feature 3: Driver Location Updates "

Pattern chosen : WebSockets
Reasoning :

- Business requirement analysis : Customers expect to see the driver's movement on a live map during the delivery window (30–45 minutes). This is a real-time, highly interactive feature. Only the customer who placed the order should see the location, ensuring privacy.
- Technical considerations : Drivers send updated coordinates every 10–15 seconds. With WebSockets, updates can flow efficiently from driver → server → customer without the overhead of repeated HTTP requests. WebSockets are full-duplex, so both driver and customer can stay synced on the same connection.
- User experience impact :The map feels smooth because updates are pushed immediately. Clients can interpolate positions between updates for even smoother animation. WebSockets work well over mobile networks and support reconnects if the signal drops.
- Scalability factors: WebSockets are more resource-intensive than SSE, but since this feature is only active during delivery and applies to a limited subset of users at a time, the load remains manageable.
- Alternatives :
    - SSE: Rejected because location updates require two-way communication (driver sending positions, server distributing to the right customer). SSE only supports one-way server → client.
    - Polling: Rejected because fetching location via REST every few seconds would cause battery drain, network overhead, and a choppy user experience.
- Trade-offs :
    - Persistent connections: WebSockets keep a connection open, which consumes more server and client resources compared to polling or SSE.
    - Scaling complexity: Requires sticky sessions or a pub/sub adapter for load balancing across multiple servers.

Network drops: Clients must handle reconnection logic, but modern libraries like Socket.IO simplify this.

Feature 4: Restaurant Order Notifications :

Pattern chosen : WebSockets

Reasoning :

- Business requirement analysis : Orders must reach restaurants within seconds; missing even one leads to unhappy customers and lost revenue. Multiple staff members may be logged in, and all of them must see the order at the same time without refreshing their dashboards.
- Technical considerations : Using queue ensures that order events are reliably delivered to the backend. provides durability and guarantees delivery even if a consumer is temporarily disconnected. Once the message is queued, the backend uses WebSockets to push the order immediately to all connected staff clients. This combination provides both reliability and real-time delivery.
- User experience impact :Orders appear instantly on the restaurant's dashboard, without requiring manual refresh. Multiple staff devices stay synchronized, reducing human errors and delays.
- Alternatives :
    - Pure WebSockets without queue: Rejected because if a staff member is offline, the order might be missed. Reliability is not guaranteed without durable messaging.
    - SSE: Rejected because multiple staff members need to send acknowledgments (e.g., when accepting orders). SSE is one-way only, which limits functionality.
    - Polling: Rejected because it would introduce delays and increase server load, violating the "within 5 seconds" requirement.
- Trade-offs :
    - Added complexity: Introducing queue adds infrastructure overhead (setup, monitoring, maintenance).
    - Persistent connections: WebSockets consume server resources, but this is acceptable for the relatively small number of restaurant staff users.
    - Failover handling: If queue goes down, a fallback mechanism (e.g., DB polling) may be needed temporarily.

Feature 5: Customer Support Chat

Pattern Chosen: WebSockets

Reasoning

- **Business requirement analysis:**
  Chat must feel instant, like WhatsApp or Messenger. Customers expect quick replies, typing indicators, and delivery confirmations. Agents may handle multiple concurrent conversations, so the system needs to manage many active chat sessions reliably.

- **Technical considerations:**
  WebSockets provide bi-directional, low-latency communication, which is ideal for chat. Both customer and agent can send and receive messages instantly.
- **User experience impact:**
  Customers see messages appear instantly, know when the agent is typing, and receive delivery/read confirmations. Agents can switch between multiple conversations seamlessly. Chat history persists so users can revisit past conversations.

Alternatives

- **SSE:** Rejected because it only supports server → client. Chat requires bi-directional communication.

- **Polling / Long Polling:** Rejected because delays are unacceptable for chat, and it would waste bandwidth and server resources.

Trade-offs

- Persistent connections: WebSockets consume more server resources than polling.

- Delivery guarantees: Need extra logic (e.g., message IDs, acknowledgments) to prevent duplicates or message loss in case of reconnections.

Feature 6: System-Wide Announcements

Pattern Chosen: Pub/Sub with Server-Sent Events (SSE)

Reasoning

- Business requirement analysis:
  Announcements must reach thousands of users, but they are not critical and can tolerate small delays. The system must avoid overwhelming servers during peak traffic, and messages should appear automatically if users are active.

- Technical considerations:
  A publish/subscribe mechanism is efficient for broadcasting the same message to many users simultaneously.

For delivery, SSE provides a one-way channel from server → client that allows lightweight broadcasting without requiring each client to poll. Offline users can fetch announcements from a database when they next open the app.

- **User experience impact:**
  Active users see announcements pop up in near real-time, without refreshing. Inactive users won't miss important messages since the system can load missed announcements when they return.

- **Scalability factors:**
  Pub/Sub fan-out ensures efficient distribution to thousands of users. SSE maintains a lightweight persistent connection per client, which is much cheaper than polling and scales better under load.

Alternatives

- **WebSockets:** Rejected because two-way communication isn't needed; SSE is simpler and more efficient for one-way broadcasts.

- **Polling:** Rejected because it would generate unnecessary traffic from thousands of clients, wasting server resources.

- **Push notifications (mobile):** Could complement announcements, but out of scope since cloud push services are restricted.

Trade-offs

- **One-way only:** SSE doesn't allow client responses; users can't acknowledge announcements through the same channel (but that isn't required).

- **Connection limits:** Some browsers limit concurrent SSE connections, but in practice, most apps only need one per user.

- **Offline users:** Messages aren't received if the app is closed, so the system must also store announcements in a database for later retrieval.

Feature 7: Image Upload for Menu Items

Pattern Chosen: HTTP Multipart Upload + Message Queue + Server-Sent Events (SSE)

Reasoning

- **Business requirement analysis:**
  Restaurants upload relatively large images (2–10MB). Processing (resizing, compression, quality checks) can take several minutes, so the system must not block the client during processing. Managers need to see progress updates and know when processing is complete.

- **Technical considerations:**
  - The image is first uploaded over HTTP multipart request (standard and reliable for large files).
  - Once uploaded, the server creates a job and pushes it to a message queue .

  - A worker processes the image asynchronously, sending progress updates (10%, 50%, 90%, complete) through SSE.
    The client stays subscribed to the SSE stream to get live progress and the final status (success/failure).

- **User experience impact:**
  The manager sees upload progress immediately, followed by processing progress updates. If something fails (bad file, network issue), the system can notify them clearly so they can retry.

- **Scalability factors:**
  The queue smooths out spikes in uploads by buffering jobs. Multiple workers can be added to process images in parallel. SSE ensures real-time status updates without overloading the server.

Alternatives

- **Synchronous upload + processing:** Rejected because it would force the client to wait 30–180 seconds, leading to timeouts and poor UX.
- **Polling for status:** Rejected because constant polling wastes resources and adds delays in showing progress.
  WebSockets: Could work, but SSE is simpler and sufficient since only server → client updates are required.

Trade-offs

- **Complexity:** Requires extra components (queue + workers + SSE handling), but gains reliability and scalability.
- **Connectivity dependency:** SSE requires the client to stay online during processing; if they disconnect, they'll need to fetch the final status later.
- **Resource usage:** Storing intermediate job states (progress %) in Redis or DB adds overhead, but ensures recovery in case of client reconnection.