Technical Report – IoT Operations Platform (Lab 8)

Transactional Integrity, Concurrency, and Access Control in SQL Server
AdventureWorks 2019 – IoT Schema

1. Executive Summary

This lab focused on building several key components of a reliable IoT backend system using SQL Server.
The main goals were to design a proper database schema for IoT devices, implement a multi-step transactional workflow for telemetry ingestion, analyze concurrency behavior through different isolation levels, and set up role-based access control that follows the principles of least privilege.

By completing all six tasks, I was able to create a working IoT schema, build a transactional stored procedure that handles data validation and error logging, test how concurrent sessions interact, and configure multiple user roles that match the business requirements. Overall, the lab helped me understand how SQL Server handles transactions, locking, and permission management in a realistic IoT scenario where thousands of devices may send telemetry at the same time.

2. IoT Business Case Summary

AdventureWorks is expanding into IoT products, including SmartBike devices and factory-floor monitoring equipment.
Each device reports telemetry data every 30 seconds, and the backend must handle more than 250,000 telemetry messages per hour.

Because of that scale, the system must deal with several issues:

- Incomplete or corrupted telemetry cycles
- Conflicts caused by concurrent updates
- The need for atomic transactions (either insert all required records or none)
- Strict separation of permissions due to privacy rules

The purpose of this lab was to build the database foundation required for an enterprise IoT platform.
Tasks included designing the schema, building reliable ingestion logic, handling concurrency, and enforcing security based on user responsibilities.

3. Transaction Workflow Description (Stored Procedure)

The telemetry ingestion stored procedure (`IoT.IngestTelemetry`) was written to simulate how a real IoT backend processes one complete telemetry cycle.
The procedure performs the following steps inside a single transaction:

Step 1 — Device validation

The procedure first checks if the DeviceID exists. If not, it throws an error.

Step 2 — Sensor data validation

Several checks ensure that data is within acceptable ranges:

- Temperature cannot be lower than -50 or higher than 150
- Battery level must be between 0 and 100
- GPS coordinates cannot be NULL

If any validation fails, the procedure throws an error.

Step 3 — Insert telemetry record

The raw telemetry packet is inserted into `IoT.Telemetry`.

Step 4 — Update LastHeartbeat

The `IoT.Device` table stores the time of the most recent telemetry update.

Step 5 — Insert DeviceHealth record

The system calculates a temperature status, battery status, and vibration status and inserts a summary row into `IoT.DeviceHealth`.

Step 6 — Generate alerts

Depending on the telemetry values, several alerts may be created:

- Low battery
- High temperature
- Invalid GPS coordinates

Step 7 — Write success audit

A summary of the processed telemetry cycle is written into `IoT.TelemetryAudit`.

Error handling

If anything fails:

- The entire transaction is rolled back
- A new error log entry is inserted into `IoT.TelemetryErrorLog`
- The error is re-thrown to the user

This ensures atomicity: **either all steps succeed, or none do**.

(Reference: Stored Procedure.sql)

4. Concurrency Test Results

To observe the behavior of concurrent transactions, I used two SSMS sessions (Session A and Session B) to update the same rows and run SELECT statements under different isolation levels.

Blocking Scenario

When both sessions attempted to update the same DeviceID, one session became blocked until the other finished its transaction. This demonstrated how SQL Server enforces locking to prevent inconsistent updates.

(Reference: Session A.sql and Session B.sql)

Dirty Read (READ UNCOMMITTED)

Under this isolation level, Session B was able to read uncommitted data from Session A. This proved that dirty reads are possible when using READ UNCOMMITTED.

Read Committed

When using READ COMMITTED, Session B waited until Session A committed before it could read the updated row. This prevented dirty reads.

Snapshot Isolation

After enabling snapshot isolation, Session B read the old committed version of the row while Session A was still updating it. No blocking occurred, and no dirty reads happened.

(Reference: Snapshot Isolation Setup.sql)

Serializable

Under SERIALIZABLE, SQL Server prevented both sessions from inserting rows that could lead to phantom reads.
This was the strictest isolation level and caused the most blocking.

Summary of Observed Behaviors

Isolation Level	Dirty Reads	Blocking	Notes
READ UNCOMMITTED	Yes	No	Fast but unsafe
READ COMMITTED	No	Yes	SQL Server default
SNAPSHOT	No	No	Reads older version
SERIALIZABLE	No	Yes (most)	Strictest isolation

Overall, the tests matched what is expected from SQL Server's concurrency model.

5. Role-Based Permission Design

The roles were designed according to the business rules in the case description.

IoTDeviceAgent

- Can insert/update only IoT.Device
- Cannot access telemetry or customer data

TelemetryIngestionService

- Can insert into Telemetry and DeviceHealth
- Can update only LastHeartbeat
- Cannot read customer tables

IoTAnalyst

- Read-only access to Telemetry and DeviceHealth
- No write permissions
- No access to customer data

IoTFieldTechnician

- Can insert into MaintenanceLog
- Cannot query DeviceOwner
- UPDATE on MaintenanceLog fails because this role lacks SELECT

(This matches the "least privilege" policy.)

SecurityComplianceOfficer

- Read-only access to Device, DeviceOwner, and Telemetry
- No modification privileges

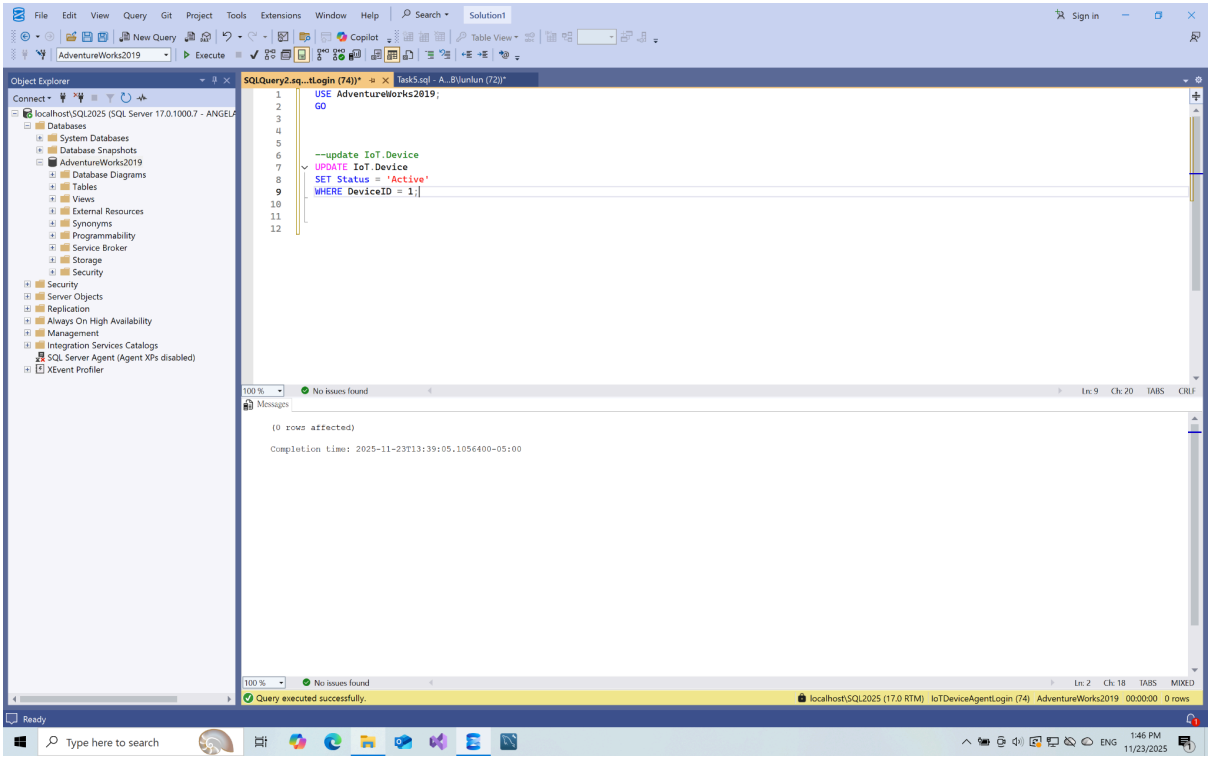
(Reference: Task5.sql)

6. Permission Validation (Task 6)

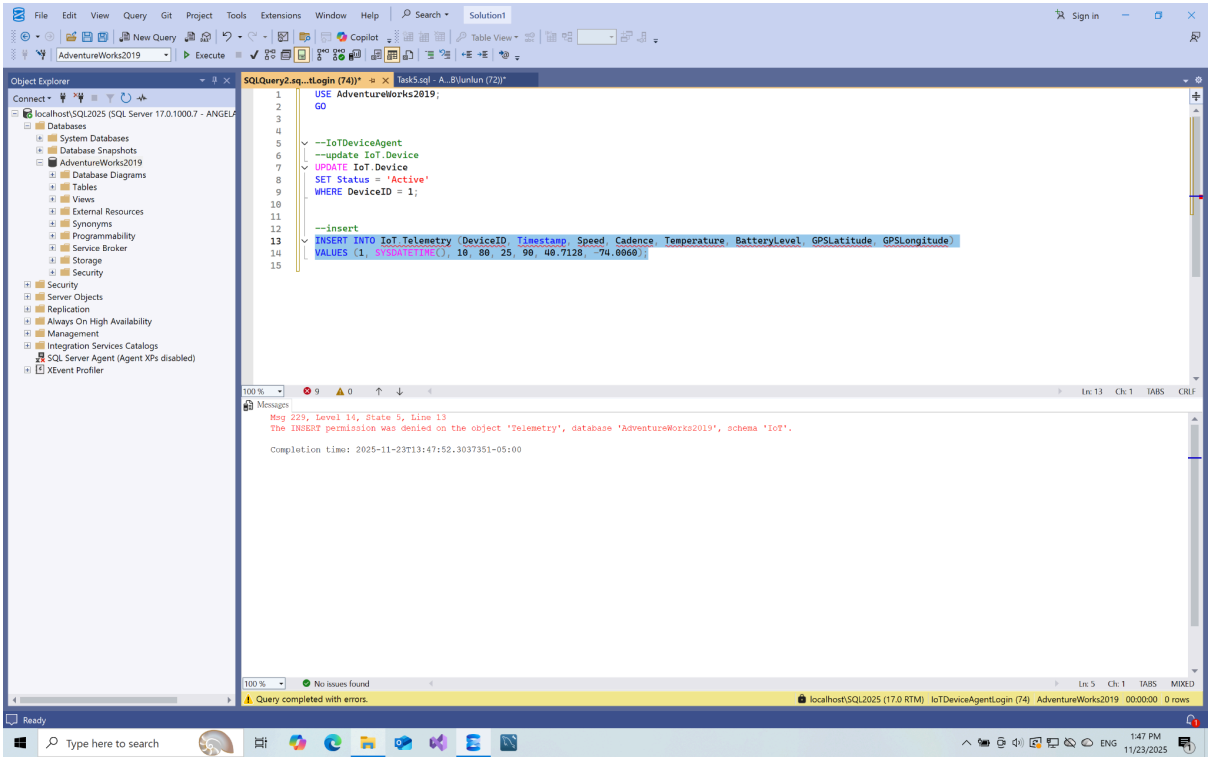
Each role was tested by logging in under a separate SQL Server login and executing statements allowed or denied by the role.

IoTDeviceAgent

- UPDATE Device → Success

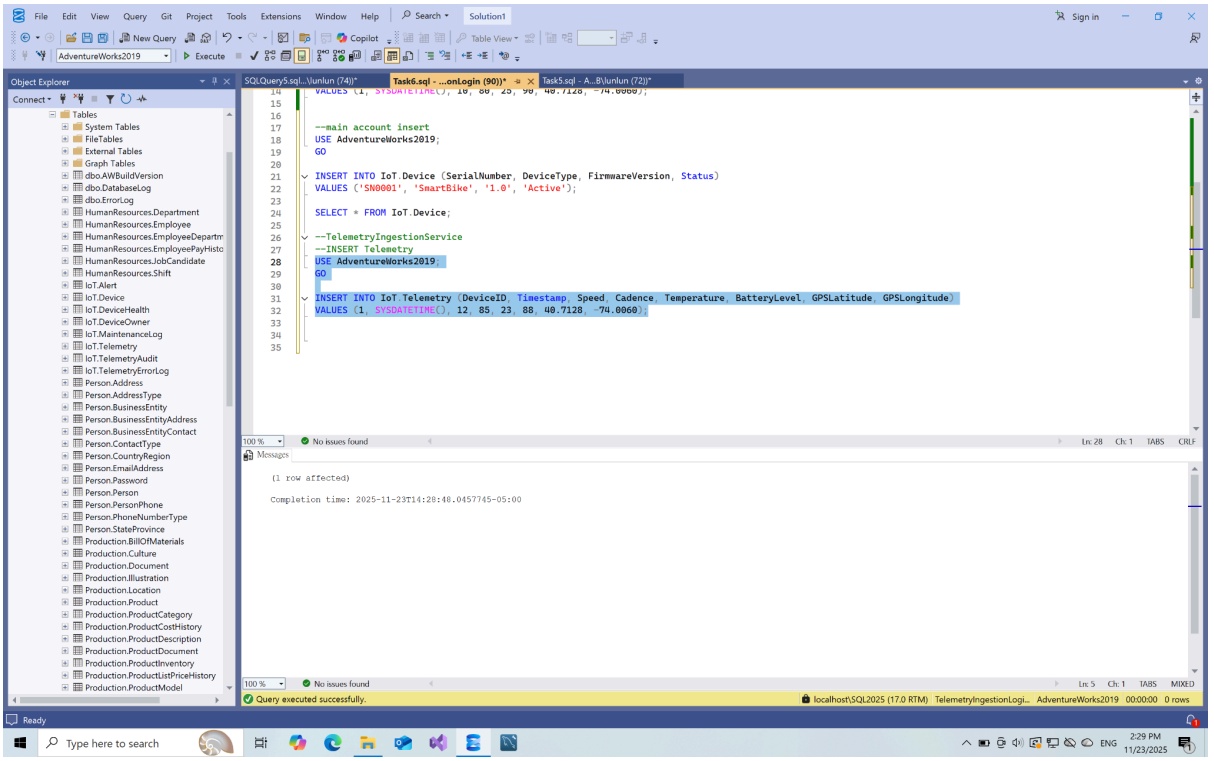


- INSERT Telemetry → Permission denied (Correct)

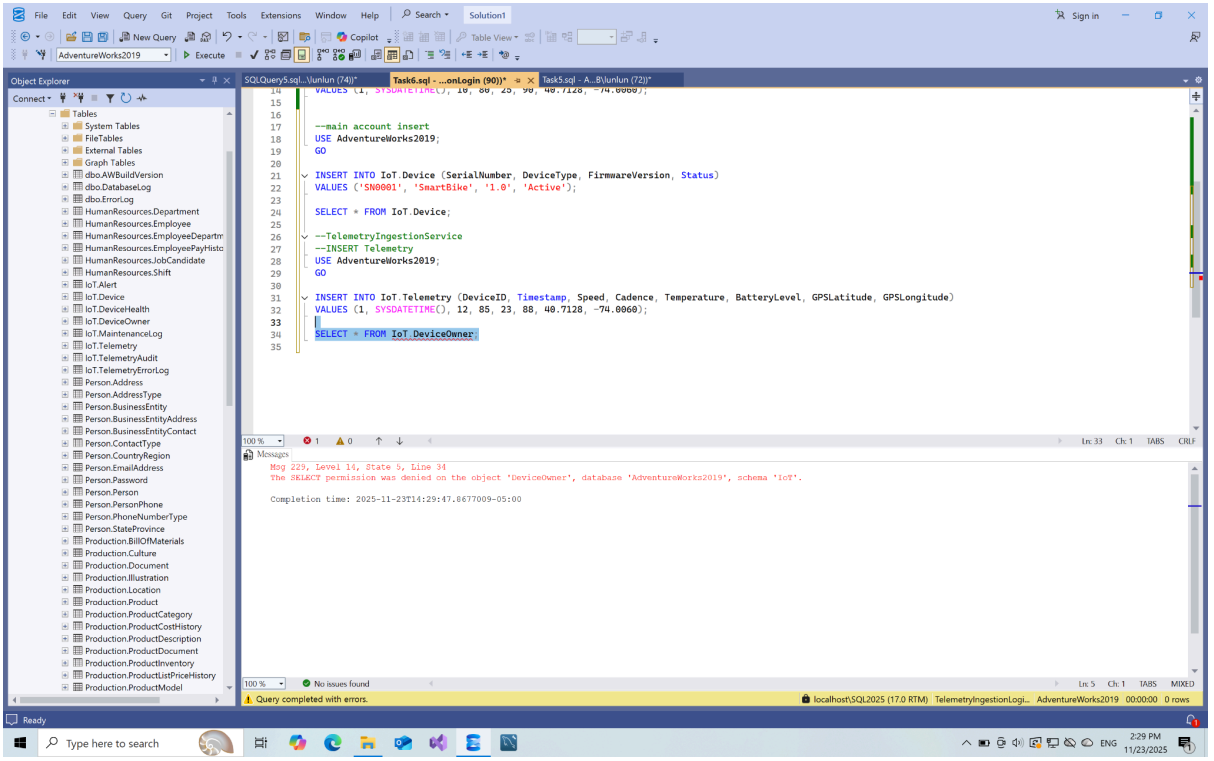


TelemetryIngestionService

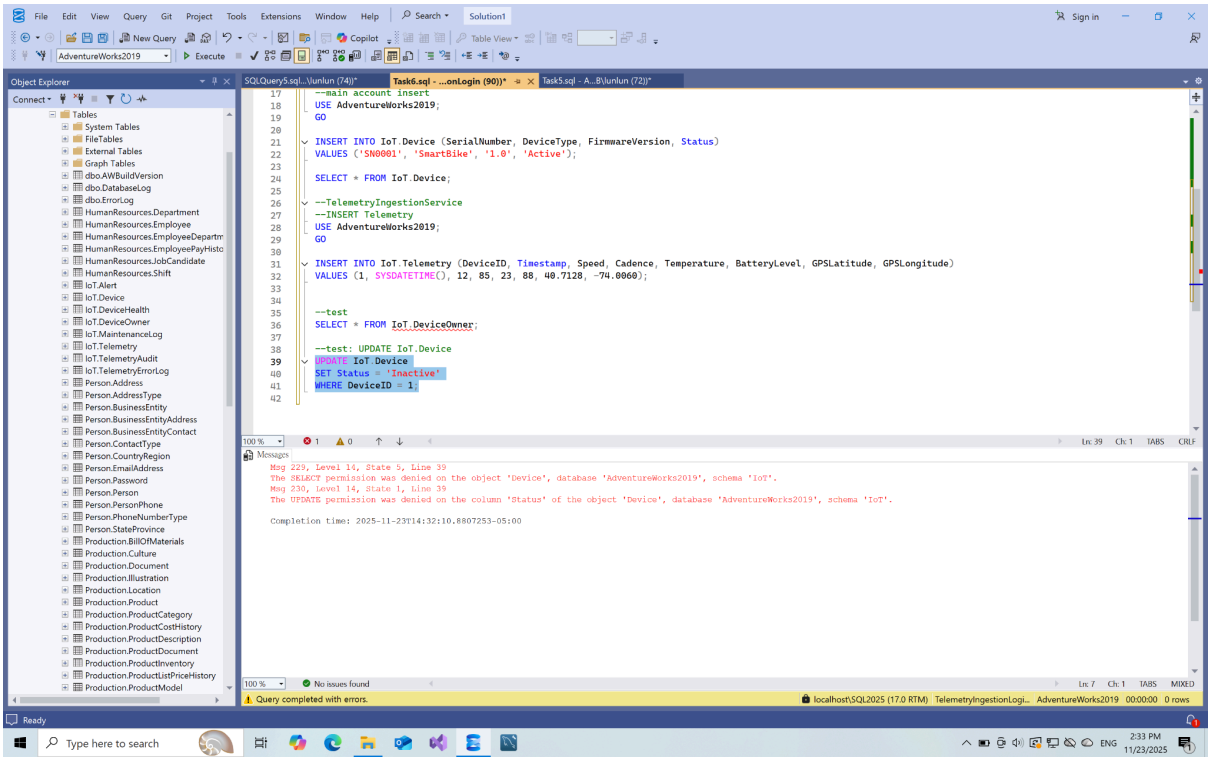
- INSERT Telemetry → Success



- SELECT DeviceOwner → Permission denied

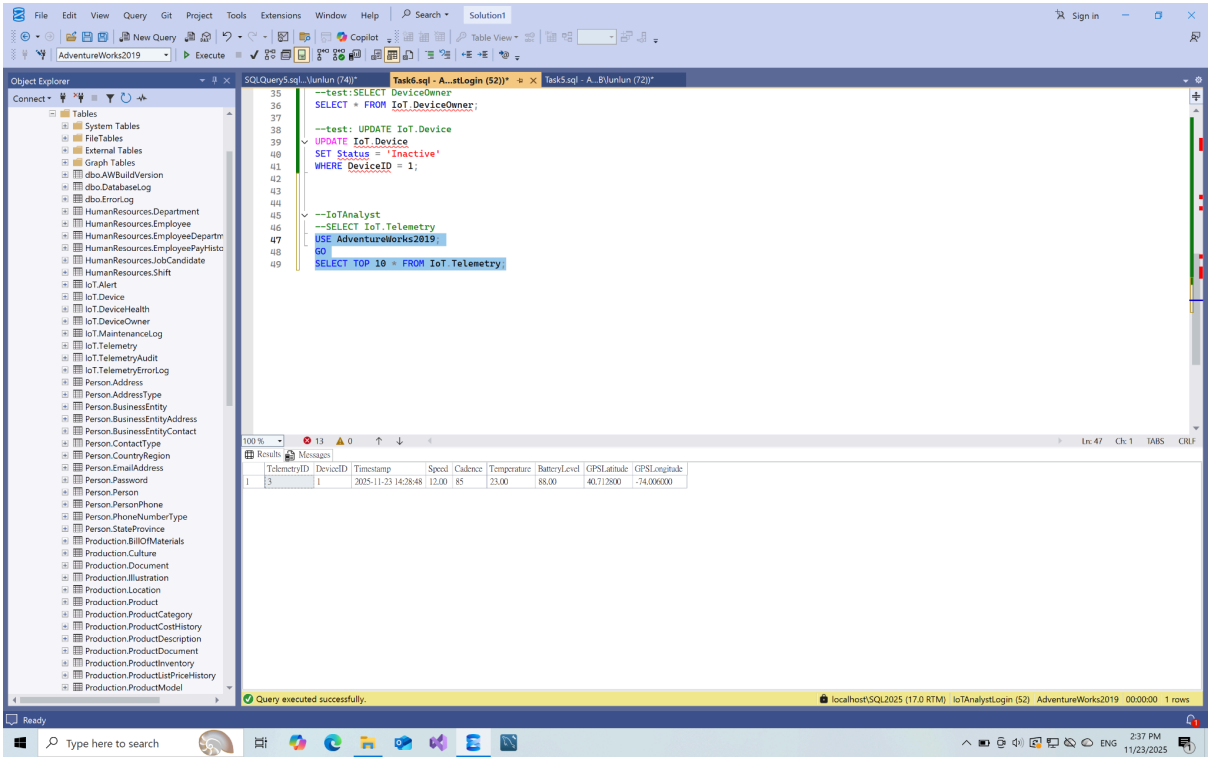


- UPDATE Device → Permission denied (except LastHeartbeat)

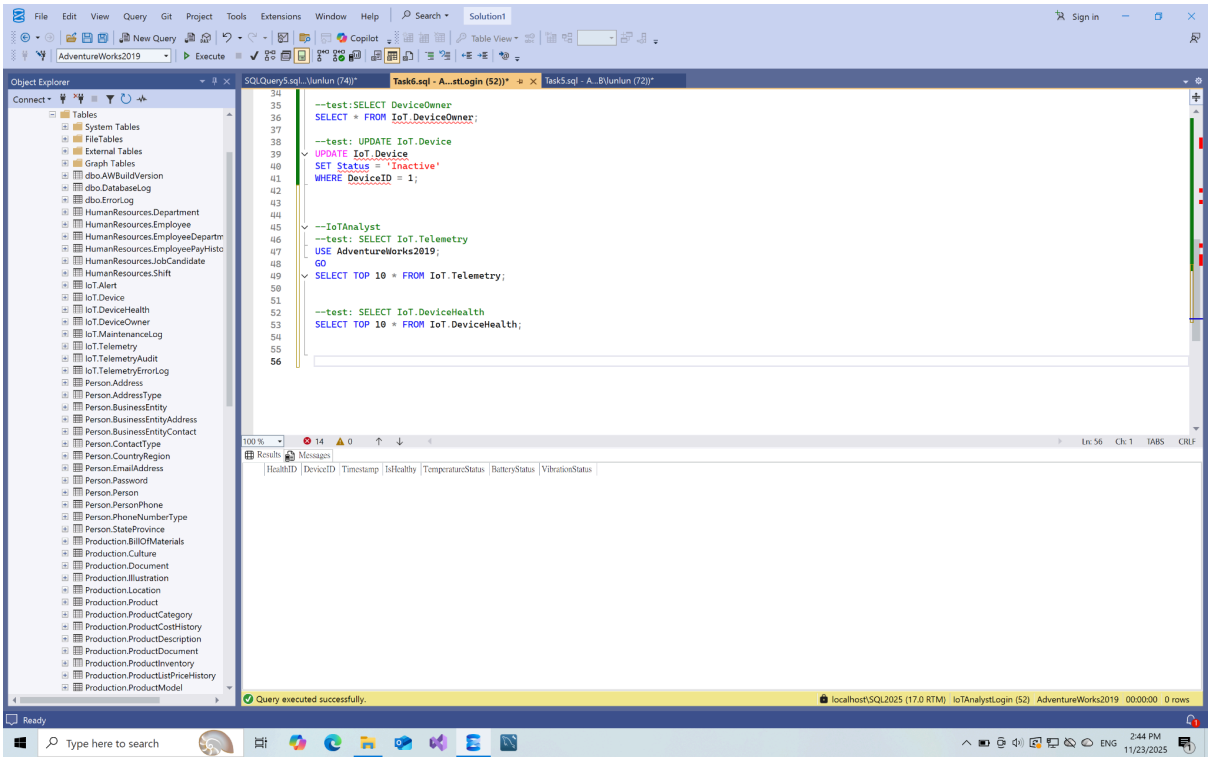


IoTAnalyst

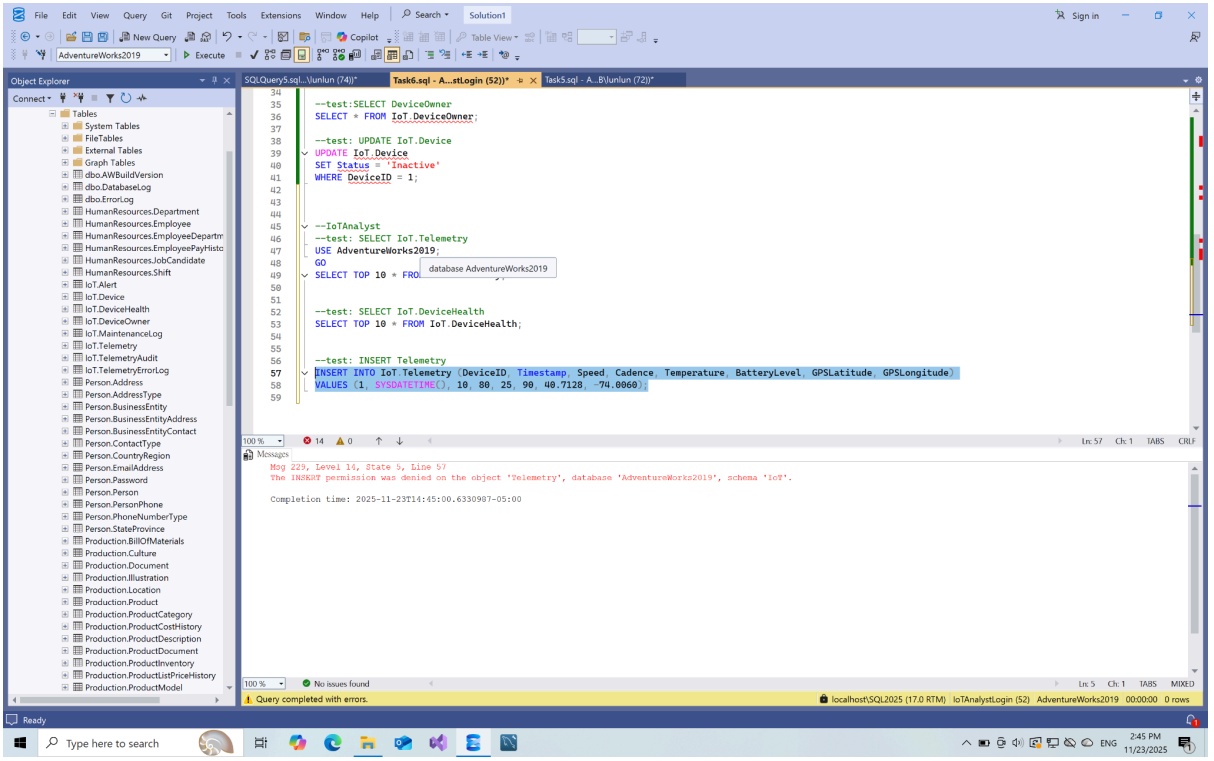
- SELECT Telemetry → Success



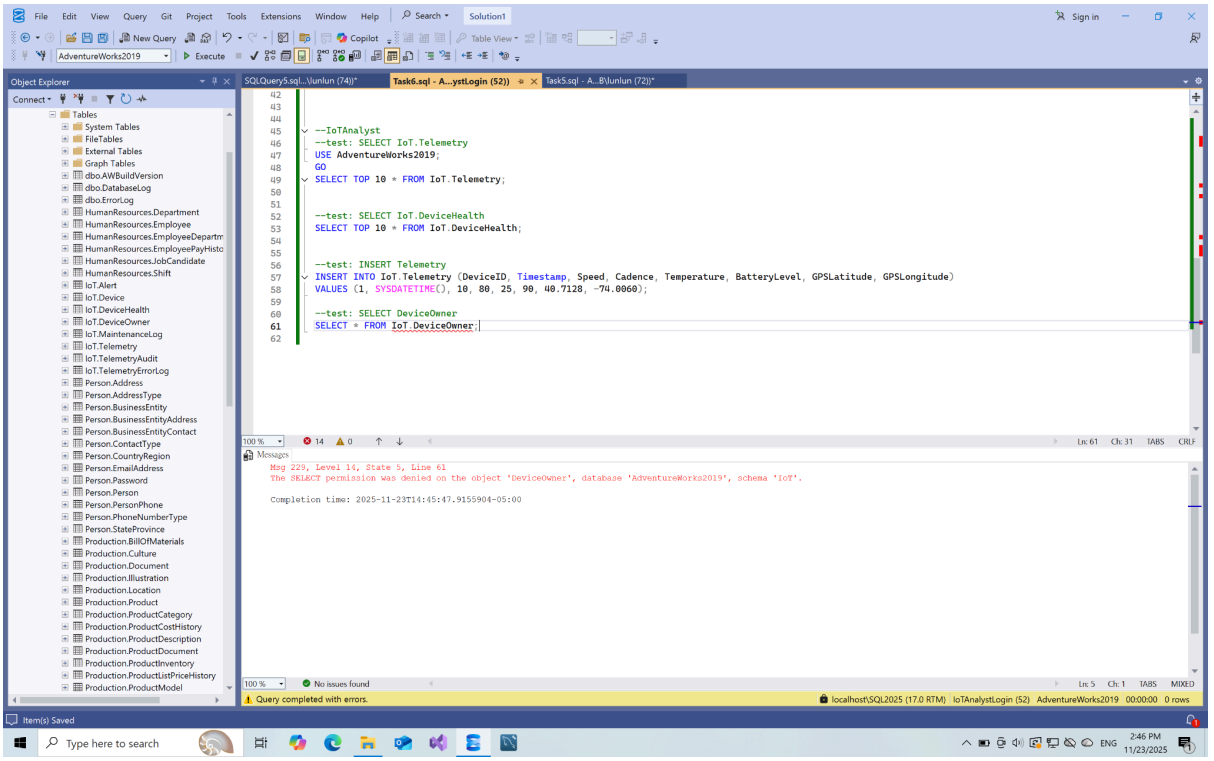
- SELECT DeviceHealth → Success



- INSERT Telemetry → Denied

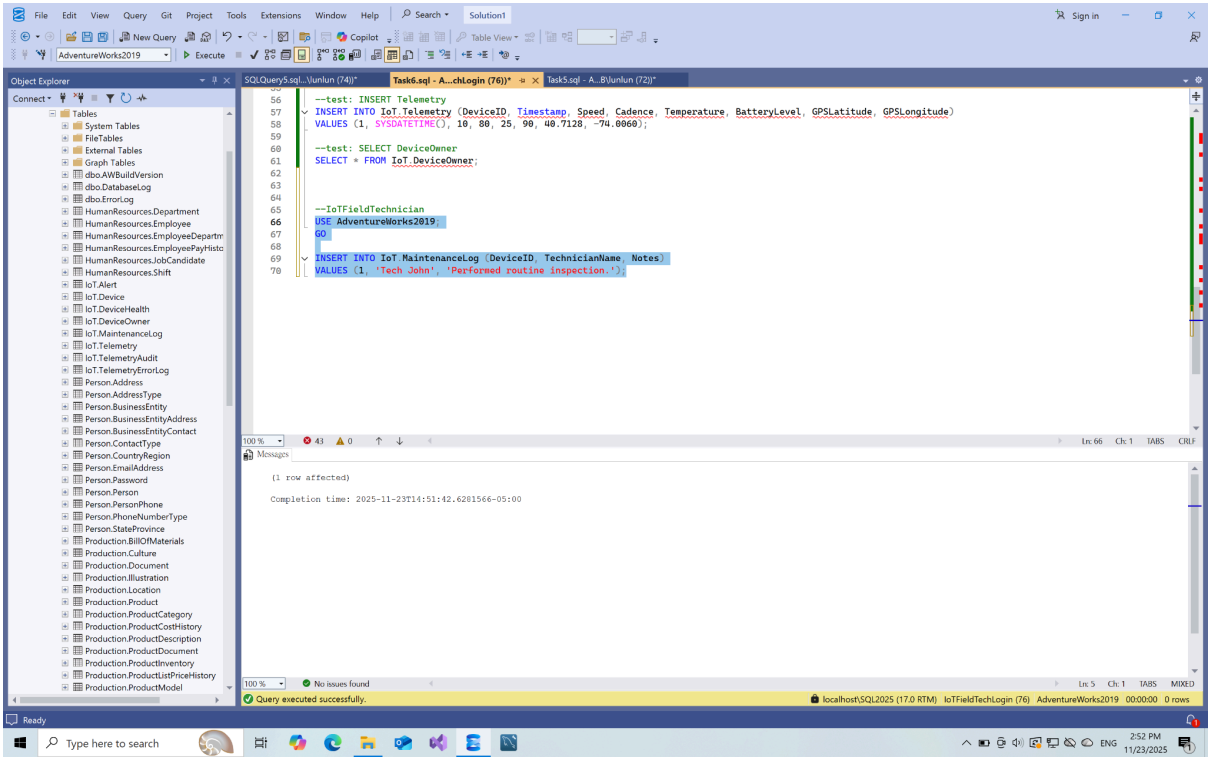


- SELECT DeviceOwner → Denied

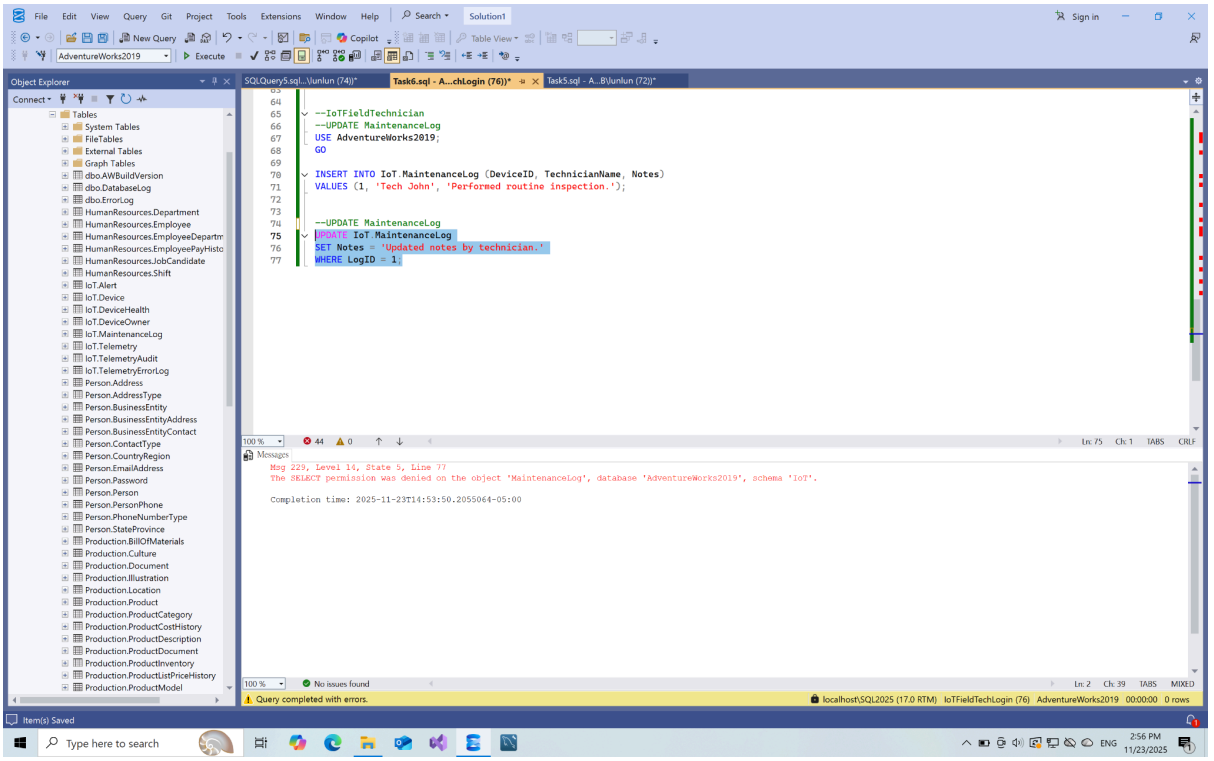


IoTFieldTechnician

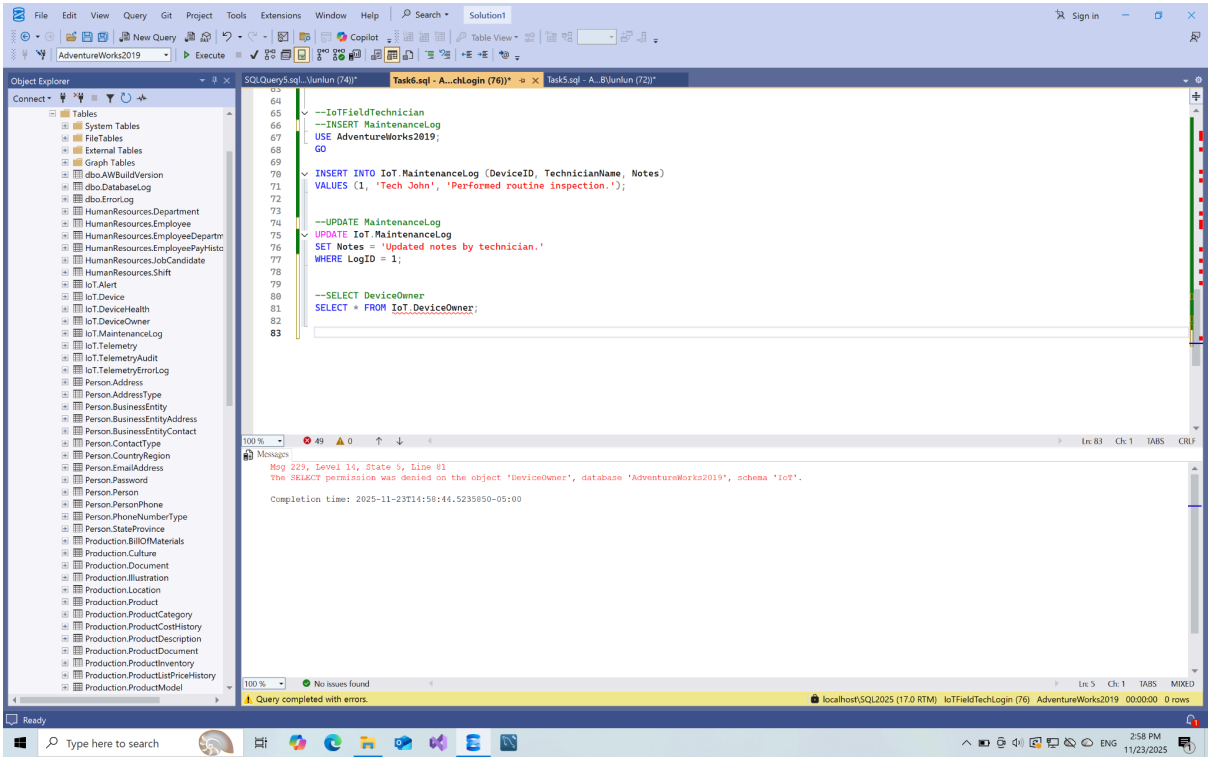
- INSERT MaintenanceLog → Success



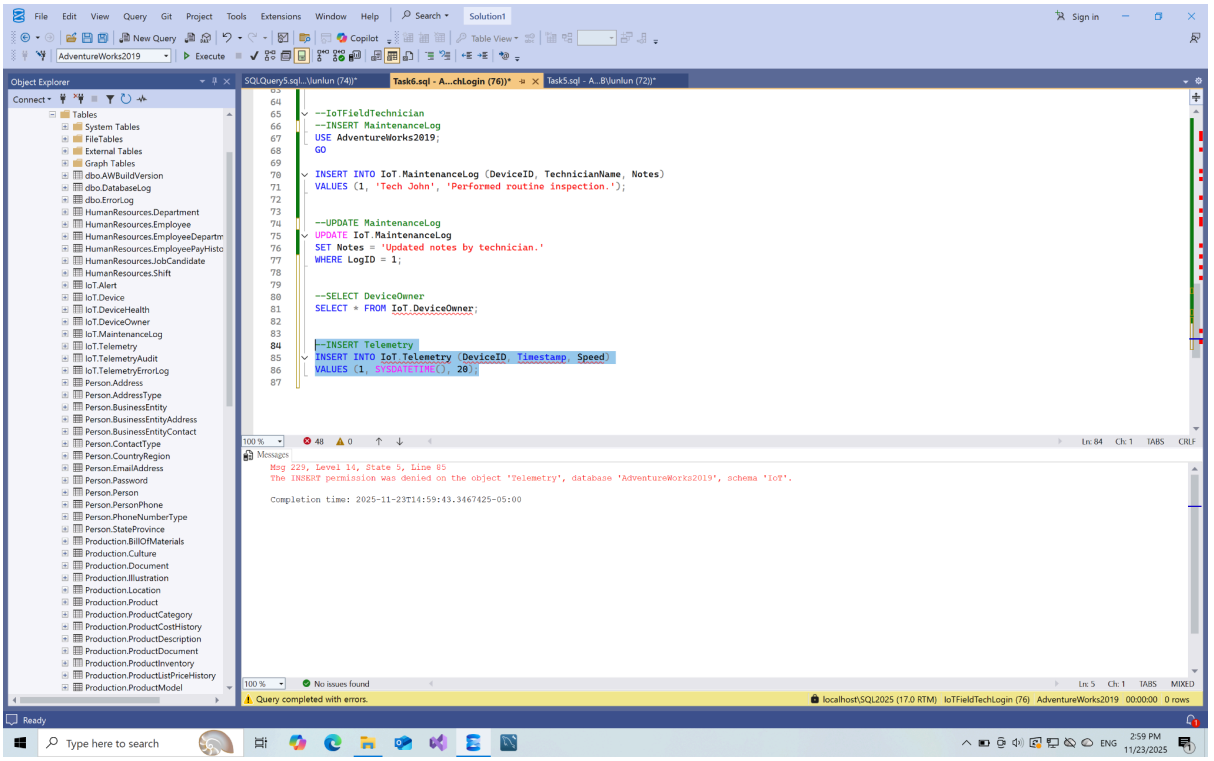
- UPDATE MaintenanceLog → Fails due to missing SELECT (expected)



- SELECT DeviceOwner → Denied

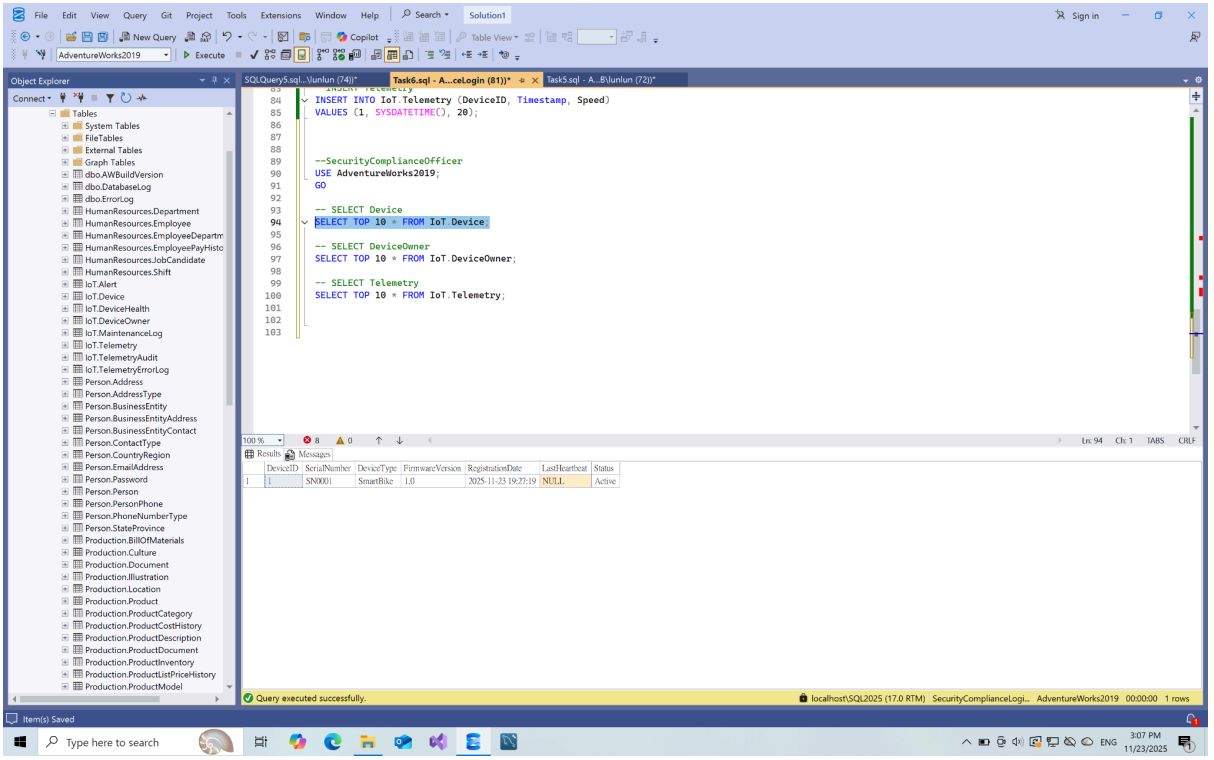


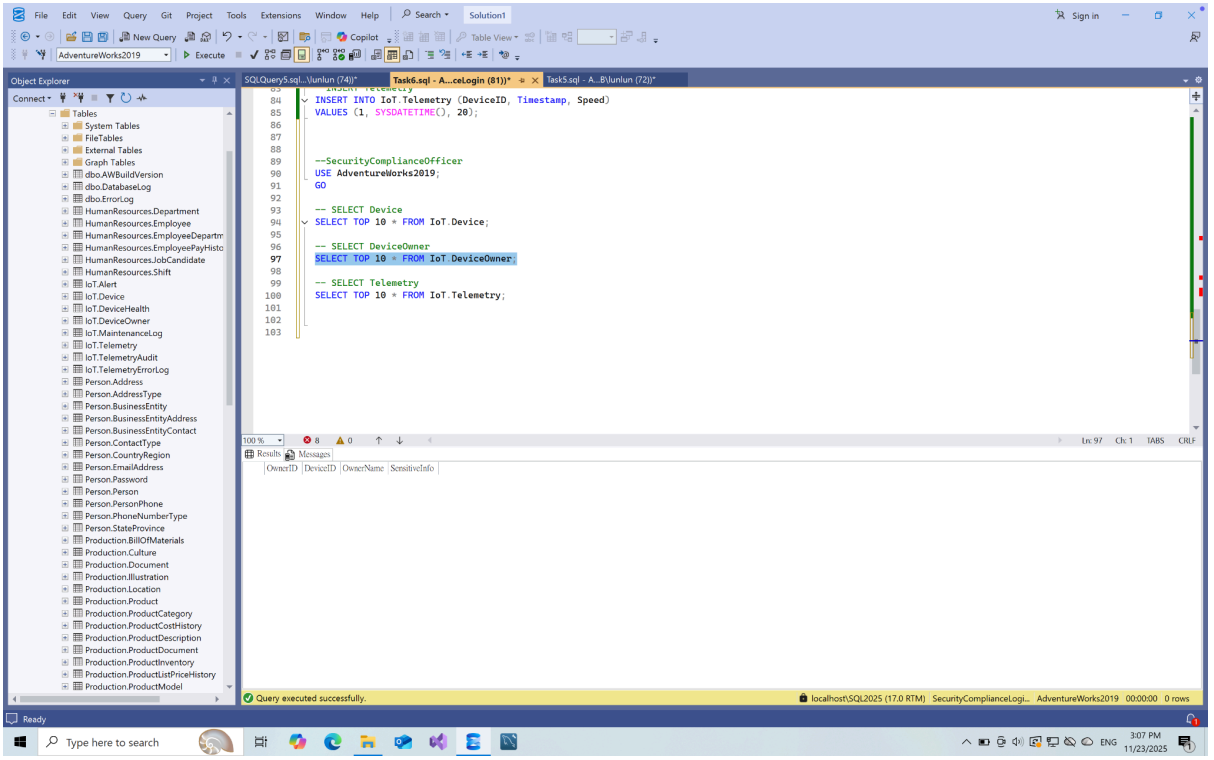
- INSERT Telemetry → Denied

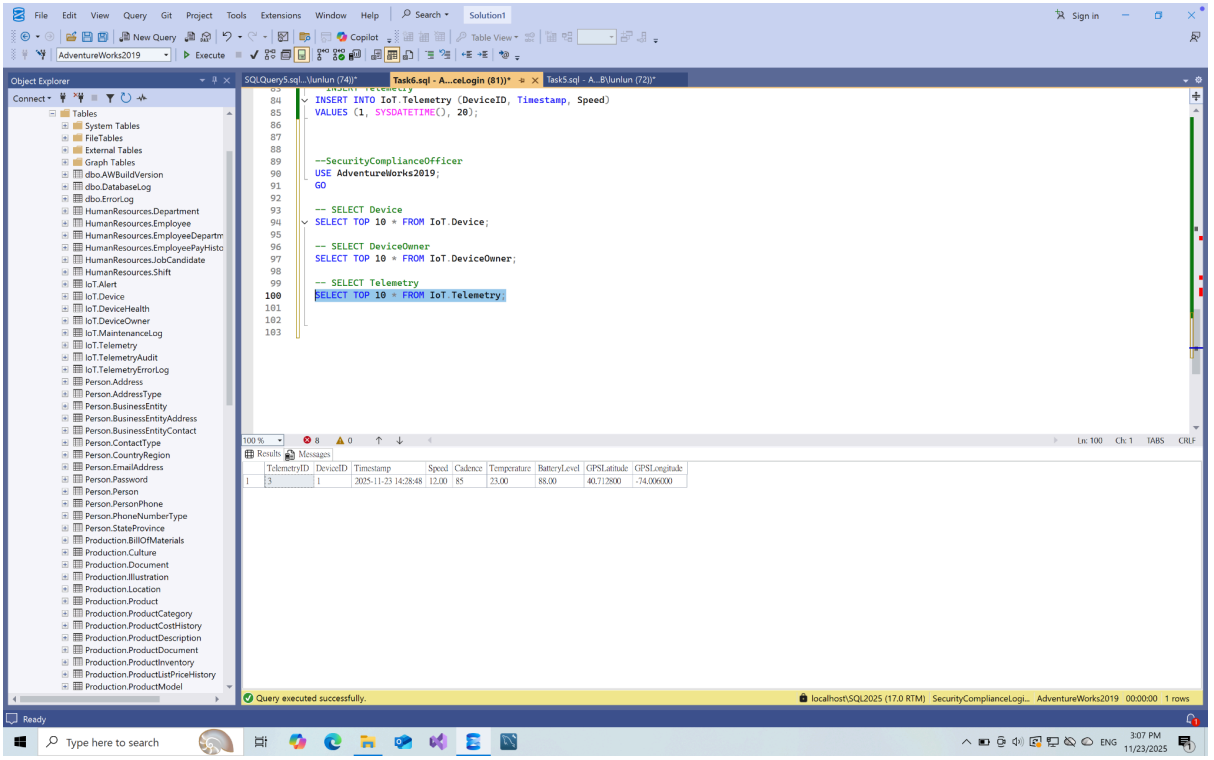


SecurityComplianceOfficer

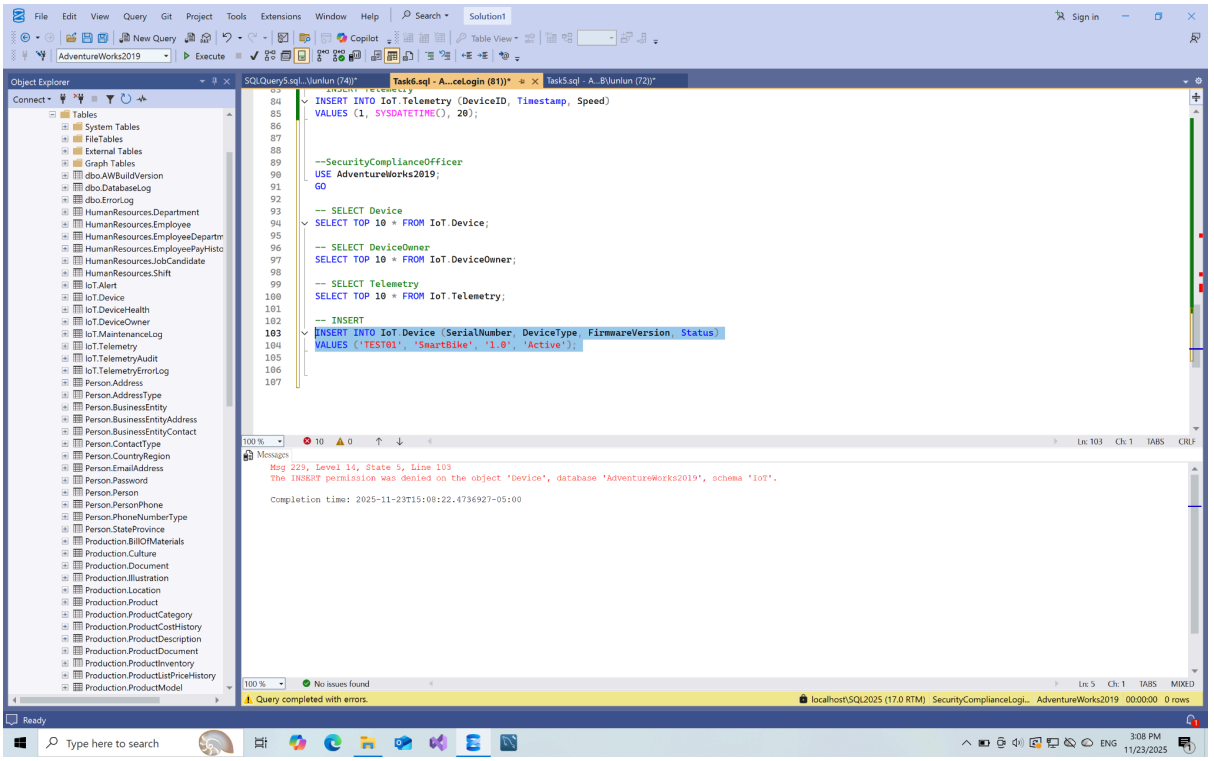
- All SELECT operations → Success

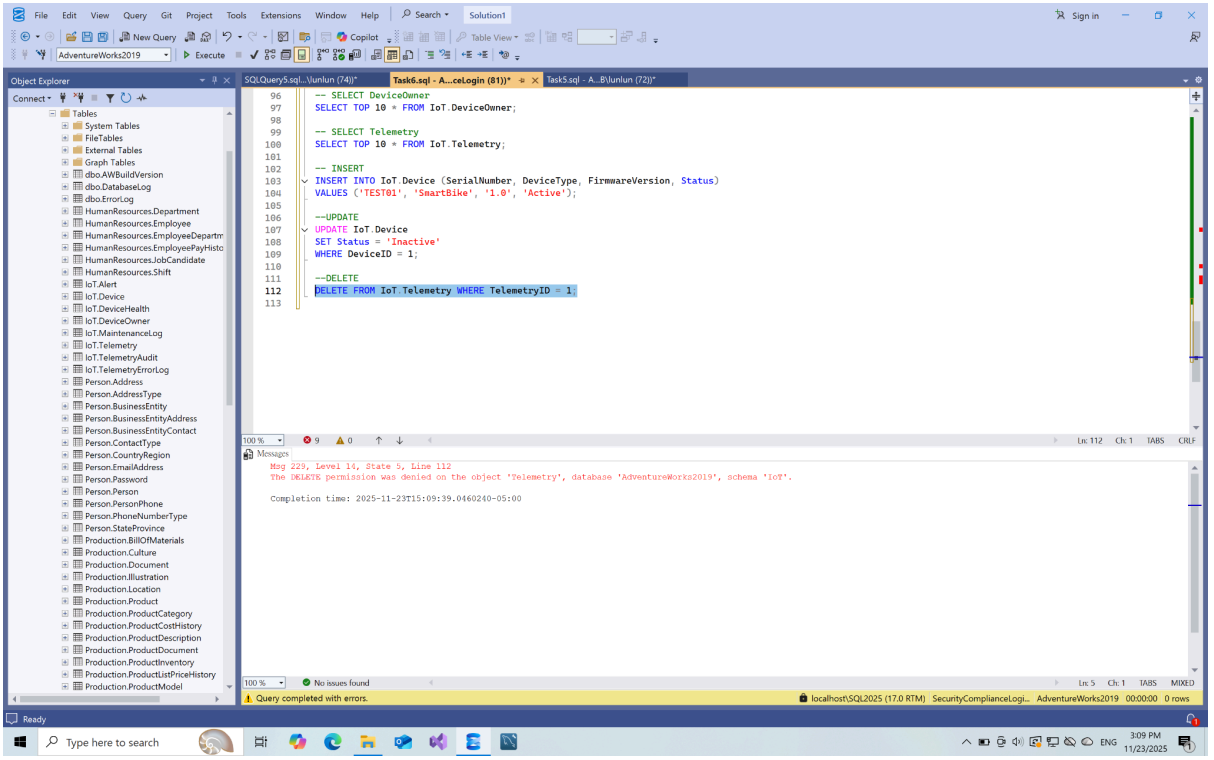

INSERT INTO IoT.Telemetry (DeviceID, Timestamp, Speed)
VALUES (1, SYSDATETIME(), 20);
--SecurityComplianceOfficer
USE AdventureWorks2019;
GO
-- SELECT Device
SELECT TOP 10 * FROM IoT.Device;
-- SELECT DeviceOwner
SELECT TOP 10 * FROM IoT.DeviceOwner;
-- SELECT Telemetry
SELECT TOP 10 * FROM IoT.Telemetry;
DeviceID SerialNumber DeviceType FirmwareVersion RegistrationDate LastHeartbeat Status
1 1 SN0001 SmartBike 1.0 2025-11-23 19:27:19 NULL Active
Query executed successfully.
Item(s) Saved

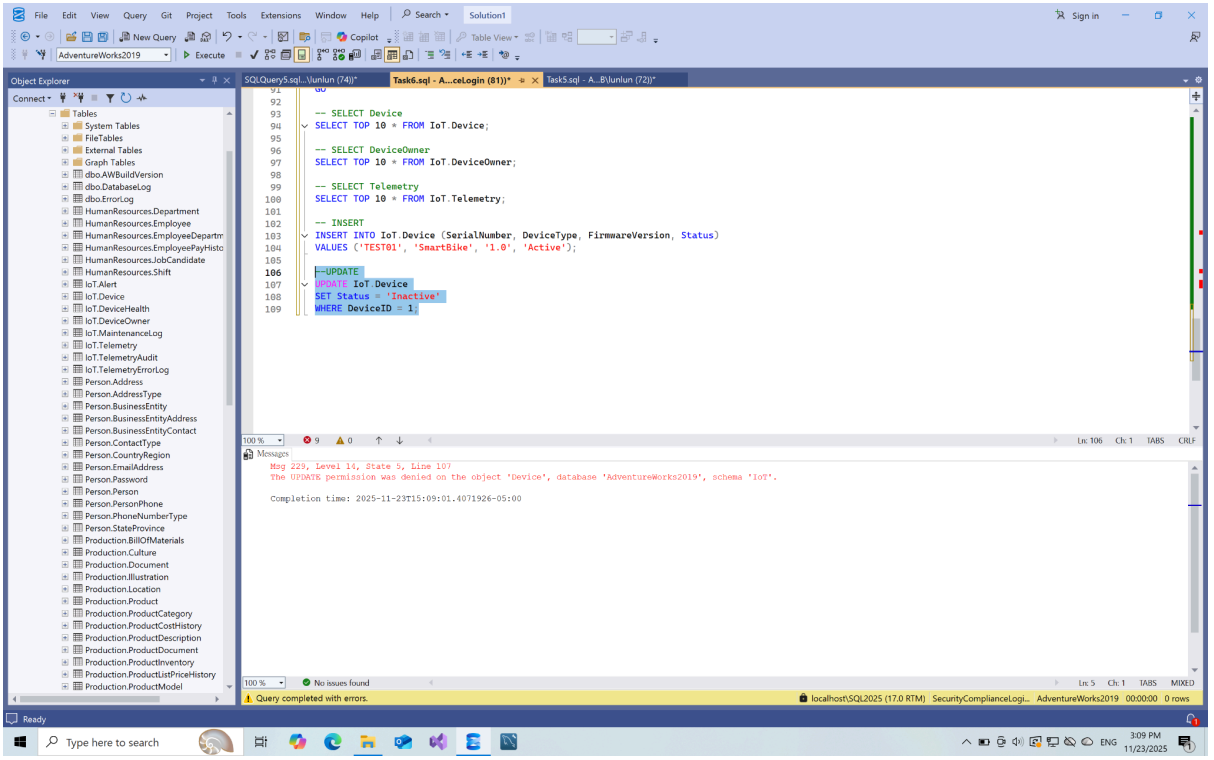

INSERT INTO IoT.Telemetry (DeviceID, Timestamp, Speed)
VALUES (1, SYSDATETIME(), 20);
--SecurityComplianceOfficer
USE AdventureWorks2019;
GO
-- SELECT Device
SELECT TOP 10 * FROM IoT.Device;
-- SELECT DeviceOwner
SELECT TOP 10 * FROM IoT.DeviceOwner;
-- SELECT Telemetry
SELECT TOP 10 * FROM IoT.Telemetry;
OwnerID DeviceID OwnerName SensitiveInfo
Query executed successfully.
Ready



- All DML operations (INSERT/UPDATE/DELETE) → Denied







(Reference: Task6.sql)

All results matched the expected behavior for each role.

7. Conclusion

This lab gave me hands-on experience with building an IoT-oriented SQL Server backend using real database engineering principles.
I learned how to design a schema that supports high-volume telemetry, how to write a transactional ingestion procedure that ensures data integrity, and how to handle concurrency issues that appear when multiple sessions access the same data.

I also gained practical experience with SQL Server's security model by creating customized roles with specific permissions and testing each one individually.
The combination of transactions, isolation levels, and RBAC helped me understand how real IoT systems protect data consistency and enforce privacy rules.

Overall, the lab strengthened my understanding of SQL Server in a realistic and meaningful way.