1. **QUICK FIX**

**Q1 (25 points): List three essential features Bistro 92’s system needs for customer satisfaction and efficient order processing.**

**Ans:**

Here are three must-have features for Bistro 92’s system that are needed to make customer happy and ensure super smooth order processing.

**1. Instant Order Delivery**

When a customer places an order that should be immediately sent to the kitchen. No real-time delay. Customers won't have to wait for the order, and this will reduce time and speed up food preparation.

**2. Simple and Easy to Use UI**

The menu should be easy to navigate with clear item lists.

Easy menu browsing, item selection, quantity adjustment, and order confirmation should be dealt with. Besides, we need to ensure that even first-time or tech novice customers can use it without confusion.

**3. Order Tracking and Status Updates**

A quick glance at the pad should show them if their order is “Placed,” “Being Cooked,” or “Ready to Serve.” It’s like giving them a little peace of mind, so they’re not wondering, “Where’s my burger?!”

This reduces anxiety about order delays and improves the overall experience with the restaurant.

**Q2 (25 points): Describe two design principles to make the smart pad interface intuitive for all users, including tech novices.**

**Ans:**

Here are the **two key design principles** for an **intuitive smart pad Interface**

**1. Simplicity and Clarity**

The interface should be kept **simple, clean, and focused.**

Showing just the essentials: browse dishes, pick items, tweak quantities, and hit “order.”

**Impact:**

* Tech novices won’t feel confused or overwhelmed.
* **Clear icons, large buttons, and minimal text** make it easy to understand without any training. Even a senior citizen can use this easily with just a go.

**Design Example:**

* **"Add to Cart" and "Place Order" buttons** should be **big and labeled clearly.**
* No hidden menus or complicated navigation.

**2. Feedback and Confirmation**

Every tap should spark **a little response,** like a **quick beep** or an **audio confirmation** on the screen, and also at the end of the order, they should know what they ordered. It’s like the pad saying, “Got it, you’re good!” This way, users know they’re on the right track and won’t accidentally order 10 pizzas.

**Impact:**

* Users feel confident that their actions are registered.
* Reduces mistakes like double ordering or uncertainty.

**Example for Bistro 92:**

* After pressing "Place Order," show **a thank you message or a simple animation.**
* When adjusting quantity, **highlight the changed number** instantly.

**Q3. (30 points): Identify three potential security vulnerabilities in Bistro 92’s system (e.g., theft, order tampering) and suggest one solution for each.**

**Ans:**

**1. Order Tampering (Fake or AlteredOrders)**

**Problem:**

If the connection between the pads and the server isn’t locked tight, a hacker could snoop on order details or customer data.

**Solution:**

* Use **authentication tokens and input validation** on the backend.
* Only accept orders from verified devices with secure API keys or device IDs.

**2. Data Theft (Customer/Order Information Leak)**

**Problem:**

If the connection between the pads and the server isn’t locked tight and encrypted, hackers could easily steal customer data or order details.

**Solution:**

Use HTTPS/TLS encryption for all communications between smart pads, servers, and dashboards.

**3. Denial of Service (Overloading the Server)**

**Problem:**

An attacker could send thousands of fake requests to overload and crash the order server (DDoS attack).

**Solution:**

Implement rate limiting and IP blocking.

For example, limit each device or IP address to a safe number of requests per minute.

**Q4 (30 points): Explain two strategies to keep Bistro 92’s system responsive and stable during peak hours.**

**Ans:**

**1. Auto-Scaling with Load Balancing**

Run multiple backend servers (API + WebSocket servers) behind a load balancer like Nginx.

**Impact:**

* Traffic is distributed across many servers.
* When the restaurant gets very busy (peak dinner rush), more servers can be automatically added (auto-scaling).
* No single server becomes overloaded or crashes.

**Implementation:**

* Use Azure App Service Scaling + Azure Load Balancer.
* Deploy backend in Docker containers for easy scaling.

**2. Use Redis Queue to Handle Order Bursts**

When too many orders are placed at the same time, the API does not have to directly write to the database.

Instead, it should quickly push orders into a Redis queue, like a to-do list.

**Impact:**

* The backend responds instantly without waiting for slow database operations.
* A separate background worker reads from the queue and saves orders into the database safely and steadily.
* Prevents server slowdowns and database overload during busy times.

**Implementation:**

* Redis (for queuing)
* Flask Worker (background service)
* Azure Redis Cache (managed Redis in cloud)

**Q5 (40 points): Describe one method to integrate the existing inventory system with Bistro 92’s new system without disrupting operations.**

**Ans:**

**Merging Bistro 92’s shiny new system with the old inventory setup without causing a kitchen meltdown? Here’s how:**

* **Use an API-Based Synchronization Layer**

Create a middleware API that connects the existing inventory system with the new Bistro 92 order system.

**How it works:**

Think of this as a translator between the new order system and the existing inventory setup. Create a middleware API that lets the two systems chat without stepping on each other’s toes. When a customer orders, say, a pizza, the new system sends a polite request to the inventory API to update the stock (like subtracting tomatoes and cheese). The old system keeps chugging along, untouched.

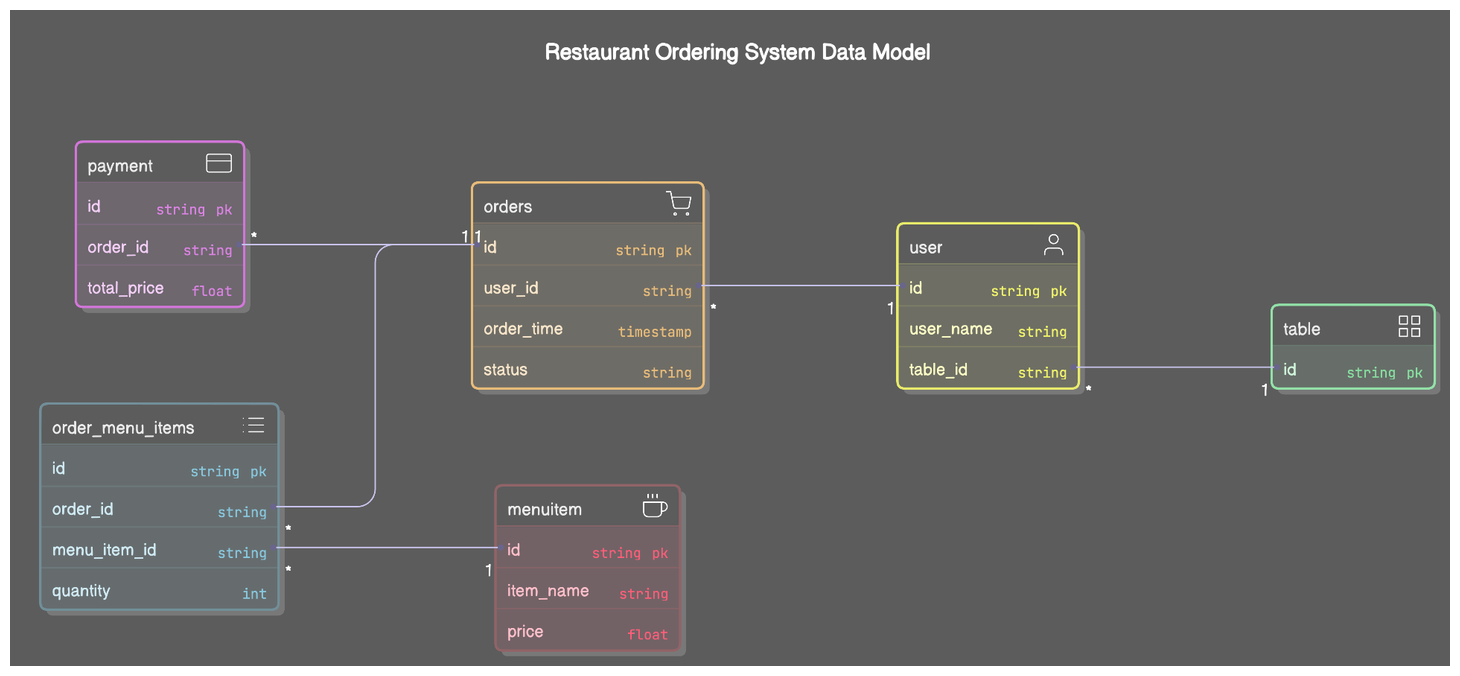
**Why it’s safe:**

* The old inventory system keeps working normally.
* The new system only communicates through APIs — no risky database modifications.
* If the API fails, orders still process, and inventory sync can retry later.

**B) Tech Tricks**

**Q1 (70 points): Design a database schema for Bistro 92 to track users, orders, menu items, tables, and payments, optimized for fast queries.**

**Answer:**

****

We are calling this "Restaurant Ordering System Data Model." This design helps track users, orders, menu items, tables, and payments in a way that’s organized and quick to search, making sure the restaurant runs smoothly. I’ll explain it in a simple, friendly way while keeping the details clear and professional.

Here are five main tables: `user`, `table`, `orders`, `order\_menu\_items`, `menuitem`, and `payment`. Each table stores specific information, and they’re connected to keep everything linked and easy to look up. Now, here’s how it works:

**1. The `user` Table**

**This stores Info about the customers or staff using the system.**

`id` (string, primary key): A unique ID for each user, like “USER001.”

`user\_name` (string): The user’s name, like “Mr. X”

{

userid :"UUID"

user\_name:"string"

table\_id:"UUID"

-- foreign key from table

}

**2. The `table` Table**

**It stores details about the restaurant’s tables.**

This tracks which table an order comes from, so the staff know where to deliver the food. A simple table with just an `id` keeps queries fast and lightweight.

`id` (string, primary key): A unique ID for each table, like “TABLE05.”

{

id : "uuid"

}

**3. The `orders` Table**

**It stores the main details of each order.**

- `id` (string, primary key): A unique ID for the order, like “ORDER123.”

- `user\_id` (string): Links to the `user` who placed the order (e.g., “USER001”).

- `order\_time` (timestamp): When the order was placed, like “2025-04-27 19:32:00.”

- `status` (string): The order’s status, like “Placed,” “Cooking,” or “Delivered.”

- `table\_id` (string): Links to the `table` the order came from, like “TABLE05.”

**- Connections:**

- `user\_id` connects to the `user` table (1:1 relationship, marked with a “1”).

- `table\_id` connects to the `table` table (also 1:1)

{

orderId: "UUID",

userid: "UUID", // foreign key from user id

menu\_items:[

{

"item\_id" : "UUID" // foreign key from menuitem table id

Quantity:"number",

}

]

ordertime: "TIMESTAMP DEFAULT CURRENT\_TIMESTAMP",

status: "ENUM('pending', 'completed', 'cancelled') DEFAULT 'pending'",

}

}.

This table is the heart of the system, tying together who ordered, when, and from which table. The `status` helps the kitchen track progress, and indexing `user\_id` and `table\_id` makes lookups fast—like finding all orders from a specific table.

**4. The `menuitem` Table**

**It stores the restaurant’s menu items.**

- `id` (string, primary key): A unique ID for each menu item, like “ITEM001.”

- `item\_name` (string): The name of the item, like “Cheeseburger.”

- `price` (float): The cost, like “12.99.”

{

id : "uuid",

item\_name:"string",

price:"float",

}

**5. The `payment` Table**

It stores payment details for each order.

- `id` (string, primary key): A unique ID for the payment, like “PAY001.”

- `order\_id` (string): Links to the `orders` table, like “ORDER123.”

- `total\_price` (float): The total cost of the order, like “25.98.”

**- Connections:**

- `order\_id` connects to the `orders` table (many-to-one, marked with a “\*”).

It’s handy for generating receipts or checking sales, and indexing `order\_id` makes it quick to find a payment for a specific order.

{

id:"uuid"

orderId:"uuid" //foreign key(order

total\_price: "float"

}

**How It All Ties Together for Bistro 92**

A customer at Table 5 (ID: “TABLE05”) places an order through the smart pad. The `user` (say, “USER001”) is linked to the order in the `orders` table, along with the `table\_id` and the time. The `order\_menu\_items` table lists what they ordered—like 2 Cheeseburgers (linked to `menuitem` ID “ITEM001”). Once the order is done, the `payment` table records the total cost. The kitchen can quickly pull up all orders for Table 5, see what’s cooking, and check the status—all without any delays.

**Q2 (80 points): Write an SQL query to retrieve all orders from the last hour, including table number, items ordered, and order time, optimized for speed.**

**Answer:**

**Here is the SQL query to retrieve all orders from the last hour -**

SELECT

t.table\_id,

m.item\_name,

oi.quantity,

o.order\_time

FROM

orders o

JOIN

user u ON o.user\_id = u.user\_id

JOIN

table t ON u.table\_id = t.table\_id

JOIN

JSON\_TABLE(

o.menu\_items,

'$[\*]'

COLUMNS (

item\_id VARCHAR(255) PATH '$.id',

quantity INT PATH '$.quantity'

)

) AS oi ON 1=1

JOIN

menuitem m ON oi.item\_id = m.id

WHERE

o.order\_time >= NOW() - INTERVAL 1 HOUR

ORDER BY

o.order\_time ASC;

**Working Explanation:**

**It Gets - Table ID (like “TABLE05”), item name (like “Cheeseburger”), quantity (like “2”), and order time (like “7:32 PM”).**

- Starts with the `orders` table.

- Joins with `user` and `table` to get the table number.

- Uses `JSON\_TABLE` to unpack the `menu\_items` JSON (e.g., item IDs and quantities).

- Joins with `menuitem` to get item names.

- Filters for orders from the last hour (`order\_time >= NOW() - 1 HOUR`).

- Sorts by order time (oldest first).

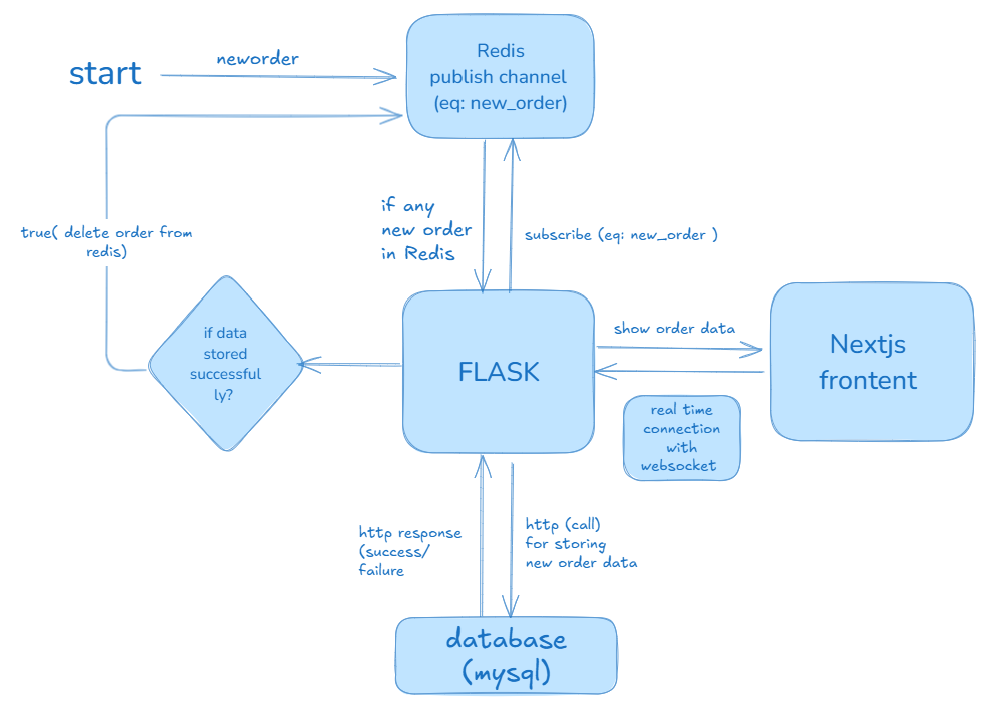
The kitchen gets a fast, clear list of recent orders—like “Table 5: 2 Cheeseburgers at 7:00 PM”—so they can keep up with the rush!

**Q3 (90 points): Implement a feature to notify kitchen staff in real-time when a new order is placed, and describe your tech stack.**

**Answer:**

Here is the flowchart for implementing a real-time notification feature to alert Bistro 92’s kitchen staff the moment a new order comes in. This solution ensures the kitchen stays on top of things during a busy dinner rush.

**Flowchart:**

****

The flowchart outlines a smooth system where orders placed by customers are instantly sent to the kitchen staff through a combination of a backend, a message queue, and a frontend dashboard.

1. **Starting: A New Order is Placed**

It all kicks off when a customer taps “Place Order” on their smart pad. This action sends a message, labeled as new\_order, to the system.

1. **Flask Backend Receives the Order**

The new\_order message lands at the Flask backend, which is the brain of our operation. The incoming HTTP request contains all the order details - like the table number, items ordered, and quantities. Flask’s job is to process this order quickly and reliably.

1. **Storing the Order in the Database**

Flask takes the order data and saves it into a MySQL database (labeled as database (mysql)).

Once the data is stored, Flask gets a response back—either a success or failure (labeled as http response (success/failure)). If the save is successful, we move forward; if it fails, we’d handle the error (though the flowchart focuses on the happy path here).

1. **Pushing the Order to Redis for Real-Time Updates**

Here’s where the magic of real-time notifications happens. After saving the order, Flask pushes the new order into a Redis channel (labeled as Redis publish channel (eg: new\_order)).

By publishing the order to a Redis channel, we’re essentially broadcasting a message that says, “Hey, there’s a new order.”

1. **Checking Redis for Any New Orders**

Meanwhile, the system continuously checks Redis to see if there are any new orders in the channel (labeled as if any new\_order in Redis). Redis also has a handy feature to clean up processed orders (labeled as TRUE (delete order) from redis), ensuring we don’t clog the system with old messages.

1. **Notifying the Kitchen via WebSocket**

The kitchen staff’s dashboard, built with Next.js (a React-based framework), is subscribed to the Redis channel (labeled as subscribe (eg: new\_order)). This subscription happens through a WebSocket connection, which allows real-time, two-way communication between the backend and the frontend (labeled as a real-time connection with WebSocket). When Redis broadcasts a new order, the Next.js frontend picks it up instantly and displays the order details on the kitchen dashboard (labeled as show order\_data).

1. **Verification of Data Storage**

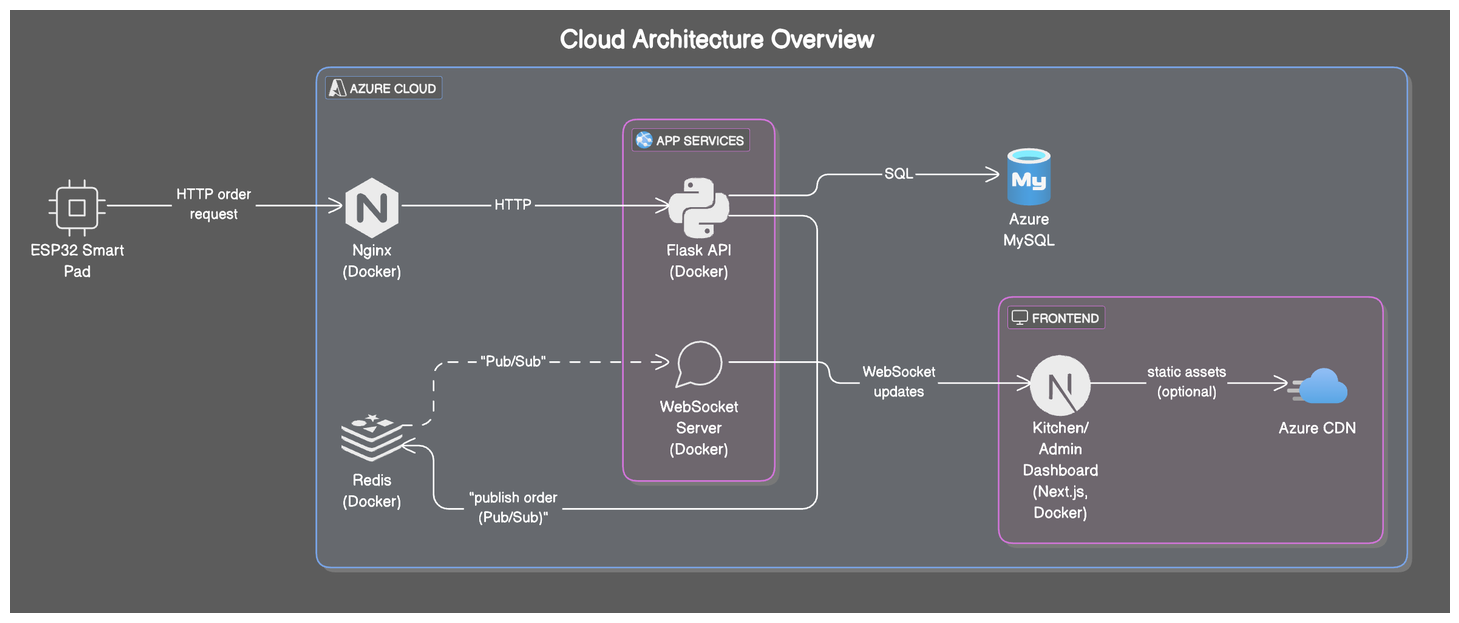
There’s also a side check to ensure the order was stored successfully in the database (labeled as if data was stored successfully?). If yes, the system proceeds as described. If not, we’d loop back to handle the issue, but the flowchart assumes success for simplicity.

Techs that are used to implement:

* Flask (Backend)
* MySQL (Database)
* Redis (Message Queue)
* Next.js (Frontend)
* WebSocket (Real-Time Connection)

**Q4 (100 points): Describe a cloud-based system architecture for real-time updates, data storage, and smart pad communication, ensuring low latency and high availability.**

**Answer:**



Overview of the Cloud-Based System Architecture

Let’s dive into the cloud-based system architecture for Bistro 92, as shown in the diagram, to ensure real-time updates, efficient data storage, and seamless communication with the smart pads. This setup is designed to keep things fast (low latency) and reliable (high availability), even during the busiest dinner rushes.

The Cloud Architecture Overview, illustrates a robust system hosted on Azure Cloud. It connects the ESP32 Smart Pads at the tables, the backend services, the kitchen/admin dashboard, and the data storage—all working in harmony to handle orders with speed and reliability. Here’s how it all fits together:

**1. Starting Point: Smart Pads Send Orders**

The journey begins with the **ESP32 Smart Pad** on each table at Bistro 92. When a customer places an order (like a juicy burger and fries), the smart pad sends an **HTTP order request** to the backend. The ESP32 is perfect for sending these requests over the internet. This is the first step in getting the order from the table to the kitchen, and it needs to be fast and secure.

**2. Load Balancing with Nginx for High Availability**

The HTTP request from the smart pad doesn’t go straight to the backend—it first hits **Nginx**, which is running in a **Docker** container. **Nginx acts as a load balancer**, playing traffic cop by distributing incoming requests across multiple backend servers.

During peak hours, when every table is ordering at once, Nginx ensures no single server gets overwhelmed. It spreads the workload, keeping the system responsive (low latency) and ensuring it stays up and running (high availability). Docker makes it easy to scale Nginx and the backend servers by spinning up more containers as needed.

**3. Flask API: The Brain of the Whole Operation**

Next, the request reaches the **Flask API**, also running in a Docker container within **Azure App Services**. Flask is a lightweight Python framework that handles the core logic of processing orders. When it receives the HTTP request, it does a few key things:

* **Parses the Order:** It extracts details like the table number, items ordered, and quantities.
* **Stores the Data:** It sends the order data to a database for permanent storage
* **Triggers Real-Time Updates:** It also sends a notification to the kitchen staff so they can start cooking right away.

**4. Data Storage with Azure MySQL**

For storing order data, the Flask API communicates with **Azure MySQL** using **SQL** queries. Here, MySQL is used because it’s a reliable database for structured data like orders, tables, and menu items. When an order comes in, **Flask** saves the details (like “Table 5 ordered a pizza at 7:32 PM”) **to MySQL**, ensuring the data is safely stored for future reference—like generating sales reports or tracking order history.

This **Azure MySQL** handles backups, scaling, and maintenance, which boosts high availability. It’s also optimized for fast read/write operations, keeping latency low when saving or retrieving order data.

**5. Real-Time Updates with Redis and WebSocket**

Now, here’s where the real-time magic happens. **To notify the kitchen staff instantly,** the Flask API publishes the new order to a **Redis server** (also in Docker) using a **Pub/Sub** (publish/subscribe) mechanism. When Flask sends a “publish\_order” message to Redis, it’s like shouting, “Hey, there’s a new order for the kitchen to handle!”

Redis is used because its an in-memory data store, which means it’s incredibly fast—perfect for real-time updates.

The kitchen and admin staff are using a **Kitchen/Admin Dashboard** built with **Next.js** (a React framework), also running in Docker. This dashboard connects to Redis via a **WebSocket Server** (also in Docker).

When Redis publishes a new order, the WebSocket server picks it up and pushes it to the dashboard in real time (labeled as “WebSocket updates”). The kitchen staffs see the order pop up on their screen instantly—no refreshing needed! This ensures low latency, as updates happen in milliseconds, keeping the kitchen in sync with customer orders.

**6. Serving the Dashboard with Azure CDN (Optional)**

There is an optional connection to **Azure CDN** for serving **static assets** like images, styles, and scripts. A CDN stores these files on servers around the world, so they load quickly no matter where the dashboard is accessed from. While this isn’t directly tied to order processing, it ensures the dashboard itself loads fast and stays responsive, contributing to a smooth user experience for the staff. This is marked as optional, but it’s a nice touch for optimizing performance.

**Q5 (110 points): Design a real-time dashboard showing pending orders, average fulfillment time, and total sales, specifying tools and justifying your choices.**

**Answer:**

**The real-time dashboard will display:**

1. Pending Orders → List of all active orders waiting to be prepared.

2. Average Fulfillment Time → Tracks how long, on average, it takes to complete an order.

3. Total Sales → Summarizes daily revenue.

**Tech stack and justification:**

| **Component** | **Tech Stack** | **Why?** |
| --- | --- | --- |
| **Frontend (UI)** | **Next.js + Tailwind CSS** | **Fast, lightweight, and great for real-time updates.** |
| **Real-Time Data** | **WebSockets (Socket.IO)** | **Instant order updates without page refresh.** |
| **Backend API** | **Flask (Python) or Node.js (Express)** | **Flexible, lightweight, and easy to integrate with WebSockets.** |

Here is the link of the video where we’ve designed a real-time dashboard showing all of the criterias and explained.

[**https://drive.google.com/file/d/1q8keD1cPIZMvhaUrgVcsHLf\_o7U4biMf/view?usp=drivesdk**](https://drive.google.com/file/d/1q8keD1cPIZMvhaUrgVcsHLf_o7U4biMf/view?usp=drivesdk)

**C) BONUS BOOSTERS**

**Q1 (250 points): Design a RESTful API for order placement that handles high concurrency without errors, detailing endpoints, request/response formats, and concurrency controls.**

**Answer:**

**For order Creation ( placing a new order) :**

Endpoints: POST /api/create-order

**Request Format :**

{

orderId: "UUID"

items:

[

{

menuItemId : "UUID from Menu Item Table"

"quantity" : "number"

}

]

,

orderCreationTime : TIMESTAMPS

}

**Response Format:**

{

"message": "Order placed successfully",

"orderId": "order-uuid-123",

"status": "Pending"

}

**Error Handling :**

| **Problem** | **Solution** |
| --- | --- |
| Same order submitted twice | Idempotency with unique Order IDs |
| Two users ordering at same time | Transactions and Isolation levels |
| Database crash on peak load | Redis queue + background workers |
| Server overloaded | Load balancing + multiple backend servers |

**How to Handle High Concurrency Without Errors:**

We need to make sure that Bistro 92’s system can handle tons of orders at once without messing up—like during a crazy dinner rush!

Here’s how to do it -

* **Group Actions Together (Database Transactions)** When saving an order, wrap all steps in a “transaction.” It’s like saying, “Do everything or nothing.” If something goes wrong, it cancels everything—no half-saved orders!  
   **Example**: Start, save the order, save the items, and finish. If anything fails, it all rolls back.
* **Avoid Duplicates (Idempotency)** Give each order a unique ID, like “order-123.” If the same order comes twice (like a glitchy retry), the system spots the ID and skips it. No double orders!  
   **Example**: Before saving, check if “order-123” already exists—if it does, ignore it.
* **Keep Data Safe (Database Isolation)** Set the database to a strict mode like “SERIALIZABLE.” This stops the system from seeing half-finished orders, so there’s no confusion.  
   **Example**: Tell the database, “Don’t let anyone see an order until it’s fully saved.”
* **Line Up Orders (Queue for High Load)** If tons of orders flood in, don’t save them straight to the database—it might crash! Instead, put them in a Redis queue (like a to-do list). A helper slowly saves them one by one.  
   **Why?** It keeps the system steady, even if 10,000 orders hit at once.
* **Share the Work (Horizontal Scaling)** Use multiple servers behind an Nginx load balancer. It’s like having extra cashiers—orders get split across servers, so no single one gets overwhelmed.
* **Double-Check Updates (Optimistic Locking)** When updating things like inventory, add a “version number.” If two orders try to update at the same time, the system checks the version and retries if needed—no mistakes!  
   **Example**: If someone else updates the inventory first, your update waits and tries again.

**Q2 (250 points): Enhance your API to support extreme scalability, processing numerous simultaneous orders without data loss, and explain your strategies.**

**Answer:**

Here’s how to Enhance the Order API for Extreme Scalability and Data Safety -

## **Step 1: Make the System Handle Tons of Orders (Scalability)**

| **Part** | **How to Do It** |
| --- | --- |
| **API Layer** | Run multiple copies of the API behind a load balancer (like Nginx). |
| **Load Balancing** | Use Nginx or Azure Load Balancer to split orders across API copies evenly. |
| **Async Handling** | Use fast tools like FastAPI or Node.js to handle orders without waiting around. |
| **Queueing** | Put orders in a Redis Queue (RQ) or Azure Service Bus to process them smoothly. |
| **Database Scaling** | Use MySQL Read/Write Replicas: save to the main DB, read from copies. |
| **Caching** | Store menu and table info in Redis so you don’t keep bugging the database. |

## **Step 2: Keep Data Safe (No Lost Orders)**

| **Problem** | **Fix** |
| --- | --- |
| **Order Collision (Double Orders)** | Use a unique order ID for each order—skip duplicates if they happen. |
| **Race Conditions (Clashing Updates)** | Use database transactions—save everything together or not at all. |
| **Backend Crash During Rush** | Push orders to a Redis queue first; a worker saves them to MySQL safely. |
| **Server Down or Crashes** | Let smart pads retry orders—they’re safe because of unique order IDs. |
| **Database Failure** | Set up MySQL Auto Failover and back up data often. |

**D) Big Idea (up to 300 Points)**

## Problem

Usually it is seen in any restaurant that the customer doesn't know how long their food will take to arrive after ordering. This uncertainty causes frustration, frequent questions to staff, and a less enjoyable dining experience, especially during busy times. The Bisrto 92 restaurant also has these shortcomings.

## Solution: AI Order Time Estimator

We are proposing to add an AI-powered system to Bistro 92’s to predict and show how long it takes to cook and serve each order. The AI uses kitchen data and order details to give accurate, real-time estimates, making customers happier and reducing staff workload.

### **Key Features**

1. The AI model will be trained based on the resturent daily operational data that has included the daily order preparation based on number of customer on the queue, available waiter and available cooks, order complexity, meal matarials quantity and other factors.
2. The estimated time will be shown to the smart pad also it will show the status of the ordered meal (“pending”, “preparing”, “ready to serve”, “delivered”).