Real-Time Clinical Activity Tracking

Making hospital data speak

Topic PageNumber

Abstract:

This project implements a real-time hospital monitoring system that integrates a transactional SQL Server database with a streaming analytics and visualization layer. The relational model captures core hospital processes including patients, visits, visit status history, admissions, appointments, operations, emergency details, and billing. Change Data Capture (CDC) is enabled via Debezium's SQL Server connector, which streams all inserts, updates, and deletes from key tables (Visits, VisitStatusHistory, Admissions, Bills) into Apache Kafka topics.

A Python service consumes these Kafka topics, maintains in-memory aggregates, and exposes live operational metrics through a RESTful API built with Flask. The service computes several key performance indicators, such as the total number of visits today, admissions created today, current inpatients (open admissions), the number of patients currently waiting per visit classification and per medical speciality, and the total revenue paid today based on bill payments. A lightweight HTML/JavaScript dashboard periodically calls the /stats endpoint and visualizes these KPIs through dynamic cards and charts, providing an up-to-date view of hospital activity.

In addition to the streaming layer, a weekly snapshot process is implemented in SQL Server to persist historical data into dedicated snapshot tables and export them as CSV files for further batch analysis. Together, this architecture demonstrates how traditional hospital information systems can be extended with modern streaming technologies to deliver near real-time operational insight and improved decision support.

Problem Statement:

Hospitals generate large volumes of operational data across many departments (reception, clinics, emergency, wards, billing, etc.). In most traditional systems, this data is stored in a relational database and accessed through manual queries or static reports. As a result:

- Management and staff do not have real-time visibility into what is happening right now (who is waiting, how many inpatients, how much revenue is collected today, etc.).
- Key indicators such as current waiting load per speciality, occupancy of beds, and daily revenue are either calculated manually or only available at the end of the day or week.
- There is no unified view that connects live transactional data (from SQL Server) with an easy-to-read dashboard for monitoring operations.
- Historical analysis is also limited, because raw transactional tables are not structured for weekly or monthly analytics and reporting.

This leads to slower decision-making, difficulty in detecting bottlenecks (e.g., overcrowded emergency, overloaded specialities), and limited ability to analyze performance over time.

The problem, therefore, is to design and implement a system that:

- Streams changes from the hospital SQL Server database in real time.
- Computes meaningful live KPIs.
- Presents them in a clear dashboard.
- Preserves weekly snapshots for batch analysis and reporting.

Project Objectives

High-Level Objective

To build an end-to-end data pipeline that connects a hospital's SQL Server transactional database to a Kafka-based streaming layer, a Python aggregation service, and a web dashboard, in order to provide near real-time operational insight and weekly analytical snapshots.

Specific Objectives

1. **Data Modeling & SQL Server Layer**
   - Design and implement a relational schema for hospital operations, including: Patients, Doctors, Specialties, Wards, Beds, Visit Status History, Admissions, Operation Sessions, Emergency Details, Appointments, Bills, Bill Items, Operation Types and Visit Status.
   - Ensure referential integrity and realistic sample data for at least one full week of operations.
2. **Change Data Capture (CDC) & Kafka Integration**

   Enable SQL Server CDC / **Debezium** integration on key transactional tables: Visits, Visit Status History, Admissions, Bills.

   Configure a Debezium SQL Server connector to stream all **inserts, updates, and deletes into Kafka topics with 3 partitions each.**

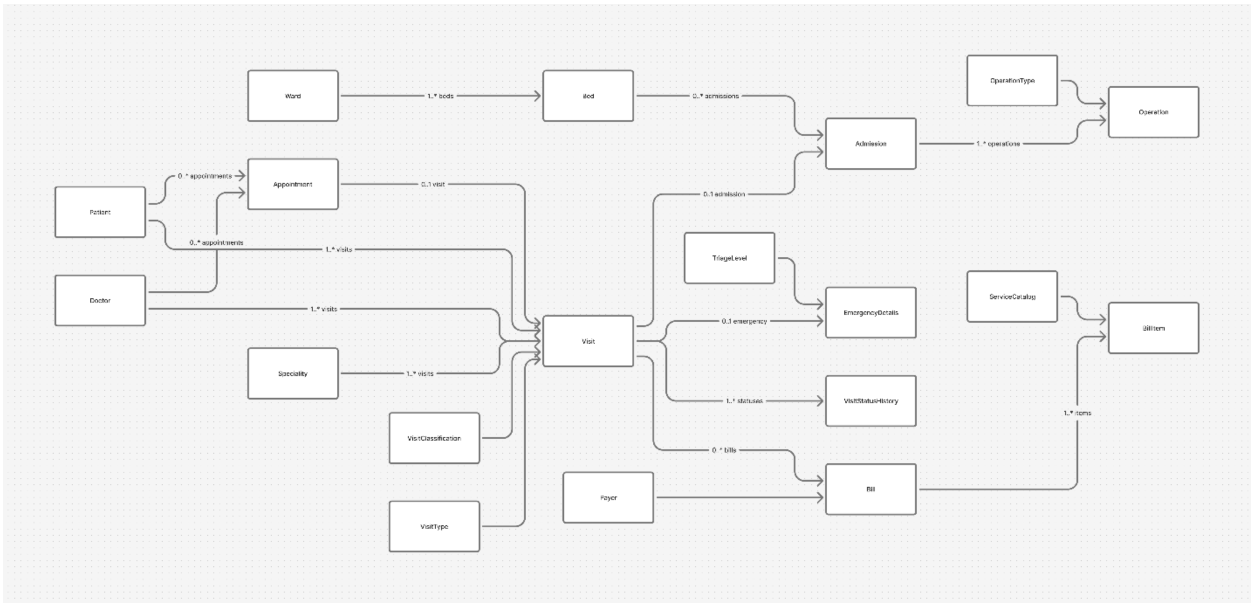
   Verify that each table's changes are correctly written to the corresponding Kafka topic.

3. **Real-Time Aggregation Service**
   - Implement a Python Kafka consumer that:

    - Subscribes to the hospital Kafka topics.
    - Maintains in-memory state and aggregates for the current day.
  - Compute the **following live KPIs**:
    - Total number of visits today.
    - Number of admissions created today.
    - Current number of inpatients (DischargeDate IS NULL).
    - Number of patients currently in Waiting status per:
      - Visit classification (Inpatient / Outpatient / Emergency / Ambulatory).
      - Medical speciality (Emergency, Internal Medicine, General Surgery, etc.).
    - Total revenue paid today based on bills with PaymentStatus = 'Paid' and PaymentDate = today.
  - Expose these metrics through a RESTful endpoint /stats using Flask.

4. **Real-Time Dashboard**
   - Build an **HTML/JavaScript** dashboard that periodically **calls the /stats API** and displays:
     - Big numeric tiles (cards) for total visits today, admissions today, inpatients now, and revenue paid today.
     - Visual indicators (bars / triangles / lists) for:
       - Waiting patients by classification.
       - Waiting patients by speciality.
   - Ensure the dashboard auto-refreshes every few seconds and clearly highlights changes.
5. **Weekly Snapshot & Batch Analytics**
   - Create weekly snapshot tables (e.g. Weekly_Visits, Weekly_Admissions, Weekly_Bills, etc.) in SQL Server.
   - Implement a stored procedure RunWeeklySnapshot that:
     - Calculates the last week’s date range.
     - Copies all relevant rows from operational tables into snapshot tables.
   - Configure a SQL Server Agent job to run this procedure automatically once per week.
   - Export weekly snapshot data to CSV files for later analysis or loading into Spark / data warehouse.

Database Design (SQL Server & ERD)



1- Logical Data Model (ERD)

Above figure – Entity–Relationship Diagram (ERD) of the Hospital Operational Database.
The database is designed to model core hospital operations: **patient registration, clinical encounters (visits), admissions, emergency episodes, appointments, and billing.** The model is **normalized** and centered around the Visit as the main clinical encounter.

Below is an explanation of the main entities (tables) and their roles, followed by the key relationships.

1.1- Main Entities and Their Roles:

1. **Patients**
   - Table: **Patients**
   - **Purpose**: Stores demographic and contact information about each patient.
   - **Key columns:**
     - PatientID (PK): unique identifier for each patient.
     - FirstName, LastName, Phone
     - Age, Gender, Nationality
     - InsuranceProvider: optional, used for billing and payer type logic.
   - Usage: All clinical encounters and appointments are linked to a specific patient.

---

2. **Doctors**
   - **Table: Doctors**
   - Purpose: Holds basic information about doctors working in the hospital.
   - **Key columns:**
     - DoctorID (PK)
     - FirstName, LastName

    - Phone, Email
- **Usage: Linked to visits, appointments, and operation sessions to show who treated the patient.**

---

3. **Speciality & Doctor_Speciality**

- **Tables:**
    - **Speciality**
    - **Doctor_Speciality**
- Purpose:
    - Speciality: defines medical specialities such as *Emergency, Internal Medicine, General Surgery, Orthopedics, Pediatrics, Cardiology, Radiology, Laboratory, ICU*.
    - Doctor_Speciality: link table (many-to-many) assigning one or more specialities to each doctor.
- **Key columns:**
    - Speciality.SpecialityID, SpecialityName, Location
    - Doctor_Speciality.DoctorID, SpecialityID (composite PK)
- **Usage: Used in analytics (e.g. "patients waiting per speciality") and for routing visits to the correct department.**

---

4. **Wards and Beds**

- **Tables:**
    - **Wards**
    - **Beds**
- **Purpose:**
    - Wards: defines hospital wards (e.g. Medical Ward A, Surgical Ward A, Orthopedic Ward, Pediatrics Ward, ICU).
    - Beds: physical beds inside each ward.
- **Key columns:**
    - Wards.WardID, WardName, WardType, Location
    - Beds.BedID, BedNumber, WardID, BedType
- **Usage: Used by Admissions to know where inpatients are staying.**

---

5. **VisitClassification and VisitType**

- **Tables:**
    - **VisitClassification**
    - **VisitType**
- **Purpose:**
    - VisitClassification: high-level category of the visit – *Inpatient, Outpatient, Emergency, Ambulatory*.
    - VisitType: more granular type within a classification, such as:
        - Initial Examination, Follow-up Visit, Pre-Operation Assessment,
        - Emergency Examination, Trauma Emergency,
        - Injection, Dressing/Wound Care, Remove Cast, Vitals Check, etc.

- **Key columns:**
    - VisitClassification.ClassificationID, ClassificationName

    - VisitType.TypeID, TypeName, ClassificationID (FK → VisitClassification)
- **Usage: Used to classify and analyze visits (e.g. visits per classification, type of emergency, etc.).**

---

6. Visits

- **Table: Visits**
- **Purpose:** Represents a single clinical encounter between a patient and a doctor (outpatient, emergency, or inpatient-related).
- **Key columns:**
    - VisitID (PK)
    - PatientID (FK → Patients)
    - DoctorID (FK → Doctors)
    - ClassificationID (FK → VisitClassification)
    - TypeID (FK → VisitType)
    - VisitDate (date and time of the encounter)
- **Usage: Central table for all clinical activity. Most other operational tables reference VisitID.**

---

7. VisitStatus and VisitStatusHistory

- **Tables:**
    - **VisitStatus**
    - **VisitStatusHistory**
- **Purpose:**
    - VisitStatus: defines possible states of a visit (Registered, Waiting, In Consultation, Finished, Admitted, In Ward, Discharged, Cancelled, No-Show).
    - VisitStatusHistory: logs the timeline of statuses for each visit.
- **Key columns:**
    - VisitStatus.StatusID, StatusName
    - VisitStatusHistory.HistoryID (PK),
      VisitID (FK → Visits),
      StatusID (FK → VisitStatus),
      StatusTime (when the status changed).
- **Usage:**
    - **Used for real-time analytics like "how many patients are currently in Waiting status".**
    - **Provides audit trail of each visit's progress.**

---

8. Admissions

- **Table: Admissions**
- **Purpose:** Represents a hospital inpatient admission linked to a visit.
- **Key columns:**
    - AdmissionID (PK)
    - VisitID (FK → Visits, unique – one admission per visit)
    - BedID (FK → Beds)
    - AdmissionDate, DischargeDate
- **Usage:**
    - **Used to track current inpatients (DischargeDate IS NULL).**

- **Supports analytic KPIs like *number of admissions today*, *current inpatients*, and bed utilization.**

---

9. **OperationType and OperationSessions**
   - **Tables:**
     - **OperationType**
     - **OperationSessions**
   - **Purpose:**
     - OperationType: defines types of surgical procedures (Appendectomy, Hernia Repair, Cesarean Section, Orthopedic Fixation, Cardiac Catheterization, etc.).
     - OperationSessions: records actual surgical operations conducted during an admission.
   - **Key columns:**
     - OperationType.OperationTypeID, OperationName
     - OperationSessions.OperationSessionID (PK),
       AdmissionID (FK → Admissions),
       OperationTypeID (FK → OperationType),
       SurgeonID (FK → Doctors),
       StartTime, EndTime
   - **Usage: For surgical workload analysis, OT utilization, and linking to billing packages.**

---

10. **TriageLevels and EmergencyDetails**
    - **Tables:**
      - **TriageLevels**
      - **EmergencyDetails**
    - **Purpose:**
      - TriageLevels: defines emergency severity levels (Level 1 – Resuscitation, Level 2 – Emergency, … Level 5 – Non-Urgent).
      - EmergencyDetails: additional info for emergency visits.
    - **Key columns:**
      - TriageLevels.TriageLevelID, LevelDescription
      - EmergencyDetails.EmergencyDetailID (PK),
        VisitID (FK → Visits, unique),
        TriageLevelID (FK → TriageLevels),
        ChiefComplaint
    - **Usage: Supports emergency triage analysis and differentiating critical vs non-critical emergency visits.**

---

11. **Appointments**
    - **Table: Appointments**
    - **Purpose:** Stores scheduled visits (future or past) and links them to actual visits when the patient arrives.
    - **Key columns:**
      - AppointmentID (PK)
      - PatientID (FK → Patients)
      - DoctorID (FK → Doctors)

- AppointmentDate, AppointmentTime
- Purpose
- Status (Scheduled, Completed, Cancelled, No-Show)
- VisitID (nullable FK → Visits)

- **Usage:**
  - **When patient comes → Status = 'Completed' and VisitID is set.**
  - **If patient doesn't show → Status = 'No-Show'.**
  - **Supports analysis of appointment adherence (no-show rates, cancellations, etc.).**

---

12. PayerTypes and ServiceCatalog

- **Tables:**
  - **PayerTypes**
  - **ServiceCatalog**
- **Purpose:**
  - PayerTypes: defines who pays (Cash, Insurance, Company).
  - ServiceCatalog: defines individual billable services (consultation, bed-day, ICU day, lab tests, imaging, procedures, surgery packages, etc.) with standard prices.
- **Key columns:**
  - PayerTypes.PayerTypeID, PayerTypeName
  - ServiceCatalog.ServiceID, ServiceName, Price, Description
- **Usage: Used by the billing system to calculate charges.**

---

13. Bills and BillItems

- **Tables:**
  - **Bills**
  - **BillItems**
- **Purpose:**
  - Bills: represents a financial bill for either a visit or an admission.
  - BillItems: breakdown of services within each bill.
- **Key columns (Bills):**
  - BillID (PK)
  - VisitID (nullable FK → Visits)
  - AdmissionID (nullable FK → Admissions)
  - PayerTypeID (FK → PayerTypes)
  - BillDate
  - TotalAmount
  - PaymentStatus (Pending, Paid, Cancelled)
  - PaymentDate
- Key columns (BillItems):
  - BillItemID (PK)
  - BillID (FK → Bills)
  - ServiceID (FK → ServiceCatalog)
  - Quantity, Price

- **Usage:**
  - **Used for real-time revenue analytics (e.g. *revenue_paid_today*).**
  - **Allows detailed analysis of which services are driving revenue.**

1.2- Key Relationships:

Some of the main relationships in the ERD are:

1. **Patient to Visits**
   - **Relationship:** One Patient → Many Visits
   - **Meaning:** A single patient can have many clinical encounters over time.
2. **Doctor to Visits**
   - **Relationship:** One Doctor → Many Visits
   - **Meaning:** A doctor may see many patients in different visit types (outpatient, emergency, inpatient rounds).
3. **VisitClassification to VisitType**
   - **Relationship:** One VisitClassification → Many VisitType
   - **Meaning:** Each visit type belongs to exactly one classification (e.g. Trauma Emergency is always Emergency).
4. **Visit to VisitStatusHistory**
   - **Relationship:** One Visit → Many VisitStatusHistory records
   - **Meaning:** Each visit changes status over time (Registered → Waiting → In Consultation → Finished or Admitted, etc.).
5. **Visit to Admission**
   - **Relationship:** One Visit → Zero or One Admission (unique constraint on VisitID in Admissions)
   - **Meaning:** Some visits result in inpatient admission; others are discharged directly.
6. **Ward to Beds**
   - **Relationship:** One Ward → Many Beds
7. **Admission to OperationSessions**
   - **Relationship:** One Admission → Zero or Many OperationSessions
8. **Visit to EmergencyDetails**
   - **Relationship:** One Visit → Zero or One EmergencyDetails
   - **Meaning:** Only emergency visits have emergency-specific details.
9. **Patient / Doctor to Appointments**
   - **Relationship:**
     - One Patient → Many Appointments
     - One Doctor → Many Appointments
   - **Link to visits:** One Appointment → Zero or One Visit
10. **Visit / Admission to Bills**
    - **Relationship:**
      - One Visit → Zero or Many Bills
      - One Admission → Zero or Many Bills
    - But each Bill is for either a single visit **or** a single admission (check constraint).
11. **Bill to BillItems**
    - **Relationship:** One Bill → Many BillItems
    - **Meaning:** Each bill contains multiple services.
12. **Doctor to Speciality (via Doctor_Speciality)**
    - **Relationship:** Many-to-Many:
      - One Doctor can have multiple Speciality entries.
      - One Speciality can have multiple doctors.

Change Data Capture (CDC)

What is CDC?

Change Data Capture (CDC) is a mechanism for tracking and capturing *row-level changes* (INSERT, UPDATE, DELETE) that occur in a database table, and then making these changes available as a stream of events.

Traditional systems often work with full snapshots of tables:

- Periodically reading the entire table
- Comparing states to detect what has changed

This approach is slow, resource-heavy, and not suitable for near real-time analytics.

CDC solves this by recording only the *changes*, not the whole table, and exposing them in a structured form that other systems (like Kafka consumers) can process continuously.

CDC in SQL Server

In SQL Server, CDC can be enabled on a database and specific tables. Once enabled:

- SQL Server monitors changes to those tables (e.g. Visits, Admissions, Bills, VisitStatusHistory).
- For every INSERT / UPDATE / DELETE, a corresponding entry is written to CDC change tables or derived from the transaction log.
- Each change record contains:
    - The old values (before)
    - The new values (after)
    - The type of operation (insert / update / delete)
    - A timestamp / log sequence number

Inside SQL Server (CDC)

1. You enable CDC at the database level and for specific tables.
2. SQL Server creates:
    - a capture job (SQL Agent job),
    - special CDC tables like cdc.dbo_Visits_CT.
3. The capture job reads the transaction log (not the base table directly) and writes row changes for CDC-enabled tables into the corresponding CDC change tables.
4. Each change in the CDC table has:
    - the operation (insert / update / delete),
    - before/after values (depending on how CDC is configured),
    - LSN and timestamps.

So:
Log → CDC change tables (by SQL Server's CDC mechanism).

Debezium

Debezium is an open-source Change Data Capture (CDC) platform.
Its job is to listen to a database (SQL Server, MySQL, Postgres, etc.) and stream every change (INSERT, UPDATE, DELETE) as events into a messaging system like Apache Kafka.

Key ideas:

- Instead of asking the database all the time:
  "What changed? Give me the new rows…"
  Debezium watches the transaction log and sends every change as an event.
- It works in real time: as soon as a row changes in SQL Server, a message is written to a Kafka topic.
- It is non-intrusive: your application and database work normally; Debezium just "taps" into the log.

So Debezium is the bridge between:

Operational database (SQL Server) → Streaming platform (Kafka)

In this project, **Debezium's role is**:

1. Connect to SQL Server (Hospital_Salam database).
   The SQL Server Debezium connector is configured with:
   - Host, port, user, password
   - Database name: Hospital_Salam
2. Monitor specific tables:
   We only care about some transactional tables:
   - Visits
   - VisitStatusHistory
   - Admissions
   - Bills

So we configure:

"table.include.list": "dbo.Visits,dbo.VisitStatusHistory,dbo.Admissions,dbo.Bills"

Publish every change as Kafka events.
For example:

- New visit → event in topic: hospital.Hospital_Salam.dbo.Visits
- Status changed to "Waiting" → event in hospital.Hospital_Salam.dbo.VisitStatusHistory
- New admission → event in hospital.Hospital_Salam.dbo.Admissions
- Bill paid → event in hospital.Hospital_Salam.dbo.Bills

Feed the real-time dashboard.
A Python Kafka consumer subscribes to those topics and updates in-memory statistics:

- Visits today per classification (Inpatient / Outpatient / Emergency / Ambulatory)
- How many patients are admitted now (inpatients)
- How many are waiting now, per speciality and per classification
- Revenue paid today from Bills

Then a Flask API (/stats) exposes these metrics, and the HTML dashboard calls this endpoint and displays the numbers with live refresh.

So the full chain is:

SQL Server (Hospital_Salam)
→ Debezium SQL Server Connector
→ Kafka Topics (one per table)
→ Python Consumer (aggregations in memory)
→ Flask API
→ HTML/JS Dashboard

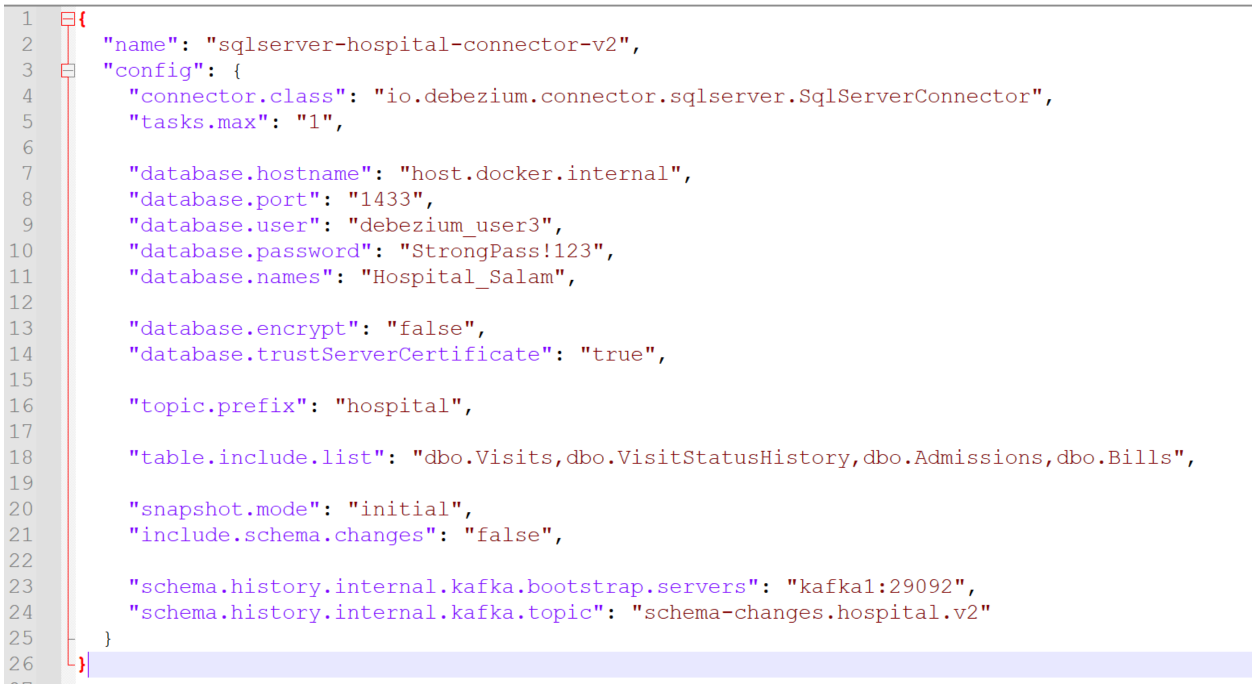
Each Kafka message has a structure like:

```
{
  "payload": {
    "before": { ... },
    "after":  { ... },
    "op": "c/u/d/r",
    "ts_ms":  1764774620...
  }
}
```

Where:

- before = row state before the change (for updates/deletes)
- after = row state after the change (for inserts/updates)
- op = operation type (c = create, u = update, d = delete, r = snapshot read)

Debezium SQL Server Connector Configuration



```
{
  "name": "sqlserver-hospital-connector-v2",
  "config": {
    "connector.class": "io.debezium.connector.sqlserver.SqlServerConnector",
    "tasks.max": "1",

    "database.hostname": "host.docker.internal",
    "database.port": "1433",
    "database.user": "debezium_user3",
    "database.password": "StrongPass!123",
    "database.names": "Hospital_Salam",

    "database.encrypt": "false",
    "database.trustServerCertificate": "true",

    "topic.prefix": "hospital",

    "table.include.list": "dbo.Visits,dbo.VisitStatusHistory,dbo.Admissions,dbo.Bills",

    "snapshot.mode": "initial",
    "include.schema.changes": "false",

    "schema.history.internal.kafka.bootstrap.servers": "kafka1:29092",
    "schema.history.internal.kafka.topic": "schema-changes.hospital.v2"
  }
}
```

This Debezium connector reads **change data** from SQL Server (Hospital_Salam database) and publishes it to **Kafka topics**.

It listens only to **four transactional tables**:

- dbo.Visits
- dbo.VisitStatusHistory
- dbo.Admissions
- dbo.Bills

It writes events into Kafka with the prefix hospital, so a table like dbo.Visits becomes a topic:

hospital.Hospital_Salam.dbo.Visits

On first startup, it takes an **initial snapshot** of existing data, then switches to **CDC (change data capture)** and streams new changes.

Field-by-field explanation

Top level

- **name: "sqlserver-hospital-connector-v2"**
  Logical name of the connector inside Kafka Connect.
  You will see this name in:

Core connector settings

- **connector.class: "io.debezium.connector.sqlserver.SqlServerConnector"**
  Tells Kafka Connect which connector plugin to use.
  Here we use Debezium’s **SQL Server** connector.

- **tasks.max: "1"**
  Maximum number of tasks (parallel workers) for this connector.

  - 1 = single task (simple, enough for a small project/demo).

  - In larger systems, you might increase this to read from many tables in parallel.

Database connection

These settings define how Debezium connects to SQL Server:

- **database.hostname: "host.docker.internal"**
  Hostname or IP of the SQL Server instance.

  - host.docker.internal is commonly used when Kafka Connect is running in Docker and SQL Server is running on the host machine (Windows).

- **database.port: "1433"**
  TCP port for SQL Server (default is 1433).

- **database.user: "debezium_user3"**
  SQL login used by the connector.
  This user must have enough permissions to:

  - read the database
  - read CDC tables / transaction log (depending on Debezium setup)

- **database.password: "StrongPass!123"**
  Password for the Debezium user.

- **database.names: "Hospital_Salam"**
  The SQL Server **database name** that Debezium will monitor.
  In this project, the transactional schema lives in Hospital_Salam.

Encryption / TLS

- **database.encrypt: "false"**
  Controls whether the connector encrypts the connection to SQL Server using TLS/SSL.

  - "false" → plain TCP connection (no TLS).
  - In production, you would typically enable encryption.

- **database.trustServerCertificate: "true"**
  When using encryption, this flag tells the client whether it should trust the server certificate without full validation.
    - Here set to "true" mainly to simplify local setups (self-signed certs, etc.).
    - With database.encrypt = false this has no real effect, but it's harmless.

Kafka topics naming

- **topic.prefix: "hospital"**
  This string is used as a **prefix** for all generated Kafka topics.

Debezium topic name pattern:

```
{topic.prefix}.{databaseName}.{schemaName}.{tableName}
```

- For this configuration:
    - Visits table → hospital.Hospital_Salam.dbo.Visits
    - VisitStatusHistory → hospital.Hospital_Salam.dbo.VisitStatusHistory
    - Admissions → hospital.Hospital_Salam.dbo.Admissions
    - Bills → hospital.Hospital_Salam.dbo.Bills

This is why the Python consumer subscribes to topics like:

- hospital.Hospital_Salam.dbo.Visits
- hospital.Hospital_Salam.dbo.Admissions
- etc.

Table selection (which tables are captured)

- **table.include.list: "dbo.Visits,dbo.VisitStatusHistory,dbo.Admissions,dbo.Bills"**

This is a **whitelist** of tables that Debezium will capture.

- Only these tables are streamed to Kafka.
- Any other tables in Hospital_Salam (e.g. Patients, Doctors, etc.) are **ignored** by this connector.

This keeps the system focused on the **transactional / operational** tables needed for the real-time dashboard.

Snapshot behavior

- **snapshot.mode: "initial"**

Controls what Debezium does when the connector starts for the first time.

- "initial" means:
    1. Take a **one-time snapshot** of existing data in all included tables.
    2. After the snapshot completes, start streaming **ongoing changes** (CDC) from the SQL Server transaction log.

Effect in your project:

- When you first start the connector, it sends all existing Visits, Admissions, Bills, etc. to Kafka (so the dashboard can start with full data).
- Then it continues to send new inserts/updates/deletes as they happen.

- **include.schema.changes: "false"**

Controls whether Debezium also publishes schema changes (DDL events) to Kafka topics.

- "false" → **do not** send DDL events.
- This keeps topics clean and focused only on **row-level data changes**.
- For this dashboard use case, you don't need to know when a column was added/changed.

Schema history storage

Debezium needs to remember how the DB schema looked over time, so it can correctly interpret messages and changes.

- **schema.history.internal.kafka.bootstrap.servers: "kafka1:29092"**

Kafka broker where Debezium stores **schema history**.

- kafka1:29092 is the internal hostname/port of the Kafka broker in your Docker compose (for example).

- **schema.history.internal.kafka.topic: "schema-changes.hospital.v2"**

Kafka topic used internally by Debezium to store schema history for this connector.

- This topic is **not** meant for normal consumers.
- It's used by Debezium itself to:
    - track table structure over time
    - be able to restart and still decode old + new messages correctly

---

Real-time stats service (Python + Kafka + Flask)

Architecture

The **real-time stats service** is responsible for consuming change events from Kafka and exposing aggregated metrics to the dashboard.

High-level flow:

SQL Server (Hospital_Salam)

- Operational database where all hospital transactions happen (visits, admissions, bills, etc.).

Debezium SQL Server Connector

- Reads the SQL Server transaction log and publishes every change (INSERT / UPDATE / DELETE) as CDC events to Kafka.

Kafka Topics

One topic per table we care about, for example:

- hospital.Hospital_Salam.dbo.Visits
- hospital.Hospital_Salam.dbo.VisitStatusHistory
- hospital.Hospital_Salam.dbo.Admissions
- hospital.Hospital_Salam.dbo.Bills

Python KafkaConsumer

Subscribes to those topics.

For each incoming event, it updates a set of **in-memory counters** and helper structures.

In-memory Stats Model

- The service keeps current metrics in Python dictionaries and sets (no database).
- This allows fast, real-time aggregation without extra queries.

Flask /stats API

- A lightweight Flask app exposes a single endpoint: GET /stats.
- It returns the current in-memory metrics as JSON.

HTML Dashboard

- The frontend periodically calls /stats (e.g. every 1–3 seconds)
- It displays the latest numbers (total visits, waiting per speciality, revenue, etc.) in real time.

In-memory model / metrics

The real-time service keeps its state in a main stats dictionary plus some helper structures.

Main stats dictionary

```
stats = {
    "visits_today_total": 0,
    "visits_today_by_classification": defaultdict(int),

    "admissions_today": 0,
    "inpatients_now": 0,

    "waiting_now_by_speciality": defaultdict(int),
    "waiting_now_by_classification": defaultdict(int),

    "revenue_paid_today": 0.0,

    "last_update": None,
}
```

Fields:

- **visits_today_total**
  Total number of distinct visits that started **today** (based on VisitDate).
- **visits_today_by_classification**
  A dictionary:

```
{
  "Inpatient": N1,
  "Outpatient": N2,
  "Emergency": N3,
  "Ambulatory": N4
}
```

It counts how many visits today for each **VisitClassification** (Inpatient / Outpatient / Emergency / Ambulatory).

- **admissions_today**
  Number of admissions whose AdmissionDate is **today**.
- **inpatients_now**
  Current number of **active inpatients**.
  Typically defined as admissions where DischargeDate is NULL (not yet discharged).
- **waiting_now_by_speciality**
  A dictionary mapping **Speciality name → current number of patients in "Waiting" status**. For example:

```
{
  "Emergency": 5,
  "Internal Medicine": 3,
  "Pediatrics": 2
}
```

  This is updated whenever a visit's status changes to or from Waiting.
- **waiting_now_by_classification**
  Similar idea, but aggregated by **visit classification** instead of speciality. For example:

```
{
  "Emergency": 4,
  "Outpatient": 6,
  "Inpatient": 1
}
```

- **revenue_paid_today**
  The total **amount of money actually paid today**, based on Bills events where:
  PaymentStatus = 'Paid', and PaymentDate is today.
  The service uses the bill's TotalAmount and adjusts the metric whenever a bill is created/updated.
- **last_update**
  ISO 8601 timestamp (UTC) of the **last processed event**.
  Used to know when the stats were last refreshed.

Helper structures

Besides stats, the service keeps a few helper structures to be able to update the metrics correctly as events come in:

```
visits_today_ids = set()
active_admissions = set()
visit_info_by_id = {}
current_status_by_visit = {}
```

- **visits_today_ids** (set)
    - Contains the set of VisitIDs whose VisitDate is today.
    - Purpose: avoid double-counting the same visit if multiple update events arrive for it.
    - When a new visit event comes in for today, the service:
        - Checks if VisitID not in visits_today_ids
        - If not there → adds it and increments visits_today_total and the corresponding classification counter.

- **active_admissions** (set)
    - Contains the AdmissionIDs for patients currently admitted (not yet discharged).
    - Used to keep inpatients_now in sync when new admissions occur or when a discharge is recorded
- **visit_info_by_id** (dict)

```
visit_info_by_id[VisitID] = {
    "doctor_id": ...,
    "classification_id": ...
}
```

Stores key information about each visit needed for later aggregations:

- Which doctor is responsible for the visit → to map to a speciality.
- Which classification (Inpatient / Outpatient / Emergency / Ambulatory).

This is used mainly when processing **status events** to update:

- waiting_now_by_speciality
- waiting_now_by_classification

- **current_status_by_visit** (dict)

```
current_status_by_visit[VisitID] = StatusID
```

- Tracks the **latest known status** of each visit (from VisitStatusHistory).
- When a new status event arrives:
    - The service looks up the old status.
    - If the old status was "Waiting" and the new status is something else, it **decrements** the waiting counters.
    - If the new status is "Waiting", it **increments** the waiting counters.

This ensures that waiting_now_* reflects the real-time number of patients currently in "Waiting" state.

Metric definitions

This section describes the real-time metrics that the Python service computes in memory based on the Debezium change events coming from SQL Server.

1.Visits today

Definition
Visits today is the number of visits whose VisitDate is equal to the current calendar date (server local date).

How it is computed

- The service listens to the **Visits** topic (Debezium CDC events).
- For each message where:
  - op (operation) is c (create) or r (read/snapshot), and
  - VisitDate is **today**,
- The corresponding VisitID is added to an in-memory set visits_today_ids.
- The metric visits_today_total is the size of this set.
- At the same time, we increment a per-classification counter:
  - Look up ClassificationID of the visit.
  - Map it to a human-readable name, e.g. Inpatient, Outpatient, Emergency, Ambulatory.
  - Increment visits_today_by_classification[classification_name].

2.Admissions today

Definition
Admissions today is the number of new admissions where AdmissionDate is equal to the current date.

How it is computed

- The service listens to the **Admissions** topic.
- For each new admission event (typically op = 'c'):
  - If AdmissionDate is **today**, increment admissions_today by 1.

---

3. Inpatients now

Definition
Inpatients now is the current number of patients who are still admitted in the hospital (i.e., they have an open admission with no discharge date yet).

How it is computed

- The service maintains a set active_admissions containing the IDs of all admissions that are currently open.
- When an admission is created:
    - Add its AdmissionID to active_admissions.
- When an admission is updated with a DischargeDate (i.e., the patient leaves the hospital):
    - Remove its AdmissionID from active_admissions.
- The metric inpatients_now is simply the size of this set at any moment.

<u>4.Waiting now per classification / speciality</u>

Definition
These metrics show how many patients are **currently waiting** (status = "Waiting") grouped by:

- **Visit classification** (Inpatient, Outpatient, Emergency, Ambulatory)
- **Doctor speciality** (Emergency, General Surgery, Internal Medicine, Pediatrics, etc.)

We derive this from the **latest status** of each visit.

<u>How it is computed</u>

- The service listens to the **VisitStatusHistory** topic.
- For each status event:
    - It updates an in-memory map current_status_by_visit[VisitID] = StatusID.
    - It also knows, from previous **Visits** events:
        - ClassificationID for each VisitID
          → mapped to a classification name CLASSIFICATION_NAMES.
        - DoctorID for each VisitID, and from that the doctor's main speciality via DOCTOR_SPECIALITY.
- For each new status event:
    - Check the **old** status for this visit (if any).
    - Check the **new** StatusID.
    - If the old status was Waiting → decrement the counters for that visit's classification and speciality.
    - If the new status is Waiting → increment the counters for that visit's classification and speciality.
- The result is stored in:
    - waiting_now_by_classification[classification_name]
    - waiting_now_by_speciality[speciality_name]

So for any point in time, these counters reflect the **current** number of patients whose **latest** status is "Waiting".

5.Revenue paid today

Definition
Revenue paid today is the total value of all bills that:

- Have PaymentStatus = 'Paid', and
- Have PaymentDate equal to today.

It represents the **actual collected revenue** for the current day (not just billed amounts).

How it is computed

- The service listens to the **Bills** topic.
- For each bill event, it looks at both:
    - the before record (previous state), and
    - the after record (current state).
- For each side (before / after), it computes a helper value paid_today_amount(record):
    - If PaymentStatus != 'Paid' → amount = 0.
    - If PaymentDate is not today → amount = 0.
    - Else, parse TotalAmount as a decimal:
        - If it is a normal numeric type → convert directly to float.
        - If it comes from Debezium as a precise decimal:
            - It might be a string, or a small JSON object { "scale": 2, "value": "..." }, or Base64-encoded bytes like "C3Gw".
            - The service decodes the Base64 value into bytes, converts it to an integer, then divides by 10^scale to obtain the real decimal value.
    - Return that value.
- The **delta** is:
    - delta = after_paid_today - before_paid_today
- If delta is non-zero:
    - Add delta to revenue_paid_today.
    - Ensure revenue_paid_today never becomes negative (just in case of out-of-order or correction events).

This way, the metric always reflects the **net paid amount for today**, even if bills are updated or corrected.

Handlers logic (high-level)

This section summarizes the main responsibilities of each event handler in the Python service.

1.handle_visits_event

- Listens to Debezium events from the **Visits** topic.
- For each visit event:
  - Stores the visit's metadata in visit_info_by_id[VisitID] = { doctor_id, classification_id }.
  - If the VisitDate is today:
    - Add VisitID to visits_today_ids (set).
    - Increment visits_today_total.
    - Look up ClassificationID and increment visits_today_by_classification[classification_name].
- This handler is responsible for all **"visits today"** metrics and for providing doctor/classification info that other handlers (like status) reuse.

---

2. handle_status_event

- Listens to Debezium events from the **VisitStatusHistory** topic.
- For each status event:
  - Updates the latest status per visit: current_status_by_visit[VisitID] = StatusID.
  - Looks up the visit's classification and speciality using:
    - visit_info_by_id[VisitID] → ClassificationID and DoctorID
    - DOCTOR_SPECIALITY[DoctorID] → speciality
    - CLASSIFICATION_NAMES[ClassificationID] → classification name
  - Compares:
    - **Old** status (previous value in current_status_by_visit)
    - **New** status (StatusID from the event)
- Logic:
  - If the **old** status was "Waiting":
    - Decrement waiting_now_by_speciality and waiting_now_by_classification counters for that visit.
  - If the **new** status is "Waiting":
    - Increment those same counters.
- This ensures both "waiting now per classification" and "waiting now per speciality" always reflect the **current** states.

3.handle_admissions_event

- Listens to Debezium events from the **Admissions** topic.
- When a new admission is created:
  - If AdmissionDate is today → increment admissions_today.
  - Add the AdmissionID to active_admissions.

- When an admission is updated with a DischargeDate:
    - Remove the AdmissionID from active_admissions.
- Then recompute:
    - inpatients_now = len(active_admissions)
- This handler owns the “admissions today” and “inpatients now” metrics.

4.handle_bills_event

- Listens to Debezium events from the **Bills** topic.
- For each event:
    - Computes before_amt = paid_today_amount(before).
    - Computes after_amt = paid_today_amount(after).
    - Calculates delta = after_amt - before_amt.
- If delta != 0:
    - Add delta to revenue_paid_today.
    - Ensure revenue_paid_today >= 0.
- This approach correctly handles:
    - New paid bills today.
    - Bills that change from “Pending” to “Paid”.
    - Bills edited or cancelled, as long as the change appears in Debezium events.

Flask API

The Python service exposes a simple HTTP API via Flask for the dashboard to consume.

1. Endpoint

- **URL:** /stats
- **Method:** GET
- **Response format:** application/json

2. Response structure

The endpoint returns the current in-memory metrics as a single JSON object. Example:

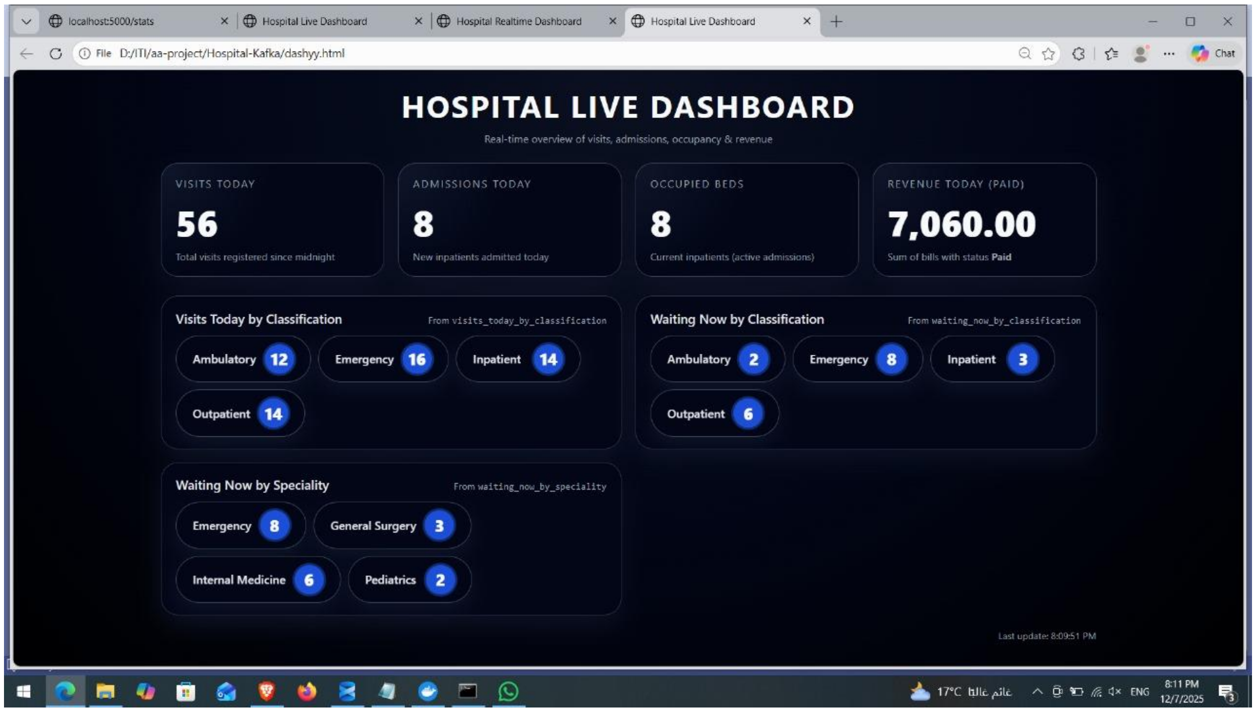
```
{
  "visits_today_total": 52,
  "visits_today_by_classification": {
    "Ambulatory": 12,
    "Emergency": 16,
    "Inpatient": 12,
    "Outpatient": 12
  },
  "admissions_today": 6,
  "inpatients_now": 6,
  "waiting_now_by_speciality": {
    "Emergency": 8,
    "General Surgery": 3,
    "Internal Medicine": 4,
    "Pediatrics": 2
  },
  "waiting_now_by_classification": {
    "Ambulatory": 2,
    "Emergency": 8,
    "Inpatient": 3,
    "Outpatient": 4
  },
  "revenue_paid_today": 6060.0,
  "last_update": "2025-12-07T17:52:44.759748+00:00"
}
```

Field meanings

- **visits_today_total**
  Total number of visits whose VisitDate is today.
- **visits_today_by_classification**
  A breakdown of today's visits by visit classification (Inpatient, Outpatient, Emergency, Ambulatory).
- **admissions_today**
  Number of admissions created today.
- **inpatients_now**
  Number of patients currently admitted (open admissions with no discharge date).
- **waiting_now_by_speciality**
  Number of *currently waiting* patients, grouped by doctor speciality. A visit is counted if its **latest** status is "Waiting".
- **waiting_now_by_classification**
  Number of *currently waiting* patients, grouped by visit classification (Inpatient, Outpatient, Emergency, Ambulatory).
- **revenue_paid_today**
  Total amount of bills paid today (PaymentStatus = 'Paid' and PaymentDate = today), correctly handling Debezium decimal encoding.
- **last_update**
  ISO-8601 timestamp (UTC) indicating when the last Kafka event was processed and the metrics were last refreshed.

Dashboard (HTML / JavaScript)

This section describes the web dashboard that visualizes the real-time hospital KPIs exposed by the Flask /stats API.



localhost:5000/stats
Hospital Live Dashboard
Hospital Realtime Dashboard
Hospital Live Dashboard
File D:/ITI/aa-project/Hospital-Kafka/dashyy.html

HOSPITAL LIVE DASHBOARD

Real-time overview of visits, admissions, occupancy & revenue

VISITS TODAY	ADMISSIONS TODAY	OCCUPIED BEDS	REVENUE TODAY (PAID)
56	8	8	7,060.00
Total visits registered since midnight	New inpatients admitted today	Current inpatients (active admissions)	Sum of bills with status **Paid**

Visits Today by Classification — From visits_today_by_classification

Ambulatory 12 · Emergency 16 · Inpatient 14 · Outpatient 14

Waiting Now by Classification — From waiting_now_by_classification

Ambulatory 2 · Emergency 8 · Inpatient 3 · Outpatient 6

Waiting Now by Speciality — From waiting_now_by_speciality

Emergency 8 · General Surgery 3 · Internal Medicine 6 · Pediatrics 2

Last update: 8:09:51 PM

1. UI layout

The dashboard is a single-page HTML application built using plain HTML, CSS and JavaScript. The layout is divided into several logical sections.

1.1 Top KPI cards

At the top of the page there is a row of four KPI cards. Each card shows a key real-time metric:

1. **Visits today**
   - Shows visits_today_total.
   - Represents how many visits have occurred today.
2. **Admissions today**
   - Shows admissions_today.
   - Represents how many patients were newly admitted today.
3. **Inpatients now**
   - Shows inpatients_now.
   - Represents how many patients are currently admitted (open admissions).
4. **Revenue today**
   - Shows revenue_paid_today.
   - Represents the total amount of bills that have been paid today.
   - Usually displayed with a currency suffix (e.g. “EGP”).

Each card typically contains:

- A **title** (the metric name).
- A **large numeric value** (the real-time value).
- Optional small subtitle or icon to visually distinguish the cards.

This gives a quick "at a glance" overview of the hospital's current activity.

1.2Visits by classification

Below the KPIs, there is a section that shows **Visits by classification**.

- It uses the visits_today_by_classification object returned from /stats.
- Keys are the classification names:
  - Inpatient
  - Outpatient
  - Emergency
  - Ambulatory
- Values are counts of visits today in each classification.

The UI can be rendered as:

- A group of horizontal bars, or
- A row of small cards, each showing:
  - Classification name.
  - Count of visits.

This section helps understand the distribution of today's visits across different care types.

1.3 Waiting now by speciality

Another section displays **current waiting patients grouped by speciality**.

- It uses the waiting_now_by_speciality object.
- The keys are speciality names, for example:
  - Emergency
  - Internal Medicine
  - General Surgery
  - Pediatrics
  - etc.

The value is how many patients currently have Status = Waiting for visits that belong to this speciality.

In the UI, each speciality is shown as a small card or block that may contain:

- The speciality name.
- A **triangle icon** or badge representing the queue.

- The numeric count of waiting patients.

This lets the user instantly see where the highest pressure / queues are in the hospital.

1.4 Waiting now by classification

There is also a section for **waiting now per classification**.

- It uses waiting_now_by_classification, with keys such as:
    - Inpatient
    - Outpatient
    - Emergency
    - Ambulatory
- Each classification shows how many patients are currently in the **Waiting** status in that class.

The visualization is similar to the speciality section (cards, triangles, or badges), but grouped by classification instead of speciality. Together, both views give an operational picture of queues per type of service.

1.5 Last update time

At the bottom or in the header, there is a small text label that shows:

- last_update from the API, displayed as an ISO timestamp (UTC) or formatted into local time.

This indicates when the dashboard metrics were last refreshed from Kafka events.

---

2. Data fetching and live updates

The dashboard uses a simple polling mechanism to keep the metrics up to date.

2.1 Fetching from /stats

- The frontend makes HTTP GET requests to the Flask endpoint:
- GET /stats
- The response is JSON with all metrics in one object.
- In the JavaScript code, a function such as fetchStats() is responsible for:
    1. Calling fetch("http://localhost:5000/stats").
    2. Parsing the JSON using response.json().
    3. Updating the corresponding DOM elements (text nodes, numbers, bars, triangles, etc.) with the latest values.

2.2 Update interval

- After the page loads, the dashboard calls fetchStats() once to populate the initial values.
- Then it sets a timer using setInterval(fetchStats, 2000) so that:
    - Every **2 seconds** (or the configured interval), it re-requests /stats.
    - The UI is refreshed with new numbers whenever Kafka events update the in-memory stats.

This approach is simple but effective for near-real-time monitoring, and does not require WebSockets or complex streaming in the frontend.

How to Run (Setup & Runbook)

This section describes how to set up and run the full pipeline:

SQL Server → Debezium → Kafka → Python (consumer + API) → HTML Dashboard

---

1 Prerequisites

Before running the system, make sure you have the following components installed and running:

1.1 Database

- Microsoft SQL Server (on Windows or Linux / Docker).
- The hospital database scripts:
  - CREATE DATABASE Hospital_Salam;
  - All CREATE TABLE scripts for:
    - Reference tables
      VisitClassification, VisitType, VisitStatus, OperationType, PayerTypes, TriageLevels, ServiceCatalog, Speciality, Doctors, Doctor_Speciality, Wards, Beds
    - Core entity tables
      Patients, Visits, VisitStatusHistory, Admissions, OperationSessions, EmergencyDetails, Appointments, Bills, BillItems
  - Optional weekly snapshot tables (Weekly_Visits, Weekly_Admissions, …) if you are using them.

1.2. Kafka & Debezium

- Apache Kafka (single broker or cluster).
- Zookeeper (if using the classic Kafka setup).
- Kafka Connect with the Debezium SQL Server connector plugin installed.
- A running Kafka Connect instance that can reach:
  - The SQL Server instance.
  - The Kafka broker(s).

1.3. Python environment

- Python 3.x
- Required libraries (for the real-time stats service):

```
pip install kafka-python flask flask-cors
```

- The Python app file, e.g. dashy.py, which:
  - Consumes from Kafka topics.
  - Maintains in-memory statistics.
  - Exposes the /stats HTTP endpoint via Flask.

1.4. Frontend

- A simple static web page, e.g. dashboard.html, that:
  - Uses JavaScript fetch() to call http://localhost:5000/stats.
  - Displays the metrics in cards/sections.

2. Step 1 – Create the database and schema

1. Start SQL Server.
2. Create the **Hospital_Salam** database:

```
CREATE DATABASE Hospital_Salam;

GO
```

3. Execute your schema scripts in order:
   1. Reference tables (Speciality, VisitClassification, VisitType, VisitStatus, etc.).
   2. Core tables (Patients, Visits, VisitStatusHistory, Admissions, Appointments, Bills, BillItems, …).
4. Insert the **reference data** (lookup tables), for example:
   1. Speciality rows (Emergency, Internal Medicine, …).
   2. VisitClassification (Inpatient, Outpatient, Emergency, Ambulatory).
   3. VisitStatus (Registered, Waiting, In Consultation, …).
   4. PayerTypes, ServiceCatalog, TriageLevels, OperationType, etc.

This ensures the database is structurally ready and has all fixed reference values.

3. Step 2 – Enable CDC and configure Debezium

3.1. Enable CDC on the database

On SQL Server, enable CDC on the database:

```
USE master;

GO

ALTER DATABASE Hospital_Salam

SET ALLOW_SNAPSHOT_ISOLATION ON;

GO


USE Hospital_Salam;

GO

EXEC sys.sp_cdc_enable_db;

GO
```

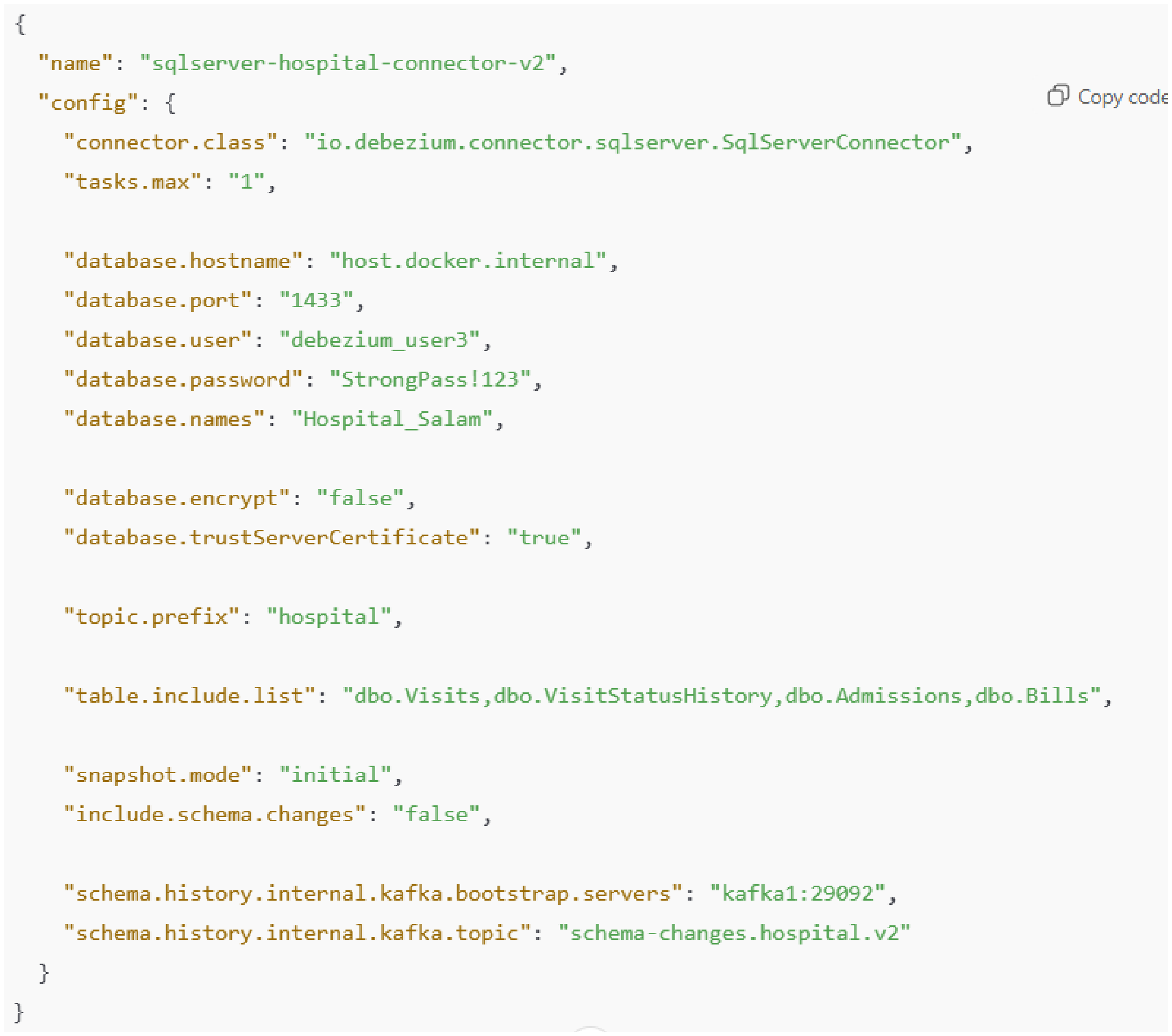
3.2. Enable CDC on the tables you want to capture

For example:

```
EXEC sys.sp_cdc_enable_table
    @source_schema = 'dbo',
    @source_name   = 'Visits',
    @role_name     = NULL;

EXEC sys.sp_cdc_enable_table
    @source_schema = 'dbo',
    @source_name   = 'VisitStatusHistory',
    @role_name     = NULL;

EXEC sys.sp_cdc_enable_table
    @source_schema = 'dbo',
    @source_name   = 'Admissions',
    @role_name     = NULL;

EXEC sys.sp_cdc_enable_table
    @source_schema = 'dbo',
    @source_name   = 'Bills',
    @role_name     = NULL;
```

3.3. Configure Debezium SQL Server connector

In Kafka Connect, create a connector using a JSON config similar to:



```json
{
  "name": "sqlserver-hospital-connector-v2",
  "config": {
    "connector.class": "io.debezium.connector.sqlserver.SqlServerConnector",
    "tasks.max": "1",

    "database.hostname": "host.docker.internal",
    "database.port": "1433",
    "database.user": "debezium_user3",
    "database.password": "StrongPass!123",
    "database.names": "Hospital_Salam",

    "database.encrypt": "false",
    "database.trustServerCertificate": "true",

    "topic.prefix": "hospital",

    "table.include.list": "dbo.Visits,dbo.VisitStatusHistory,dbo.Admissions,dbo.Bills",

    "snapshot.mode": "initial",
    "include.schema.changes": "false",

    "schema.history.internal.kafka.bootstrap.servers": "kafka1:29092",
    "schema.history.internal.kafka.topic": "schema-changes.hospital.v2"
  }
}
```

Post this JSON to the Kafka Connect REST API (e.g. POST /connectors).
After the connector starts:

- An initial snapshot of the listed tables is sent to Kafka.
- Then every change (INSERT/UPDATE/DELETE) is streamed live.

The main topics used by the dashboard are:

- hospital.Hospital_Salam.dbo.Visits
- hospital.Hospital_Salam.dbo.VisitStatusHistory
- hospital.Hospital_Salam.dbo.Admissions
- hospital.Hospital_Salam.dbo.Bills

4. Step 3 – Start Kafka and Kafka Connect

Make sure the following services are running in the correct order:

1. **Zookeeper** (if using the classic deployment).
2. **Kafka broker(s)**.
3. **Kafka Connect** with the Debezium connector configured.
4. Verify that topics for the hospital tables exist (e.g. using kafka-topics or a UI like Kafka UI).

5. Step 4 – Run the Python real-time stats service

5. From the project directory (where `dashy.py` is located):

   ```
   python dashy.py
   ```

This will:

- Start a **KafkaConsumer** that subscribes to:
  - hospital.Hospital_Salam.dbo.Visits
  - hospital.Hospital_Salam.dbo.Admissions
  - hospital.Hospital_Salam.dbo.VisitStatusHistory
  - hospital.Hospital_Salam.dbo.Bills
- Maintain the in-memory statistics dictionary (stats).
- Run a **Flask** HTTP server exposing the /stats endpoint.

You should see output similar to:

```
HTTP API running on http://localhost:5000/stats
Starting Kafka consumer...
 * Running on http://127.0.0.1:5000
```

6. Step 5 – Open the dashboard

Open dashboard.html in your browser (e.g. by double-clicking it or serving it via a simple HTTP server).

The JavaScript inside the page will:

- Periodically call http://localhost:5000/stats using fetch().
- Update the UI components with the values returned from the API:
  - visits_today_total
  - admissions_today
  - inpatients_now
  - revenue_paid_today

- visits_today_by_classification
- waiting_now_by_speciality
- waiting_now_by_classification
- last_update

Initially, everything may be zero until events arrive from Kafka.

7. Step 6 – Insert sample data and observe the dashboard

To test the full flow:

1. Insert sample rows into **Patients**, then create **Visits** for today:

```
INSERT INTO Patients (FirstName, LastName, Phone, Age, Gender, Nationality, InsuranceProv
VALUES ('Omar', 'Ali', '01011111111', 32, 'M', 'Egyptian', 'AXA Insurance');

INSERT INTO Visits (PatientID, DoctorID, ClassificationID, TypeID, VisitDate)
VALUES (1, 2, 2, 1, GETDATE());  -- Outpatient visit today
```

2. Insert corresponding **VisitStatusHistory** entries to simulate status changes (Registered → Waiting → In Consultation → Finished).
3. Insert **Admissions** for inpatients and set DischargeDate to NULL for active inpatients.
4. Insert **Bills** with:
   1. PaymentStatus = 'Paid'
   2. PaymentDate = today
   3. A valid TotalAmount.
5. After each insert/update, Debezium streams the change to Kafka, the Python consumer updates stats, and the dashboard shows:
   1. Increasing **Visits today**.
   2. Updated **Admissions today** and **Inpatients now**.
   3. **Waiting** counts per classification and speciality.
   4. Non-zero **Revenue today**.

This completes the end-to-end runbook for your real-time hospital dashboard system.

Lessons Learned & Challenges

During the implementation of the real-time hospital dashboard, several practical challenges appeared. This section summarises the most important ones and how they were handled.

---

1. Handling Debezium decimal (Base64) values

Challenge:
Debezium represents SQL Server DECIMAL / NUMERIC in different ways depending on the serialization format.
In JSON, some decimal fields (like TotalAmount in Bills) may arrive as:

- A plain number (e.g. 270.0), or
- A numeric string (e.g. "270.00"), or
- A Base64-encoded binary representation of the unscaled decimal (e.g. "C3Gw"), or
- A structured object like:

```
{ "scale": 2, "value": "C3Gw" }
```

Trying to directly convert "C3Gw" to float obviously fails and caused errors in the consumer.

Solution:
A helper function decode_decimal_value was implemented to normalize any Debezium decimal into a Python float:

- If the value is already an int/float → cast directly.
- If it’s a plain numeric string → parse with float().
- If it’s a dict → read scale and value recursively.
- If it’s Base64 → decode bytes, convert to integer, then divide by 10^scale.

This allowed the **revenue_paid_today** metric to work correctly regardless of the internal decimal representation.

2. Messages with payload = null (tombstones / schema events)

Challenge:
Kafka topics from Debezium sometimes include messages where:

```
{
  "schema": { ... },
  "payload": null
}
```

These “tombstone” or metadata messages are not real row changes. If the consumer tries to treat them as normal records, it leads to NoneType errors or invalid processing.

Solution:

In the consumer loop, every message is validated:

- Check that the message value is a dict.
- Extract payload = data.get("payload").
- If payload is None or empty, **skip** the message and continue.

This simple guard makes the consumer robust to non-data messages produced by Debezium and Kafka Connect.

3. Keeping “waiting” counters consistent (per speciality & classification)

Challenge:
The dashboard needs live metrics for:

- waiting_now_by_speciality
- waiting_now_by_classification

These depend on the **latest status** of each visit from VisitStatusHistory. A visit can change status many times (Registered → Waiting → In Consultation → Finished, etc.). If we only increment counters when we see a Waiting status, the numbers will drift over time as patients move out of “Waiting”.

Solution:

A small in-memory state model is maintained:

- current_status_by_visit[VisitID] = last StatusID
- visit_info_by_id[VisitID] = {doctor_id, classification_id}

In handle_status_event:

1. Read the **old status** (if any) from current_status_by_visit.
2. Update the **new status**.
3. If the old status was Waiting → decrement the relevant counters for that visit’s:
    - speciality (via doctor → speciality mapping),
    - classification (Inpatient, Outpatient, …).
4. If the new status is Waiting → increment the same counters.

This “increment/decrement” approach guarantees that:

- Numbers reflect the **current** waiting list, not the history.
- Counters do not grow infinitely.
- A visit is counted as “waiting” in exactly one classification and one speciality at a time.

4. In-memory state across topics (visits, statuses, admissions, bills)

Challenge:
The metrics combine information from multiple topics:

- Visits → Visit date, classification, doctor.
- VisitStatusHistory → current status of each visit.
- Admissions → inpatient admissions and discharges.
- Bills → paid revenue.

Events from these topics arrive **asynchronously** and not necessarily in the “ideal order”. For example, a status event may arrive before or after you have processed the corresponding Visit event.

Solution:

Several helper structures are used:

- visits_today_ids → set of VisitIDs that occurred today (for de-duplicating counts).
- active_admissions → set of AdmissionIDs currently in hospital.
- visit_info_by_id → central mapping from VisitID to {doctor_id, classification_id}.
- current_status_by_visit → tracks the latest known status for each visit.

Access to these structures is protected by a thread lock (lock) because both the Kafka consumer thread and the Flask HTTP handler access them.

This design keeps the logic simple and lets the system build up a consistent picture of the current day, even if messages arrive slightly out of order.

5. Consumer group, partitions, and why a single consumer

Challenge:
Each Kafka topic is configured with multiple partitions. In Kafka, messages are distributed across these partitions, and consumers in the same **consumer group** share them.

If multiple consumers in the same group update shared in-memory stats, you can easily get:

- Race conditions (simultaneous updates without proper locking).
- Double counting or missed messages (if different consumers manage different partitions but share state incorrectly).
- Very complex synchronization logic.

Solution:

For this real-time dashboard, the design intentionally uses:

- **One consumer process** (the dashy.py script).
- **One consumer group** for the dashboard (e.g. "hospital-dashboard-group").

- A single thread that processes all partitions sequentially.

This gives:

- **Simplicity**: one place where all stats are updated.
- **Consistency**: every event is processed exactly once by the dashboard.
- Easy debugging and reasoning about metrics behavior.
