GROUP ASSIGNMENT

TECHNOLOGY PARK MALAYSIA

CT069-3-3-DBS

DATABASE SECURITY

APD3F2502CS(DA)

GROUP 12

HAND OUT DATE : 20th September 2025

HAND IN DATE : 26th June 2025

Name	TP Number
Lee Yi En	**TP072576**
Ooi Chong Ming	**TP072667**
Samantha Kok Jia Yin	**TP066133**
Sim Sau Yang	**TP065596**
Yeoh Zi Qing Bryan	**TP072717**

Table of Contents

Table of Figures

1.0 Introduction

Database security is a major issue about information systems, particularly when dealing with sensitive and confidential information like medical records. The information of patients and staff members in a healthcare system should not be exposed to unauthorized access, misuse, or possible breaches to guarantee privacy and regulatory integrity (Sabin, 2024).

The MedicalInfoSystem that is provided in this project, therefore, stores various classes of data with varying levels of sensitivity, such as patient information, staff information, and diagnosis information.

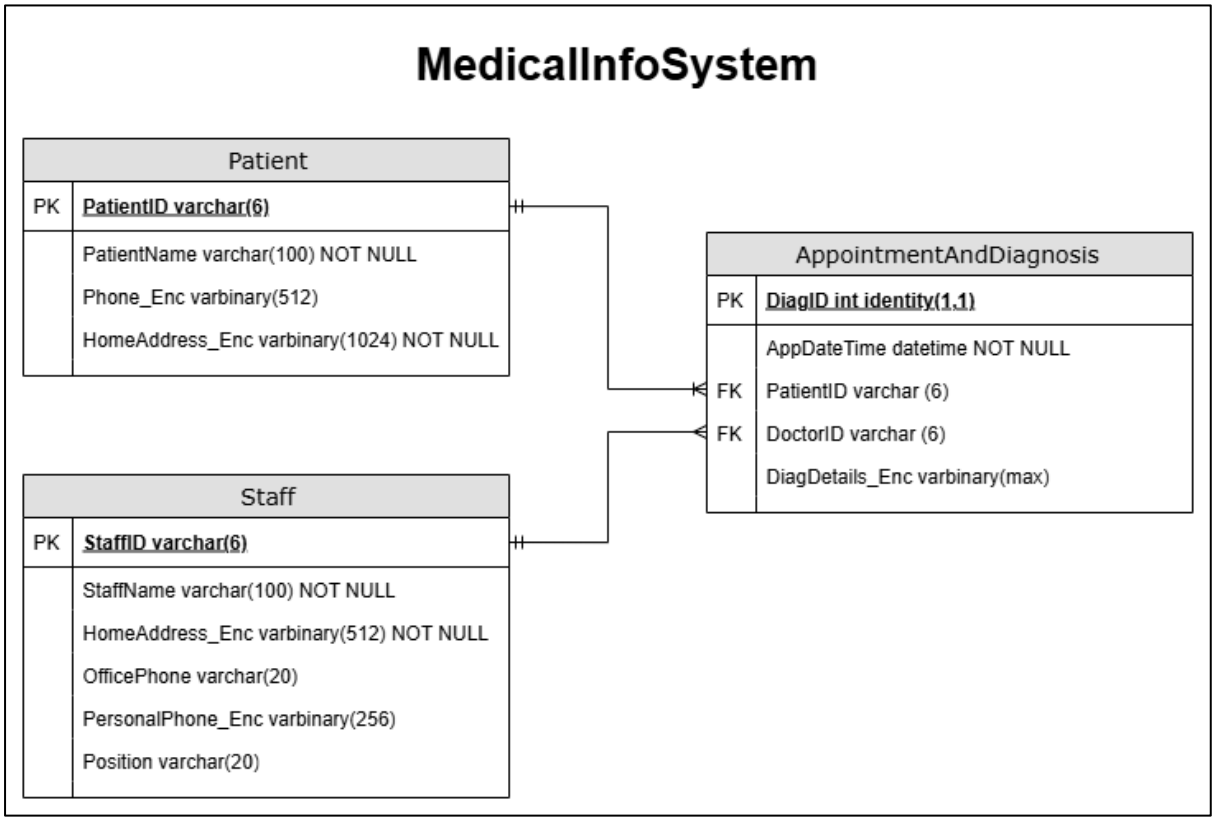


Figure 1: Medical Info System ERD

As illustrated in Figure 1, the MedicalInfoSystem consists of three principal structures: Patient, Staff, and AppointmentAndDiagnosis. Key demographic and contact data are saved in the Patient and Staff tables. In contrast, the AppointmentAndDiagnosis table serves as a hub, as patients are connected with medical personnel, and the most important characteristics of the consultation and diagnosis are entered in this table. This relational design ensures data integrity through the use of primary keys (PK) to uniquely identify a record and foreign keys (FK) to define relationships between entities, while the overall system must still meet recognized

security frameworks for confidentiality, integrity, and availability (International Organization for Standardization, 2022).

The system design emphasizes the need to protect data on various levels. As an example, the personal data of patients, employee information, and diagnostic records should be safeguarded against unauthorized access to avoid privacy breaches. In addition, the system handles sensitive, highly sensitive, and extremely sensitive medical information. Therefore, there is a need for proper database security controls.

Thus, to overcome these difficulties, this report examines various security levels, including data protection and obfuscation, encryption, masking, backup, permission management, role-based and object-level access, and auditing processes. All these measures will help maintain the confidentiality, integrity, and availability of MedicalInfoSystem, and ensure compliance with healthcare standards without compromising the trust between patients, staff, and the institution (Pietervanhove, 2025).

2.0 Data Protection

To design a safe medical information database system, data protection is the first important aspect to be considered. It defines how information in the medical system is kept safe from accidental exposure, misuse, and loss throughout its life cycle. This section explains the controls that keep sensitive values unreadable to unauthorized parties, limit on-screen exposure to only what is necessary, and maintain reliable recovery after failures. The approach applies to the Staff, Patient, and Appointment & Diagnosis tables, and database schema, including:

- **Encryption:** It serves as the primary database safeguard that converts sensitive values into unreadable ciphertext before storage (Ikram, 2024). There are two cryptographic modes applied, including **symmetric encryption**, which utilizes a single secret key for both lock and unlock, that protects routine personally identifiable information (PII) that is sensitive, and also **asymmetric encryption,** which uses a public key for locking and a different private key for unlocking, that provides stronger separation of duties for highly sensitive medical narratives (Thilakanathan et al., 2016).
- **Masking**: It functions as an obfuscation layer at query time that replaces sensitive fields with unknown when the viewer does not have a legitimate need to know (*What Is Data Masking? - Static and Dynamic Data Masking Explained - AWS*, n.d.). In this system, self-view, patient, and staff basic contexts reveal clear values. In contrast, cross-record browsing and non-clinical contexts display masked outputs for private identifiers and medical text to preserve the operational usability without exposing unnecessary plaintext (Satori Cyber, 2022).
- **Backup & Recovery:** It is a resilience control that preserves restorable copies of encrypted data and associated keys, ensuring that information remains recoverable after errors or failures, with an emphasis on both traceability and recoverability (*Summary of the HIPAA Security Rule*, n.d.; Team, 2022).

Overall, data protection encompasses encryption, masking, and backup and recovery, which work together to ensure confidentiality, integrity, and availability without compromising day-to-day usability (Nail, 2025).

2.1 Data Protection & Storage Posture (Requirements 1 & 3)

Before determining the action solution of requirements, the database schema structure needs to be examined to identify the data security level, understand how to address it, and incorporate it into the solution.

Table	Column	Classification	Protection Mechanism	Justification
Staff	StaffID	Internal	Shown in views	Staff IDs are required for authentication and linking records, not secret but restricted within the system.
	StaffName	Internal	Visible in views	Names are visible to all authenticated users to support collaboration.
	HomeAddress	Private	**Symmetric key encryption & Masking in views**	Highly sensitive personal data, which needs encryption to protect at rest, masking prevents unauthorized exposure.
	OfficePhone	Internal	Visible in views	Shared for work contact purposes, and safe for internal use.
	PersonalPhone	Private	**Symmetric key encryption & Masking in views**	Sensitive contact information is encrypted at rest and masked for unauthorized users.
	Position	Internal	Shown in views	Needed for role recognition and has low sensitivity.
Patient	PatientID	Internal	Shown in views	It is simply an identifier for linking records, neither directly sensitive nor public.
	PatientName	Internal	Shown in views	Names must be seen by doctors or nurses, but considered internal.
	Phone	Private	**Symmetric key encryption & Masking in views**	Sensitive contact info is encrypted and masked when needed.
	HomeAddress	Private	**Symmetric key encryption & Masking in views**	Highly sensitive information is to be protected similarly to staff addresses.

Apmnt & Diag	DiagID	Internal	Not encrypted	ID is used as a reference key and has no sensitive meaning itself.
	AppDateTime	Confidential	Visible in views	Important medical scheduling data is accessible only to relevant roles.
	PatientID	Internal	Linked to Patient	This is a foreign key with low sensitivity, but internal only.
	DoctorID	Internal	Linked to Staff	This is a foreign key with low sensitivity, but internal only.
	DiagDetails	Private	**Asymmetric key encryption & Masking in views**	Medical diagnoses require the strongest protection and are encrypted asymmetrically and masked for unauthorized roles.

Table 1: Data Classification Matrix

Accordingly, every attribute in Staff, Patient, and Appointment & Diagnosis is classified as **Internal**, **Confidential**, or **Private** based on sensitivity and operational need. PII such as HomeAddress and Personal Phone is labeled Private and designed to be protected with symmetric encryption and masking in views, while DiagDetails is also Private and safeguarded with asymmetric encryption plus masking for unauthorized viewers. Scheduling metadata, such as AppDateTime, is Confidential and visible only where clinically necessary. Identifiers and linkage fields, such as StaffID, PatientID, and DoctorID, as well as directory-safe fields like StaffName and OfficePhone, are Internal and displayed in views to support routine operations. With this schema design, **Requirements 1 and 3 can be achieved** to ensure confidentiality and sufficient protection from exposure. Plus, to meet the assignment requirements, **these protection mechanisms are interrelated with permission management.** However, this section will focus solely on the data protection techniques themselves, while the authorization is documented separately under Permission Management.

2.1.1 Staff & Patient PII & Diagnosis Protection

Since the personal details for both staff and patients include phone numbers and home addresses, which are highly sensitive, and the diagnosis records contain extremely private clinical information, these values must be encrypted to prevent casual exposure, insider misuse, or leakage through stolen files and backups. To achieve confidentiality and provide suitable protection as required by the assignment (**Requirements 1 and 3**), the solution applies two cryptographic approaches. **Symmetric encryption (AES-256)** is used for routine personally identifiable information (PII), such as staff and patient contact details, because it is efficient and secure for frequent read and write operations. **Asymmetric encryption (certificate-based)** is reserved for diagnostic text, as this data carries the highest sensitivity and requires stronger separation of duties, which has a public key to encrypt the diagnosis, but only the private key within the certificate can decrypt it. Overall, these methods ensure that all PII and diagnosis details are stored as ciphertext at rest, visible in plain text only through controlled decryption when the user has a legitimate need.

```
IF NOT EXISTS (SELECT 1 FROM sys.symmetric_keys WHERE name='##MS_DatabaseMasterKey##')
    CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'Str0ng!DBMK_P@ss_2025';
GO

IF NOT EXISTS (SELECT 1 FROM sys.certificates WHERE name='Cert_StaffKEK')
    CREATE CERTIFICATE Cert_StaffKEK   WITH SUBJECT='Protector for Staff symmetric key';
IF NOT EXISTS (SELECT 1 FROM sys.certificates WHERE name='Cert_PatientKEK')
    CREATE CERTIFICATE Cert_PatientKEK WITH SUBJECT='Protector for Patient symmetric key';
IF NOT EXISTS (SELECT 1 FROM sys.certificates WHERE name='Cert_Diag')
    CREATE CERTIFICATE Cert_Diag       WITH SUBJECT='Asymmetric cert for Diagnosis details';
GO
```

Figure 2: Create Certificates

	What	name
1	Certs	Cert_Diag
2	Certs	Cert_PatientKEK
3	Certs	Cert_StaffKEK

Figure 3: Created Certificates

To implement the solution, first, during the database schema construction, the cryptographic foundation is established to enable encryption and decryption within the database. It begins by creating a **Database Master Key (DMK)**, which is the root protector of all cryptographic materials in the database. The DMK ensures that any private keys or certificates stored within the database can be securely encrypted and later opened when needed. After establishing this root, **three certificates** are created, and each of them has its own use. The first certificate is to **protect the symmetric key for staff** data, another to **protect the symmetric key for patient**

data, and a third to **provide an asymmetric key pair for diagnosis details** directly. These certificates act as key-encrypting keys, which means they do not encrypt the data themselves, but they safeguard the symmetric keys.

```
IF NOT EXISTS (SELECT 1 FROM sys.symmetric_keys WHERE name='SK_StaffPII')
    CREATE SYMMETRIC KEY SK_StaffPII
        WITH ALGORITHM = AES_256
        ENCRYPTION BY CERTIFICATE Cert_StaffKEK;
IF NOT EXISTS (SELECT 1 FROM sys.symmetric_keys WHERE name='SK_PatientPII')
    CREATE SYMMETRIC KEY SK_PatientPII
        WITH ALGORITHM = AES_256
        ENCRYPTION BY CERTIFICATE Cert_PatientKEK;
GO
```

Figure 4: Create Symmetric Keys

	What	name
1	SymKeys	SK_PatientPII
2	SymKeys	SK_StaffPII

Figure 5: Created Symmetric Keys

Finally, two **symmetric keys** (SK_StaffPII and SK_PatientPII) are created, then each is protected by its corresponding certificate. These symmetric keys are the working keys used to **encrypt and decrypt** sensitive values like **phone numbers and home addresses**. Finally, a clear layered hierarchy is established, ranging from the master key to certificates and symmetric and asymmetric keys, which ensures that sensitive attributes can be encrypted securely and decrypted only under controlled conditions, and protected from exposure even if the database files are copied.

```
IF OBJECT_ID('SecureData.Staff','U') IS NULL
BEGIN
    CREATE TABLE SecureData.Staff (
        StaffID            VARCHAR(6)   NOT NULL CONSTRAINT PK_Staff PRIMARY KEY,
        StaffName          VARCHAR(100) NOT NULL,
        HomeAddress_Enc    VARBINARY(512)  NOT NULL, -- SYMMETRIC (SK_StaffPII)
        OfficePhone        VARCHAR(20)   NULL,       -- plaintext (directory)
        PersonalPhone_Enc  VARBINARY(256)  NULL,     -- SYMMETRIC (SK_StaffPII)
        Position           VARCHAR(20)   NULL        -- Doctor / Nurse
    );
END;
```

Figure 6: Create Staff Table

With the keys on hand, the staff table is designed with a mix of encrypted and plaintext fields to balance confidentiality with usability. The unique identifier (StaffID) and name (StaffName) remain in plaintext for authentication and collaboration, while OfficePhone and Position are also left readable since they are directory-safe and needed for daily operations. In contrast, highly sensitive personal details such as HomeAddress and PersonalPhone are stored in **VARBINARY** columns (HomeAddress_Enc and PersonalPhone_Enc) that hold ciphertext

encrypted with the **symmetric key SK_StaffPII**. This ensures that private information is never written in plain text to the database.

```
IF OBJECT_ID('SecureData.Patient','U') IS NULL
BEGIN
    CREATE TABLE SecureData.Patient (
        PatientID       VARCHAR(6)   NOT NULL CONSTRAINT PK_Patient PRIMARY KEY,
        PatientName     VARCHAR(100) NOT NULL,
        Phone_Enc       VARBINARY(512)   NULL,       -- SYMMETRIC (SK_PatientPII)
        HomeAddress_Enc VARBINARY(1024)  NOT NULL    -- SYMMETRIC (SK_PatientPII)
    );
END;

IF OBJECT_ID('SecureData.AppointmentAndDiagnosis','U') IS NULL
```

Figure 7: Create Patient Table

The same protection design is applied to the patient table, where the identifier (PatientID) and name (PatientName) are stored in plaintext for record linking and operational use, while the sensitive columns Phone_Enc and HomeAddress_Enc are defined as **VARBINARY** fields. These columns are encrypted using the **symmetric key SK_PatientPII**, protected by its certificate, ensuring that patient contact information and residential address are secured as ciphertext in the database.

```
IF OBJECT_ID('SecureData.AppointmentAndDiagnosis','U') IS NULL
BEGIN
    CREATE TABLE SecureData.AppointmentAndDiagnosis (
        DiagID          INT IDENTITY(1,1) NOT NULL CONSTRAINT PK_AppointmentAndDiagnosis PRIMARY KEY,
        AppDateTime     DATETIME    NOT NULL,
        PatientID       VARCHAR(6)  NOT NULL,
        DoctorID        VARCHAR(6)  NOT NULL,
        DiagDetails_Enc VARBINARY(MAX) NULL          -- ASYMMETRIC (Cert_Diag)
        -- (FKs omitted intentionally; RLS will be added later)
    );
END;
GO
```

Figure 8: Create AppointmentAndDiagnosis Table

On the other hand, the diagnosis information is protected through a stronger approach because of its extremely sensitive nature. In the AppointmentAndDiagnosis table, operational attributes such as DiagID, AppDateTime, PatientID, and DoctorID are stored in plaintext since they are required for scheduling, record linking, and workflow management. However, the medical notes column DiagDetails_Enc is defined as a **VARBINARY(MAX)** field so that it stores only ciphertext. This column is encrypted using **asymmetric encryption with the certificate Cert_Diag**, where the public key encrypts the diagnosis details and only the private key within the certificate can decrypt them. This design provides stronger separation of duties by ensuring that diagnosis records cannot be easily decrypted and can only be revealed through controlled procedures for authorized roles, offering the highest level of protection among all sensitive data in the system.

```
-- STAFF (symmetric)
OPEN SYMMETRIC KEY SK_StaffPII DECRYPTION BY CERTIFICATE Cert_StaffKEK;

IF NOT EXISTS (SELECT 1 FROM SecureData.Staff WHERE StaffID='D001')
BEGIN
    INSERT INTO SecureData.Staff
        (StaffID, StaffName, HomeAddress_Enc, OfficePhone, PersonalPhone_Enc, Position)
    VALUES
        ('D001','Dr. Ooi',
            EncryptByKey(Key_GUID('SK_StaffPII'), CONVERT(VARBINARY(4000), '12 Jalan Bukit, KL')),
            '03-11112222',
            EncryptByKey(Key_GUID('SK_StaffPII'), CONVERT(VARBINARY(4000), '012-3456789')),
            'Doctor'),
```

Figure 9: Insert into Staff by SuperAdmin

```
-- PATIENT (symmetric)
OPEN SYMMETRIC KEY SK_PatientPII DECRYPTION BY CERTIFICATE Cert_PatientKEK;

IF NOT EXISTS (SELECT 1 FROM SecureData.Patient WHERE PatientID='P001')
BEGIN
    INSERT INTO SecureData.Patient
        (PatientID, PatientName, Phone_Enc, HomeAddress_Enc)
    VALUES
        ('P001','Ali Musa',
            EncryptByKey(Key_GUID('SK_PatientPII'), CONVERT(VARBINARY(4000), '012-3456789')),
            EncryptByKey(Key_GUID('SK_PatientPII'), CONVERT(VARBINARY(4000), '22, Jalan Bukit, KL')));
END;
```

Figure 10: Insert into Patient by SuperAdmin

After that, the SuperAdmin tries to insert user details into the tables. It shows how staff and patient PII are protected with symmetric encryption before being stored. Each section first opens the relevant symmetric key (SK_StaffPII or SK_PatientPII) using its certificate, then applies EncryptByKey to convert sensitive fields such as home address and phone into ciphertext saved in VARBINARY columns.

	StaffID	StaffName	HomeAddress_Enc	OfficePhone	PersonalPhone_Enc	Position
1	D001	Dr. Ooi	0x007C31C56EBEC549B540D8E84200651A020000009EB2806...	03-11112222	0x007C31C56EBEC549B540D8E84200651A02000000E26029E...	Doctor
2	D002	Dr. Bryan	0x007C31C56EBEC549B540D8E84200651A02000000B0217D...	03-11113333	0x007C31C56EBEC549B540D8E84200651A0200000066E6153...	Doctor
3	D003	Dr. D003	0x007C31C56EBEC549B540D8E84200651A020000029C2978...	03-110003	0x007C31C56EBEC549B540D8E84200651A02000000EB4F58A...	Doctor
4	D004	Dr. D004	0x007C31C56EBEC549B540D8E84200651A02000000A7DC7F...	03-110004	0x007C31C56EBEC549B540D8E84200651A020000008E53F94...	Doctor
5	D005	Dr. D005	0x007C31C56EBEC549B540D8E84200651A02000000960E70C...	03-110005	0x007C31C56EBEC549B540D8E84200651A020000006C23C0F...	Doctor

Figure 11: Staff Table (by SuperAdmin)

	PatientID	PatientName	Phone_Enc	HomeAddress_Enc
1	P001	Ali Musa	0x0050740392DABE41A7E5DBDD6AFCD92602000000D30BDF...	0x0050740392DABE41A7E5DBDD6AFCD9260200000071FF9FE...
2	P002	Patient P002	0x0050740392DABE41A7E5DBDD6AFCD92602000000EC8CAC...	0x0050740392DABE41A7E5DBDD6AFCD92602000000D7419E6...
3	P003	Patient P003	0x0050740392DABE41A7E5DBDD6AFCD926020000003F737BD...	0x0050740392DABE41A7E5DBDD6AFCD926020000002A0B328...
4	P004	Patient P004	0x0050740392DABE41A7E5DBDD6AFCD92602000000A3C7B9E...	0x0050740392DABE41A7E5DBDD6AFCD92602000000ED8932E...
5	P005	Patient P005	0x0050740392DABE41A7E5DBDD6AFCD92602000000A93DF4F...	0x0050740392DABE41A7E5DBDD6AFCD92602000000A4C5A64...

Figure 12: Patient Table (by SuperAdmin)

As a result, personal details are never written in plaintext, while identifiers and names remain readable for operational use. This ensures that even if database files or backups are accessed, only encrypted values are exposed.

	DiagID	AppDateTime	PatientID	DoctorID	DiagDetails_Enc
1	1	2025-09-20 12:20:34.120	P001	D001	0x500160730C3DC900840F45A3C83E7BF4244B1C83131DB4F...

Figure 13: AppointmentAndDiagnosis Table (by SuperAdmin)

This record from the AppointmentAndDiagnosis table also shows that medical information is protected at rest. The sensitive medical notes are stored in the DiagDetails_Enc column as ciphertext. The value appears as a long hexadecimal string, which is the output of encrypting the diagnosis text with the asymmetric certificate **Cert_Diag**.

By doing so, the diagnosis content cannot be read directly from the table or backups. It can only be revealed later through controlled decryption functions in views or procedures for authorized users. This ensures that extremely sensitive clinical details remain protected while the operational parts of the appointment record remain fully usable.

2.2 View over protected data (Requirements 8 & 11 & 13)

After the database structure is defined, the first problem with data protection is to ensure that authorized users can view necessary details in plain text while the same data remains securely stored as ciphertext. **Requirement 8** specifies that staff must be able to see their own records in full, and **Requirement 11** applies the same rule to patients. In addition, **Requirement 13** states that clinicians need to view patient names and phone numbers clearly for operational purposes.

To achieve this, the system decrypts sensitive columns only at query time using the appropriate symmetric key, ensuring that staff, patients, and clinicians can see the required details in plain text. At the same time, all protected fields remain encrypted at rest.

2.2.1 Staff Decrypt Own Information for Display

```
WHEN USER_NAME() = s.StaffID THEN
    CONVERT(varchar(200),
        DecryptByKeyAutoCert(CERT_ID('Cert_StaffKEK'), NULL, s.HomeAddress_Enc)
    )
```

Figure 14: Decrypt HomeAddress (by Staff)

```
WHEN USER_NAME() = s.StaffID THEN
    CONVERT(varchar(50),
        DecryptByKeyAutoCert(CERT_ID('Cert_StaffKEK'), NULL, s.PersonalPhone_Enc)
    )
```

Figure 15: Decrypt PersonalPhone (by Staff)

When a staff member views the self-detail's view, the pre-set query checks USER_NAME() against the row's StaffID. If it matches, the encrypted columns are decrypted inline using the certificate-assisted helper, for example, **DecryptByKeyAutoCert (CERT_ID ('Cert_StaffKEK'), NULL, s.HomeAddress_Enc),** and the same for s.PersonalPhone_Enc, then cast back to readable text. Non-encrypted fields, such as StaffName, OfficePhone, and Position, are selected as usual. If the viewer is not the owner, the decrypted branch is not taken, and those sensitive columns are returned as masked placeholders (discuss later).

	StaffID	StaffName	OfficePhone	HomeAddress	PersonalPhone	Position
1	D001	Dr. Ooi	03-11112222	12 Jalan Bukit, KL	012-3456789	Doctor

Figure 16: Staff Personal Details (by Doctor)

	StaffID	StaffName	OfficePhone	HomeAddress	PersonalPhone	Position
1	N001	Nurse Britney	03-22224444	55 Jalan Klang Lama, KL	017-8877665	Nurse

Figure 17: Staff Personal Details (by Nurse)

The results show that when staff access their own records, the encrypted fields for home address and personal phone are correctly decrypted and displayed in plaintext, while other non-sensitive fields, such as name, office phone, and position, are shown directly. This confirms that the staff self-view mechanism works as intended **(Requirement 8)** while ensuring that the data is stored as ciphertext at rest, thereby maintaining confidentiality.

2.2.2 Patient Decrypt Own Information for Display

```
CONVERT(varchar(50),
    DecryptByKeyAutoCert(CERT_ID('Cert_PatientKEK'), NULL, Phone_Enc)
) AS Phone,
CONVERT(varchar(4000),
    DecryptByKeyAutoCert(CERT_ID('Cert_PatientKEK'), NULL, HomeAddress_Enc)
) AS HomeAddress
```

Figure 18: Decrypt Phone and HomeAddress (by Patient)

Similar to the staff self-view, the patient query uses the **certificate-protected symmetric key SK_PatientPII** to decrypt the encrypted columns Phone_Enc and HomeAddress_Enc, converting them back into readable text at query time. The purpose is to let patients see their own full details in plaintext.

	PatientID	PatientName	Phone	HomeAddress
1	P001	Ali Musa	012-3456789	22, Jalan Bukit, KL

Figure 19: Patient Personal Details (by Patient)

The result shows that the patient record is displayed with both the phone number and home address in clear, readable form when they are decrypted using the certificate-protected symmetric key at runtime. The outcome justifies that **Requirement 11** is met, as patients can see their own full details, while the confidentiality requirement is also upheld since the sensitive information remains encrypted at rest.

2.2.3 Staff Decrypt Patient Information for Display

```
CONVERT(varchar(50),
    DecryptByKeyAutoCert(CERT_ID('Cert_PatientKEK'), NULL, p.Phone_Enc)
) AS Phone
```

Figure 20: Decrypt Patient Phone (by Staff)

This view expression allows authorized staff to see a patient's phone number in plain text, satisfying **Requirement 13**. It uses the certificate-assisted helper **DecryptByKeyAutoCert(CERT_ID('Cert_PatientKEK'), NULL, p.Phone_Enc)** to open the symmetric key that protects patient PII and decrypt the Phone_Enc ciphertext to plain text.

	PatientID	PatientName	Phone
1	P001	Siti NMDNMDNMDNMDNMD	012-01010101
2	P002	Patient P002	012-0000002
3	P003	Patient P003	012-0000003
4	P004	Patient P004	012-0000004
5	P005	Patient P005	012-0000005

Figure 21: Patient Details (by Staff)

The result displays a list of patients, with both names and phone numbers clearly visible. This confirms that the decryption logic for Phone_Enc worked correctly, allowing staff to see the patient contact numbers alongside their names. This fulfils **Requirement 13**, which allows staff to view patient names and phones, while the confidentiality requirement is still preserved because the actual database columns remain.

2.3 Safe updates to keep PII encrypted (Requirements 9 & 12 & 14)

Moving to the next problem, let authorized users edit contact details without ever writing plaintext back to disk. **Requirement 9** requires staff to update and verify their own details in full, **Requirement 12** requires patients to do the same for their records, and **Requirement 14** allows only nurses to update a patient's name and phone. These actions must not weaken protection for sensitive fields such as phone numbers and home addresses.

Since the database stores these attributes as ciphertext, every update path must **re-encrypt on write**. Incoming values must be first converted to binary and then sealed with the correct symmetric key before being persisted, so the sensitive data always contains encrypted bytes. Procedures or triggers handle this consistently, ensuring that only the ciphertext is stored, and non-sensitive fields remain plaintext as designed.

2.3.1 Staff Encrypt Own Information for Update

```
OPEN SYMMETRIC KEY SK_StaffPII DECRYPTION BY CERTIFICATE Cert_StaffKEK;

UPDATE s
SET
    s.StaffName    = @StaffName,
    s.OfficePhone  = @OfficePhone,
    s.HomeAddress_Enc =
        CASE
            WHEN @HomeAddress IS NULL OR @HomeAddress = '**********'
                THEN s.HomeAddress_Enc
            ELSE EncryptByKey(Key_GUID('SK_StaffPII'),
                    CONVERT(VARBINARY(4000), @HomeAddress))
        END,
    s.PersonalPhone_Enc =
        CASE
            WHEN @PersonalPhone IS NULL OR @PersonalPhone = '**********'
                THEN s.PersonalPhone_Enc
            ELSE EncryptByKey(Key_GUID('SK_StaffPII'),
                    CONVERT(VARBINARY(4000), @PersonalPhone))
        END
FROM SecureData.Staff AS s
WHERE s.StaffID = @me;

CLOSE SYMMETRIC KEY SK_StaffPII;
```

Figure 22: Update Self Details (by Staff)

The staff self-update procedure opens the symmetric key **SK_StaffPII** and updates the staff record while ensuring sensitive fields remain encrypted. Plaintext attributes such as name and office phone are written directly, but for HomeAddress_Enc and PersonalPhone_Enc, the routine checks if the input is null or a masked placeholder. If so, the existing ciphertext is preserved. Otherwise, the new value is re-encrypted with EncryptByKey. This design ensures that only the ciphertext is stored on disk, prevents accidental overwrites with placeholder values, and allows staff to securely update and verify their own details only while maintaining confidentiality as required.

```
UPDATE SecureData.vwStaff
    SET StaffName = 'Dr. OoiOoiOoiOoi' WHERE StaffID = 'D001'
```

Figure 23: Update Staff Personal Information (by Doctor)

	StaffID	StaffName	OfficePhone	HomeAddress	PersonalPhone	Position
1	D001	Dr. OoiOoiOoiOoi	03-11112222	12 Jalan Bukit, KL	012-3456789	Doctor

Figure 24: Staff Personal Details (by Doctor)

	StaffID	StaffName	HomeAddress_Enc	OfficePhone	PersonalPhone_Enc	Position
1	D001	Dr. OoiOoiOoiOoi	0x007C31C56EBEC549B540D8E84200651A0200...	03-11112222	0x007C31C56EBEC549B540D8E84200651A0200000...	Doctor

Figure 25: Staff Personal Details (by SuperAdmin)

When the doctor updates his details, the system ensures that the sensitive fields remain encrypted in the database to fulfil Requirement 9, while still displaying the decrypted values to him through the self-view, as in Requirement 8. As seen in the results by SuperAdmin, the doctor's name and other plaintext fields are updated directly. Still, the home address and personal phone are stored in ciphertext (HomeAddress_Enc, PersonalPhone_Enc) at the database level.

2.3.2 Patient Encrypt Own Information for Update

```
OPEN SYMMETRIC KEY SK_PatientPII DECRYPTION BY CERTIFICATE Cert_PatientKEK;

UPDATE p
   SET PatientName     = @PatientName,
       Phone_Enc       = CASE WHEN @Phone IS NULL THEN NULL
                              ELSE EncryptByKey(Key_GUID('SK_PatientPII'),
                                   CONVERT(VARBINARY(4000), @Phone)) END,
       HomeAddress_Enc = CASE WHEN @HomeAddress IS NULL THEN HomeAddress_Enc
                              ELSE EncryptByKey(Key_GUID('SK_PatientPII'),
                                   CONVERT(VARBINARY(4000), @HomeAddress)) END
  FROM SecureData.Patient AS p
 WHERE p.PatientID = @me;

CLOSE SYMMETRIC KEY SK_PatientPII;
```

Figure 26: Update Self Details (by Patient)

After that, the update routine for patients works the same way as the staff update. It opens the symmetric key **SK_PatientPII** and re-encrypts sensitive fields (Phone_Enc, HomeAddress_Enc) whenever new values are provided, while leaving existing ciphertext intact if the input is null. Plaintext fields such as PatientName are updated directly to fulfill Requirement 12.

```
EXEC SecureData.usp_Patient_Self_Update
     @PatientName='Ali KNNKNNKNNKNNKNN', @Phone='012-01010101', @HomeAddress='22, Jalan Habis, KL';
```

Figure 27: Update Patient Personal Information (by Patient)

	PatientID	PatientName	Phone	HomeAddress
1	P001	Ali KNNKNNKNNKNNKNN	012-01010101	22, Jalan Habis, KL

Figure 28: Patient Personal Details (by Patient)

	PatientID	PatientName	Phone_Enc	HomeAddress_Enc
1	P001	Ali KNNKNNKNNKNNKNN	0x0050740392DABE41A7E5DBDD6AFCD926020...	0x0050740392DABE41A7E5DBDD6AFCD926020...

Figure 29: Patient Personal Details (by SuperAdmin)

Similar to the staff update, the patient update process re-encrypts sensitive fields before storing them. The patient's view shows the updated phone and address in plaintext (Requirement 11). In contrast, the superadmin view shows only ciphertext in the database (Requirement 12), ensuring usability for the patient and confidentiality at rest.

2.3.3 Nurse Encrypt Patient Information for Update

```
OPEN SYMMETRIC KEY SK_PatientPII DECRYPTION BY CERTIFICATE Cert_PatientKEK;

UPDATE p
   SET p.PatientName = COALESCE(@PatientName, p.PatientName),
       p.Phone_Enc   = CASE
                           WHEN @Phone IS NULL THEN NULL
                           ELSE EncryptByKey(Key_GUID('SK_PatientPII'),
                                  CONVERT(VARBINARY(4000), @Phone))
                       END
 FROM SecureData.Patient AS p
WHERE p.PatientID = @PatientID;

CLOSE SYMMETRIC KEY SK_PatientPII;
```

Figure 30: Update Patient Details (by Nurse)

This update procedure corresponds to **Requirement 14**, where nurses are allowed to update a patient's name and phone number only. The data protection process is the same as what patients do when updating their own records. The PatientName is updated directly in plaintext, while the phone is stored in the encrypted column Phone_Enc using the symmetric key **SK_PatientPII**. This ensures that even though nurses can perform the update, the sensitive phone number remains encrypted at rest.

```
UPDATE SecureData.vwPatient
    SET PatientName = 'Siti NMDNMDNMDNMDNMD'
    WHERE PatientID = 'P001'
```

Figure 31: Update Patient Information (by Nurse)

	PatientID	PatientName	Phone
1	P001	Siti NMDNMDNMDNMDNMD	012-01010101

Figure 32: Patient Details (by Nurse)

	PatientID	PatientName	Phone_Enc
1	P001	Siti NMDNMDNMDNMDNMD	0x0050740392DABE41A7E5DBDD6AFCD926020000009A817E6...

Figure 33: Patient Details (by SuperAdmin)

Here, the nurse performs an update on the patient record, modifying the name and phone number as allowed by **Requirement 14**. On the patient's side, the updated details are displayed

clearly in plaintext so the patient can view and verify their own information. On the superadmin side, however, the phone field is stored only as ciphertext in the Phone_Enc column, confirming that the sensitive data remains encrypted at rest.

2.4 Diagnosis confidentiality (Requirements 17 & 18 & 19 & 20)

The problem with diagnosis information is that it is the most sensitive type of medical data in the system. It cannot be stored in plaintext, yet doctors and patients need to work with it directly in their roles**. Requirement 17** states that doctors may only add diagnosis details after an appointment has been scheduled, ensuring that diagnoses are tied to real patient encounters. **Requirement 18** requires that patients can view all of their own diagnosis records, including appointment time, doctor name, and the diagnosis details themselves. **Requirement 19** requires that doctors can view all patients' diagnosis details for clinical purposes, while **Requirement 20** restricts editing so that each doctor may only update the diagnoses that they originally added. These combined requirements demand strong protection of diagnosis data at rest, careful decryption only in the correct contexts, and controlled write operations.

The solution is to protect all diagnosis text using **asymmetric encryption**. The column DiagDetails_Enc stores ciphertext produced by the public key of the certificate Cert_Diag, so the information is never written in plaintext to the database or backups. Whenever a doctor adds a new diagnosis or updates an existing one, the input is encrypted with the public key before being stored, ensuring confidentiality at rest **(Requirements 17 & 20)**. When a doctor or patient needs to view diagnosis details, the database uses the private key portion of the same certificate to decrypt the DiagDetails_Enc value on the fly, returning plaintext only to the authorized view **(Requirements 18 & 19)**. This asymmetric design ensures a strong separation of duties, allowing encryption to be performed widely through the public key. However, decryption requires controlled access to the certificate's private key, thereby reducing the risk of misuse while still meeting the functional requirements of patient care and record management.

2.4.1 Doctor Encrypt Diagnosis for Add and Update

```
UPDATE ad
   SET ad.DiagDetails_Enc = EncryptByCert(CERT_ID('Cert_Diag'),
                                          CONVERT(varbinary(max), @Diagnosis))
FROM SecureData.AppointmentAndDiagnosis ad
WHERE ad.DiagID = @DiagID;
```

Figure 34: Update Patient Diagnosis (by Doctor)

The update statement allows a doctor to add or modify a diagnosis by encrypting the new details before saving them, using **EncryptByCert(CERT_ID('Cert_Diag'), …)** to seal the input with the public key of **Cert_Diag** and storing the result as ciphertext in DiagDetails_Enc. The purpose is to ensure that diagnosis content is never written in plaintext but always protected at rest. Asymmetric encryption enables doctors to encrypt data using the public key freely, yet decryption requires the private key, which is tightly controlled. This design allows doctors to securely add and update diagnoses **(Requirements 17 and 20)** while ensuring that only authorized views or procedures can later decrypt the details for patients and clinicians (Requirements 18 and 19).

```
EXEC SecureData.usp_Doctor_SetDiagnosis @DiagID=1, @Diagnosis=N'Drink more water';
```

Figure 35: Add or Update Diagnosis (by Doctor)

	DiagID	AppDateTime	PatientID	PatientName	DoctorID	DoctorName	Diagnosis
1	1	2025-09-20 12:20:34.120	P001	Siti NMDNMDNMDNMDNMD	D001	Dr. OoiOoiOoiOoi	Drink more water

Figure 36: Diagnosis Details (by Doctor)

	DiagID	AppDateTime	PatientID	DoctorID	DiagDetails_Enc
1	1	2025-09-20 12:20:34.120	P001	D001	0x4016E78B79AEE30C8FB88F754CEDFA6EFA1067A123E659...

Figure 37: Diagnosis Details (by SuperAdmin)

The doctor can successfully add or update a diagnosis, and once submitted, the diagnosis text is stored in the database (DiagDetails_Enc) as ciphertext **(Requirements 17 and 20)**. From the superadmin perspective, the record remains encrypted to preserve confidentiality at rest. At the same time, in the doctor's view, the decrypted diagnosis is shown to confirm that the update was applied correctly. This validates that the system supports the secure creation and modification of diagnosis records, with the mechanism for how doctors and patients view decrypted details explained in the following section.

2.4.2 Doctor and Patient Decrypt Diagnosis for Display

```
CONVERT(nvarchar(max), DecryptByCert(CERT_ID('Cert_Diag'), ad.DiagDetails_Enc)) AS Diagnosis
```

Figure 38: Decrypt Diagnosis (by Doctor and Patient)

The mechanism for retrieving diagnosis details is the same for both doctors and patients. The encrypted column DiagDetails_Enc is decrypted on the fly using the private key from the certificate **Cert_Diag**. The function **DecryptByCert(CERT_ID('Cert_Diag'), ad.DiagDetails_Enc)** converts the ciphertext back into readable text, which is then displayed in the query output as Diagnosis. This ensures that the sensitive medical information remains stored securely as ciphertext in the database.

```
SELECT * FROM SecureData.vwAppointments_Doctor
```

Figure 39: View Appointment (by Doctor)

	DiagID	AppDateTime	PatientID	PatientName	DoctorID	DoctorName	Diagnosis
1	1	2025-09-20 12:20:34.120	P001	Siti NMDNMDNMDNMDNMD	D001	Dr. OoiOoiOoiOoi	Drink more water

Figure 40: Appointment (by Doctor)

This result shows that when the doctor queries through **vwAppointments_Doctor**, the encrypted diagnosis field is successfully decrypted and displayed in plaintext alongside appointment details. It demonstrates that doctors can view all their diagnosis records as required (**Requirement 19**), while the data itself remains stored securely as ciphertext in the database. This maintains confidentiality at rest.

```
EXEC SecureData.usp_Patient_Diagnosis_ListSelf;
```

Figure 41: View Appointment (by Patient)

	DiagID	AppDateTime	DoctorID	DoctorName	Diagnosis
1	1	2025-09-20 12:20:34.120	D001	Dr. OoiOoiOoiOoi	Drink more water

Figure 42: Appointment (by Patient)

The result shows a patient viewing their own diagnosis records, where the encrypted DiagDetails_Enc column is decrypted and displayed together with appointment and doctor information. The mechanism is essentially the same as for doctors, since both rely on the same certificate private key to decrypt the diagnosis text. This ensures that patients can view their complete medical records in plaintext (**Requirement 18**), while the database continues to store sensitive details securely as ciphertext.

2.5 Directory listings with masking (Requirements 10 & 16 & 21)

Another problem is that certain directory views must remain helpful in day-to-day operations while still protecting sensitive information that staff are not authorized to see. **Requirement 10** specifies that all authenticated users must be able to see all staff names and office phone numbers, but if a general query such as SELECT * is executed, columns like personal phone and home address are also part of the result. To prevent leakage, those fields must be **masked** when the viewer is not the record owner. **Requirement 16** requires nurses to update or cancel appointments when no diagnosis exists yet. However, they are not allowed to view the diagnosis text directly (**Requirement 21**), and it would be inappropriate to return the raw encrypted value. Together, these create a need for **masking** to present directory information in a safe and controlled way.

The solution is to implement **dynamic masking** in the views, where the database returns substituted values (such as "**********") in place of the actual sensitive content whenever the user does not meet the conditions for decryption. For staff listings, only StaffName and OfficePhone are always visible, while fields like HomeAddress and PersonalPhone are masked for all other staff except the owner. For patient records, staff see PatientName and phone as required. However, nurses see diagnosis columns replaced with masked text, ensuring that they can still confirm whether a diagnosis exists (to satisfy **Requirement 16**) without reading its content (to satisfy **Requirement 21**). This approach is classified as **dynamic masking** because the underlying data remains encrypted and intact in storage, while the masking is applied at query time based on the context of who is accessing the data.

2.5.1 Staff Sensitive Data Masking

```
SELECT
    s.StaffID,
    --CASE WHEN USER_NAME() = s.StaffID THEN s.StaffID ELSE '**********' END AS StaffID,
    s.StaffName,
    s.OfficePhone,
    CASE
        WHEN USER_NAME() = s.StaffID THEN
            CONVERT(varchar(200),
                DecryptByKeyAutoCert(CERT_ID('Cert_StaffKEK'), NULL, s.HomeAddress_Enc)
            )
        ELSE '**********'
    END AS HomeAddress,
    CASE
        WHEN USER_NAME() = s.StaffID THEN
            CONVERT(varchar(50),
                DecryptByKeyAutoCert(CERT_ID('Cert_StaffKEK'), NULL, s.PersonalPhone_Enc)
            )
        ELSE '**********'
    END AS PersonalPhone,
    CASE
        WHEN USER_NAME() = s.StaffID THEN s.Position
        ELSE '**********'
    END AS Position
FROM SecureData.Staff AS s;
```

Figure 43: Create Masking for Staff Data (by Staff)

Masking is implemented for staff records to ensure that sensitive details are only visible to the record owner. When the logged-in user matches the StaffID, the encrypted fields HomeAddress_Enc and PersonalPhone_Enc are decrypted and displayed in plaintext, along with the staff position. For all other users, these fields are replaced with masked placeholders ('**********'), while non-sensitive attributes like StaffName and OfficePhone remain openly visible to everyone. The purpose of this design is to allow directory-style queries to return useful information without leaking private data. It satisfies **Requirement 10** by keeping staff name and office phone accessible, while maintaining confidentiality of PII through dynamic masking, ensuring that only authorized users can view their own details, and preventing exposure of others' sensitive information.

```
SELECT * FROM SecureData.vwStaff
```

Figure 44: View Staff table (by Staff)

	StaffID	StaffName	OfficePhone	HomeAddress	PersonalPhone	Position
1	D001	Dr. OoiOoiOoiOoi	03-11112222	12 Jalan Bukit, KL	012-3456789	Doctor
2	D002	Dr. Bryan	03-11113333	**********	**********	**********
3	D003	Dr. D003	03-110003	**********	**********	**********
4	D004	Dr. D004	03-110004	**********	**********	**********
5	D005	Dr. D005	03-110005	**********	**********	**********

Figure 45: Staff Table (by Staff)

The result shows the output view, where directory-safe fields such as StaffName and OfficePhone are visible to all users, while sensitive attributes like HomeAddress, PersonalPhone, and Position are only shown in plaintext to the record owner. For other staff, those columns are replaced with "**********", demonstrating that **dynamic masking** is applied at query time.

2.5.2 Diagnosis Masking for Nurses

```
SELECT
    ad.DiagID,
    ad.AppDateTime,
    ad.PatientID,
    p.PatientName,
    ad.DoctorID,
    s.StaffName AS DoctorName,
    CASE
        WHEN ad.DiagDetails_Enc IS NULL
            THEN CAST(NULL AS nvarchar(max))
        ELSE N'**********'
    END AS Diagnosis
FROM SecureData.AppointmentAndDiagnosis AS ad
JOIN SecureData.Patient AS p ON p.PatientID = ad.PatientID
JOIN SecureData.Staff   AS s ON s.StaffID   = ad.DoctorID;
```

Figure 46: Create Masking for Diagnosis (by Nurse)

This nurse-facing query intentionally **never decrypts** diagnosis text and instead applies **dynamic masking** at query time. It joins appointments with patient and doctor info, then uses a CASE expression on DiagDetails_Enc. If no diagnosis has been recorded (IS NULL), it returns NULL. Otherwise, it returns a neutral placeholder ('**********') in the diagnosis column. It lets nurses confirm whether a diagnosis exists to support scheduling changes or cancellations **(Requirement 16)** while preventing any disclosure of the diagnosis content **(Requirement 21)**. The design keeps diagnosis data encrypted at rest and avoids revealing ciphertext or plaintext to nurses, providing a clear “exists/doesn’t exist” signal without exposing sensitive details.

```
SELECT * FROM SecureData.vwAppointments_Nurse;
```

Figure 47: View Appointment Table (by Nurse)

	DiagID	AppDateTime	PatientID	PatientName	DoctorID	DoctorName	Diagnosis
1	1	2025-09-20 12:20:34.120	P001	Siti NMDNMDNMDNMDNMD	D001	Dr. OoiOoiOoiOoi	**********

Figure 48: Appointment Table (by Nurse)

This output shows that nurses can see scheduling context (DiagID, appointment time, patient, and doctor info) while the **Diagnosis** column is masked as **********. The view never decrypts DiagDetails_Enc, because there is no need to do so. Instead, it returns a placeholder to indicate that a diagnosis exists without revealing its content.

2.6 Backup & Restoration (Requirement 4)

Requirement 4 emphasizes that the database must remain recoverable even in the event of system failure, accidental deletion, or corruption. To meet this need, implementing a structured backup and restoration plan is essential, as it ensures that secure copies of the database are consistently created and can be used to restore operations with minimal disruption.

```
SET NOCOUNT ON;

-- === CONFIG ===
DECLARE @Db             sysname        = N'MedicalInfoSystem';
DECLARE @DbB            sysname        = N'[' + REPLACE(N'MedicalInfoSystem', N']', N']
DECLARE @BackupDir      nvarchar(4000) = N'C:\Users\user\Documents\APU\04 APD3F2502CS(D
DECLARE @TDEPassword    nvarchar(200)  = N'Str0ng!DBMK_P@ss_2025';      -- master key p
DECLARE @PrivKeyPass    nvarchar(200)  = N'Str0ng!PVK_P@ss_2025';       -- password for

-- Ensure backup folder exists (best effort)
BEGIN TRY EXEC master.dbo.xp_create_subdir @BackupDir; END TRY BEGIN CATCH END CATCH;
```

Figure 49: Set Constant Variable

Before the backup process begins, all configuration values the backup/TDE script needs are initialized, including database name (both plain and bracket-quoted), backup directory, and strong passwords for the master key and the exported TDE certificate's private key.

2.6.1 Set the database to FULL recovery

```
-- === A) FULL recovery model ===
IF (SELECT recovery_model FROM sys.databases WHERE name=@Db) <> 1  -- 1 = FULL
BEGIN
    DECLARE @sqlRec nvarchar(400) = N'ALTER DATABASE ' + @DbB + N' SET RECOVERY FULL WITH NO_WAIT;';
    EXEC sys.sp_executesql @sqlRec;
END
```

Figure 50: Set Database to Full Recovery

	name	recovery_model_desc
1	MedicalInfoSystem	FULL

Figure 51: Full Recovery is Enabled

The first step is to set the database to **FULL recovery mode**, so that every change is captured in the transaction log, enabling **point-in-time restore**. The script checks the current recovery model. This is important for Requirement 4 because the **LOG backups** will be enabled for this system, which only works in FULL recovery, letting the backup chain streamline from FULL to DIFF and to LOGs.

2.6.2 Create TDE protectors (in master) and back up the certificate

```
-- === B) TDE protectors (in master) + BACKUP of the cert ===
USE master;

IF NOT EXISTS (SELECT 1 FROM sys.symmetric_keys WHERE name = '##MS_DatabaseMasterKey##')
BEGIN
    DECLARE @sqlMK nvarchar(max) = N'CREATE MASTER KEY ENCRYPTION BY PASSWORD = N'''
        + REPLACE(@TDEPassword,'''','''''') + N''';';
    EXEC sys.sp_executesql @sqlMK;
END

IF NOT EXISTS (SELECT 1 FROM sys.certificates WHERE name = 'Cert_TDE_MIS')
BEGIN
    CREATE CERTIFICATE Cert_TDE_MIS
      WITH SUBJECT = 'TDE protector for MedicalInfoSystem';
END

DECLARE @Stamp   varchar(32)    = REPLACE(REPLACE(REPLACE(CONVERT(char(19),GETDATE(),120),':',''),'-',''),' ','_');
DECLARE @CerFile nvarchar(4000) = @BackupDir + N'Cert_TDE_MIS_' + @Stamp + N'.cer';
DECLARE @PvkFile nvarchar(4000) = @BackupDir + N'Cert_TDE_MIS_' + @Stamp + N'.pvk';

DECLARE @sqlBkpCert nvarchar(max) =
N'BACKUP CERTIFICATE Cert_TDE_MIS
   TO FILE = N''' + REPLACE(@CerFile,'''','''''') + N'''
   WITH PRIVATE KEY (
        FILE = N''' + REPLACE(@PvkFile,'''','''''') + N''',
        ENCRYPTION BY PASSWORD = N''' + REPLACE(@PrivKeyPass,'''','''''') + N'''
   );';
EXEC sys.sp_executesql @sqlBkpCert;
```

Figure 52: Create TDE and Backup Certificate

After that, **TDE protectors** are prepared and exported for disaster recovery. It runs in master, first creating a **Database Master Key** (if missing) to safely store private keys, then creating the server certificate **Cert_TDE_MIS** that will protect the database's encryption key. Next, it generates timestamped paths and **backs up the certificate and its private key** to the backup folder (.cer and .pvk). This enables Transparent Data Encryption and makes sure the keys needed to **restore an encrypted database or its backups on another server** are safely archived. Without this certificate backup, TDE-encrypted files are unreadable during recovery. Keeping the protector in the master and exporting it with a password-protected private key balances security and restorability.

2.6.3 Create a Database Encryption Key (DEK) and enable TDE

```
-- === C) Create DEK in DB + turn TDE ON (tempdb will be encrypted too) ===
DECLARE @sqlCreateDEK nvarchar(max) =
N'IF NOT EXISTS (SELECT 1 FROM sys.dm_database_encryption_keys WHERE database_id = DB_ID())
BEGIN
    CREATE DATABASE ENCRYPTION KEY WITH ALGORITHM = AES_256
        ENCRYPTION BY SERVER CERTIFICATE Cert_TDE_MIS;
END;';
EXEC ('USE ' + @DbB + '; ' + @sqlCreateDEK);

IF (SELECT is_encrypted FROM sys.databases WHERE name = @Db) = 0
BEGIN
    DECLARE @sqlEnc nvarchar(200) = N'ALTER DATABASE ' + @DbB + N' SET ENCRYPTION ON;';
    EXEC sys.sp_executesql @sqlEnc;
END

-- Evidence: 2 = encrypting, 3 = encrypted
SELECT db_name(database_id) AS DBName, encryption_state, encryptor_type
FROM sys.dm_database_encryption_keys
WHERE database_id IN (DB_ID(@Db), DB_ID('tempdb'));
```

Figure 53: Create DEK and Enable TDE

	DBName	encryption_state	encryptor_type
1	tempdb	3	ASYMMETRIC KEY
2	MedicalInfoSystem	2	CERTIFICATE

Figure 54: Database Encryption State

With TDE, the next step is to create a Database Encryption Key (DEK) for the **MedicalInfoSystem** using AES-256 and secure it with the certificate Cert_TDE_MIS. After that, **TDE** can be turned on so that all data files and logs are automatically encrypted. The result shows that both the system database tempdb and the MedicalInfoSystem database are now protected. This ensures all stored data, including temporary files, is safe from being read in plain text if storage or backups are exposed.

2.6.4 Take on-demand backups (FULL, DIFF, LOG) with verification

```
-- === D) On-demand BACKUPS NOW (FULL smart, DIFF, LOG) ===
DECLARE @ts       varchar(32)    = REPLACE(REPLACE(REPLACE(CONVERT(char(19),GETDATE(),120),':',''),'-',''),' ','_');
DECLARE @fullFile nvarchar(4000) = @BackupDir + @Db + N'_FULL_' + @ts + N'.bak';
DECLARE @diffFile nvarchar(4000) = @BackupDir + @Db + N'_DIFF_' + @ts + N'.bak';
DECLARE @logFile  nvarchar(4000) = @BackupDir + @Db + N'_LOG_'  + @ts + N'.trn';

DECLARE @HasBaseFull bit =
    CASE WHEN EXISTS (
        SELECT 1 FROM msdb.dbo.backupset
        WHERE database_name=@Db AND type='D' AND is_copy_only=0
    ) THEN 1 ELSE 0 END;

DECLARE @sqlFull nvarchar(max);
IF @HasBaseFull = 0
    SET @sqlFull = N'BACKUP DATABASE ' + @DbB + N' TO DISK = @file WITH COMPRESSION, CHECKSUM, STATS=5;';
ELSE
    SET @sqlFull = N'BACKUP DATABASE ' + @DbB + N' TO DISK = @file WITH COPY_ONLY, COMPRESSION, CHECKSUM, STATS=5;';

EXEC sys.sp_executesql @sqlFull, N'@file nvarchar(4000)', @file=@fullFile;

DECLARE @sqlDiff nvarchar(max) = N'BACKUP DATABASE ' + @DbB + N' TO DISK = @file WITH DIFFERENTIAL, COMPRESSION, CHECKSUM, STATS=5;';
EXEC sys.sp_executesql @sqlDiff, N'@file nvarchar(4000)', @file=@diffFile;

DECLARE @sqlLog nvarchar(max)  = N'BACKUP LOG ' + @DbB + N' TO DISK = @file WITH COMPRESSION, CHECKSUM, STATS=5;';
EXEC sys.sp_executesql @sqlLog, N'@file nvarchar(4000)', @file=@logFile;

-- Verify the files we just wrote (uses variables, no hardcoded timestamps)
RESTORE VERIFYONLY FROM DISK = @fullFile;
RESTORE VERIFYONLY FROM DISK = @diffFile;
RESTORE VERIFYONLY FROM DISK = @logFile;
```

Figure 55: Enable On-demand Backups

	database_name	backup_type	is_copy_only	backup_start_date	backup_finish_date	physical_device_name
1	MedicalInfoSystem	LOG	0	2025-09-19 20:23:31.000	2025-09-19 20:23:31.000	C:\Users\user\Documents\APU\04 APD3F2502CS(DA)\Se...
2	MedicalInfoSystem	DIFF	0	2025-09-19 20:23:31.000	2025-09-19 20:23:31.000	C:\Users\user\Documents\APU\04 APD3F2502CS(DA)\Se...
3	MedicalInfoSystem	FULL	0	2025-09-19 20:23:30.000	2025-09-19 20:23:31.000	C:\Users\user\Documents\APU\04 APD3F2502CS(DA)\Se...

Figure 56: Backup Details

To test whether the backup actually works, on-demand backups of the MedicalInfoSystem database are implemented, including a **full backup**, a **differential backup**, and a **transaction log backup**. Each backup file is saved with a timestamp, compressed, and verified using RESTORE VERIFYONLY to confirm its validity and restorability. The full backup captures the entire database, the differential backup stores only the changes since the last full backup, and the log backup records all recent transactions. These backups ensure the database can be restored to a safe and consistent state in the event of a failure. This fulfils Requirement 4 for reliable data protection.

2.6.5 Schedule recurring backups with SQL Agent jobs

Name	PID	Description	Status
SQLSERVERAGENT	19992	SQL Server Agent (MSSQLSERVER)	Running

Figure 57: Enable SQL Server Agent

	servicename	status_desc
1	SQL Server Agent (MSSQLSERVER)	Running

Figure 58: SQL Server Agent is Enabled

Before setting up recurring backup jobs, it is necessary to ensure that the **SQL Server Agent service is running**, because this service is responsible for executing scheduled tasks in SQL Server. Without it, even if the backup jobs are correctly created and scheduled, they will not run automatically.

```
-- === E) SQL Agent Jobs (FULL @ 02:00; DIFF / 4h; LOG / 15m) ===
DECLARE @JobFull sysname = N'APU_Backup_FULL_' + @Db;
DECLARE @JobDiff sysname = N'APU_Backup_DIFF_' + @Db;
DECLARE @JobLog  sysname = N'APU_Backup_LOG_'  + @Db;

DECLARE @SchFull sysname = N'SCH_Backup_FULL_Daily_02';
DECLARE @SchDiff sysname = N'SCH_Backup_DIFF_Every4h';
DECLARE @SchLog  sysname = N'SCH_Backup_LOG_Every15min';

-- step commands (generate timestamped filenames inside the job)
DECLARE @cmdFULL nvarchar(max) =
N'DECLARE @Dir nvarchar(4000)=N''' + REPLACE(@BackupDir,'''','''''') + N''';
  DECLARE @ts  varchar(32)=REPLACE(REPLACE(REPLACE(CONVERT(char(19),GETDATE(),120),'':'',''''),''-'',''''),'' '',''_'' );
  DECLARE @file nvarchar(4000)=@Dir+N''' + REPLACE(@Db,'''','''''') + N''' + N''_FULL_''+@ts+N''.bak'';
  BACKUP DATABASE ' + @DbB + N' TO DISK = @file WITH COMPRESSION, CHECKSUM, STATS=5;';

DECLARE @cmdDIFF nvarchar(max) =
N'DECLARE @Dir nvarchar(4000)=N''' + REPLACE(@BackupDir,'''','''''') + N''';
  DECLARE @ts  varchar(32)=REPLACE(REPLACE(REPLACE(CONVERT(char(19),GETDATE(),120),'':'',''''),''-'',''''),'' '',''_'' );
  DECLARE @file nvarchar(4000)=@Dir+N''' + REPLACE(@Db,'''','''''') + N''' + N''_DIFF_''+@ts+N''.bak'';
  BACKUP DATABASE ' + @DbB + N' TO DISK = @file WITH DIFFERENTIAL, COMPRESSION, CHECKSUM, STATS=5;';

DECLARE @cmdLOG nvarchar(max) =
N'DECLARE @Dir nvarchar(4000)=N''' + REPLACE(@BackupDir,'''','''''') + N''';
  DECLARE @ts  varchar(32)=REPLACE(REPLACE(REPLACE(CONVERT(char(19),GETDATE(),120),'':'',''''),''-'',''''),'' '',''_'' );
  DECLARE @file nvarchar(4000)=@Dir+N''' + REPLACE(@Db,'''','''''') + N''' + N''_LOG_''+@ts+N''.trn'';
  BACKUP LOG ' + @DbB + N' TO DISK = @file WITH COMPRESSION, CHECKSUM, STATS=5;';
```

Figure 59: Schedule Backup

```
-- FULL job (create or update)
IF NOT EXISTS (SELECT 1 FROM msdb.dbo.sysjobs WHERE name = @JobFull)
BEGIN
  EXEC msdb.dbo.sp_add_job     @job_name=@JobFull, @enabled=1, @description=N'Nightly FULL backup (TDE)';
  EXEC msdb.dbo.sp_add_jobstep @job_name=@JobFull, @step_name=N'FULL', @subsystem=N'TSQL', @database_name=N'master', @command=@cmdFULL;

  IF NOT EXISTS (SELECT 1 FROM msdb.dbo.sysschedules WHERE name = @SchFull)
    EXEC msdb.dbo.sp_add_schedule @schedule_name=@SchFull, @freq_type=4, @freq_interval=1, @active_start_time=020000; -- daily 02:00

  IF NOT EXISTS (
      SELECT 1
      FROM msdb.dbo.sysjobschedules js
      JOIN msdb.dbo.sysschedules sc ON sc.schedule_id = js.schedule_id
      JOIN msdb.dbo.sysjobs      j  ON j.job_id = js.job_id
      WHERE j.name=@JobFull AND sc.name=@SchFull)
    EXEC msdb.dbo.sp_attach_schedule @job_name=@JobFull, @schedule_name=@SchFull;

  EXEC msdb.dbo.sp_add_jobserver  @job_name=@JobFull, @server_name=@@SERVERNAME;
END
ELSE
BEGIN
  EXEC msdb.dbo.sp_update_jobstep @job_name=@JobFull, @step_id=1, @step_name=N'FULL', @command=@cmdFULL;
END
```

Figure 60: Schedule Full Backup

```
-- DIFF job
IF NOT EXISTS (SELECT 1 FROM msdb.dbo.sysjobs WHERE name = @JobDiff)
BEGIN
  EXEC msdb.dbo.sp_add_job     @job_name=@JobDiff, @enabled=1, @description=N'Differential backup every 4 hours (TDE)';
  EXEC msdb.dbo.sp_add_jobstep @job_name=@JobDiff, @step_name=N'DIFF', @subsystem=N'TSQL', @database_name=N'master', @command=@cmdDIFF;

  IF NOT EXISTS (SELECT 1 FROM msdb.dbo.sysschedules WHERE name = @SchDiff)
    EXEC msdb.dbo.sp_add_schedule @schedule_name=@SchDiff, @freq_type=4, @freq_interval=1,
                                  @freq_subday_type=8, @freq_subday_interval=4, @active_start_time=000000; -- every 4h

  IF NOT EXISTS (
      SELECT 1
      FROM msdb.dbo.sysjobschedules js
      JOIN msdb.dbo.sysschedules sc ON sc.schedule_id = js.schedule_id
      JOIN msdb.dbo.sysjobs      j  ON j.job_id = js.job_id
      WHERE j.name=@JobDiff AND sc.name=@SchDiff)
    EXEC msdb.dbo.sp_attach_schedule @job_name=@JobDiff, @schedule_name=@SchDiff;

  EXEC msdb.dbo.sp_add_jobserver @job_name=@JobDiff, @server_name=@@SERVERNAME;
END
ELSE
BEGIN
  EXEC msdb.dbo.sp_update_jobstep @job_name=@JobDiff, @step_id=1, @step_name=N'DIFF', @command=@cmdDIFF;
END
```

Figure 61: Schedule Differential Backup

```
-- LOG job
IF NOT EXISTS (SELECT 1 FROM msdb.dbo.sysjobs WHERE name = @JobLog)
BEGIN
  EXEC msdb.dbo.sp_add_job     @job_name=@JobLog, @enabled=1, @description=N'Log backup every 15 minutes (TDE)';
  EXEC msdb.dbo.sp_add_jobstep @job_name=@JobLog, @step_name=N'LOG', @subsystem=N'TSQL', @database_name=N'master', @command=@cmdLOG;

  IF NOT EXISTS (SELECT 1 FROM msdb.dbo.sysschedules WHERE name = @SchLog)
    EXEC msdb.dbo.sp_add_schedule @schedule_name=@SchLog, @freq_type=4, @freq_interval=1,
                                  @freq_subday_type=4, @freq_subday_interval=15, @active_start_time=000000; -- every 15m

  IF NOT EXISTS (
      SELECT 1
      FROM msdb.dbo.sysjobschedules js
      JOIN msdb.dbo.sysschedules sc ON sc.schedule_id = js.schedule_id
      JOIN msdb.dbo.sysjobs      j  ON j.job_id = js.job_id
      WHERE j.name=@JobLog AND sc.name=@SchLog)
    EXEC msdb.dbo.sp_attach_schedule @job_name=@JobLog, @schedule_name=@SchLog;

  EXEC msdb.dbo.sp_add_jobserver  @job_name=@JobLog, @server_name=@@SERVERNAME;
END
ELSE
BEGIN
  EXEC msdb.dbo.sp_update_jobstep @job_name=@JobLog, @step_id=1, @step_name=N'LOG', @command=@cmdLOG;
END
```

Figure 62: Schedule Transaction Log Backup

SQL Agent job names, schedules, and backup commands that will run automatically to protect the MedicalInfoSystem database are defined. Three types of jobs are prepared, including a full backup job scheduled daily at 2:00 AM, a differential backup job every 4 hours, and a transaction log backup job every 15 minutes. Each job generates timestamped filenames to prevent old backups from being overwritten, and all backups utilize compression and checksums for enhanced efficiency and integrity. This setup ensures continuous and automated protection of data while also minimizing data loss.

	job_name	job_enabled	schedule_name	schedule_enabled	next_run	status
1	APU_Backup_DIFF_MedicalInfoSystem	1	SCH_Backup_DIFF_Every4h	1	2025-09-20 00:00:00.000	Scheduled
2	APU_Backup_FULL_MedicalInfoSystem	1	SCH_Backup_FULL_Daily_02	1	2025-09-20 02:00:00.000	Scheduled
3	APU_Backup_LOG_MedicalInfoSystem	1	SCH_Backup_LOG_Every1...	1	2025-09-19 20:15:00.000	Scheduled

Figure 63: Show next run time

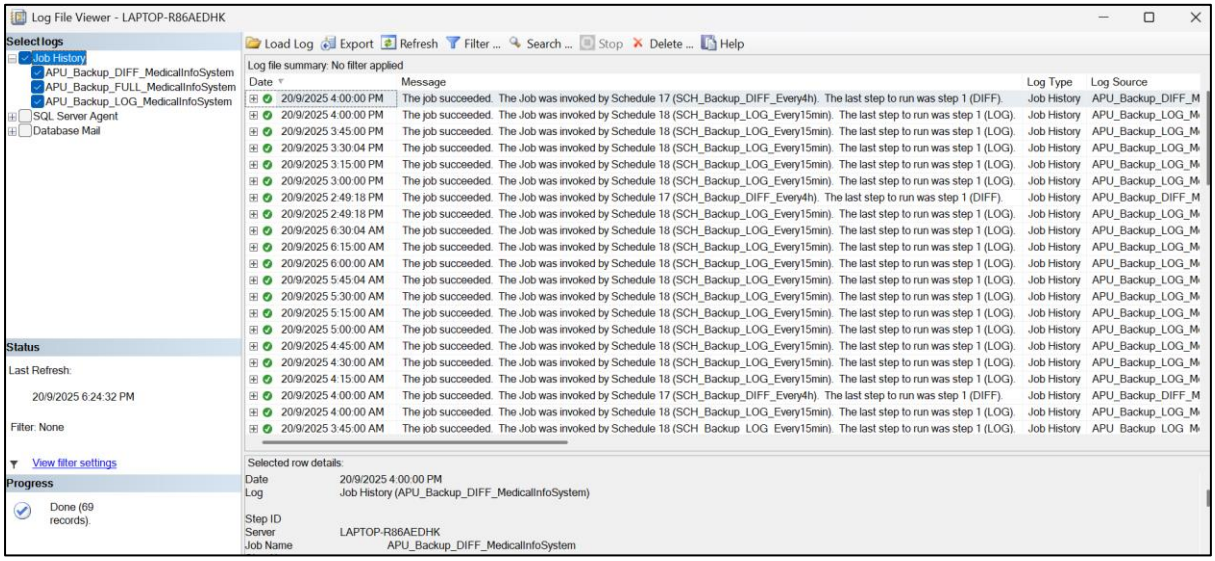


Figure 64: Backup History

This result shows that the scheduled backup jobs for the MedicalInfoSystem database are active, properly configured, and successfully run. Each job and schedule is enabled, and the "next_run" times confirm when the tasks will be executed. This confirms that the backup automation is functioning as planned, providing continuous database protection without requiring manual intervention.

In conclusion, the backup process ensures this database is continuously protected through a layered strategy. The database is first secured with **TDE**, so even backup files remain unreadable without the certificate. Then, a combination of **full, differential, and transaction log backups** is used to balance recovery speed and storage efficiency, with verification steps included to guarantee backup integrity. Finally, the use of **SQL Server Agent jobs** automates this cycle on a reliable schedule. It can reduce human error and ensure timely execution. Overall, these measures provide strong assurance of data availability, integrity, and recoverability.

3.0 Permission Management

This section documents how access to the APU Hospital database is controlled so that every user can do exactly what their role requires, no more or no less, while keeping sensitive medical data protected. The design follows the principles of least privilege, separation of duties, and defence in depth, where users authenticate as individual logins, are authorized through roles, and interact only with carefully scoped database objects that enforce business rules at multiple layers (Joint Task Force, 2020).

At a high level, the solution combines four complementary mechanisms:

- **Role-Based Access Control (RBAC):** Server logins are mapped to database users and grouped into roles like SuperAdmins, Doctors, Nurses, and Patients, so permissions are assigned once per role instead of per user. This makes the model auditable, scalable, and easy to maintain as staff change (Joint Task Force, 2020).
- **Object-Level Permissions:** Direct access to base tables is revoked. Instead, users interact via views and stored procedures that expose only the columns and actions relevant to their role (e.g., nurses can update patient contact details, doctors can set diagnoses, and patients can only view their own information). GRANT/REVOKE/DENY statements on these objects implement least privilege.
- **Advanced Access Control with RLS:** Row-Level Security predicates and policies restrict which rows a role can read or modify (e.g., "self-only" updates for staff records; patients see only their own row). This prevents horizontal data leakage even if someone discovers object names or tries ad-hoc queries (Microsoft SQL, 2025).
- **Encryption Permissions (with selective decryption):** Sensitive columns are encrypted at rest using symmetric keys (PII) and an asymmetric certificate (diagnosis). Key/certificate permissions and module signing ensure that only authorized roles, or in some cases, only approved procedures, can decrypt data, allowing patients to access their own data without being granted broad key rights (Microsoft SQL, 2023).

Together, these controls satisfy the application's functional rules while protecting confidentiality and integrity. For example, staff can view and update their own details, but others see only directory-safe fields. Another example includes that nurses can manage appointments but never view diagnosis content. In contrast, doctors can view and set diagnoses but only for the correct patients, and patients can view their own diagnoses and update their own details without seeing anyone else's data.

3.1 Authorization Table/Matrix

Role	Permission Type	Object	Scope / Condition	Privilege(s)
SuperAdmins	GRANT (role membership)	db_owner on MedicalInfoSystem	Administrative maintenance	Full DDL and DML within the database
SuperAdmins	GRANT	Security objects: Cert_PatientKEK, SK_PatientPII, Cert_Diag	Key and certificate maintenance	CONTROL / VIEW DEFINITION, open/rotate keys and certs
SuperAdmins	GRANT	Security policies, predicate functions	RLS administration	CREATE/ ALTER/ DROP policy and functions
Doctors	GRANT	SecureData.vwStaff	Directory for all staff; self-row shows full details	SELECT
Doctors	GRANT	SecureData.vwPatient	Directory for all patients	SELECT
Doctors	GRANT	SecureData.vwAppointments_Doctor	All patients' appointments with decrypted diagnoses	SELECT
Doctors	GRANT EXECUTE	SecureData.usp_Doctor_SetDiagnosis	Only for appointments owned by the caller's doctor	Add/update diagnosis (encrypted at write)
Doctors	REVOKE	Base tables (SecureData.Staff, SecureData.Patient, SecureData.AppointmentAndDiagnosis)	—	No direct SELECT/ INSERT/ UPDATE/ DELETE

Nurses	GRANT	SecureData.vwStaff	Directory for all staff; self-row shows full details	SELECT
Nurses	GRANT	SecureData.vwPatient	Directory for all patients	SELECT
Nurses	GRANT	SecureData.vwAppointments_Nurse	Appointments only; diagnosis masked or null	SELECT
Nurses	GRANT	SecureData.vwPatient	Update path for patient name and phone only	UPDATE (columns: PatientName, Phone)
Nurses	GRANT EXECUTE	SecureData.usp_Nurse_AddAppointment	Appointment creation	Add appointment
Nurses	GRANT EXECUTE	SecureData.usp_Nurse_UpdateAppointment	Only when DiagDetails is NULL	Update appointment datetime
Nurses	GRANT EXECUTE	SecureData.usp_Nurse_CancelAppointment	Only when DiagDetails is NULL	Cancel appointment
Nurses	REVOKE	Base tables (SecureData.Staff, SecureData.Patient, SecureData.AppointmentAndDiagnosis)	—	No direct SELECT/ INSERT/ UPDATE/ DELETE
Patients	REVOKE	SecureData.vwPatient	Directory surface	No SELECT (cannot enumerate others)
Patients	GRANT EXECUTE	SecureData.usp_Patient_Self_Get	Self only	View own details in plaintext

Patients	GRANT EXECUTE	SecureData.usp_Patient_Self_Update	Self only	Update own details, encrypted at rest
Patients	GRANT EXECUTE	SecureData.usp_Patient_Diagnosis_ListSelf	Self only	View own diagnosis, appointment time, doctor name, decrypted via signed module
Patients	REVOKE	Base tables (SecureData.Staff, SecureData.Patient, SecureData.AppointmentAndDiagnosis)	—	No direct SELECT/ INSERT/ UPDATE/ DELETE

Table 2: Authorization Table/Matrix

The Authorization table/matrix as shown in Table 2 formalizes least-privilege access by separating authentication identities from authorization surfaces and routing every action through curated objects. Direct access to base tables is removed for Doctors, Nurses, and Patients, so ad-hoc queries cannot bypass masking, row ownership checks, or encryption rules. Read access is exposed through views that return only the columns each role needs. Staff and patient directories are delivered from vwStaff and vwPatient, where directory fields are always visible and sensitive attributes appear only on the owner's row. Diagnosis visibility is split by profession: doctors read decrypted diagnosis via vwAppointments_Doctor, while nurses see appointments with diagnosis masked in vwAppointments_Nurse.

Write access is funnelled through a few stored procedures and updateable views that enforce business rules and re-encrypt data at the boundary. Nurses manage appointment creation, time changes, and cancellations only while no diagnosis exists, and they edit patient name and phone through the clinician view so the INSTEAD OF UPDATE trigger re-encrypts into the _Enc columns. Doctors add or amend diagnosis only through usp_Doctor_SetDiagnosis, which checks that the caller is the assigned doctor before writing the encrypted value. Patients never touch tables or clinician views; instead they use signed self-service procedures that return and update their own record in plaintext while keeping key and certificate permissions out of the Patients role.

SuperAdmin is isolated as the single maintenance principal. Membership in db_owner and explicit control of keys, certificates, and security policies allow creation and rotation of cryptographic material, adjustment of RLS, and other DDL as needed, without over-granting operational roles. The result is a predictable and auditable permission scheme: each role can do exactly what the requirements specify, nothing more, with encryption boundaries, row-level filters, and object-level grants working together to protect confidentiality, integrity, and usability.

3.2 Identity & Role Provisioning (Requirements 2 & 5 & 7)

Requirement 2 - SuperAdmin must be able to perform appropriate DDL (such as create tables, views, logins, users, encryption keys, etc) and DML tasks to maintain a high level of functionality, availability, and security of this DB

Requirement 5 - All users must be able to log into the MS-SQL system using SQL Server Management Studio and perform their own tasks. Assume that all users have sufficient knowledge of writing and running SQL queries in the query window.

Requirement 7 - There are two staff positions: doctors and nurses.

In the hospital database, distinct users must be authenticated to SQL Server and then authorized inside the database according to their job function. Without a formal identity and role provisioning model, several risks would arise, including the use of shared or ad-hoc accounts, over-granted or inconsistent privileges across users, and the inability to apply subsequent controls (object permissions, RLS, and encryption access) predictably. It must also be ensured that a **Superadmin** can perform maintenance DDL/DML (creating schemas, logins, users, roles, keys, views, policies) while ordinary users remain least-privileged. Additionally, every user must be able to sign in via SSMS and perform only the tasks appropriate to their position, like **doctors** and **nurses** as staff roles with different capabilities, and **patients** as end users, so that later permission rules can be enforced cleanly.

3.2.1 Server logins (authentication)

Figure 65: Bulk Provisioning of Server Logins

Parameterized loops are employed to provision many authentication principals consistently and quickly. A patterned password is set for demonstration purposes and is guarded by CHECK_POLICY=ON. A PRINT trail is produced for operational visibility, and the same idempotent existence checks are used to prevent duplicate login creation on re-run. Setting the default database streamlines first-connect behaviour for each login.

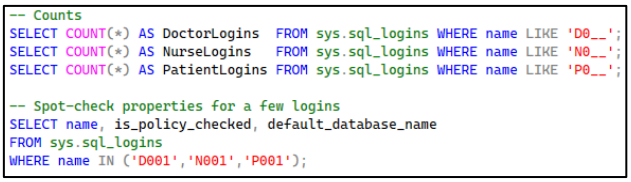
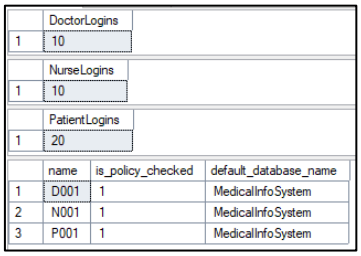


Figure 66: Server Logins Checking



	DoctorLogins
1	10

	NurseLogins
1	10

	PatientLogins
1	20

	name	is_policy_checked	default_database_name
1	D001	1	MedicalInfoSystem
2	N001	1	MedicalInfoSystem
3	P001	1	MedicalInfoSystem

Figure 67: Server Logins Output

The script in Figure 66 verifies that bulk provisioning created the expected number of logins by counting doctor (D0__), nurse (N0__), and patient (P0__) accounts in sys.sql_logins. The results labeled DoctorLogins, NurseLogins, and PatientLogins report the totals for each group in the shown output, which includes 10 doctors, 10 nurses, and 20 patients, matched with the bulk provision of server logins as shown in Figure 67. It then spot-checks three representative

logins (D001, N001, and P001) retrieving is_policy_checked and default_database_name to confirm security policy enforcement and that each account defaults to the MedicalInfoSystem database, as shown in Figure 67.

3.2.2 Database users & role membership (authorization)

```
USE MedicalInfoSystem;
SET NOCOUNT ON;

-- Ensure roles exist
IF NOT EXISTS (SELECT 1 FROM sys.database_principals WHERE type='R' AND name='Doctors')  CREATE ROLE Doctors;
IF NOT EXISTS (SELECT 1 FROM sys.database_principals WHERE type='R' AND name='Nurses')   CREATE ROLE Nurses;
IF NOT EXISTS (SELECT 1 FROM sys.database_principals WHERE type='R' AND name='Patients') CREATE ROLE Patients;

DECLARE @i int, @id sysname, @sql nvarchar(max);
```

Figure 68: Create Roles

```
-- Doctors D001..D010
SET @i = 1;
WHILE @i <= 10
BEGIN
    SET @id = N'D' + RIGHT('000' + CAST(@i AS varchar(3)), 3);

    IF NOT EXISTS (SELECT 1 FROM sys.database_principals WHERE name = @id)
    BEGIN
        SET @sql = N'CREATE USER [' + @id + N'] FOR LOGIN [' + @id + N'];';
        EXEC (@sql);
        PRINT 'Created user ' + @id;
    END
    ELSE
        PRINT 'User ' + @id + ' already exists (skipped).';

    IF NOT EXISTS (
        SELECT 1
        FROM sys.database_role_members drm
        JOIN sys.database_principals r ON r.principal_id = drm.role_principal_id AND r.name='Doctors'
        JOIN sys.database_principals m ON m.principal_id = drm.member_principal_id AND m.name=@id
    )
    BEGIN
        SET @sql = N'ALTER ROLE Doctors ADD MEMBER [' + @id + N'];';
        EXEC (@sql);
        PRINT 'Added ' + @id + ' to Doctors';
    END
    ELSE
        PRINT @id + ' already in Doctors (skipped).';

    SET @i += 1;
END
```

```
-- Nurses N001..N010
SET @i = 1;
WHILE @i <= 10
BEGIN
    SET @id = N'N' + RIGHT('000' + CAST(@i AS varchar(3)), 3);

    IF NOT EXISTS (SELECT 1 FROM sys.database_principals WHERE name = @id)
    BEGIN
        SET @sql = N'CREATE USER [' + @id + N'] FOR LOGIN [' + @id + N'];';
        EXEC (@sql);
        PRINT 'Created user ' + @id;
    END
    ELSE
        PRINT 'User ' + @id + ' already exists (skipped).';

    IF NOT EXISTS (
        SELECT 1
        FROM sys.database_role_members drm
        JOIN sys.database_principals r ON r.principal_id = drm.role_principal_id AND r.name='Nurses'
        JOIN sys.database_principals m ON m.principal_id = drm.member_principal_id AND m.name=@id
    )
    BEGIN
        SET @sql = N'ALTER ROLE Nurses ADD MEMBER [' + @id + N'];';
        EXEC (@sql);
        PRINT 'Added ' + @id + ' to Nurses';
    END
    ELSE
        PRINT @id + ' already in Nurses (skipped).';

    SET @i += 1;
END
```

```
-- Patients P001..P020
SET @i = 1;
WHILE @i <= 20
BEGIN
    SET @id = N'P' + RIGHT('000' + CAST(@i AS varchar(3)), 3);

    IF NOT EXISTS (SELECT 1 FROM sys.database_principals WHERE name = @id)
    BEGIN
        SET @sql = N'CREATE USER [' + @id + N'] FOR LOGIN [' + @id + N'];';
        EXEC (@sql);
        PRINT 'Created user ' + @id;
    END
    ELSE
        PRINT 'User ' + @id + ' already exists (skipped).';

    IF NOT EXISTS (
        SELECT 1
        FROM sys.database_role_members drm
        JOIN sys.database_principals r ON r.principal_id = drm.role_principal_id AND r.name='Patients'
        JOIN sys.database_principals m ON m.principal_id = drm.member_principal_id AND m.name=@id
    )
    BEGIN
        SET @sql = N'ALTER ROLE Patients ADD MEMBER [' + @id + N'];';
        EXEC (@sql);
        PRINT 'Added ' + @id + ' to Patients';
    END
    ELSE
        PRINT @id + ' already in Patients (skipped).';

    SET @i += 1;
END
GO
```

Figure 69: Bulk Creation of Database Users and Role Membership

In Figure 68, the query ensures the three database roles (**Doctors, Nurses, Patients**) exist, and it creates any that are missing. Then, for each created login in Figure 65, a corresponding database user is created so that object permissions can be evaluated within the database, as shown in Figure 69. Role membership is then enforced with a join against sys.database_role_members to avoid duplicate adds. This pattern ensures that authorization is handled uniformly and that least-privilege access is achieved through role inheritance instead of direct grants to users. The output messages provide an audit-friendly trail of what was created or skipped.

By having these practices, each login is bound to a database user so permissions can be evaluated inside the database. Users are also grouped into roles (Doctors, Nurses, Patients). The membership checks against sys.database_role_members keep the script re-runnable without duplicates. This implements clean **RBAC** so later GRANT/REVOKE/RLS is role-scoped, not user-scoped.

```
-- Users created for bulk principals?
SELECT type_desc, COUNT(*) AS Cnt
FROM sys.database_principals
WHERE name LIKE '[DNP]0__'
GROUP BY type_desc;

-- Role membership overview
SELECT r.name AS RoleName, COUNT(*) AS Members
FROM sys.database_role_members drm
JOIN sys.database_principals r ON r.principal_id = drm.role_principal_id
WHERE r.name IN ('Doctors','Nurses','Patients')
GROUP BY r.name
ORDER BY r.name;

-- Detailed membership (optional)
EXEC sp_helprolemember Doctors;
EXEC sp_helprolemember Nurses;
EXEC sp_helprolemember Patients;
```

Figure 70: Database Users & Role Membership Checking

	type_desc	Cnt
1	SQL_USER	40

	RoleName	Members
1	Doctors	10
2	Nurses	10
3	Patients	20

	DbRole	MemberName	MemberSID
1	Doctors	D001	0xF2548F5BE962504E8C5FDB20798B1F1D
2	Doctors	D002	0xCC2E2DCDDA4AE641887CEDE1F02CCDFB
3	Doctors	D003	0xB48364C62B7CE345BA24D9C6737E408C
4	Doctors	D004	0x9BFBE0F0BA55374490EA88F2AF41F738
5	Doctors	D005	0xD596DDFF96463344914D9BE1A54C8FB6
6	Doctors	D006	0xC5C8B3B7EAC3514B918DD2EEBB338876
7	Doctors	D007	0x67CF4419DC466E4A9BFB83962D9BCF79
8	Doctors	D008	0xC0F169C17B1A1F449AD53F6B886C1A71

	DbRole	MemberName	MemberSID
1	Nurses	N001	0x08CAF447C9491348AC934B4E2070BE99
2	Nurses	N002	0xF4E921F7916BD54F95C57EA18C25FB03
3	Nurses	N003	0xFACA13FC5750F04DAA48308756132443
4	Nurses	N004	0x5AC1EE2E00C6A942A477BE80F8E48D...
5	Nurses	N005	0x847F5D61F25B90499120048F3F293569
6	Nurses	N006	0x1AF8840E724BCC449B007502AFBEEB...
7	Nurses	N007	0x86300CA65972D344A8FEE3F1278C1A39
8	Nurses	N008	0x350C548DC715564C92B4659A5F1090A8

	DbRole	MemberName	MemberSID
1	Patients	P001	0x8EF74FC1672AD94B996FF3739AC65BF5
2	Patients	P002	0xB5846D54D2AFA642A16E6BA51A2D4BCC
3	Patients	P003	0x7D170AA16ED7114AAED7FC78E4DD7D...
4	Patients	P004	0x9A0924CDD568904E8FFC7FAE579E2AD4
5	Patients	P005	0xDD877200A5BF334DA4F3C0EC3F3FBE99
6	Patients	P006	0xE7165CA224E05D4C92178AAAA6D1042D
7	Patients	P007	0xACC0C713DA4FB441956C769059407243

Figure 71: Sample Output for Database Users & Role Membership Checking

The script in Figure 70 validates bulk user provisioning and role assignments in the MedicalInfoSystem. It queries sys.database_principals for names matching [DNP]0__ to confirm 40 database users were created (10 Doctors, 10 Nursers, 20 Patients). Next, it summarizes role membership via a join on sys.database_role_members, showing the role Doctors has 10 members, Nurses 10, and Patients 20. Then, sp_helprolemember lists the expected members per role, confirming D001–D010 are in Doctors, N001–N010 in Nurses, and P001–P020 in Patients. This proves authorization identities and RBAC grouping are in place for all principals.

3.2.3 Role setup & SuperAdmin maintenance path

```
USE MedicalInfoSystem;
GO
IF NOT EXISTS (SELECT 1 FROM sys.database_principals WHERE name='SuperAdmin') CREATE USER SuperAdmin FOR LOGIN SuperAdmin;
IF NOT EXISTS (SELECT 1 FROM sys.database_principals WHERE name='D001')       CREATE USER D001       FOR LOGIN D001;
IF NOT EXISTS (SELECT 1 FROM sys.database_principals WHERE name='D002')       CREATE USER D002       FOR LOGIN D002;
IF NOT EXISTS (SELECT 1 FROM sys.database_principals WHERE name='N001')       CREATE USER N001       FOR LOGIN N001;
IF NOT EXISTS (SELECT 1 FROM sys.database_principals WHERE name='N002')       CREATE USER N002       FOR LOGIN N002;
IF NOT EXISTS (SELECT 1 FROM sys.database_principals WHERE name='P001')       CREATE USER P001       FOR LOGIN P001;

IF NOT EXISTS (SELECT 1 FROM sys.database_principals WHERE type='R' AND name='SuperAdmins') CREATE ROLE SuperAdmins;
IF NOT EXISTS (SELECT 1 FROM sys.database_principals WHERE type='R' AND name='Doctors')     CREATE ROLE Doctors;
IF NOT EXISTS (SELECT 1 FROM sys.database_principals WHERE type='R' AND name='Nurses')      CREATE ROLE Nurses;
IF NOT EXISTS (SELECT 1 FROM sys.database_principals WHERE type='R' AND name='Patients')    CREATE ROLE Patients;
```

Figure 72: Map Login to User and Create Role

```
IF NOT EXISTS (
    SELECT 1 FROM sys.database_role_members
    WHERE role_principal_id = USER_ID('SuperAdmins') AND member_principal_id = USER_ID('SuperAdmin'))
    ALTER ROLE SuperAdmins ADD MEMBER SuperAdmin;

IF NOT EXISTS (
    SELECT 1 FROM sys.database_role_members
    WHERE role_principal_id = USER_ID('Doctors') AND member_principal_id = USER_ID('D001'))
    ALTER ROLE Doctors ADD MEMBER D001;
IF NOT EXISTS (
    SELECT 1 FROM sys.database_role_members
    WHERE role_principal_id = USER_ID('Doctors') AND member_principal_id = USER_ID('D002'))
    ALTER ROLE Doctors ADD MEMBER D002;

IF NOT EXISTS (
    SELECT 1 FROM sys.database_role_members
    WHERE role_principal_id = USER_ID('Nurses') AND member_principal_id = USER_ID('N001'))
    ALTER ROLE Nurses ADD MEMBER N001;
IF NOT EXISTS (
    SELECT 1 FROM sys.database_role_members
    WHERE role_principal_id = USER_ID('Nurses') AND member_principal_id = USER_ID('N002'))
    ALTER ROLE Nurses ADD MEMBER N002;

IF NOT EXISTS (
    SELECT 1 FROM sys.database_role_members
    WHERE role_principal_id = USER_ID('Patients') AND member_principal_id = USER_ID('P001'))
    ALTER ROLE Patients ADD MEMBER P001;
```

Figure 73: Add Members to Roles

```
-- Convenience (db_owner)
EXEC sp_addrolemember 'db_owner', 'SuperAdmin';
GO
```

Figure 74: Grant db_owner to SuperAdmin

As shown in Figure 72, server logins are bound to database users so that authorization can be applied inside the database. A defensive, existence-checked creation pattern is used for both users and roles to keep the script re-runnable. Four application roles, SuperAdmins, Doctors, Nurses, and Patients are established so that privileges can be assigned at role scope rather than per user, enabling least-privilege management and simpler auditing. Then in Figure 73Role membership is then assigned so that each user inherits the permissions associated with their job function. Conditional checks are applied before assignment so that a clean, idempotent state is maintained across repeated runs. In Figure 74, SuperAdmin is added to db_owner to perform security-sensitive DDL/DML (creating schemas, keys, policies, etc.) required by Requirement 2, without elevating operational roles.

```
USE MedicalInfoSystem;
GO
SELECT name, type_desc
FROM sys.database_principals
WHERE name IN ('SuperAdmin','D001','D002','N001','N002','P001',
               'SuperAdmins','Doctors','Nurses','Patients');
```

Figure 75: User and Role Identity Checking

	name	type_desc
1	SuperAdmin	SQL_USER
2	D001	SQL_USER
3	D002	SQL_USER
4	N001	SQL_USER
5	N002	SQL_USER
6	P001	SQL_USER
7	SuperAdmi...	DATABAS...
8	Doctors	DATABAS...
9	Nurses	DATABAS...
10	Patients	DATABAS...

Figure 76: User and Role Identity

Figure 75 queries sys.database_principals to list the specified principals and their types. As shown in Figure 76, SuperAdmin, D001, D002, N001, N002, P001 are SQL_USER accounts (users in this database), while SuperAdmins, Doctors, Nurses, Patients are database roles (DATABASE_ROLE), confirming both the user identities and the roles defined in the database.

```
-- SuperAdmin has db_owner?
EXEC AS LOGIN='SuperAdmin';
SELECT IS_ROLEMEMBER('db_owner') AS IsDbOwner;
REVERT;
```

	IsDbOwner
1	1

Figure 77: Permission of SuperAdmin Checking

Query in Figure 77 impersonates the SuperAdmin login and asks SQL Server whether that principal is a member of the db_owner role in the current database using. The output IsDbOwner = 1 means SuperAdmin *is* a member of db_owner in this database, proving the solution's success. Thus, the result of Figure 76 and Figure 77 proves the role framework exists from the base schema and that SuperAdmin has a privileged, auditable maintenance path as required.

3.2.4 Solution Summary

The provisioning approach in Section 3.2.1 was designed to establish unique identities for every user, so that each action could be attributed to an individual principal. Server logins were created in bulk with password policy enabled, and a safe default database was set so

connections would open in the correct context. Idempotent patterns were used so repeated executions would converge on a consistent state without manual correction. As a result, authentication was stabilized and accountability was strengthened from the first connection onward.

In 3.2.2, authorization was centralized through database users and roles, so privileges would be granted once and inherited by members. By mapping logins to users and grouping them under Doctors, Nurses, Patients, and SuperAdmins, permission changes were simplified and least privilege was enforced uniformly. Reviews and audits were made more reliable because grants were attached to roles rather than scattered across individual accounts. Subsequent object permissions, row-level rules, and decryption rights could then be reasoned about with clarity.

The design in 3.2.3 established a controlled maintenance path, allowing sensitive DDL and DML to be executed only by SuperAdmin, while operational users remained constrained to their designated duties. Creation of roles at initialization ensured predictable prerequisites for downstream scripts. By concentrating powerful capabilities under a single audited principal, the attack surface was reduced and administrative actions were performed with clear accountability. In combination, these three steps produced a secure and scalable foundation upon which all later permission rules could be enforced consistently.

3.3 Least-Privilege Object Access (Requirements 10 & 13 & 15 & 16 & 17 & 19 & 21)

Requirement 10 - All authenticated users must be able to see all staff name and office phone numbers only

Requirement 13 - Nurses and doctors must be able to see all patients' name and phone numbers

Requirement 15 - Only nurses can add or cancel appointments for patients to see doctor

Requirement 16 - Nurses can cancel an appointment or update the appointment datetime but only if the doctor have not added any diagnosis details

Requirement 19 - Doctors must be able to see ALL patients diagnosis details

Requirement 21 - Nurses must not be able to see diagnosis details

The hospital database holds a mixture of directory fields and highly sensitive clinical content, so the surface presented to each role must be narrowed to only what is necessary for that role to function. If direct table access were granted, vertical overexposure of columns and horizontal overreach of actions would occur. A single SELECT * on staff or patient tables would reveal far more than the directory views intended by policy, and unguarded DML would permit actions that violate clinical workflow. It is therefore required that object boundaries be drawn so that reading and writing happen only through controlled shapes of data and only by the correct actors.

The directory requirements establish the first constraint. All authenticated users are expected to see staff name and office phone only, while clinicians are expected to see patient name and phone across the population. If raw tables were exposed, personal addresses and other sensitive attributes would be visible, defeating confidentiality. The problem is compounded by the diagnosis domain, where asymmetry is mandated. Doctors must see every patient’s diagnosis in plain text, while nurses must never see diagnosis content at all. Without purpose-built objects, any shared query surface would either hide too much for doctors or reveal too much for nurses, and ad hoc filters would be fragile and unenforceable.

Operational constraints create a second pressure point. Only nurses are permitted to add or cancel appointments, and only doctor-facing operations should touch diagnosis content. If procedures were not interposed and if EXECUTE rights were not scoped by role, an

authenticated user could attempt appointment management or diagnosis access outside of policy, relying on discipline rather than enforcement. In sum, the problem to be solved is the construction of object interfaces that expose the minimum necessary columns for each reader, that encapsulate writes behind role-scoped procedures, and that make it impossible for nurses to observe diagnosis data while still allowing doctors full visibility and nurses full responsibility for appointment operations.

3.3.1 Staff Directory & Self-Only Details via View (Requirement 10)

```
CREATE VIEW SecureData.vwStaff
AS
SELECT
    s.StaffID,
    --CASE WHEN USER_NAME() = s.StaffID THEN s.StaffID ELSE '**********' END AS StaffID,
    s.StaffName,
    s.OfficePhone,
    CASE
        WHEN USER_NAME() = s.StaffID THEN
            CONVERT(varchar(200),
                DecryptByKeyAutoCert(CERT_ID('Cert_StaffKEK'), NULL, s.HomeAddress_Enc)
            )
        ELSE '**********'
    END AS HomeAddress,
    CASE
        WHEN USER_NAME() = s.StaffID THEN
            CONVERT(varchar(50),
                DecryptByKeyAutoCert(CERT_ID('Cert_StaffKEK'), NULL, s.PersonalPhone_Enc)
            )
        ELSE '**********'
    END AS PersonalPhone,
    CASE
        WHEN USER_NAME() = s.StaffID THEN s.Position
        ELSE '**********'
    END AS Position
FROM SecureData.Staff AS s;
GO

GRANT SELECT, UPDATE ON OBJECT::SecureData.vwStaff TO Doctors;
GRANT SELECT, UPDATE ON OBJECT::SecureData.vwStaff TO Nurses;
GO
```

Figure 78: SecureData.vwStaff View

Figure 78 shows a single view, SecureData.vwStaff, that narrows exposure to the minimum set of columns needed for routine lookups while preserving self-service for staff. StaffName and OfficePhone are always shown so the directory remains useful, meeting the “name and office phone” requirement. For sensitive fields (HomeAddress, PersonalPhone, Position), USER_NAME() is compared to StaffID, and only when the row belongs to the connected user is decryption performed using DecryptByKeyAutoCert, otherwise a fixed mask is returned. Because **GRANT SELECT, UPDATE** is applied only to Doctors and Nurses on the **view** rather than on the base table, least-privilege is enforced and business logic is centralized. This creates a stable, auditable surface that prevents accidental “SELECT *” overexposure from the underlying table.

```
EXEC AS LOGIN='N001';
SELECT TOP 3 StaffID, StaffName, OfficePhone, HomeAddress, PersonalPhone, Position
FROM SecureData.vwStaff
ORDER BY StaffID;
REVERT;
```

Figure 79: Testing SecureData.vwStaff as Nurse

	StaffID	StaffName	OfficePhone	HomeAddress	PersonalPhone	Position
1	D001	Dr. Ooi	03-11112222	**********	**********	**********
2	D002	Dr. Bryan	03-11113333	**********	**********	**********
3	D003	Dr. D003	03-110003	**********	**********	**********

Figure 80: SecureData.vwStaff Output as Nurse

```
EXEC AS LOGIN='D001';
SELECT StaffID, StaffName, OfficePhone, HomeAddress, PersonalPhone, Position
FROM SecureData.vwStaff
WHERE StaffID='D001';
REVERT;
```

Figure 81: Testing SecureData.vwStaff as Doctor

	StaffID	StaffName	OfficePhone	HomeAddress	PersonalPhone	Position
1	D001	Dr. Ooi	03-11112222	12 Jalan Bukit, KL	012-3456789	Doctor

Figure 82: SecureData.vwStaff Output as Doctor

As shown in Figure 79, the query tries to view top 3 of the Staff table using the view as Nurse N001. As shown in Figure 80, the directory fields are readable for many staff but sensitive columns show as **********, proving the masking works. In Figure 81, the query tries to view the doctor's own row, which in fact returns plaintext for the encrypted columns, as shown in Figure 82, demonstrating selective, self-only decryption exactly as designed.

3.3.2 Patient Directory (Name + Phone) via View (Requirement 13)

```
CREATE VIEW SecureData.vwPatient
AS
SELECT
    p.PatientID,
    p.PatientName,
    CONVERT(varchar(50),
        DecryptByKeyAutoCert(CERT_ID('Cert_PatientKEK'), NULL, p.Phone_Enc)
    ) AS Phone
FROM SecureData.Patient AS p;
GO

-- Grants for the view
GRANT SELECT ON OBJECT::SecureData.vwPatient TO Nurses;
GRANT UPDATE ON OBJECT::SecureData.vwPatient TO Nurses;
GRANT SELECT ON OBJECT::SecureData.vwPatient TO Doctors;
-- Explicitly keep Patients off the view
REVOKE SELECT ON OBJECT::SecureData.vwPatient FROM Patients;
REVOKE UPDATE ON OBJECT::SecureData.vwPatient FROM Patients;
GO
```

Figure 83: SecureData.vwPatient View

Clinicians require a quick, minimal directory of patients and contact numbers, while patients must not enumerate other patients. Thus, the view shown in Figure 83 decrypts only the Phone column through DecryptByKeyAutoCert and omits home address and other PII. Access is

strictly role-scoped, where Doctors and Nurses get **SELECT** and Nurses additionally receive **UPDATE** for their allowed edits, while Patients are explicitly **REVOKEd** to prevent any directory read. Placing grants at the view rather than the table ensures that only the curated shape of data is exposed.

```
EXEC AS LOGIN='D001';
SELECT TOP 5 PatientID, PatientName, Phone
FROM SecureData.vwPatient
ORDER BY PatientID;
REVERT;
```

Figure 84: Testing SecureData.vwPatient as Doctor

	PatientID	PatientName	Phone
1	P001	Ali Musa	012-3456789
2	P002	Patient P002	012-0000002
3	P003	Patient P003	012-0000003
4	P004	Patient P004	012-0000004
5	P005	Patient P005	012-0000005

Figure 85: SecureData.vwPatient Output as Doctor

```
EXEC AS LOGIN='P001';
BEGIN TRY
    SELECT TOP 1 * FROM SecureData.vwPatient;
END TRY
BEGIN CATCH
    SELECT ERROR_NUMBER() AS Err, ERROR_MESSAGE() AS Msg;
END CATCH
REVERT;
```

Figure 86: Testing SecureData.vwPatient as Patient

	Err	Msg
1	229	The SELECT permission was denied on the object 'v...

Figure 87: SecureData.vwPatient Output as Patient

Query in Figure 84 tries to use the SecureData.vwPatient view as a doctor and as shown in Figure 85, it returns rows with decrypted phone numbers, showing clinicians' read access works as intended. However in Figure 86, when the patient tries to do the same, their attempt returns an error due to the explicit REVOKE, proving that non-clinician principals cannot read the directory.

3.3.3 Separation of Appointment Views for Nurses vs Doctors (Requirements 19 & 21)

```
CREATE VIEW SecureData.vwAppointments_Nurse
AS
SELECT
    ad.DiagID,
    ad.AppDateTime,
    ad.PatientID,
    p.PatientName,
    ad.DoctorID,
    s.StaffName AS DoctorName,
    CASE
        WHEN ad.DiagDetails_Enc IS NULL
            THEN CAST(NULL AS nvarchar(max))
        ELSE N'**********'
    END AS Diagnosis
FROM SecureData.AppointmentAndDiagnosis AS ad
JOIN SecureData.Patient AS p ON p.PatientID = ad.PatientID
JOIN SecureData.Staff   AS s ON s.StaffID   = ad.DoctorID;
GO
```

Figure 88: SecureData.vwAppointments_Nurse View

```
CREATE VIEW SecureData.vwAppointments_Doctor
AS
SELECT
    ad.DiagID,
    ad.AppDateTime,
    ad.PatientID,
    p.PatientName,
    ad.DoctorID,
    s.StaffName AS DoctorName,
    CONVERT(nvarchar(max), DecryptByCert(CERT_ID('Cert_Diag'), ad.DiagDetails_Enc)) AS Diagnosis
FROM SecureData.AppointmentAndDiagnosis ad
JOIN SecureData.Patient p ON p.PatientID = ad.PatientID
JOIN SecureData.Staff   s ON s.StaffID   = ad.DoctorID;
GO
```

Figure 89: SecureData.vwAppointments_Doctor

```
-- Grants
GRANT SELECT ON OBJECT::SecureData.vwAppointments_Nurse  TO Nurses;
GRANT SELECT ON OBJECT::SecureData.vwAppointments_Doctor TO Doctors;
-- Keep others off these views
REVOKE SELECT ON OBJECT::SecureData.vwAppointments_Nurse  FROM Patients;
REVOKE SELECT ON OBJECT::SecureData.vwAppointments_Doctor FROM Patients, Nurses;
GO
```

Figure 90: Grant and Revoke Permission by Role

The two views, as shown in Figure 88 and Figure 89 carve out distinct read surfaces over the same table. The doctor view joins patient and doctor names and uses DecryptByCert to return diagnosis text in plaintext. The nurse view returns the same structural information but replaces any non-NULL diagnosis with a fixed mask, or null if still unset. This design satisfies asymmetric visibility, where doctors must see all diagnosis content for clinical decision making, while nurses are intentionally prevented from viewing diagnosis details at all. Query in Figure 90 then grants restrict each view to its intended role and revoke access from others to prevent accidental crossover.

```
EXEC AS LOGIN='N001';
SELECT TOP 3 DiagID, AppDateTime, PatientID, DoctorID, Diagnosis
FROM SecureData.vwAppointments_Nurse
ORDER BY AppDateTime DESC;
REVERT;
```

Figure 91: Testing SecureData.vwAppointments_Nurse as Nurse

	DiagID	AppDateTime	PatientID	DoctorID	Diagnosis
1	2	2025-09-28 10:00:00.000	P002	D002	NULL
2	1	2025-09-20 12:09:03.307	P001	D001	**********

Figure 92: SecureData.vwAppointments_Nurse Output as Nurse

```
EXEC AS LOGIN='D001';
SELECT TOP 3 DiagID, AppDateTime, PatientID, DoctorID, Diagnosis
FROM SecureData.vwAppointments_Doctor
ORDER BY AppDateTime DESC;
REVERT;
```

Figure 93: Testing SecureData.vwAppointments_Doctor as Doctor

	DiagID	AppDateTime	PatientID	DoctorID	Diagnosis
1	2	2025-09-28 10:00:00.000	P002	D002	NULL
2	1	2025-09-20 12:09:03.307	P001	D001	Acute viral pharyngitis; rest, hydration, and an...

Figure 94: SecureData.vwAppointments_Doctor Output as Doctor

Figure 91 tested the SecureData.vwAppointments_Nurse and as shown in Figure 92, it shows either NULL if no diagnosis exists or a masked string, never plaintext. On the other hand, Figure 92 tested SecureData.vwAppointments_Doctor and as shown in Figure 94, the output shows decrypted diagnosis text for the doctor's own appointment and NULL for other doctors' appointments, evidencing correct least-privilege separation.

3.3.4 Nurse-Only Appointment Operations via EXECUTE Grants (Requirements 15 & 16 & 21)

```
CREATE PROCEDURE SecureData.usp_Nurse_AddAppointment
    @PatientID   varchar(6),
    @DoctorID    varchar(6),
    @AppDateTime datetime
AS
BEGIN
    SET NOCOUNT ON;
    SET XACT_ABORT ON;

    IF IS_MEMBER('Nurses') <> 1 AND IS_MEMBER('SuperAdmins') <> 1
    BEGIN
        RAISERROR('Only Nurses may add appointments.', 16, 1);
        RETURN;
    END

    -- Validate Patient exists
    IF NOT EXISTS (SELECT 1 FROM SecureData.Patient WHERE PatientID = @PatientID)
    BEGIN
        RAISERROR('PatientID not found.', 16, 1); RETURN;
    END
    -- Validate Doctor exists and is a Doctor
    IF NOT EXISTS (SELECT 1 FROM SecureData.Staff WHERE StaffID = @DoctorID AND Position = 'Doctor')
    BEGIN
        RAISERROR('DoctorID not found or not a Doctor.', 16, 1); RETURN;
    END

    INSERT INTO SecureData.AppointmentAndDiagnosis
        (AppDateTime, PatientID, DoctorID, DiagDetails_Enc)
    VALUES
        (@AppDateTime, @PatientID, @DoctorID, NULL);

    SELECT SCOPE_IDENTITY() AS NewDiagID;
END
GO
```

Figure 95: SecureData.usp_Nurse_AddAppointment

```
CREATE PROCEDURE SecureData.usp_Nurse_CancelAppointment
    @DiagID int
AS
BEGIN
    SET NOCOUNT ON;
    SET XACT_ABORT ON;

    IF IS_MEMBER('Nurses') <> 1 AND IS_MEMBER('SuperAdmins') <> 1
    BEGIN
        RAISERROR('Only Nurses may cancel appointments.', 16, 1);
        RETURN;
    END

    DELETE FROM SecureData.AppointmentAndDiagnosis
     WHERE DiagID = @DiagID
       AND DiagDetails_Enc IS NULL;

    IF @@ROWCOUNT = 0
        RAISERROR('Cannot cancel: diagnosis already exists or appointment not found.', 16, 1);
END
GO
```

Figure 96: SecureData.usp_Nurse_CancelAppointment

```
CREATE PROCEDURE SecureData.usp_Nurse_UpdateAppointment
    @DiagID       int,
    @NewDateTime  datetime
AS
BEGIN
    SET NOCOUNT ON;
    SET XACT_ABORT ON;

    IF IS_MEMBER('Nurses') <> 1 AND IS_MEMBER('SuperAdmins') <> 1
    BEGIN
        RAISERROR('Only Nurses may update appointments.', 16, 1);
        RETURN;
    END

    UPDATE ad
       SET ad.AppDateTime = @NewDateTime
    FROM SecureData.AppointmentAndDiagnosis ad
    WHERE ad.DiagID = @DiagID
      AND ad.DiagDetails_Enc IS NULL;

    IF @@ROWCOUNT = 0
        RAISERROR('Cannot update time: diagnosis already exists or appointment not found.', 16, 1);
END
GO

GRANT EXECUTE ON SecureData.usp_Nurse_AddAppointment     TO Nurses;
GRANT EXECUTE ON SecureData.usp_Nurse_CancelAppointment  TO Nurses;
GRANT EXECUTE ON SecureData.usp_Nurse_UpdateAppointment  TO Nurses;
REVOKE EXECUTE ON SecureData.usp_Nurse_AddAppointment     FROM Doctors, Patients;
REVOKE EXECUTE ON SecureData.usp_Nurse_CancelAppointment  FROM Doctors, Patients;
REVOKE EXECUTE ON SecureData.usp_Nurse_UpdateAppointment  FROM Doctors, Patients;
GO
```

Figure 97: SecureData.usp_Nurse_UpdateAppointment and Respective Permission Control

Figure 95, Figure 96 and Figure 97 shows that all appointment writes are funnelled through stored procedures that first check IS_MEMBER('Nurses') and then enforce business rules. As shown in Figure 96 and Figure 97, cancel and update operations are allowed only when DiagDetails_Enc IS NULL, preventing time changes or cancellations after diagnosis has been added. EXECUTE rights are restricted to the Nurses role, leaving other roles unable to call these pathways. This ensures only nurses can perform appointment scheduling work and that those actions are safe with respect to diagnosis state.

```
-- Nurse can add an appointment (safe form)
EXEC AS LOGIN = 'N001';
DECLARE @dt datetime2 = DATEADD(DAY, 1, SYSUTCDATETIME());
EXEC SecureData.usp_Nurse_AddAppointment
    @PatientID   = 'P001',
    @DoctorID    = 'D001',
    @AppDateTime = @dt;
REVERT;

-- Doctor cannot call the nurse procedure
EXEC AS LOGIN='D001';
BEGIN TRY
    EXEC SecureData.usp_Nurse_AddAppointment
    PatientID='P001',
    @DoctorID='D001',
    @AppDateTime=@dt;
END TRY
BEGIN CATCH
    SELECT ERROR_NUMBER() AS Err, ERROR_MESSAGE() AS Msg; -- expect permission or role error
END CATCH
REVERT;
```

Figure 98: Testing SecureData.usp_Nurse_AddAppointment as Nurse and Doctor

	NewDiagID
1	4

Figure 99: SecureData.usp_Nurse_AddAppointment Output as Nurse

	Err	Msg
1	229	The EXECUTE permission was denied on the object ...

Figure 100: SecureData.usp_Nurse_AddAppointment Output as Doctor

Figure 98 shows query that tries to execute SecureData.usp_Nurse_AddAppointment as the nurse and doctor. As shown in Figure 99, the Nurse call succeeds and returns a new DiagID. However in Figure 100, the doctor's attempt fails with a permission denial, confirming that appointment manipulation is constrained to the Nurses role and that ad-hoc writes are impossible.

```
EXEC AS LOGIN='N001';
SELECT DiagID, AppDateTime, PatientID, DoctorID, Diagnosis
FROM SecureData.vwAppointments_Nurse
ORDER BY AppDateTime DESC;
REVERT;
```

	DiagID	AppDateTime	PatientID	DoctorID	Diagnosis
1	2	2025-09-28 10:00:00.000	P002	D002	NULL
2	5	2025-09-28 10:00:00.000	P001	D001	NULL
3	3	2025-09-20 16:48:51.610	P001	D001	**********
4	1	2025-09-20 12:09:03.307	P001	D001	**********

Figure 101: Current AppointmentAndDiagnosis Table

```
-- Try Cancel Appointment with NULL Diagnosis
EXEC AS LOGIN = 'N001';
EXEC SecureData.usp_Nurse_CancelAppointment @DiagID = '5';
REVERT;

-- Try Cancel Appointment with NON-NULL Diagnosis
EXEC AS LOGIN = 'N001';
EXEC SecureData.usp_Nurse_CancelAppointment @DiagID = '3';
REVERT;
```

Figure 102: Testing SecureData.usp_Nurse_CancelAppointment

```
Commands completed successfully.

Completion time: 2025-09-20T01:00:44.7643638+08:00
```

Figure 103: SecureData.usp_Nurse_CancelAppointment Output for DiagID = 5

```
Msg 50000, Level 16, State 1, Procedure SecureData.usp_Nurse_CancelAppointment, Line 20 [Batch Start Line 44]
Cannot cancel: diagnosis already exists or appointment not found.

Completion time: 2025-09-20T01:02:20.7394695+08:00
```

Figure 104: SecureData.usp_Nurse_CancelAppointment Output for DiagID = 3

	DiagID	AppDateTime	PatientID	DoctorID	Diagnosis
1	2	2025-09-28 10:00:00.000	P002	D002	NULL
2	3	2025-09-20 16:48:51.610	P001	D001	**********
3	1	2025-09-20 12:09:03.307	P001	D001	**********

Figure 105: Final AppointmentAndDiagnosis Table after Cancel

Figure 101 shows the current AppointmentAndDiagnosis Table (at the point of documenting this part) and Figure 102 shows query that tries to Cancel Appointment on DiagID = 5 (Null Diagnosis) and DiagID = 3 (Non-Null Diagnosis). As shown in Figure 103, the cancel appointment action on appointment with Null Diagnosis is successful (DiagID = 5) while as shown in Figure 104, the cancel appointment action on appointment with Non-Null Diagnosis is rejected with error message (DiagID = 5). This shows that the cancel appointment operation can only be done on appointments with Non-Null Diagnosis, fulfilling the requirement. Figure 105 shows the final table after change, where DiagID = 5 is removed but DiagID = 3 isn't.

```
EXEC AS LOGIN='N001';
SELECT DiagID, AppDateTime, PatientID, DoctorID, Diagnosis
FROM SecureData.vwAppointments_Nurse
ORDER BY AppDateTime DESC;
REVERT;
```

	DiagID	AppDateTime	PatientID	DoctorID	Diagnosis
1	2	2025-09-28 10:00:00.000	P002	D002	NULL
2	3	2025-09-20 16:48:51.610	P001	D001	**********
3	1	2025-09-20 12:09:03.307	P001	D001	**********

Figure 106: Current AppointmentAndDiagnosis Table

```
-- Try Update Appointment with NULL Diagnosis
EXEC AS LOGIN = 'N001';
DECLARE @NewDateTime datetime  = '2025-09-22T10:30:00';
EXEC SecureData.usp_Nurse_UpdateAppointment
     @DiagID      = 2,
     @NewDateTime = @NewDateTime;
REVERT

-- Try Update Appointment with NON-NULL Diagnosis
EXEC AS LOGIN = 'N001';
DECLARE @NewDateTime2 datetime  = '2025-09-22T10:30:00';
EXEC SecureData.usp_Nurse_UpdateAppointment
     @DiagID      = 3,
     @NewDateTime = @NewDateTime2;
REVERT
```

Figure 107: Testing SecureData.usp_Nurse_UpdateAppointment

```
Commands completed successfully.

Completion time: 2025-09-20T01:12:44.9941218+08:00
```

Figure 108: SecureData.usp_Nurse_UpdateAppointment Output for DiagID = 2

```
Msg 50000, Level 16, State 1, Procedure SecureData.usp_Nurse_UpdateAppointment, Line 23 [Batch Start Line 62]
Cannot update time: diagnosis already exists or appointment not found.

Completion time: 2025-09-20T01:13:17.0923912+08:00
```

Figure 109: SecureData.usp_Nurse_UpdateAppointment Output for DiagID = 3

	DiagID	AppDateTime	PatientID	DoctorID	Diagnosis
1	2	2025-09-22 10:30:00.000	P002	D002	NULL
2	3	2025-09-20 16:48:51.610	P001	D001	**********
3	1	2025-09-20 12:09:03.307	P001	D001	**********

Figure 110: Final AppointmentAndDiagnosis Table after Update

Similarly with the SecureData.usp_Nurse_CancelAppointment, the update operation have the exact same setting as shown in the figures above, where you can see that Updating on appointments with Null Diagnosis is available (Figure 108) while Updating on appointments with Non-Null Diagnosis is not available (Figure 109). Figure 110 also shows the changes where DiagID = 2 (Null Diagnosis) successfully updated the date while DiagID = 3 date update is unsuccessful.

3.3.5 Doctor-Only Diagnosis Entry via EXECUTE Grant (Requirement 17 & 19)

```
CREATE PROCEDURE SecureData.usp_Doctor_SetDiagnosis
    @DiagID     int,
    @Diagnosis  nvarchar(max)
AS
BEGIN
    SET NOCOUNT ON;
    SET XACT_ABORT ON;

    IF IS_MEMBER('Doctors') <> 1 AND IS_MEMBER('SuperAdmins') <> 1
    BEGIN
        RAISERROR('Only Doctors may set diagnosis.', 16, 1);
        RETURN;
    END

    -- Must be the assigned doctor
    IF NOT EXISTS (
        SELECT 1
        FROM SecureData.AppointmentAndDiagnosis ad
        WHERE ad.DiagID = @DiagID
          AND (ad.DoctorID = SUSER_SNAME() OR IS_MEMBER('SuperAdmins') = 1)
    )
    BEGIN
        RAISERROR('Appointment not found or not assigned to you.', 16, 1);
        RETURN;
    END

    -- Update diagnosis (encrypt with Cert_Diag)
    UPDATE ad
       SET ad.DiagDetails_Enc = EncryptByCert(CERT_ID('Cert_Diag'),
                                CONVERT(varbinary(max), @Diagnosis))
    FROM SecureData.AppointmentAndDiagnosis ad
    WHERE ad.DiagID = @DiagID;

    -- Return the updated row (decrypted)
    SELECT
        ad.DiagID,
        ad.AppDateTime,
        ad.PatientID,
        p.PatientName,
        ad.DoctorID,
        s.StaffName AS DoctorName,
        CONVERT(nvarchar(max), DecryptByCert(CERT_ID('Cert_Diag'), ad.DiagDetails_Enc)) AS Diagnosis
    FROM SecureData.AppointmentAndDiagnosis ad
    JOIN SecureData.Patient p ON p.PatientID = ad.PatientID
    JOIN SecureData.Staff   s ON s.StaffID   = ad.DoctorID
    WHERE ad.DiagID = @DiagID;
END
GO

GRANT EXECUTE ON SecureData.usp_Doctor_SetDiagnosis TO Doctors;
REVOKE EXECUTE ON SecureData.usp_Doctor_SetDiagnosis FROM Nurses, Patients;
GO
```

Figure 111: SecureData.usp_Doctor_SetDiagnosis and Permission Control

Figure 111 shows that diagnosis DML is isolated behind a single procedure that first confirms the principal is a doctor or SuperAdmin, then checks ownership by matching DoctorID to SUSER_SNAME() for the supplied DiagID. Only then is the diagnosis written, and it is encrypted with the asymmetric certificate. Returning the row with DecryptByCert provides an immediate, auditable confirmation of what was saved. EXECUTE is limited to the Doctors role so nurses and patients cannot touch diagnosis content, aligning with least-privilege access.

```
EXEC AS LOGIN='D001';
SELECT DiagID, AppDateTime, PatientID, DoctorID, Diagnosis
FROM SecureData.vwAppointments_Doctor
ORDER BY AppDateTime DESC;
REVERT;
```

	DiagID	AppDateTime	PatientID	DoctorID	Diagnosis
1	2	2025-09-22 10:30:00.000	P002	D002	NULL
2	6	2025-09-20 17:27:39.460	P003	D001	NULL
3	3	2025-09-20 16:48:51.610	P001	D001	Acute viral pharyngitis; rest, hydration, and an...
4	1	2025-09-20 12:09:03.307	P001	D001	Acute pharyngitis

Figure 112: Current AppointmentAndDiagnosis Table

```
-- Doctor owning the appointment can set diagnosis
EXEC AS LOGIN='D001';
EXEC SecureData.usp_Doctor_SetDiagnosis @DiagID=6, @Diagnosis=N'Acute pharyngitis';
REVERT;

-- Nurse cannot call the doctor procedure
EXEC AS LOGIN='N001';
BEGIN TRY
    EXEC SecureData.usp_Doctor_SetDiagnosis @DiagID=1, @Diagnosis=N'Test';
END TRY
BEGIN CATCH
    SELECT ERROR_NUMBER() AS Err, ERROR_MESSAGE() AS Msg;
END CATCH
REVERT;
```

Figure 113: Testing SecureData.usp_Doctor_SetDiagnosis

	DiagID	AppDateTime	PatientID	PatientName	DoctorID	DoctorName	Diagnosis
1	6	2025-09-20 17:27:39.460	P003	Patient P003	D001	Dr. Ooi	Acute pharyngitis

Figure 114: SecureData.usp_Doctor_SetDiagnosis Output as Doctor

	Err	Msg
1	229	The EXECUTE permission was denied on the object ...

Figure 115: SecureData.usp_Doctor_SetDiagnosis Output as Nurse

	DiagID	AppDateTime	PatientID	DoctorID	Diagnosis
1	2	2025-09-22 10:30:00.000	P002	D002	NULL
2	6	2025-09-20 17:27:39.460	P003	D001	Acute pharyngitis
3	3	2025-09-20 16:48:51.610	P001	D001	Acute viral pharyngitis; rest, hydration, and an...
4	1	2025-09-20 12:09:03.307	P001	D001	Acute pharyngitis

Figure 116: Final AppointmentAndDiagnosis Table after Set Diagnosis

Query in Figure 113 shows doctor and nurse try to set diagnosis text using SecureData.usp_Doctor_SetDiagnosis and as shown in Figure 114, the doctor call succeeds and returns the row with decrypted diagnosis for verification. However, the nurse attempt fails, as shown in Figure 115, demonstrating that diagnosis entry is doctor-only and that procedures enforce both role and ownership constraints. Figure 116 also shows the results where the DiagID=6 row had updated Diagnosis Text.

3.3.6 Locking Down Base Tables (Support for all least-privilege items)

```
-------------------------------------------------------------
-- 6) Lock down base tables (least-privilege in later files)
-------------------------------------------------------------
REVOKE SELECT, INSERT, UPDATE, DELETE ON SecureData.Staff  FROM Doctors, Nurses, Patients;
REVOKE SELECT, INSERT, UPDATE, DELETE ON SecureData.Patient FROM Doctors, Nurses, Patients;
REVOKE SELECT, INSERT, UPDATE, DELETE ON SecureData.AppointmentAndDiagnosis FROM Doctors, Nurses, Patients;
GO
```

Figure 117: Lock Down Base Tables

As shown in Figure 117, by removing direct privileges on base tables, all reads and writes are forced through the curated objects above. This guarantees that directory projections, masking, decryption rules, and business constraints cannot be bypassed with ad-hoc SQL. It is a foundational least-privilege measure that keeps the data surface small and predictable.

```
EXEC AS LOGIN='D001';
BEGIN TRY
  SELECT TOP 1 * FROM SecureData.AppointmentAndDiagnosis;
END TRY
BEGIN CATCH
  SELECT ERROR_NUMBER() AS Err, ERROR_MESSAGE() AS Msg;
END CATCH
REVERT;
```

Figure 118: Test Select on AppointmentAndDiagnosis Table as Doctor

	Err	Msg
1	229	The SELECT permission was denied on the object '...

Figure 119: Output of Select on AppointmentAndDiagnosis Table

Figure 118 shows a doctor trying to access the AppointmentAndDiagnosis table directly, and the output shown in Figure 119 shows that a clinician cannot read directly from the base table and must instead use the role-specific views, confirming that least-privilege is enforced end-to-end.

3.3.7 Solution Summary

The least-privilege design in 3.2 was constructed so that each role is shown only what is necessary to perform its duties, and every other path is quietly closed. Staff information is served through a view that always presents directory fields, while sensitive attributes are revealed only when the row belongs to the connected staff member, therefore routine lookups remain convenient and privacy is preserved. Patient lookups are delivered through a compact directory that clinicians can use efficiently, while patients are prevented from enumerating other patients, which maintains confidentiality without disrupting clinical workflows. Diagnosis visibility is separated cleanly, doctors read decrypted diagnosis where required for care, nurses encounter masked values so clinical coordination is possible without exposure to

sensitive content. All modifications to appointments and diagnoses are funneled through stored procedures that verify role, confirm required preconditions, and return consistent shapes, so policy is enforced exactly where state changes occur and accidental violations are prevented. Direct access to base tables is removed, so ad hoc queries cannot bypass masking, decryption rules, or business constraints, and every interaction is forced through curated, auditable objects.

This arrangement is optimal because it reaches least privilege without sacrificing usability, it concentrates rules in a small number of views and procedures, it ensures encryption boundaries are respected by allowing decryption only within doctor surfaces and signed paths, and it makes cross-role mistakes unlikely. The surface area of the database is reduced, the intent of each object is clear, reviews and audits become simpler, and evidence of compliance with the requirements is produced naturally through the same interfaces that users employ every day.

3.4 Self-Service & Row Ownership Controls (Requirements 8 & 9 & 11 & 12 & 14 & 17 & 18 & 20)

Requirement 8 - Staffs must be able to see their own details in full and in plain text form no matter how it is stored in the DB.

Requirement 9 - Staff must be able to update and verify their own details in full.

Requirement 11 - Patient must be able to see their own details in full and in plain text form no matter how it is stored in the DB.

Requirement 12 - Patient must be able to update and verify their own details in full.

Requirement 14 - Only nurses can update patient name and phone.

Requirement 17 - Doctors can only add diagnosis details after an appointment is scheduled.

Requirement 18 - Patients must be able to see all their own diagnosis records including appointment datetime, doctor name and diagnosis details.

Requirement 20 - A doctor must be able to update diagnosis details added by him/her only.

In the hospital database, each person is expected to view and maintain only their own information, while still experiencing the system as simple and responsive. Staff are expected to view and update their own records in full and in plain text, patients are expected to view and update their own records in full and in plain text, and both expectations must hold even though the underlying storage uses encryption for sensitive fields. If raw tables were queried directly, horizontal exposure would occur because a user could select or update rows that belong to someone else. Encrypted values could be mishandled or revealed in the wrong context. A stable link between a row and the authenticated principal must therefore be enforced on every read and write, so that self-access is enabled, cross-record access is denied, and plaintext is produced only for the rightful owner.

Clinical authorship introduces an additional constraint. Doctors are expected to update diagnosis entries that they authored, while being prevented from altering diagnoses written by other doctors. If ownership of diagnosis rows were not verified, an authenticated doctor could unintentionally or deliberately change another doctor's clinical note, which would compromise integrity and accountability. At the same time, patients are expected to see all of their own diagnosis records, including appointment time, doctor name, and diagnosis details, while being unable to see any other patient's information. These expectations require that row filters,

ownership checks, and controlled decryption be applied consistently to every path, so that self-service remains smooth, privacy is preserved, and authorship integrity is maintained across all clinical data.

3.4.1 Staff Self-Ownership with RLS and a View Trigger (Requirements 8 & 9)

```
CREATE VIEW SecureData.vwStaff
AS
SELECT
    s.StaffID,
    --CASE WHEN USER_NAME() = s.StaffID THEN s.StaffID ELSE '**********' END AS StaffID,
    s.StaffName,
    s.OfficePhone,
    CASE
        WHEN USER_NAME() = s.StaffID THEN
            CONVERT(varchar(200),
                DecryptByKeyAutoCert(CERT_ID('Cert_StaffKEK'), NULL, s.HomeAddress_Enc)
            )
        ELSE '**********'
    END AS HomeAddress,
    CASE
        WHEN USER_NAME() = s.StaffID THEN
            CONVERT(varchar(50),
                DecryptByKeyAutoCert(CERT_ID('Cert_StaffKEK'), NULL, s.PersonalPhone_Enc)
            )
        ELSE '**********'
    END AS PersonalPhone,
    CASE
        WHEN USER_NAME() = s.StaffID THEN s.Position
        ELSE '**********'
    END AS Position
FROM SecureData.Staff AS s;
GO

GRANT SELECT, UPDATE ON OBJECT::SecureData.vwStaff TO Doctors;
GRANT SELECT, UPDATE ON OBJECT::SecureData.vwStaff TO Nurses;
GO
```

Figure 120: SecureData.vwStaff

```
CREATE TRIGGER SecureData.tr_vwStaff_Update
ON SecureData.vwStaff
INSTEAD OF UPDATE
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @me sysname = USER_NAME();

    IF UPDATE(Position)
    BEGIN
        RAISERROR('Updating Position is not allowed via this view.', 16, 1);
        RETURN;
    END

    IF NOT EXISTS (SELECT 1 FROM inserted WHERE StaffID = @me)
    BEGIN
        RAISERROR('You can only update your own details.', 16, 1);
        RETURN;
    END

    DECLARE
        @StaffName      varchar(100),
        @OfficePhone    varchar(20),
        @HomeAddress    varchar(200),
        @PersonalPhone  varchar(50);

    SELECT TOP (1)
        @StaffName     = i.StaffName,
        @OfficePhone   = i.OfficePhone,
        @HomeAddress   = i.HomeAddress,
        @PersonalPhone = i.PersonalPhone
    FROM inserted AS i
    WHERE i.StaffID = @me;

    OPEN SYMMETRIC KEY SK_StaffPII DECRYPTION BY CERTIFICATE Cert_StaffKEK;

    UPDATE s
    SET
        s.StaffName   = @StaffName,
        s.OfficePhone = @OfficePhone,
        s.HomeAddress_Enc =
            CASE
                WHEN @HomeAddress IS NULL OR @HomeAddress = '**********'
                    THEN s.HomeAddress_Enc
                ELSE EncryptByKey(Key_GUID('SK_StaffPII'),
                        CONVERT(VARBINARY(4000), @HomeAddress))
            END,
        s.PersonalPhone_Enc =
            CASE
                WHEN @PersonalPhone IS NULL OR @PersonalPhone = '**********'
                    THEN s.PersonalPhone_Enc
                ELSE EncryptByKey(Key_GUID('SK_StaffPII'),
                        CONVERT(VARBINARY(4000), @PersonalPhone))
            END
    FROM SecureData.Staff AS s
    WHERE s.StaffID = @me;

    CLOSE SYMMETRIC KEY SK_StaffPII;
END
GO
```

Figure 121: SecureData.tr_vwStaff_Update

```
/* ============================================
   5) RLS: drop policy FIRST, then (re)create functions
   ============================================ */
/* Predicate: TRUE for owner (or SuperAdmins) - for UPDATE operations */
CREATE FUNCTION SecureData.fn_IsOwnStaffRow (@StaffID sysname)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN
    SELECT 1 AS fn_result
    WHERE @StaffID = USER_NAME()
       OR IS_MEMBER('SuperAdmins') = 1;
GO

/* Delete policy: ONLY SuperAdmins can delete (everyone else blocked) */
CREATE FUNCTION SecureData.fn_CanDeleteStaff (@StaffID sysname)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN
    SELECT 1 AS fn_result
    WHERE IS_MEMBER('SuperAdmins') = 1;
GO
```

Figure 122: RLS Functions

```
/* ============================================
   6) RLS policy (CREATE + ALTER pattern)
      - AFTER UPDATE      -> must own the row
      - BEFORE UPDATE     -> must own the row
      - BEFORE DELETE     -> only SuperAdmins may delete
   ============================================ */
CREATE SECURITY POLICY SecureData.StaffRlsPolicy
ADD BLOCK PREDICATE SecureData.fn_IsOwnStaffRow(StaffID)
    ON SecureData.Staff AFTER UPDATE
WITH (STATE = ON);
GO

ALTER SECURITY POLICY SecureData.StaffRlsPolicy
ADD BLOCK PREDICATE SecureData.fn_IsOwnStaffRow(StaffID)
    ON SecureData.Staff BEFORE UPDATE;
GO

ALTER SECURITY POLICY SecureData.StaffRlsPolicy
ADD BLOCK PREDICATE SecureData.fn_CanDeleteStaff(StaffID)
    ON SecureData.Staff BEFORE DELETE;
GO

-- Ensure policy is ON
ALTER SECURITY POLICY SecureData.StaffRlsPolicy WITH (STATE = ON);
GO
```

Figure 123: RLS Policies

Figure 120 above is the view SecureData.vwStaff that shows directory fields to everyone but reveals sensitive columns only on the caller's own row. It is GRANTed to Doctors and Nurses. Figure 121 is the SecureData.tr_vwStaff_Update, which is a trigger on SecureData.vwStaff where it allows changes only to the caller's own row and re-encrypts updated values, blocking changing position (INSTEAD OF UPDATE). Figure 122 and Figure 123, on the other hand, there are RLS (Row-Level Security) functions and policies that implement owner-only UPDATEs and DELETEs blocked except for SuperAdmins.

With these implementations, the staff experience is shaped around a single view that always returns directory fields, while sensitive attributes, such as home address, personal phone number, and position, are revealed only when the row belongs to the connected principal. This arrangement supports everyday lookups without leaking PII horizontally. The INSTEAD OF

UPDATE trigger on the view acts as the write-gate, where it confirms that the update statement targets the caller's own row, blocks disallowed attribute changes such as Position and re-encrypts any updated PII using the symmetric key so confidentiality is maintained at rest. Row-Level Security (RLS) reinforces these rules under the view by adding block predicates that allow UPDATE only by the row owner or SuperAdmin, and by reserving DELETE for SuperAdmin. The effect is a layered design where reads are safely filtered at projection time, writes are guarded at the view boundary, and RLS guarantees are enforced in the storage layer.

```
EXEC AS LOGIN='N001';
SELECT StaffID, StaffName, OfficePhone, HomeAddress, PersonalPhone, Position
FROM SecureData.vwStaff
ORDER BY StaffID;
REVERT;
```

Figure 124: Directory Read as Nurse

	StaffID	StaffName	OfficePhone	HomeAddress	PersonalPhone	Position
1	D001	Dr. Ooi	03-11112222	**********	**********	**********
2	D002	Dr. Bryan	03-11113333	**********	**********	**********
3	D003	Dr. D003	03-110003	**********	**********	**********
4	D004	Dr. D004	03-110004	**********	**********	**********
5	D005	Dr. D005	03-110005	**********	**********	**********
6	D006	Dr. D006	03-110006	**********	**********	**********
7	D007	Dr. D007	03-110007	**********	**********	**********
8	D008	Dr. D008	03-110008	**********	**********	**********
9	D009	Dr. D009	03-110009	**********	**********	**********
10	D010	Dr. D010	03-110010	**********	**********	**********
11	N001	Nurse Britney	03-22224444	55 Jalan Klang Lama, KL	017-8877665	Nurse
12	N002	Nurse Samantha	03-22225555	**********	**********	**********
13	N003	Nurse N003	03-220003	**********	**********	**********
14	N004	Nurse N004	03-220004	**********	**********	**********
15	N005	Nurse N005	03-220005	**********	**********	**********
16	N006	Nurse N006	03-220006	**********	**********	**********
17	N007	Nurse N007	03-220007	**********	**********	**********
18	N008	Nurse N008	03-220008	**********	**********	**********

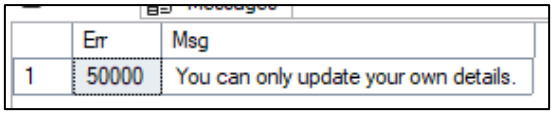
Figure 125: Output for Directory Read

```
-- Self update (N001 updates N001)
EXEC AS LOGIN='N001';
UPDATE s
  SET OfficePhone = '03-1234567'
FROM SecureData.vwStaff AS s
WHERE s.StaffID = 'N001';
REVERT;

-- Cross update (N001 tries to update D001) -> should be blocked
EXEC AS LOGIN='N001';
BEGIN TRY
  UPDATE s
    SET OfficePhone = '03-7654321'
  FROM SecureData.vwStaff AS s
  WHERE s.StaffID = 'D001';
END TRY
BEGIN CATCH
  SELECT ERROR_NUMBER() AS Err, ERROR_MESSAGE() AS Msg;
END CATCH
REVERT;
```

Figure 126: Self-Update and Cross-Update Test

```
(1 row affected)

Completion time: 2025-09-20T02:36:31.4035116+08:00
```

Figure 127: Self Update Output



	Err	Msg
1	50000	You can only update your own details.

Figure 128: Cross Update Output

	StaffID	StaffName	OfficePhone	HomeAddress	PersonalPhone	Position
1	D001	Dr. Ooi	03-11112222	**********	**********	**********
2	D002	Dr. Bryan	03-11113333	**********	**********	**********
3	D003	Dr. D003	03-110003	**********	**********	**********
4	D004	Dr. D004	03-110004	**********	**********	**********
5	D005	Dr. D005	03-110005	**********	**********	**********
6	D006	Dr. D006	03-110006	**********	**********	**********
7	D007	Dr. D007	03-110007	**********	**********	**********
8	D008	Dr. D008	03-110008	**********	**********	**********
9	D009	Dr. D009	03-110009	**********	**********	**********
10	D010	Dr. D010	03-110010	**********	**********	**********
11	N001	Nurse Britney	03-1234567	55 Jalan Klang Lama, KL	017-8877665	Nurse
12	N002	Nurse Samantha	03-22225555	**********	**********	**********
13	N003	Nurse N003	03-220003	**********	**********	**********
14	N004	Nurse N004	03-220004	**********	**********	**********

Figure 129: After Update Table

In the query shown in Figure 124, as Nurse N001 tries to read the entire table using the view, directory fields appear for many staff, while HomeAddress, PersonalPhone, and Position display masked values except on the caller's own row, as shown in Figure 125, demonstrating selective disclosure. In Figure 126, where the Nurse N001 tries to self-update and cross-update. The self-update completes successfully as shown in Figure 127 because the trigger accepts the change and re-encrypts where needed. However, the cross-update returns a user-friendly error "You can only update your own details" as shown in Figure 128, proving the view trigger and RLS block predicates are correctly preventing horizontal writes. Figure 129 shows that the change of OfficePhone by Nurse N001 is successful.

3.4.2 Patient Self-Service with RLS Filters and Module-Signed Procedures (Requirements 11 & 12 & 14 & 18)

```
/* ========================================================
   5) Consolidated RLS for Patient table (ONE policy)
   ======================================================== */
-- FILTER: Doctors/Nurses see all; Patients see self; Admin/db_owner see all
CREATE FUNCTION SecureData.fn_Patient_VisibleToCareTeam (@PatientID sysname)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN
    SELECT 1 AS fn_result
    WHERE IS_MEMBER('Doctors') = 1
       OR IS_MEMBER('Nurses') = 1
       OR IS_MEMBER('SuperAdmins') = 1
       OR IS_MEMBER('db_owner') = 1
       OR @PatientID = SUSER_SNAME();
GO
```

Figure 130: fn_Patient_VisibleToCareTeam

```
-- UPDATE allowed: Nurses (and SuperAdmins)
CREATE OR ALTER FUNCTION SecureData.fn_Patient_UpdateAllowed (@PatientID sysname)
RETURNS TABLE WITH SCHEMABINDING AS
RETURN
    SELECT 1 AS fn_result
    WHERE IS_MEMBER('Nurses') = 1
       OR IS_MEMBER('SuperAdmins') = 1
       OR @PatientID = SUSER_SNAME()
       OR IS_MEMBER('db_owner') = 1;
GO

-- DELETE allowed: SuperAdmins only
CREATE FUNCTION SecureData.fn_Patient_DeleteAllowed (@PatientID sysname)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN
    SELECT 1 AS fn_result
    WHERE IS_MEMBER('SuperAdmins') = 1;
GO
```

Figure 131: fn_Patient_UpdateAllowed and fn_Patient_DeleteAllowed

```
-- Create ONE policy and add predicates
CREATE SECURITY POLICY SecureData.Policy_Patient_RLS
ADD FILTER PREDICATE SecureData.fn_Patient_VisibleToCareTeam(PatientID)
    ON SecureData.Patient
WITH (STATE = ON);
GO

ALTER SECURITY POLICY SecureData.Policy_Patient_RLS
ADD BLOCK PREDICATE SecureData.fn_Patient_UpdateAllowed(PatientID)
    ON SecureData.Patient BEFORE UPDATE;
GO
ALTER SECURITY POLICY SecureData.Policy_Patient_RLS
ADD BLOCK PREDICATE SecureData.fn_Patient_UpdateAllowed(PatientID)
    ON SecureData.Patient AFTER UPDATE;
GO
ALTER SECURITY POLICY SecureData.Policy_Patient_RLS
ADD BLOCK PREDICATE SecureData.fn_Patient_DeleteAllowed(PatientID)
    ON SecureData.Patient BEFORE DELETE;
GO
ALTER SECURITY POLICY SecureData.Policy_Patient_RLS WITH (STATE = ON);
GO
```

Figure 132: RLS Block Predicates

```
/* ========================================================
   3) Patient self view (decrypt via module signing; newest first)
   ======================================================== */
CREATE PROCEDURE SecureData.usp_Patient_Diagnosis_ListSelf
AS
BEGIN
    SET NOCOUNT ON;

    SELECT
        ad.DiagID,
        ad.AppDateTime,
        ad.DoctorID,
        s.StaffName AS DoctorName,
        CONVERT(nvarchar(max), DecryptByCert(CERT_ID('Cert_Diag'), ad.DiagDetails_Enc)) AS Diagnosis
    FROM SecureData.AppointmentAndDiagnosis ad
    JOIN SecureData.Staff s ON s.StaffID = ad.DoctorID
    WHERE ad.PatientID = SUSER_SNAME()
    ORDER BY ad.AppDateTime DESC;
END
GO

GRANT EXECUTE ON SecureData.usp_Patient_Diagnosis_ListSelf TO Patients;
REVOKE EXECUTE ON SecureData.usp_Patient_Diagnosis_ListSelf FROM Doctors, Nurses;
GO
```

Figure 133: SecureData.usp_Patient_Diagnosis_ListSelf

Figure 130 shows the RLS filter function, fn_Patient_VisibleToCareTeam, which allows doctors and nurses to see all patients, a patient sees only themselves, while admins and db_owner see all. Figure 131 and Figure 132 shows the RLS filter functions fn_Patient_UpdateAllowed and fn_Patient_DeleteAllowed with the respective RLS block predicates, which ensure that UPDATEs are allowed to nurses, SuperAdmin, db_owner, or the patient self, while DELETEs are allowed only to SuperAdmin. The policy SecureData.Policy_Patient_RLS also attaches FILTER and BLOCK predicates to SecureData.Patient. Figure 133 on the other hand shows patient self-diagnosis list procedure SecureData.usp_Patient_Diagnosis_ListSelf, plus module signing so patients can decrypt their own diagnosis without direct certificate rights.

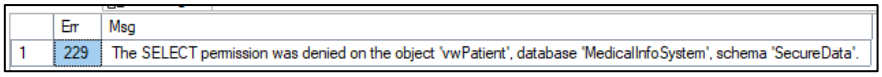
By implementing these practices, it ensures that as patients must enjoy a smooth self-service path, yet the care team still needs population-wide visibility. The RLS FILTER predicate enforces that doctors and nurses can query all rows for care delivery, while a patient can only see their own row by matching PatientID to the current security context. The BLOCK predicates protect writes, allowing nurses, SuperAdmin, db_owner, or the patient to perform name and phone updates, while DELETE is restricted to SuperAdmin as a safety measure. For diagnosis viewing, a dedicated patient procedure returns the caller's own diagnosis history. It decrypts it using DecryptByCert, but the ability to decrypt is granted to a special signer principal created from a certificate, not to the Patients role. The procedure is signed with that certificate, so the code path can decrypt even though end users have no certificate permissions. This preserves confidentiality while enabling the intended self-view.

```
-- Nurse sees a list of patients via the directory view (allowed)
EXEC AS LOGIN='N001';
SELECT TOP 5 PatientID, PatientName, Phone
FROM SecureData.vwPatient
ORDER BY PatientID;
REVERT;

-- Patient tries to read the directory view (blocked by your REVOKE)
EXEC AS LOGIN='P001';
BEGIN TRY
  SELECT TOP 1 * FROM SecureData.vwPatient;
END TRY
BEGIN CATCH
  SELECT ERROR_NUMBER() AS Err, ERROR_MESSAGE() AS Msg;
END CATCH
REVERT;

-- Patient uses the sanctioned self path (allowed, row-scoped, decrypted)
EXEC AS LOGIN='P001';
EXEC SecureData.usp_Patient_Self_Get;
REVERT;
```

Figure 134: Test Nurse and Patient View Patient Table

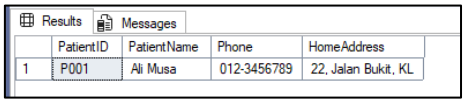
	PatientID	PatientName	Phone
1	P001	Ali Musa	012-3456789
2	P002	Patient P002	012-0000002
3	P003	Patient P003	012-0000003
4	P004	Patient P004	012-0000004
5	P005	Patient P005	012-0000005

Figure 135: Nurse View on Patient Table Output using SecureData.vwPatient



	Err	Msg
1	229	The SELECT permission was denied on the object 'vwPatient', database 'MedicalInfoSystem', schema 'SecureData'.

Figure 136: Patient try View on Patient Table Output using SecureData.vwPatient

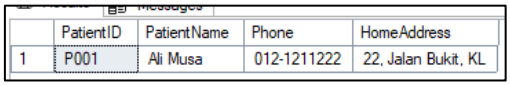


Results Messages

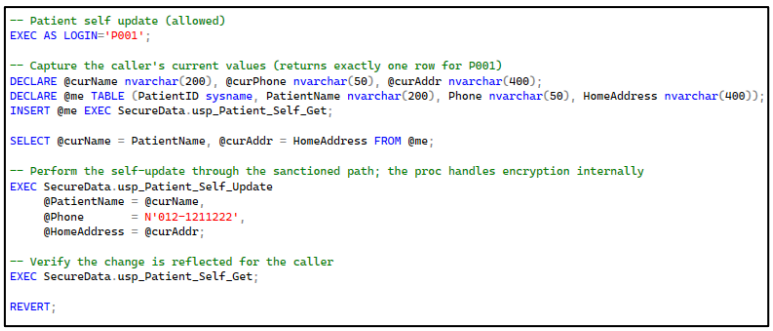
	PatientID	PatientName	Phone	HomeAddress
1	P001	Ali Musa	012-3456789	22, Jalan Bukit, KL

Figure 137: Patient View Table using SecureData.usp_Patient_Self_Get

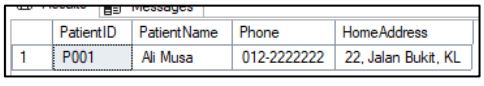
Figure 134 shows the testing query on Nurse viewing patient table using SecureData.vwPatient, Patient trying to view patient table using SecureData.vwPatient and Patient viewing patient table using SecureData.usp_Patient_Self_Get. The first viewing code in Figure 134 shows that clinician query using SecureData.vwPatient will returns multiple patients, as shown in Figure 135, confirming the care-team-wide filter. When patient tries to access using the same way, an error message as shown in Figure 136 will be prompted to show that patients cannot enumerate other patients through the directory surface. The last call in Figure 134, which shows the patient accessing the self-path stored procedure returns exactly one row for the caller with decrypted fields that belong to them, as shown in Figure 137, proving that the RLS filter and module-signed decrypt path enforce self-only access without giving patients any direct key or table permissions. The query returns a single row for the caller, and also shows that the RLS FILTER predicate trims horizontal visibility to self.



	PatientID	PatientName	Phone	HomeAddress
1	P001	Ali Musa	012-1211222	22, Jalan Bukit, KL



```
-- Patient self update (allowed)
EXEC AS LOGIN='P001';

-- Capture the caller's current values (returns exactly one row for P001)
DECLARE @curName nvarchar(200), @curPhone nvarchar(50), @curAddr nvarchar(400);
DECLARE @me TABLE (PatientID sysname, PatientName nvarchar(200), Phone nvarchar(50), HomeAddress nvarchar(400));
INSERT @me EXEC SecureData.usp_Patient_Self_Get;

SELECT @curName = PatientName, @curAddr = HomeAddress FROM @me;

-- Perform the self-update through the sanctioned path; the proc handles encryption internally
EXEC SecureData.usp_Patient_Self_Update
     @PatientName = @curName,
     @Phone       = N'012-1211222',
     @HomeAddress = @curAddr;

-- Verify the change is reflected for the caller
EXEC SecureData.usp_Patient_Self_Get;

REVERT;
```



	PatientID	PatientName	Phone	HomeAddress
1	P001	Ali Musa	012-2222222	22, Jalan Bukit, KL

Figure 138: Patient Self Update Test

```
-- Patient P001 tries to open the PII key and write an encrypted value for P002
EXEC AS LOGIN='P001';
BEGIN TRY
    -- Attempt to open the symmetric key needed for Phone_Enc updates
    OPEN SYMMETRIC KEY SK_PatientPII
        DECRYPTION BY CERTIFICATE Cert_PatientKEK;

    -- Attempt to update P002's encrypted phone (will not be reachable anyway)
    UPDATE p
       SET Phone_Enc = EncryptByKey(Key_GUID('SK_PatientPII'), CONVERT(varbinary(256), N'012-9999999'))
      FROM SecureData.Patient AS p
     WHERE p.PatientID = 'P002';

    SELECT 'RowsAffected' AS Info, @@ROWCOUNT AS RowsAffected;
END TRY
BEGIN CATCH
    SELECT
        ERROR_NUMBER()   AS Err,
        ERROR_SEVERITY() AS Sev,
        ERROR_STATE()    AS State,
        ERROR_MESSAGE()  AS Msg;
END CATCH
REVERT;
```

	Err	Sev	State	Msg
1	15151	16	1	Cannot find the symmetric key 'SK_PatientPII', because it does not exist or you do not have permission.

Figure 139: Patient Cross Update Test

	PatientID	PatientName	Phone
1	P001	Ali Musa	012-2222222

```
-- Nurse updates a patient's contact details (allowed path via the view)
EXEC AS LOGIN='N001';

-- Baseline: see current values through the clinician directory view
SELECT PatientID, PatientName, Phone
FROM SecureData.vwPatient
WHERE PatientID = 'P001';

-- Perform the update via the view so the INSTEAD OF UPDATE trigger handles encryption
UPDATE v
   SET PatientName = N'Patient One',
       Phone       = N'012-5555555'
  FROM SecureData.vwPatient AS v
 WHERE v.PatientID = 'P001';

-- Verify: the new values should appear in plaintext through the view
SELECT PatientID, PatientName, Phone
FROM SecureData.vwPatient
WHERE PatientID = 'P001';

REVERT;
```

	PatientID	PatientName	Phone
1	P001	Patient One	012-5555555

Figure 140: Nurse Update Patient Table Test

```
EXEC AS LOGIN='P001';
EXEC SecureData.usp_Patient_Diagnosis_ListSelf;
REVERT;
```

	DiagID	AppDateTime	DoctorID	DoctorName	Diagnosis
1	3	2025-09-20 16:48:51.610	D001	Dr. Ooi	Acute viral pharyngitis; rest, hydration, and an...
2	1	2025-09-20 12:09:03.307	D001	Dr. Ooi	Acute pharyngitis

Figure 141: Patient Diagnosis List Test via Signed Module

For the output shown in Figure 138, the self-update succeeds while the cross update returns a permission error, as shown in Figure 139, proving the BLOCK predicate enforces ownership and role rules. The nurse update, as shown in Figure 140 completes successfully, demonstrating the role-based write authority. The self-diagnosis procedure showed in Figure 141 returns the caller's diagnosis list with plaintext details and doctor names, confirming that decryption was performed by a signed module rather than by granting certificate rights to the Patients role.

3.4.3 Doctor Authorship Enforcement for Diagnosis Rows (Requirements 17 & 20)

```
CREATE PROCEDURE SecureData.usp_Doctor_SetDiagnosis
    @DiagID     int,
    @Diagnosis  nvarchar(max)
AS
BEGIN
    SET NOCOUNT ON;
    SET XACT_ABORT ON;

    IF IS_MEMBER('Doctors') <> 1 AND IS_MEMBER('SuperAdmins') <> 1
    BEGIN
        RAISERROR('Only Doctors may set diagnosis.', 16, 1);
        RETURN;
    END

    -- Must be the assigned doctor
    IF NOT EXISTS (
        SELECT 1
        FROM SecureData.AppointmentAndDiagnosis ad
        WHERE ad.DiagID = @DiagID
          AND (ad.DoctorID = SUSER_SNAME() OR IS_MEMBER('SuperAdmins') = 1)
    )
    BEGIN
        RAISERROR('Appointment not found or not assigned to you.', 16, 1);
        RETURN;
    END

    -- Update diagnosis (encrypt with Cert_Diag)
    UPDATE ad
       SET ad.DiagDetails_Enc = EncryptByCert(CERT_ID('Cert_Diag'),
                                              CONVERT(varbinary(max), @Diagnosis))
    FROM SecureData.AppointmentAndDiagnosis ad
    WHERE ad.DiagID = @DiagID;

    -- Return the updated row (decrypted)
    SELECT
        ad.DiagID,
        ad.AppDateTime,
        ad.PatientID,
        p.PatientName,
        ad.DoctorID,
        s.StaffName AS DoctorName,
        CONVERT(nvarchar(max), DecryptByCert(CERT_ID('Cert_Diag'), ad.DiagDetails_Enc)) AS Diagnosis
    FROM SecureData.AppointmentAndDiagnosis ad
    JOIN SecureData.Patient p ON p.PatientID = ad.PatientID
    JOIN SecureData.Staff   s ON s.StaffID   = ad.DoctorID
    WHERE ad.DiagID = @DiagID;
END
GO

GRANT EXECUTE ON SecureData.usp_Doctor_SetDiagnosis TO Doctors;
REVOKE EXECUTE ON SecureData.usp_Doctor_SetDiagnosis FROM Nurses, Patients;
GO
```

Figure 142: SecureData.usp_Doctor_SetDiagnosis

Figure 142 shows a stored procedure for setting diagnosis while restricting execution to Doctors or SuperAdmin, verifies the caller is the assigned doctor for the given DiagID and encrypts the diagnosis with Cert_Diag. This practice ensures that clinical authorship is enforced by placing all diagnosis updates behind one procedure that binds the write action to the doctor who owns the appointment. The procedure first asserts role membership, then checks the AppointmentAndDiagnosis row to confirm that the DoctorID matches the current security context. Only if that ownership test passes is the diagnosis encrypted and stored. This removes the possibility of a doctor altering another doctor's note, protects integrity of clinical documentation, and produces a uniform audit trail because every write flows through a single, predictable code path.

	DiagID	AppDateTime	PatientID	DoctorID	Diagnosis
1	9	2025-09-30 10:00:00.000	P005	D001	NULL
2	7	2025-09-27 10:00:00.000	P002	D001	NULL
3	2	2025-09-22 10:30:00.000	P002	D002	NULL
4	6	2025-09-20 17:27:39.460	P003	D001	**********
5	3	2025-09-20 16:48:51.610	P001	D001	**********
6	1	2025-09-20 12:09:03.307	P001	D001	**********
7	8	2025-09-16 10:00:00.000	P003	D002	NULL

Figure 143: Current AppointmentAndDiagnosis Table

```
-- Assigned doctor writes diagnosis
EXEC AS LOGIN='D001';
EXEC SecureData.usp_Doctor_SetDiagnosis @DiagID = 7, @Diagnosis = N'Post-viral cough';
REVERT;
```

Figure 144: Testing Assigned Doctor Write Diagnosis

	DiagID	AppDateTime	PatientID	PatientName	DoctorID	DoctorName	Diagnosis
1	7	2025-09-27 10:00:00.000	P002	Patient P002	D001	Dr. Ooi	Post-viral cough

Figure 145: Assigned Doctor Write Diagnosis Output

```
-- Non-owner doctor attempts to write
EXEC AS LOGIN='D001';
BEGIN TRY
  EXEC SecureData.usp_Doctor_SetDiagnosis @DiagID = 2, @Diagnosis = N'Test edit';
END TRY
BEGIN CATCH
  SELECT ERROR_NUMBER() AS Err, ERROR_MESSAGE() AS Msg;
END CATCH
REVERT;
```

Figure 146: Testing Non-Owner Doctor Attempts to Write Diagnosis

	Err	Msg
1	50000	Appointment not found or not assigned to you.

Figure 147: Non-Owner Doctor Attempts to Write Diagnosis Output

When the assigned doctor executes the procedure, as shown in Figure 144, the row is updated and the value is stored encrypted, as shown in Figure 145, proving that rightful authors can complete documentation. When a different doctor attempts the same change, as shown in Figure 146, the procedure raises an error such as “Appointment not found or not assigned to you,” as shown in Figure 147, confirming the authorship check prevents cross-doctor edits and that integrity of diagnosis entries is preserved.

3.4.4 Solution Summary

The self-service and row ownership design in 3.3 was arranged so that each user experiences simple, natural actions while the system quietly enforces strict boundaries. For staff, visibility of sensitive attributes is granted only on the owner’s row through a controlled view, and any edit is accepted only when the row belongs to the connected principal, with the view trigger re encrypting values so confidentiality at rest is preserved. For patients, a filter ensures that only the caller’s record is visible, while a signed self-update procedure performs the necessary decryption and re encryption on their behalf, so personal details can be reviewed and corrected

in plaintext without granting key rights or exposing other patients. For clinical notes, authorship is respected because diagnosis updates are allowed only by the doctor linked to the record, and attempts by other doctors are rejected, which protects integrity and accountability. Together these mechanisms prevent horizontal data leakage, keep all writes on deterministic and audited paths, maintain encryption boundaries, and still provide a smooth self-service experience for both staff and patients, which makes the solution both secure and usable under the assignment constraints.

3.5 Guarding Destructive Operations (Requirement 3)

Requirement 3 - Sufficient and suitable protection must be provided to protect all data from accidental and intentional exposure or deletion without compromising functionality and usability

Patient, staff, and diagnosis records represent critical clinical history, so accidental or malicious deletion would cause loss of continuity of care, break audit trails, and undermine trust. In a shared database where many authenticated users can run their own SQL, even a small mistake such as a missing filter or a misunderstood join could issue a wide delete, and a targeted malicious act could attempt to remove evidence by erasing rows. If direct delete capability were left available to operational roles, integrity and availability would be put at risk, because recovery would depend on out-of-band backups rather than being prevented at the point of action. At the same time, usability must not be harmed, routine updates must remain possible, and administrators must retain a controlled path to perform legitimate maintenance tasks.

The problem to be solved is to ensure that destructive operations are blocked by default for non-administrative users, that no alternative path exists through views or ad hoc statements, and that only a designated SuperAdmin can perform deletions in exceptional cases. The controls must resist both accidental execution and deliberate attempts, they must be enforced close to the data so bypass is not possible, and they must coexist with encryption, row filters, and least-privilege grants already in place. Clear denial behaviour is also required, so users receive immediate feedback when a deletion is not permitted, and the system remains predictable under stress or misuse.

3.5.1 Admin-Only Deletes with a Double Gate

```
DENY DELETE ON OBJECT::SecureData.Patient TO Doctors, Nurses, Patients;
GO
```

```
DENY DELETE ON OBJECT::SecureData.Staff TO Doctors;
DENY DELETE ON OBJECT::SecureData.Staff TO Nurses;
GO
```

```
DENY DELETE ON OBJECT::SecureData.AppointmentAndDiagnosis TO Doctors;
DENY DELETE ON OBJECT::SecureData.AppointmentAndDiagnosis TO Nurses;
GO
```

Figure 148: DENY DELETE on Patient, Staff and AppointmentAndDiagnosis Table

```
-- DELETE allowed: SuperAdmins only
CREATE FUNCTION SecureData.fn_Patient_DeleteAllowed (@PatientID sysname)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN
    SELECT 1 AS fn_result
    WHERE IS_MEMBER('SuperAdmins') = 1;
GO
```

```
-- Create ONE policy and add predicates
CREATE SECURITY POLICY SecureData.Policy_Patient_RLS
ADD FILTER PREDICATE SecureData.fn_Patient_VisibleToCareTeam(PatientID)
    ON SecureData.Patient
WITH (STATE = ON);
GO
ALTER SECURITY POLICY SecureData.Policy_Patient_RLS
ADD BLOCK PREDICATE SecureData.fn_Patient_DeleteAllowed(PatientID)
    ON SecureData.Patient BEFORE DELETE;
GO
```

Figure 149: RLS Delete Block and Block Predicate for Patient Table

```
/* Delete policy: ONLY SuperAdmins can delete (everyone else blocked) */
CREATE FUNCTION SecureData.fn_CanDeleteStaff (@StaffID sysname)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN
    SELECT 1 AS fn_result
    WHERE IS_MEMBER('SuperAdmins') = 1;
GO
```

```
CREATE SECURITY POLICY SecureData.StaffRlsPolicy
ADD BLOCK PREDICATE SecureData.fn_IsOwnStaffRow(StaffID)
    ON SecureData.Staff AFTER UPDATE
WITH (STATE = ON);
GO

ALTER SECURITY POLICY SecureData.StaffRlsPolicy
ADD BLOCK PREDICATE SecureData.fn_CanDeleteStaff(StaffID)
    ON SecureData.Staff BEFORE DELETE;
GO
```

Figure 150: RLS Delete Block and Block Predicate for Staff Table

Destructive operations are guarded by two independent gates so that no single mistake opens a path to data loss. As shown in Figure 148, at the permission layer, DENY DELETE is applied directly to each base table for all non-administrative roles, which means a simple DELETE is refused regardless of other grants. At the row-evaluation layer, a Row-Level Security BLOCK

predicate is attached to the table and evaluated at execution time, as shown Figure 149 and Figure 150, allowing a delete only when the caller is the SuperAdmin principal. This second gate protects even if a permission were misapplied or if a different object surface were used. Because both controls live at the data boundary, they cannot be bypassed with ad-hoc SQL, and yet a controlled, auditable path remains available for SuperAdmin to perform exceptional maintenance tasks.

```
-- Doctor tries to delete
EXEC AS LOGIN='D001';
BEGIN TRY
    DELETE FROM SecureData.Patient WHERE PatientID = 'P001';
END TRY
BEGIN CATCH
    SELECT ERROR_NUMBER() AS Err, ERROR_MESSAGE() AS Msg;
END CATCH
REVERT;
```

	Err	Msg
1	229	The SELECT permission was denied on the object 'Patient', database 'MedicalInfoSystem', schema 'SecureData'.

Figure 151: Doctor Tries to Delete Patient Directly

```
EXEC AS LOGIN='N001';
BEGIN TRY
    DELETE FROM SecureData.vwPatient WHERE PatientID = 'P001';
END TRY
BEGIN CATCH
    SELECT ERROR_NUMBER() AS Err, ERROR_MESSAGE() AS Msg;
END CATCH
REVERT;
```

	Err	Msg
1	229	The DELETE permission was denied on the object 'vwPatient', database 'MedicalInfoSystem', schema 'SecureData'.

Figure 152: Nurse Tries to Delete Patient through View

For Figure 151, the session is switched to a doctor and a direct DELETE is issued on the Patient base table. Due to the base table SecureData.Patient has SELECT revoked for clinicians, which act as the first gate that prevents the clinicians, for this case doctor, to delete patients from patient table. Figure 152 shows the nurse trying to issue a DELETE through views, which causes the DENY DELETE to prevent the operation at the permission layer, and your RLS block predicate permits deletion only for SuperAdmin, acting as the second gate. Either an error is caught that states the delete is not permitted like in Figure 152 that indicates that the statement reached execution but the RLS block predicate prevented any row from qualifying for deletion or SELECT permission was denied like in Figure 151, both outcomes prove that ordinary clinicians cannot delete patient data.

```
-- SuperAdmin deletes a disposable patient row: BEFORE → DELETE → AFTER
EXEC AS LOGIN='SuperAdmin';

-- BEFORE: quick snapshot of the table (top 10 for readability)
SELECT TOP 10
    PatientID,
    PatientName,
    CONVERT(varchar(50),
        DecryptByKeyAutoCert(CERT_ID(N'Cert_PatientKEK'), NULL, Phone_Enc)
    ) AS Phone,
    CONVERT(varchar(4000),
        DecryptByKeyAutoCert(CERT_ID(N'Cert_PatientKEK'), NULL, HomeAddress_Enc)
    ) AS HomeAddress
FROM SecureData.Patient
ORDER BY PatientID;
```

	PatientID	PatientName	Phone	HomeAddress
1	P001	Patient One	012-5555555	22, Jalan Bukit, KL
2	P002	Patient P002	012-0000002	No. 2, Jalan Pesakit, KL
3	P003	Patient P003	012-0000003	No. 3, Jalan Pesakit, KL
4	P004	Patient P004	012-0000004	No. 4, Jalan Pesakit, KL
5	P005	Patient P005	012-0000005	No. 5, Jalan Pesakit, KL
6	P006	Patient P006	012-0000006	No. 6, Jalan Pesakit, KL
7	P007	Patient P007	012-0000007	No. 7, Jalan Pesakit, KL
8	P008	Patient P008	012-0000008	No. 8, Jalan Pesakit, KL
9	P009	Patient P009	012-0000009	No. 9, Jalan Pesakit, KL
10	P010	Patient P010	012-0000010	No. 10, Jalan Pesakit,...

Figure 153: Current Patient Table

```
-- Ensure a disposable demo row exists
-- Open the symmetric key so EncryptByKey returns a ciphertext (not NULL)
OPEN SYMMETRIC KEY SK_PatientPII
    DECRYPTION BY CERTIFICATE Cert_PatientKEK;

IF NOT EXISTS (SELECT 1 FROM SecureData.Patient WHERE PatientID = N'P_ZAP')
BEGIN
    INSERT INTO SecureData.Patient (PatientID, PatientName, Phone_Enc, HomeAddress_Enc)
    VALUES (
        N'P_ZAP',
        N'Demo Patient',
        EncryptByKey(Key_GUID('SK_PatientPII'), CONVERT(varchar(50), N'012-0000000')),
        EncryptByKey(Key_GUID('SK_PatientPII'), CONVERT(varchar(4000), N'Demo Address'))
    );
END

-- Verify the inserted row (decrypt on the fly)
SELECT
    PatientID,
    PatientName,
    CONVERT(varchar(50),
        DecryptByKeyAutoCert(CERT_ID(N'Cert_PatientKEK'), NULL, Phone_Enc)
    ) AS Phone_Plain,
    CONVERT(varchar(4000),
        DecryptByKeyAutoCert(CERT_ID(N'Cert_PatientKEK'), NULL, HomeAddress_Enc)
    ) AS HomeAddress_Plain
FROM SecureData.Patient
WHERE PatientID = N'P_ZAP';

-- Close the symmetric key (good hygiene)
CLOSE SYMMETRIC KEY SK_PatientPII;
```

	PatientID	PatientName	Phone_Plain	HomeAddress_Plain
1	P_ZAP	Demo Patient	012-0000000	Demo Address

Figure 154: Insert Disposable Demo Row

```
-- DELETE: perform the admin-only deletion
DELETE FROM SecureData.Patient
WHERE PatientID = 'P_ZAP';
```

```
(1 row affected)

Completion time: 2025-09-20T13:21:43.5658247+08:00
```

Figure 155: Perform Deletion

```
-- AFTER: prove the row is gone
SELECT
    PatientID,
    PatientName,
    CONVERT(nvarchar(200),
        DecryptByKeyAutoCert(CERT_ID(N'Cert_PatientKEK'), NULL, Phone_Enc)
    ) AS Phone_Plain,
    CONVERT(nvarchar(400),
        DecryptByKeyAutoCert(CERT_ID(N'Cert_PatientKEK'), NULL, HomeAddress_Enc)
    ) AS HomeAddress_Plain
FROM SecureData.Patient
WHERE PatientID = N'P_ZAP';

-- Show a fresh snapshot after deletion
SELECT TOP 10
    PatientID,
    PatientName,
    CONVERT(varchar(50),
        DecryptByKeyAutoCert(CERT_ID(N'Cert_PatientKEK'), NULL, Phone_Enc)
    ) AS Phone,
    CONVERT(varchar(4000),
        DecryptByKeyAutoCert(CERT_ID(N'Cert_PatientKEK'), NULL, HomeAddress_Enc)
    ) AS HomeAddress
FROM SecureData.Patient
ORDER BY PatientID;
REVERT;
```

PatientID	PatientName	Phone_Plain	HomeAddress_Plain

	PatientID	PatientName	Phone	HomeAddress
1	P001	Patient One	012-5555555	22, Jalan Bukit, KL
2	P002	Patient P002	012-0000002	No. 2, Jalan Pesakit, KL
3	P003	Patient P003	012-0000003	No. 3, Jalan Pesakit, KL
4	P004	Patient P004	012-0000004	No. 4, Jalan Pesakit, KL
5	P005	Patient P005	012-0000005	No. 5, Jalan Pesakit, KL
6	P006	Patient P006	012-0000006	No. 6, Jalan Pesakit, KL
7	P007	Patient P007	012-0000007	No. 7, Jalan Pesakit, KL
8	P008	Patient P008	012-0000008	No. 8, Jalan Pesakit, KL
9	P009	Patient P009	012-0000009	No. 9, Jalan Pesakit, KL
10	P010	Patient P010	012-0000010	No. 10, Jalan Pesakit,...

Figure 156: Patient Table after Deletion

For under SuperAdmin, Figure 153 shows the current patient table, proving that SuperAdmin have access to open the key and view table directly through decryption key. Then, a temporary row is created and then deleted under SuperAdmin, as shown in Figure 154, Figure 155 and Figure 156. The confirmation query in Figure 156 returns zero remaining rows and the patient table exactly the same as before deletion, proving that the exception path for administrative maintenance works while all non-admin paths remain blocked.

3.5.2 Solution Summary

The approach to guarding destructive operations was arranged so that data could not be removed by accident or malice, while a controlled administrative path remained available for legitimate work. Deletion capability was first constrained at the permission boundary, where explicit denial ensured that ordinary roles could not issue a delete even if other grants existed, then a row level block was applied so that every attempted delete was evaluated against the caller, allowing success only when the session belonged to SuperAdmin. Because both gates were placed close to the data, no alternate route through views or ad hoc statements could bypass them, and the same behaviour was observed regardless of the tool being used, whether directly or through view. At the same time, SuperAdmin retained a predictable, auditable path to perform necessary maintenance, which preserved usability without weakening protection. Evidence was easy to produce, because denied attempts surfaced clear errors, successful administrative actions left straightforward before and after snapshots, and the model aligned with least privilege, integrity, and availability requirements.

3.6 SuperAdmin Maintenance Path (Requirement 2)

Requirement 2 - Superadmin must be able to perform appropriate DDL (such as create tables, views , logins, users , encryption keys etc) and DML tasks to maintain high level of functionality, availability and security of this DB

A hospital database must evolve, recover, and remain secure over time, which requires a trusted principal to create and modify core objects, to rotate and manage keys, to enable or adjust row level security, to seed roles and permissions, and to perform corrective maintenance when faults occur. If no dedicated maintenance path exists, everyday roles would either be over granted to perform one off administrative tasks, or critical administration would be delayed and improvised, both of which increase risk. Without a clear owner for DDL and sensitive DML, schema changes could be attempted from ad hoc accounts, encryption material could be mishandled, and fixes during incidents could be blocked by missing privileges, which would harm availability and integrity.

The problem therefore is to concentrate powerful capability in a single audited principal, so that creation of schemas, views, policies, logins, users, keys, and certificates can be executed reliably, while operational users remain least privileged. This path must be predictable, always available to the designated administrator, and separate from clinical roles, so routine workflows are never mixed with administrative authority. It must also interoperate with auditing, so that when maintenance is performed, a clear record of who acted and what changed is produced. In short, a controlled SuperAdmin path is needed to keep the system healthy, to implement and adjust security controls, and to do so without weakening the permission boundaries that protect day to day operations.

3.6.1 Privileged, Auditable Admin Channel via db_owner and Security Object Control

```
IF NOT EXISTS (SELECT 1 FROM sys.database_principals WHERE type='R' AND name='SuperAdmins') CREATE ROLE SuperAdmins;
```

```
IF NOT EXISTS (
    SELECT 1 FROM sys.database_role_members
    WHERE role_principal_id = USER_ID('SuperAdmins') AND member_principal_id = USER_ID('SuperAdmin'))
    ALTER ROLE SuperAdmins ADD MEMBER SuperAdmin;
```

```
-- Convenience (db_owner)
EXEC sp_addrolemember 'db_owner', 'SuperAdmin';
GO
```

Figure 157: SuperAdmin Schema

```
/* Predicate: TRUE for owner (or SuperAdmins) — for UPDATE operations */
CREATE FUNCTION SecureData.fn_IsOwnStaffRow (@StaffID sysname)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN
    SELECT 1 AS fn_result
    WHERE @StaffID = USER_NAME()
       OR IS_MEMBER('SuperAdmins') = 1;
GO

/* Delete policy: ONLY SuperAdmins can delete (everyone else blocked) */
CREATE FUNCTION SecureData.fn_CanDeleteStaff (@StaffID sysname)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN
    SELECT 1 AS fn_result
    WHERE IS_MEMBER('SuperAdmins') = 1;
GO

/* ================================================
   6) RLS policy (CREATE + ALTER pattern)
      - AFTER UPDATE      -> must own the row
      - BEFORE UPDATE     -> must own the row
      - BEFORE DELETE     -> only SuperAdmins may delete
   ================================================ */
CREATE SECURITY POLICY SecureData.StaffRlsPolicy
ADD BLOCK PREDICATE SecureData.fn_IsOwnStaffRow(StaffID)
    ON SecureData.Staff AFTER UPDATE
WITH (STATE = ON);
GO

ALTER SECURITY POLICY SecureData.StaffRlsPolicy
ADD BLOCK PREDICATE SecureData.fn_CanDeleteStaff(StaffID)
    ON SecureData.Staff BEFORE DELETE;
GO

ALTER SECURITY POLICY SecureData.StaffRlsPolicy
ADD BLOCK PREDICATE SecureData.fn_IsOwnStaffRow(StaffID)
    ON SecureData.Staff BEFORE UPDATE;
GO

-- Ensure policy is ON
ALTER SECURITY POLICY SecureData.StaffRlsPolicy WITH (STATE = ON);
GO
```

Figure 158: SuperAdmin Path for UPDATE and DELETE on Staff Table

```
-- FILTER: Doctors/Nurses see all; Patients see self; Admin/db_owner see all
CREATE FUNCTION SecureData.fn_Patient_VisibleToCareTeam (@PatientID sysname)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN
    SELECT 1 AS fn_result
    WHERE IS_MEMBER('Doctors') = 1
       OR IS_MEMBER('Nurses') = 1
       OR IS_MEMBER('SuperAdmins') = 1
       OR IS_MEMBER('db_owner') = 1
       OR @PatientID = SUSER_SNAME();
GO

-- UPDATE allowed: Nurses (and SuperAdmins)
CREATE OR ALTER FUNCTION SecureData.fn_Patient_UpdateAllowed (@PatientID sysname)
RETURNS TABLE WITH SCHEMABINDING AS
RETURN
    SELECT 1 AS fn_result
    WHERE IS_MEMBER('Nurses') = 1
       OR IS_MEMBER('SuperAdmins') = 1
       OR @PatientID = SUSER_SNAME()
       OR IS_MEMBER('db_owner') = 1;
GO

-- DELETE allowed: SuperAdmins only
CREATE FUNCTION SecureData.fn_Patient_DeleteAllowed (@PatientID sysname)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN
    SELECT 1 AS fn_result
    WHERE IS_MEMBER('SuperAdmins') = 1;
GO
```

```
-- Create ONE policy and add predicates
CREATE SECURITY POLICY SecureData.Policy_Patient_RLS
ADD FILTER PREDICATE SecureData.fn_Patient_VisibleToCareTeam(PatientID)
    ON SecureData.Patient
WITH (STATE = ON);
GO

ALTER SECURITY POLICY SecureData.Policy_Patient_RLS
ADD BLOCK PREDICATE SecureData.fn_Patient_DeleteAllowed(PatientID)
    ON SecureData.Patient BEFORE DELETE;
GO

ALTER SECURITY POLICY SecureData.Policy_Patient_RLS
ADD BLOCK PREDICATE SecureData.fn_Patient_UpdateAllowed(PatientID)
    ON SecureData.Patient BEFORE UPDATE;
GO
ALTER SECURITY POLICY SecureData.Policy_Patient_RLS
ADD BLOCK PREDICATE SecureData.fn_Patient_UpdateAllowed(PatientID)
    ON SecureData.Patient AFTER UPDATE;
GO
ALTER SECURITY POLICY SecureData.Policy_Patient_RLS
ADD BLOCK PREDICATE SecureData.fn_Patient_DeleteAllowed(PatientID)
    ON SecureData.Patient BEFORE DELETE;
GO
ALTER SECURITY POLICY SecureData.Policy_Patient_RLS WITH (STATE = ON);
GO
```

Figure 159: SuperAdmin Path for VIEW, UPDATE and DELETE on Patient Table

A dedicated maintenance principal is established so that sensitive DDL and guarded DML can be performed reliably without over granting operational roles. By placing SuperAdmin in

db_owner, as shown in Figure 157, full database-scope authority is provided to create and alter schemas, views, procedures, users, security policies, and other objects that keep the system healthy. This membership is complemented by explicit rights over security objects such as certificates and symmetric keys, ensuring that encryption material can be opened for maintenance, rotated safely, or re-granted when needed, as shown in Figure 158 and Figure 159 while ordinary roles remain unable to decrypt data directly. Because the capability is concentrated in one audited principal, administration becomes predictable and accountable, schema and policy changes occur through a single channel, and day-to-day users have the least privilege. The result is a clear separation of duties, a simple operational story for recoveries and upgrades, and a maintenance path that works consistently with your auditing and permission model.

```
-- SuperAdmin maintenance smoke test: role check → DDL → key use → user/role admin
EXEC AS LOGIN = 'SuperAdmin';

-- Prove SuperAdmin really has db_owner
SELECT DB_NAME() AS CurrentDB, SYSTEM_USER AS LoginName, IS_ROLEMEMBER('db_owner') AS IsDbOwner;
```

	CurrentDB	LoginName	IsDbOwner
1	MedicalInfoSystem	SuperAdmin	1

Figure 160: Prove SuperAdmin's db_owner Role

```
-- Perform safe DDL in your schema (create and drop a scratch table)
CREATE TABLE SecureData.__AdminSmoke__(Id int PRIMARY KEY, Note nvarchar(100));
INSERT INTO SecureData.__AdminSmoke__(Id, Note) VALUES (1, N'Hello from SuperAdmin');
SELECT TOP 1 * FROM SecureData.__AdminSmoke__;
DROP TABLE SecureData.__AdminSmoke__;
```

	Id	Note
1	1	Hello from SuperAdmin

Figure 161: Sample DDL Operation (CREATE table, INSERT Value, DROP Table)

```
-- Demonstrate encryption maintenance capability (open key, encrypt/decrypt a value on the fly)
OPEN SYMMETRIC KEY SK_PatientPII
    DECRYPTION BY CERTIFICATE Cert_PatientKEK;

DECLARE @cipher varbinary(4000) =
    EncryptByKey(Key_GUID('SK_PatientPII'), CONVERT(nvarchar(200), N'Key round-trip demo'));

SELECT
    CONVERT(nvarchar(200),
        DecryptByKeyAutoCert(CERT_ID(N'Cert_PatientKEK'), NULL, @cipher)
    ) AS DecryptedRoundTrip;

CLOSE SYMMETRIC KEY SK_PatientPII;
```

	DecryptedRoundTrip
1	Key round-trip demo

Figure 162: Prove Encryption Maintenance Capability

```
-- User and role administration in DB scope (create a contained user, add to role, then clean up)
IF NOT EXISTS (SELECT 1 FROM sys.database_principals WHERE name = N'__MaintUser')
    CREATE USER [__MaintUser] WITHOUT LOGIN;

IF NOT EXISTS (
    SELECT 1
    FROM sys.database_role_members drm
    WHERE drm.role_principal_id = USER_ID(N'Nurses')
      AND drm.member_principal_id = USER_ID(N'__MaintUser')
)
    ALTER ROLE Nurses ADD MEMBER [__MaintUser];

-- Show the membership just granted
EXEC sp_helprolemember N'Nurses';

-- Clean up the temporary principal
ALTER ROLE Nurses DROP MEMBER [__MaintUser];
DROP USER [__MaintUser];

REVERT;
```

	DbRole	MemberName	MemberSID
1	Nurses	__MaintUser	0x010500000000000903000000B2B147E37CFA49498000FD...
2	Nurses	N001	0x08CAF447C9491348AC934B4E2070BE99
3	Nurses	N002	0xF4E921F7916BD54F95C57EA18C25FB03
4	Nurses	N003	0xFACA13FC5750F04DAA48308756132443
5	Nurses	N004	0x5AC1EE2E00C6A942A477BE80F8E48DDF
6	Nurses	N005	0x847F5D61F25B90499120048F3F293569
7	Nurses	N006	0x1AF8840E724BCC449B007502AFBEEBC1
8	Nurses	N007	0x86300CA65972D344A8FEE3F1278C1A39
9	Nurses	N008	0x350C548DC715564C92B4659A5F1090A8
10	Nurses	N009	0x9CE30880D1A8E540A3B2C7001E13B844
11	Nurses	N010	0x64EC0E9F2DBC9548BEE56221E7D9AA3A

Figure 163: SuperAdmin's User and Role Administration Capability

The testing in this session is switched to SuperAdmin, the first query in Figure 160 confirms IsDbOwner = 1, which proves the principal holds the database-owner role and therefore has broad DDL capability. A scratch table is created, written, read, and dropped in the secure schema, as shown in Figure 161, which demonstrates that schema-level maintenance is possible without relying on elevated rights for operational roles. In Figure 162, the symmetric key is opened and a plaintext message is encrypted and immediately decrypted, which shows that SuperAdmin can perform key operations required for rotation, repair, and diagnostics, while ordinary users remain locked out. Finally, a contained user is created, added to the Nurses role, verified, and removed, as shown in Figure 163, which proves that SuperAdmin can administer principals and roles directly inside the database. Together, these outputs show that a single audited principal can execute maintenance, that encryption material is manageable without expanding end-user rights, and that database objects can be safely created, altered, and removed in support of ongoing operations.

3.6.2 Solution Summary

The maintenance model was designed to concentrate powerful actions in a single audited principal, which maintains administration as predictable, accountable, and secure. By placing SuperAdmin in db owner, schema and policy changes can be executed reliably, keys and certificates can be opened and rotated when required, and role or user adjustments can be made without improvisation. Explicit control over security objects ensures encryption material is handled correctly while ordinary roles remain unable to decrypt data, so confidentiality is preserved even during maintenance. Because administrative capability is separated from clinical work, least privilege is maintained for day to day users, emergency fixes and upgrades can proceed without delay, and every significant change is attributable to a single responsible identity. This design reduces operational risk, simplifies recovery, and keeps the system healthy over time, while avoiding the temptation to over grant permissions to operational roles.

4.0 Auditing

This section discusses how to capture, assign, and maintain activity at the APU Hospital database to enable administrators to discover, track, and verify security-related incidents. The system employs defence in depth and least privilege to ensure that evidence is captured, regardless of user permission restrictions. The result is a tamper-proof server-level copy, complete with quarriable paths at the database, suitable for both operational and forensic analyses (Microsoft SQL Server, 2025b).

At a high level, the solution combines four complementary mechanisms:

- **SQL Server Audit to File (Server/DB).** The system produces a digitally signed, append-only audit stream that captures security-critical activity at the engine boundary: successful and failed login **attempts**, permission and **role membership** changes, and comprehensive access to sensitive objects (*SELECT/INSERT/UPDATE/DELETE/EXECUTE*). Events are written out-of-band to rollover *.sqlaudit* files in a specified folder, yielding **tamper-evident** evidence suitable for incident response and compliance. Because this runs at the server level and the database audit specification funnels database-scope actions into the same audit, it gives you a single authoritative source for both successful access and failed attempts (Microsoft SQL Server, 2024g; Microsoft SQL Server, 2025b).
- **In-Database DDL Journal.** A database-level DDL trigger records object change in Audit.DDLAudit, capturing exactly what changed, where it changed, who changed it, and the precise T-SQL executed (Microsoft SQL Server, 2024d). To preserve least-privilege for normal users while keeping the journal authoritative, the trigger calls a small helper procedure that runs **EXECUTE AS OWNER**, so inserts succeed even when direct writes to the Audit schema are denied. The result is an exact, queryable ledger of *CREATE/ALTER/DROP* activity for root-cause analysis and auditing.
- **In-Database DML Journal.** Lightweight AFTER triggers on the SecureData tables record row-level *INSERT/UPDATE/DELETE*. Each event writes to Audit.DMLAudit with the action type, primary key (as JSON), and **before/after** images (as JSON), plus actor, application, host, and session metadata. These triggers also execute with owner context, so evidence is captured even under strict *DENY* rules in the Audit schema; temporal history tables are excluded to avoid duplication. This provides a forensic trail showing exactly how a row changed and who initiated the change.

- **System-Versioned Temporal Tables.** For key business entities (*SecureData.Patient, Staff, and AppointmentAndDiagnosis*), system versioning maintains complete row histories in dedicated *...History tables* with *ValidFrom/ValidTo* periods. This enables precise point-in-time reconstruction via *FOR SYSTEM_TIME* (e.g., AS OF, ALL) and complements the DML journal: triggers give high-fidelity per-statement snapshots, while temporal tables ensure every committed version remains queryable across time for reconstruction, verification, and audit sampling (Microsoft SQL Server, 2025a; Microsoft SQL Server, 2025c).

The combined design provides end-to-end accountability with coB audit serves as a single, append-only source for both successful and failed attempts; the DDL journal contributes exact T-SQL for structural changes; the DML journal provides row-level deltas for data changes; and temporal tables preserve point-in-time history for reconstruction. Evidence integrity is enforced by OWNER-context writers and DENY on the Audit schema, while writing to *.*sqlaudit* files and standardized audit tables keeps storage tamper-evident yet queryable for operations and forensics. The result aligns directly with the assignment drivers, like Req 6 (track all activities and attempts), Req 5 (login visibility), and Req 4 (recoverability and verifiable history) with low operational overhead and clear, screenshot-ready proof paths.

4.1 Audit Matrix

Requirements	Technique / Control	Scope	Events Captured (Attempts?)	Who & Context Captured	Evidence Store
6, 5, 4	**Server Audit → File + DB Audit Specification**	Server (all) + DB (*SecureData* schema)	**Logins** (success/fail), **DCL** (perm/role changes), **DDL** access, **DQL/DML** access on SecureData: SELECT/INSERT/UPDATE/DELETE/EXECUTE (includes failures/attempts) + Backup/Restore	*server_principal_name, database_name, schema_name, object_name, action_id, succeeded, statement*	**.sqlaudit** files in configured folder (discover via **sys.dm_server_audit_status.audit_file_path**; **read via sys.fn_get_audit_file**)
6 ***(supports 2)***	**In-DB DDL Journal (DB-level DDL trigger)**	Database	All **DDL** events (CREATE/ALTER/DROP …)	Actor (*SUSER_SNAME()*), TSql text, object schema/name/type, UTC timestamp	**Audit.DDLAudit** table (writes via helper proc *EXECUTE AS OWNER*)
6 ***(supports 4)***	**In-DB DML Journal (per-table AFTER triggers on *SecureData*)**	*SecureData* tables with PK (history tables excluded)	**Row changes**: INSERT/UPDATE/DELETE (normalized per statement)	Actor, AppName, HostName, SessionId; KeyJson (PK), BeforeJson/AfterJson (row images), UTC timestamp	**Audit.DMLAudit** table (triggers run *EXECUTE AS OWNER*)

4 ***(supports 1,3)***	**System-Versioned Temporal Tables**	*SecureData.Patient, Staff, AppointmentAndDiagnosis*	**All committed row versions across time** (automatic history)	Row version periods *ValidFrom/ValidTo*; reconstruct state AS OF any time	Base tables + **history tables** *SecureData.PatientHistory, StaffHistory, AppointmentAndDiagnosisHistory*

Table 3: Audit Matrix

The four-control audit matrix provides full coverage using SQL's canonical categories. Server Audit → File (with the DB Audit Specification) captures authentication and access at the engine boundary: login outcomes (LGIS/LGIF), DCL changes (GRANT/DENY/REVOKE and role membership), DDL access, and DQL/DML on the SecureData schema (SELECT/INSERT/UPDATE/DELETE/EXECUTE), with denied attempts surfaced by succeeded = 0, plus BACKUP/RESTORE; evidence resides in rollover, digitally signed .sqlaudit files (read via sys.fn_get_audit_file). Inside the database, the DDL journal (DB-level trigger) persists CREATE/ALTER/DROP with exact T-SQL, object metadata, and actor into Audit.DDLAudit. Row changes are captured by the DML journal (per-table AFTER triggers on SecureData PK tables), writing action type (I/U/D), KeyJson and BeforeJson/AfterJson with Actor/AppName/HostName/SessionId into Audit.DMLAudit. Finally, system-versioned temporal tables on SecureData.Patient, Staff, and AppointmentAndDiagnosis maintain committed versions (ValidFrom/ValidTo) in fixed history tables, enabling FOR SYSTEM_TIME ("AS OF"/"ALL") point-in-time reconstruction. Together these controls satisfy Req 4–6: breadth and attempts via Enterprise Audit, provenance via DDL/DML journals, and recoverability via temporal history.

4.2 Prove "all activities and attempts are tracked" (Requirements 5 & 6)

Requirement 5 - All users must be able to log into the MS-SQL system using SQL Server Management Studio and perform their own tasks. Assume that all users have sufficient knowledge on writing and running SQL queries in the query window.

Requirement 6 - All activities by all users (including attempt to do such activity) must be tracked.

The system capture both successful activities and unsuccessful attempts across the stack (server logins, backup/restore, and all object access in SecureData) with actor, object, time, and statement for accountability.

4.2.1 Server Audit → FILE + DB Audit Specification (breadth & attempts)

```
-- Server audit ON?
USE master;
SELECT name,is_state_enabled FROM sys.server_audits WHERE name='ServerAudit_MIS';
SELECT name,is_state_enabled FROM sys.server_audit_specifications WHERE name='ServerAuditSpec_MIS';

-- DB audit spec ON?
USE MedicalInfoSystem;
SELECT name,is_state_enabled FROM sys.database_audit_specifications WHERE name='DB_Audit_MIS';
```

Figure 164 : Status of Auditing Pipelines

	name	is_state_enabled
1	ServerAudit_MIS	1

	name	is_state_enabled
1	ServerAuditSpec_MIS	1

	name	is_state_enabled
1	DB_Audit_MIS	1

Figure 165 : Output of the status for Auditing Pipelines

Figure 164 shows the code to check the status of both Server and Database Audit.

- sys. server_audits → checks if ServerAudit_MIS is running (enterprise FILE target running).
- sys. server_audit_specifications → check if ServerAuditSpec_MIS is enabled (server-scope groups like login success/failure and backup/restore are associated with target).

- sys. database_audit_specifications (from MedicalInfoSystem) → checks if DB_Audit_MIS is active (database scoped caputure for SecureData, for obj access and DML/EXEC).

Figure 165 displays three partial result sets, one for each view intended, that all return the field audit object with *is_state_enabled = 1*.

- Once ServerAudit_MIS is enabled, the enterprise audit target collects events (Microsoft SQL Server, 2024d).
- Server level action (i.e., LGIS, LGIF, BA*, BAL) are logged when ServerAuditSpec_MIS is enabled (Microsoft SQL Server, 2024a).
- With DB_Audit_MIS on, objects accessed (specifically logged for both opens+menus and denied attempts) by SecureData are sent to the common enterprise audit stream (Microsoft SQL Server, 2024e).

4.2.1.1 Server Audit → FILE + Server Audit Specification

```
--=============================================================================
-- 0) SERVER AUDIT → FILE (safe default = SQL error-log folder)
--=============================================================================
USE master;
GO
DECLARE @UseErrorLogFolder bit = 0;   -- 1=reliable default; 0=use custom folder below
DECLARE @AuditDir nvarchar(4000);

IF @UseErrorLogFolder = 1
BEGIN
    DECLARE @ErrorLog nvarchar(4000) = CAST(SERVERPROPERTY('ErrorLogFileName') AS nvarchar(4000));
    DECLARE @BaseDir  nvarchar(4000) = LEFT(@ErrorLog, LEN(@ErrorLog) - CHARINDEX('\', REVERSE(@ErrorLog)) + 1);
    SET @AuditDir = @BaseDir + N'Audit\';
END
ELSE
BEGIN
    -- Make sure the SQL Server service account has write permission here:
    SET @AuditDir = N'C:\Users\user\Desktop\APU\APU Y3S2\DBS\Assignment\SQL_Audit\';
END

-- ensure trailing slash + create folder (best-effort)
SET @AuditDir = @AuditDir + CASE WHEN RIGHT(@AuditDir,1) IN ('\','/') THEN '' ELSE '\' END;
BEGIN TRY EXEC master.dbo.xp_create_subdir @AuditDir; END TRY BEGIN CATCH END CATCH;

-- clean + (re)create server audit and spec
IF EXISTS (SELECT 1 FROM sys.server_audit_specifications WHERE name=N'ServerAuditSpec_MIS')
BEGIN
  ALTER SERVER AUDIT SPECIFICATION [ServerAuditSpec_MIS] WITH (STATE=OFF);
  DROP  SERVER AUDIT SPECIFICATION [ServerAuditSpec_MIS];
END

IF EXISTS (SELECT 1 FROM sys.server_audits WHERE name=N'ServerAudit_MIS')
BEGIN
  ALTER SERVER AUDIT [ServerAudit_MIS] WITH (STATE=OFF);
  DROP  SERVER AUDIT [ServerAudit_MIS];
END

DECLARE @dirEsc nvarchar(4000) = REPLACE(@AuditDir,'''','''''');
EXEC (N'
CREATE SERVER AUDIT [ServerAudit_MIS]
  TO FILE (FILEPATH = N''' + @dirEsc + N''',
           MAXSIZE = 1 GB, MAX_ROLLOVER_FILES = 20)
  WITH (QUEUE_DELAY = 1000, ON_FAILURE = CONTINUE);');
```

Figure 166 : Enterprise audit readback—discovery and pattern resolution (code)

```sql
/* 1) Enterprise Audit file - robust discovery + readback */
/* ======================[ ENTERPRISE AUDIT EVIDENCE ]====================== */
/* ===== Enterprise Audit readback – handle file vs folder correctly ===== */
USE master;

DECLARE @path    nvarchar(4000) =
(SELECT TOP(1) audit_file_path
   FROM sys.dm_server_audit_status
  WHERE name = N'ServerAudit_MIS' AND status = 1);

IF @path IS NULL
BEGIN
  PRINT 'ServerAudit_MIS is not started or not found.';
  SELECT name, status_desc, audit_file_path FROM sys.dm_server_audit_status;
  RETURN;
END

-- If DMV returned a file (ends with .sqlaudit), extract its directory; else use the path as-is.
DECLARE @dir nvarchar(4000);
IF @path LIKE N'%.sqlaudit'
BEGIN
  DECLARE @lastSlash int = LEN(@path) - CHARINDEX('\', REVERSE(@path)) + 1; -- position of last '\'
  SET @dir = SUBSTRING(@path, 1, @lastSlash);  -- includes trailing '\'
END
ELSE
BEGIN
  SET @dir = @path + CASE WHEN RIGHT(@path,1) IN ('\','/') THEN '' ELSE '\' END;
END

DECLARE @pattern nvarchar(4000) = @dir + N'*.sqlaudit';

SELECT [audit_file_path_from_DMV]=@path, [folder_we_will_read]=@dir, [pattern]=@pattern;

-- Now read the audit files
SELECT TOP 200 event_time, server_principal_name, database_name, schema_name, object_name,
       action_id, succeeded, statement
FROM sys.fn_get_audit_file(@pattern, DEFAULT, DEFAULT)
ORDER BY event_time DESC;

-- Your DB only
SELECT TOP 200 event_time, server_principal_name, database_name, schema_name, object_name,
       action_id, succeeded, statement
FROM sys.fn_get_audit_file(@pattern, DEFAULT, DEFAULT)
WHERE database_name = N'MedicalInfoSystem'
ORDER BY event_time DESC;

-- Failures only (attempts)
SELECT TOP 100 event_time, server_principal_name, database_name, schema_name, object_name,
       action_id, succeeded, statement
FROM sys.fn_get_audit_file(@pattern, DEFAULT, DEFAULT)
WHERE database_name = N'MedicalInfoSystem' AND succeeded = 0
ORDER BY event_time DESC;
```

*Figure 167 : Resolved audit path, read folder, and *.sqlaudit pattern (output)*

	audit_file_path_from_DMV	folder_we_will_read	pattern
1	C:\Users\user\Desktop\APU\APU Y3S2\DBS\Assignment\SQL_Audit\ServerAudit_MIS_...	C:\Users\user\Desktop\APU\APU Y3S2\DBS\Assignment\SQL_Audit\	C:\Users\user\Desktop\APU\APU Y3S2\DBS\Assignment\SQL_Audit*.sqlaudit

Figure 168 : Audit path/pattern resolved

	event_time	server_principal_name	database_name	schema_name	object_name	action_id	succeeded	statement
11	2025-09-19 16:35:04.3009399	N001				LGIS	1	-- network protocol: LPC set quoted_identifier ...
12	2025-09-19 16:35:04.2619313	N001				LGIS	1	-- network protocol: LPC set quoted_identifier ...
13	2025-09-19 16:34:58.9096934	N001				LGIF	0	Login failed for user 'N001'. Reason: Password...
14	2025-09-19 16:31:18.1582587	NT SERVICE\SQLT...				LGIS	1	-- network protocol: LPC set quoted_identifier ...
15	2025-09-19 16:31:18.1189485	NT SERVICE\SQLT...				LGIS	1	-- network protocol: LPC set quoted_identifier ...
16	2025-09-19 16:30:00.6344585	NT SERVICE\SQLS...				LGIS	1	-- network protocol: LPC set quoted_identifier ...
17	2025-09-19 16:30:00.6074529	NT SERVICE\SQLS...	MedicalInfoSy...		MedicalInf...	BAL	1	BACKUP LOG [MedicalInfoSystem] TO DISK =...
18	2025-09-19 16:30:00.6064521	NT SERVICE\SQLS...				LGIS	1	-- network protocol: LPC set quoted_identifier ...

Figure 169 : Enterprise audit events—All events (Top 200)

	event_time	server_principal_name	database_name	schema_name	object_name	action_id	succeeded	statement
1	2025-09-19 16:30:00.6074529	NT SERVICE\SQLSERVERAGENT	MedicalInfoSystem		MedicalInfoSystem	BAL	1	BACKUP LOG [MedicalInfoSystem] TO DISK = @file WI...
2	2025-09-19 16:19:40.6181070	DESKTOP-7B1SAMI\user	MedicalInfoSystem	sys	database_audit_specifications	SL	1	SELECT name,is_state_enabled FROM sys.database_a...
3	2025-09-19 16:15:00.4331177	NT SERVICE\SQLSERVERAGENT	MedicalInfoSystem		MedicalInfoSystem	BAL	1	BACKUP LOG [MedicalInfoSystem] TO DISK = @file WI...
4	2025-09-19 16:00:00.4228013	NT SERVICE\SQLSERVERAGENT	MedicalInfoSystem		MedicalInfoSystem	BA	1	BACKUP DATABASE [MedicalInfoSystem] TO DISK = ...
5	2025-09-19 16:00:00.4228013	NT SERVICE\SQLSERVERAGENT	MedicalInfoSystem		MedicalInfoSystem	BAL	1	BACKUP LOG [MedicalInfoSystem] TO DISK = @file WI...
6	2025-09-19 15:45:00.6967446	NT SERVICE\SQLSERVERAGENT	MedicalInfoSystem		MedicalInfoSystem	BAL	1	BACKUP LOG [MedicalInfoSystem] TO DISK = @file WI...
7	2025-09-19 15:30:00.5129308	NT SERVICE\SQLSERVERAGENT	MedicalInfoSystem		MedicalInfoSystem	BAL	1	BACKUP LOG [MedicalInfoSystem] TO DISK = @file WI...
8	2025-09-19 15:15:00.7450596	NT SERVICE\SQLSERVERAGENT	MedicalInfoSystem		MedicalInfoSystem	BAL	1	BACKUP LOG [MedicalInfoSystem] TO DISK = @file WI...

Figure 170 : Enterprise audit events – MedicalInfoSystem

	event_time	server_principal_name	database_name	schema_name	object_name	action_id	succeeded	statement
1	2025-09-19 11:19:09.5326534	S-1-9-3-605790920-1249579415-3732787096-917641345	MedicalInfoSystem	SecureData	Staff	SL	0	SELECT TOP 1 * FROM SecureData.Staff
2	2025-09-19 11:02:42.3916198	S-1-9-3-3823765091-1136227105-1368295832-1219374...	MedicalInfoSystem	SecureData	Staff	SL	0	SELECT TOP 1 * FROM SecureData.Staff
3	2025-09-19 10:49:25.5704348	S-1-9-3-2145553004-1132670880-2152211082-2148984...	MedicalInfoSystem	SecureData	Staff	SL	0	SELECT TOP 1 * FROM SecureData.Staff
4	2025-09-19 10:34:30.3376824	S-1-9-3-2532647266-1159518026-3515742653-2246927...	MedicalInfoSystem	SecureData	Staff	SL	0	SELECT TOP 1 * FROM SecureData.Staff
5	2025-09-19 09:46:29.3623927	SuperAdmin	MedicalInfoSystem	Audit	LogonAudit	IN	0	INSERT MedicalInfoSystem.Audit.Logon...
6	2025-09-19 09:26:29.5332568	D001	MedicalInfoSystem	SecureData	Appointme...	SL	0	SELECT DiagID, AppDateTime, PatientI...
7	2025-09-19 09:26:29.5332568	D001	MedicalInfoSystem	SecureData	Appointme...	SL	0	SELECT TOP (1) @DiagID = DiagID FR...
8	2025-09-19 09:26:18.0138390	N001	MedicalInfoSystem	SecureData	Appointme...	SL	0	SELECT TOP (1) DiagID, AppDateTime,...
9	2025-09-19 09:26:18.0138390	N001	MedicalInfoSystem	SecureData	Appointme...	SL	0	SELECT TOP (1) @DiagID = DiagID FR...

Figure 171 : Enterprise audit events – Failures (MedicalInfoSystem, succeeded=0)

Figure 166 shows the creation and enablement of *FILE target ServerAudit_MIS, ServerAuditSpec_MIS* & server scope groups for login success/failure and backup/restore. Establishes the server-level capture path.

Figure 167 shows the code for evidence output that the Server Audit is working all activities and attempts are tracked.

Figure 168 displays the resolved the audit location: raw audit_file_path from *sys.dm_server_audit_status*, where-from directory and destination **.sqlaudit* pattern. Confirms readback queries target the correct enterprise audit files (Microsoft SQL Server, 2023).

Figure 169 shows the evidence of authentication with LGIS (*Successful Logins*) and LGIF (*Failed Login*), along with timestamp and principal, and client statement. Shows login activity and attempts tracking.

Figure 170 shows backup/log backup evidence (*BA*/BAL*) tied to the executing principal and statement. Demonstrates auditing of recovery operations.

Figure 171 shows the access to *SecureData* object with allowed and denied operations (*action_id like SL, succeeded = 1 and 0*) with full statement text is depicted. Evidence produced by *DB_Audit_MIS* (e.g., *SCHEMA_OBJECT_ACCESS_GROUP* and *DML/EXEC on SCHEMA::SecureData BY PUBLIC*) demonstrates capture of activity and attempts at the data layer.

4.2.1.2 Database Audit Specification (scope: SecureData + attempts)

```
--=============================================================================
-- 2) DATABASE AUDIT SPEC (breadth + attempts in SecureData)
--=============================================================================
IF EXISTS (SELECT 1 FROM sys.database_audit_specifications WHERE name=N'DB_Audit_MIS')
BEGIN
  ALTER DATABASE AUDIT SPECIFICATION [DB_Audit_MIS] WITH (STATE=OFF);
  DROP  DATABASE AUDIT SPECIFICATION [DB_Audit_MIS];
END
GO
CREATE DATABASE AUDIT SPECIFICATION [DB_Audit_MIS]
FOR SERVER AUDIT [ServerAudit_MIS]
    ADD (DATABASE_PERMISSION_CHANGE_GROUP),
    ADD (DATABASE_PRINCIPAL_CHANGE_GROUP),
    ADD (DATABASE_ROLE_MEMBER_CHANGE_GROUP),
    ADD (SCHEMA_OBJECT_ACCESS_GROUP),
    ADD (SELECT  ON SCHEMA::SecureData BY PUBLIC),
    ADD (INSERT  ON SCHEMA::SecureData BY PUBLIC),
    ADD (UPDATE  ON SCHEMA::SecureData BY PUBLIC),
    ADD (DELETE  ON SCHEMA::SecureData BY PUBLIC),
    ADD (EXECUTE ON SCHEMA::SecureData BY PUBLIC)
WITH (STATE = ON);
GO
```

Figure 172 : Database Audit Specification for SecureData

Figure 172 shows the database audit specification *DB_Audit_MIS* associated with *FOR SERVER AUDIT ServerAudit_MIS* and *STATE = ON*.

- Includes *SCHEMA_OBJECT_ACCESS_GROUP* to produce object-access events (both success and failure) (Microsoft SQL Server, 2024e; Microsoft SQL Server, 2024g).
- Add *SELECT/INSERT/UPDATE/DELETE/EXECUTE ON SCHEMA::SecureData BY PUBLIC* to record all user activity and failed attempts on *SecureData*.
- Adds database security change groups (*DATABASE_PERMISSION_CHANGE_GROUP, DATABASE_PRINCIPAL_CHANGE_GROUP, DATABASE_ROLE_MEMBER_CHANGE_GROUP*) to log the permission/principal/role changes.

The enterprise readback will display SecureData events having action_id as SL/IN/UP/DL/EX, with succeeded = 1 (activity) and succeeded = 0 (attempts), actor and statement text- this is to fulfil Requirements 5 & 6.

4.2.2 Solution summary

Comprehensive activity coverage is achieved by combining *Server Audit (ServerAuditSpec_MIS)* with Database Audit Specification *(DB_Audit_MIS)*. Server Audit captures logins, principal/role changes, backup/restore, and audit changes—the high-value server events no database object can see. The Database Audit Spec targets *SecureData* with *SCHEMA_OBJECT_ACCESS_GROUP* plus explicit *SELECT/INSERT/UPDATE/DELETE/EXECUTE ON SCHEMA::SecureData* BY PUBLIC, so both successes and failures surface in enterprise evidence via *sys.fn_get_audit_file (action_id, succeeded, statement)* (Microsoft SQL Server, 2024f). This technique is preferred over triggers and app logs because SELECT attempts and blocked access cannot be captured by DML/DDL triggers or reliably by applications; only SQL Server Audit reliably records attempts (including failed reads/executes) with the actor and exact statement while streaming to *.sqlaudit files for durable, centralized review.

4.3 Prove DDL is captured with who/what/when (Requirement 6; supports Requirement 2)

Requirement 2	-	Superadmin must be able to perform appropriate DDL (such as create tables, views, logins, users, encryption keys etc) and DML tasks to maintain high level of functionality, availability and security of this DB
Requirement 6	-	All activities by all users (including attempt to do such activity) must be tracked.

This section provides evidence of database-scope DDL auditing. A database-level DDL trigger captures *EVENTDATA()* for every schema change and calls a writer procedure that persists the exact T-SQL, object metadata, actor, and UTC timestamp into *Audit.DDLAudit*. Output confirms that creates, alters, drops, grants/denies, and other DDL operations are recorded with attribution.

4.3.1 Implementation & Evidence — Database-Level DDL Trigger → Audit Writer (EXECUTE AS OWNER)

```
IF OBJECT_ID(N'Audit.DDLAudit','U') IS NULL
BEGIN
  CREATE TABLE Audit.DDLAudit(
    DDLAuditID   bigint IDENTITY(1,1) PRIMARY KEY,
    PostTime     datetime2(7) NOT NULL DEFAULT SYSUTCDATETIME(),
    EventType    nvarchar(128) NOT NULL,
    ObjectSchema nvarchar(128) NULL,
    ObjectName   nvarchar(256) NULL,
    ObjectType   nvarchar(128) NULL,
    TSql         nvarchar(max) NULL,
    Actor        sysname       NOT NULL DEFAULT SUSER_SNAME(),
    EventXml     xml           NOT NULL
  );
END
```

Figure 173 : Audit.DDLAudit Object

```
--=============================================================================
-- 3) HELPER PROC (EXECUTE AS OWNER) the DDL trigger will call
--=============================================================================
-- create-or-alter pattern (avoids "already exists" and ordering issues)
IF OBJECT_ID('Audit.usp_WriteDDLAudit','P') IS NULL
    EXEC ('CREATE PROCEDURE Audit.usp_WriteDDLAudit AS RETURN 0;');
GO
ALTER PROCEDURE Audit.usp_WriteDDLAudit
  @EventXml xml,
  @Actor    sysname
WITH EXECUTE AS OWNER
AS
BEGIN
  SET NOCOUNT ON;
  INSERT Audit.DDLAudit(EventType,ObjectSchema,ObjectName,ObjectType,TSql,EventXml,Actor)
  SELECT
      @EventXml.value('(/EVENT_INSTANCE/EventType)[1]'               ,'nvarchar(128)'),
      @EventXml.value('(/EVENT_INSTANCE/SchemaName)[1]'              ,'nvarchar(128)'),
      @EventXml.value('(/EVENT_INSTANCE/ObjectName)[1]'              ,'nvarchar(256)'),
      @EventXml.value('(/EVENT_INSTANCE/ObjectType)[1]'              ,'nvarchar(128)'),
      @EventXml.value('(/EVENT_INSTANCE/TSQLCommand/CommandText)[1]' ,'nvarchar(max)'),
      @EventXml,
      @Actor;
END
GO
GRANT EXECUTE ON Audit.usp_WriteDDLAudit TO PUBLIC;
GO
```

Figure 174 : DDL writer proc (EXECUTE AS OWNER)

```
--=============================================================================
-- 4) DB-LEVEL DDL TRIGGER (CREATE must be 1st stmt in batch)
--=============================================================================
IF EXISTS (SELECT 1 FROM sys.triggers WHERE name=N'TR_DDL_DB_Audit' AND parent_class=0)
    DROP TRIGGER TR_DDL_DB_Audit ON DATABASE;
GO
CREATE TRIGGER TR_DDL_DB_Audit
ON DATABASE
FOR DDL_DATABASE_LEVEL_EVENTS
AS
BEGIN
  SET NOCOUNT ON;

  DECLARE @x xml;       SET @x = CAST(EVENTDATA() AS xml);
  DECLARE @actor sysname; SET @actor = SUSER_SNAME();

  -- ignore events on our own Audit objects (prevents bootstrap noise)
  DECLARE @schema nvarchar(128) = @x.value('(/EVENT_INSTANCE/SchemaName)[1]','nvarchar(128)');
  DECLARE @obj    nvarchar(256) = @x.value('(/EVENT_INSTANCE/ObjectName)[1]' ,'nvarchar(256)');
  IF @schema = N'Audit' AND @obj IN (N'usp_WriteDDLAudit',N'DDLAudit',N'DMLAudit',N'LogonAudit',N'TR_DDL_DB_Audit') RETURN;

  EXEC Audit.usp_WriteDDLAudit @EventXml=@x, @Actor=@actor;
END;
GO
```

Figure 175 : Database DDL trigger → writer

```
/* 2) DDL trigger → Audit.DDLAudit */
USE MedicalInfoSystem;
SELECT TOP 20
  PostTime, EventType, ObjectSchema, ObjectName, ObjectType, Actor
FROM Audit.DDLAudit
ORDER BY PostTime DESC;

-- (optional) last 24h summary
SELECT EventType, COUNT(*) AS cnt
FROM Audit.DDLAudit
WHERE PostTime > DATEADD(day,-1,SYSUTCDATETIME())
GROUP BY EventType ORDER BY cnt DESC;
```

Figure 176 : DDL evidence queries (detail/summary)

	PostTime	EventType	ObjectSchema	ObjectName	ObjectType	Actor
1	2025-09-19 12:01:50.4088298	ALTER_PROCEDURE	dbo	__drop_dc	PROCEDURE	DESKTOP-7B1SAMI\user
2	2025-09-19 12:01:50.4018282	ALTER_TRIGGER	SecureData	TR_Appoi...	TRIGGER	DESKTOP-7B1SAMI\user
3	2025-09-19 12:01:50.3978273	ALTER_TRIGGER	SecureData	TR_Patien...	TRIGGER	DESKTOP-7B1SAMI\user
4	2025-09-19 12:01:50.3938268	ALTER_TRIGGER	SecureData	TR_Staff_...	TRIGGER	DESKTOP-7B1SAMI\user
5	2025-09-19 12:01:50.2843901	CREATE_DATABAS...	NULL	DB_Audit_...	DATABASE ...	DESKTOP-7B1SAMI\user
6	2025-09-19 12:01:50.2803894	DROP_DATABASE_...	NULL	DB_Audit_...	DATABASE ...	DESKTOP-7B1SAMI\user
7	2025-09-19 12:01:50.2773879	ALTER_DATABASE...	NULL	DB_Audit_...	DATABASE ...	DESKTOP-7B1SAMI\user
8	2025-09-19 12:01:50.2703874	DENY_DATABASE		Audit	SCHEMA	DESKTOP-7B1SAMI\user

Figure 177 : DDL evidence rows (who/what/when)

	EventType	cnt
1	GRANT_DATABASE	40
2	ALTER_TRIGGER	18
3	DENY_DATABASE	17
4	REVOKE_DATABA...	11
5	CREATE_PROCED...	8
6	ALTER_TABLE	8
7	CREATE_USER	6
8	DROP_USER	6
9	CREATE_DATABA...	5
10	ALTER_AUTHORI...	5
11	ALTER_DATABAS...	5
12	DROP_DATABASE...	5
13	ALTER_SECURIT...	4
14	CREATE_FUNCTI...	3

Figure 178 : DDL event counts (last 24h)

Figure 173 shows the table Audit.DDLAudit, the persistent sink for database-scope DDL events. Figure 174 shows the writer procedure *Audit.usp_WriteDDLAudit* declared *WITH EXECUTE AS OWNER*. The procedure parses *EVENTDATA()* and inserts *EventType, ObjectSchema, ObjectName, ObjectType, TSql, raw EventXml*, and *Actor* into *Audit.DDLAudit*. This proves that the evidence can always be written in owner context and includes the exact command text. Figure 175 shows *TR_DDL_DB_Audit FOR DDL_DATABASE_LEVEL_EVENTS* which is the database-level trigger. This trigger will reads the *EVENTDATA()* and ignores the Audit schema

objects to avoid noise, and executes *Audit.usp_WriteDDLAudit*. This proves that all the DDL events at database level are routed to the audit table.

Figure 176 provides example evidence queries to Audit. *DDLAudit*: a details view (*TOP 20 ... ORDER BY PostTime DESC*) and optional last-24-hours summary (*COUNT(*) BY EventType*). Figure 177 displayed the recently-captured DDL rows with *PostTime, EventType, ObjectSchema, ObjectName, ObjectType and Actor* (e.g., *ALTER_TRIGGER, CREATE_DATABASE_AUDIT_SPECIFICATION,DENY_DATABAS*E) are displayed.

Figure 178 shows a summary of DDL activity in the last 24 hours grouped by *EventType* (e.g., *GRANT_DATABASE, ALTER_TRIGGER, CREATE_PROCEDURE*).

4.3.2 Solution summary

Database-level DDL is captured by a DB-scope DDL trigger (*TR_DDL_DB_Audit*) that parses *EVENTDATA()* and writes to *Audit.DDLAudit* via a helper proc *WITH EXECUTE AS OWNER*. This approach is chosen over relying solely on Server Audit because it ensures an in-database, queryable ledger with the exact T-SQL, normalized object metadata, and the actor, while allowing precise filtering (e.g., ignore Audit schema to avoid bootstrap noise). The trigger runs within the DDL transaction and produces deterministic, immediate entries even if tooling above the engine is unavailable, giving auditors a self-contained, searchable evidence table that aligns directly with schema objects under review.

4.4 Prove DML row changes (before/after) are captured (Requirement 6; supports Requirement 4)

Requirement 4 - All data and data changes must be traceable and recoverable including in the event of complete DB failure or loss.

Requirement 6 - All activities by all users (including attempt to do such activity) must be tracked.

This section is to demonstrate server scope login auditing and context. The Server Audit Specification allows *SUCCESSFUL_LOGIN_GROUP* and *FAILED_LOGIN_GROUP*, with the authentication events being sent to the enterprise *.*sqlaudit* target. Enterprise readback displays *LGIS/LGIF* entries with *event_time, server_principal_name*, *succeeded* and *statement* for activity as well as attempts. These fields provide accountability for who/what/when for authentication activity, satisfying Requirements 5 and 6.

4.4.1 Implementation & Evidence — Per-Table DML Triggers

```
IF OBJECT_ID(N'Audit.DMLAudit','U') IS NULL
BEGIN
  CREATE TABLE Audit.DMLAudit(
    DMLAuditID bigint IDENTITY(1,1) PRIMARY KEY,
    AtTime     datetime2(7) NOT NULL DEFAULT SYSUTCDATETIME(),
    TableName  sysname      NOT NULL,
    Action     char(1)      NOT NULL,       -- I/U/D
    KeyJson    nvarchar(2000) NOT NULL,
    BeforeJson nvarchar(max) NULL,
    AfterJson  nvarchar(max) NULL,
    Actor      sysname      NOT NULL DEFAULT SUSER_SNAME(),
    AppName    nvarchar(256) NULL DEFAULT APP_NAME(),
    HostName   nvarchar(256) NULL DEFAULT HOST_NAME(),
    SessionId  int          NOT NULL DEFAULT @@SPID
  );
END
```

Figure 179 : Audit.DMLAudit table schema

```
--=============================================================================
-- 5) PER-TABLE DML TRIGGERS (SecureData.* with PK) → Audit.DMLAudit
--=============================================================================
DECLARE @t table(SchemaName sysname, TableName sysname, TableId int, HasPK bit);
INSERT @t
SELECT s.name, t.name, t.object_id,
       CASE WHEN EXISTS (SELECT 1 FROM sys.indexes i WHERE i.object_id=t.object_id AND i.is_primary_key=1)
            THEN 1 ELSE 0 END
FROM sys.tables t
JOIN sys.schemas s ON s.schema_id=t.schema_id
WHERE s.name='SecureData' AND t.temporal_type <> 1;  -- skip history tables

DECLARE @Schema sysname, @Table sysname, @ObjId int, @HasPK bit;
DECLARE c CURSOR LOCAL FAST_FORWARD FOR SELECT SchemaName, TableName, TableId, HasPK FROM @t;
OPEN c; FETCH NEXT FROM c INTO @Schema, @Table, @ObjId, @HasPK;

WHILE @@FETCH_STATUS = 0
BEGIN
  IF @HasPK = 1
  BEGIN
    DECLARE @Trig sysname = N'TR_' + @Table + N'_DML_Audit';
    DECLARE @pkJoin nvarchar(max), @pkColsI nvarchar(max), @pkColsD nvarchar(max),
            @allColsI nvarchar(max), @allColsD nvarchar(max);

    -- PK lists & join
    SELECT
      @pkJoin = STRING_AGG(CONCAT('ISNULL(i.',QUOTENAME(c.name),', d.',QUOTENAME(c.name),')=ISNULL(d.',QUOTENAME(c.name),', i.',QUOTENAME(c.name),')'),' AND '),
      @pkColsI = STRING_AGG(CONCAT('i.',QUOTENAME(c.name)), ','),
      @pkColsD = STRING_AGG(CONCAT('d.',QUOTENAME(c.name)), ',')
    FROM sys.index_columns ic
    JOIN sys.columns c ON c.object_id=ic.object_id AND c.column_id=ic.column_id
    JOIN sys.indexes  ix ON ix.object_id=ic.object_id AND ix.index_id=ic.index_id AND ix.is_primary_key=1
    WHERE ic.object_id=@ObjId;

    -- all (non-computed) columns
    SELECT
      @allColsI = STRING_AGG(CONCAT('i.',QUOTENAME(c.name)), ','),
      @allColsD = STRING_AGG(CONCAT('d.',QUOTENAME(c.name)), ',')
    FROM sys.columns c WHERE c.object_id=@ObjId AND c.is_computed=0;

    DECLARE @firstPK sysname =
    (SELECT TOP(1) c.name
     FROM sys.index_columns ic
     JOIN sys.columns c ON c.object_id=ic.object_id AND c.column_id=ic.column_id
     JOIN sys.indexes  ix ON ix.object_id=ic.object_id AND ix.index_id=ic.index_id AND ix.is_primary_key=1
     WHERE ic.object_id=@ObjId ORDER BY ic.key_ordinal);
```

Figure 180 : Trigger generator—discover auditable tables; build PK/all-column lists.

```
    DECLARE @ddl nvarchar(max) = N'
CREATE OR ALTER TRIGGER ' + QUOTENAME(@Schema) + N'.' + QUOTENAME(@Trig) + N'
ON ' + QUOTENAME(@Schema) + N'.' + QUOTENAME(@Table) + N'
WITH EXECUTE AS OWNER
AFTER INSERT, UPDATE, DELETE
AS
BEGIN
  SET NOCOUNT ON;
  INSERT Audit.DMLAudit(AtTime, TableName, Action, KeyJson, BeforeJson, AfterJson, Actor, AppName, HostName, SessionId)
  SELECT
    SYSUTCDATETIME(),
    N''' + REPLACE(@Table,'''',''''') + N''',
    CASE WHEN i.' + QUOTENAME(@firstPK) + N' IS NOT NULL AND d.' + QUOTENAME(@firstPK) + N' IS NULL THEN ''I''
         WHEN i.' + QUOTENAME(@firstPK) + N' IS NULL AND d.' + QUOTENAME(@firstPK) + N' IS NOT NULL THEN ''D''
         ELSE ''U'' END,
    COALESCE( (SELECT ' + @pkColsI + N' FOR JSON PATH, WITHOUT_ARRAY_WRAPPER),
              (SELECT ' + @pkColsD + N' FOR JSON PATH, WITHOUT_ARRAY_WRAPPER) ),
    CASE WHEN d.' + QUOTENAME(@firstPK) + N' IS NOT NULL
         THEN (SELECT ' + @allColsD + N' FOR JSON PATH, WITHOUT_ARRAY_WRAPPER) END,
    CASE WHEN i.' + QUOTENAME(@firstPK) + N' IS NOT NULL
         THEN (SELECT ' + @allColsI + N' FOR JSON PATH, WITHOUT_ARRAY_WRAPPER) END,
    SUSER_SNAME(), APP_NAME(), HOST_NAME(), @@SPID
  FROM inserted i
  FULL OUTER JOIN deleted d ON ' + @pkJoin + N';
END;';
    EXEC sys.sp_executesql @ddl;
  END

  FETCH NEXT FROM c INTO @Schema, @Table, @ObjId, @HasPK;
END
CLOSE c; DEALLOCATE c;
GO
```

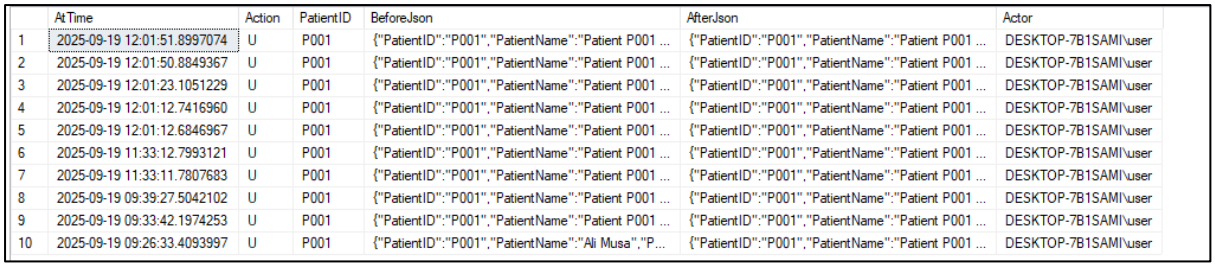
Figure 181 : Trigger body—EXECUTE AS OWNER; I/U/D → write to Audit.DMLAudit

```
/* 3) DML triggers → Audit.DMLAudit (row deltas) */
-- Recent changes across SecureData tables
SELECT TOP 50
  AtTime, TableName, Action, KeyJson, Actor, AppName, HostName, SessionId
FROM Audit.DMLAudit
ORDER BY AtTime DESC;

-- Expand one example (Patient) to show keys and before/after
SELECT TOP 20
  AtTime, Action,
  JSON_VALUE(KeyJson,'$.PatientID')    AS PatientID,
  BeforeJson, AfterJson, Actor
FROM Audit.DMLAudit
WHERE TableName = N'Patient'
ORDER BY AtTime DESC;
```

Figure 182 : Evidence queries—recent changes and Patient view

	AtTime	TableName	Action	KeyJson	Actor	AppName	HostName	SessionId
1	2025-09-19 12:01:51.9158358	AppointmentAndDiagnosis	D	{}	DESKTOP-7B1SAMI\user	Microsoft SQL Server Management Studio - Query	DESKTOP-7B1SAMI	72
2	2025-09-19 12:01:51.9128332	AppointmentAndDiagnosis	U	{"Diag...	DESKTOP-7B1SAMI\user	Microsoft SQL Server Management Studio - Query	DESKTOP-7B1SAMI	72
3	2025-09-19 12:01:51.9108330	AppointmentAndDiagnosis	I	{"Diag...	DESKTOP-7B1SAMI\user	Microsoft SQL Server Management Studio - Query	DESKTOP-7B1SAMI	72
4	2025-09-19 12:01:51.8997074	Patient	U	{"Pati...	DESKTOP-7B1SAMI\user	Microsoft SQL Server Management Studio - Query	DESKTOP-7B1SAMI	72
5	2025-09-19 12:01:50.8849367	Patient	U	{"Pati...	DESKTOP-7B1SAMI\user	Microsoft SQL Server Management Studio - Query	DESKTOP-7B1SAMI	72
6	2025-09-19 12:01:33.4031628	AppointmentAndDiagnosis	U	{"Diag...	DESKTOP-7B1SAMI\user	Microsoft SQL Server Management Studio - Query	DESKTOP-7B1SAMI	67
7	2025-09-19 12:01:33.3907102	AppointmentAndDiagnosis	I	{"Diag...	DESKTOP-7B1SAMI\user	Microsoft SQL Server Management Studio - Query	DESKTOP-7B1SAMI	67
8	2025-09-19 12:01:27.9129969	AppointmentAndDiagnosis	U	{"Diag...	DESKTOP-7B1SAMI\user	Microsoft SQL Server Management Studio - Query	DESKTOP-7B1SAMI	52

Figure 183 : DML evidence rows—who/what/when/where



	AtTime	Action	PatientID	BeforeJson	AfterJson	Actor
1	2025-09-19 12:01:51.8997074	U	P001	{"PatientID":"P001","PatientName":"Patient P001 ...	{"PatientID":"P001","PatientName":"Patient P001 ...	DESKTOP-7B1SAMI\user
2	2025-09-19 12:01:50.8849367	U	P001	{"PatientID":"P001","PatientName":"Patient P001 ...	{"PatientID":"P001","PatientName":"Patient P001 ...	DESKTOP-7B1SAMI\user
3	2025-09-19 12:01:23.1051229	U	P001	{"PatientID":"P001","PatientName":"Patient P001 ...	{"PatientID":"P001","PatientName":"Patient P001 ...	DESKTOP-7B1SAMI\user
4	2025-09-19 12:01:12.7416960	U	P001	{"PatientID":"P001","PatientName":"Patient P001 ...	{"PatientID":"P001","PatientName":"Patient P001 ...	DESKTOP-7B1SAMI\user
5	2025-09-19 12:01:12.6846967	U	P001	{"PatientID":"P001","PatientName":"Patient P001 ...	{"PatientID":"P001","PatientName":"Patient P001 ...	DESKTOP-7B1SAMI\user
6	2025-09-19 11:33:12.7993121	U	P001	{"PatientID":"P001","PatientName":"Patient P001 ...	{"PatientID":"P001","PatientName":"Patient P001 ...	DESKTOP-7B1SAMI\user
7	2025-09-19 11:33:11.7807683	U	P001	{"PatientID":"P001","PatientName":"Patient P001 ...	{"PatientID":"P001","PatientName":"Patient P001 ...	DESKTOP-7B1SAMI\user
8	2025-09-19 09:39:27.5042102	U	P001	{"PatientID":"P001","PatientName":"Patient P001 ...	{"PatientID":"P001","PatientName":"Patient P001 ...	DESKTOP-7B1SAMI\user
9	2025-09-19 09:33:42.1974253	U	P001	{"PatientID":"P001","PatientName":"Patient P001 ...	{"PatientID":"P001","PatientName":"Patient P001 ...	DESKTOP-7B1SAMI\user
10	2025-09-19 09:26:33.4093997	U	P001	{"PatientID":"P001","PatientName":"Ali Musa","P...	{"PatientID":"P001","PatientName":"Patient P001 ...	DESKTOP-7B1SAMI\user

Figure 184 : Patient row deltas—KeyJson + BeforeJson/AfterJson

Figure 179 shows the evidence table Audit.DMLAudit with AtTime, TableName, Action (I/U/D), KeyJson, BeforeJson, AfterJson, and context (Actor, AppName, HostName, SessionId). This proves that the fixed schema to store row keys and before/after payloads with attribution.

Figure shows discovery of SecureData tables with a primary key and preparation of PK/all-column lists for JSON capture. This proves that the generator targets only auditable tables and builds column metadata for snapshots.

Figure shows the emitted trigger body WITH EXECUTE AS OWNER AFTER INSERT/UPDATE/DELETE, computing Action and writing KeyJson, BeforeJson, AfterJson to Audit.DMLAudit. This proves that the consistent capture for I/U/D with owner-context writes (tamper-resistant path).

Figure 182 shows evidence queries: recent changes across SecureData and a Patient-focused view extracting PatientID via JSON_VALUE. This proves that the reproducible steps to read audit entries and expand a table-specific case.

Figure 183 shows recent DML entries with AtTime, TableName, Action, KeyJson, and actor/context fields.This proves that who/what/when/where for row changes across audited tables.

Figure 184 shows Patient updates with PatientID plus non-null BeforeJson/AfterJson pairs. This proves the precise before/after deltas persisted for review and rollback analysis.

The trigger generator applies a uniform EXECUTE AS OWNER I/U/D audit to each SecureData table with a PK, producing normalized entries in Audit.DMLAudit that include row keys and before/after JSON. Evidence confirms attributable and reviewable row-level changes, satisfying Req 6 and supporting Req 4.

4.4.2 Solution summary

Row-level delta changes are captured using per-table triggers on *SecureData.* * (tables with a primary key). Every trigger EXECUTE AS OWNER and writes a canonical record to

Audit.DMLAudit with *Action (I/U/D), TableName, KeyJson* (row key), *BeforeJson* and *AfterJson* snapshots, and *Actor/AppName/HostName/SessionId*. The *Audit* schema is secured (DENY to PUBLIC), and owner-context writers ensure the evidence gets written out even if the actor has no direct rights. Proof queries reveal chronic/newer vents and a concentrated *Patient* view with keys and deltas being responsible for, and reviewable, changes. This approach gives deterministic capture over inserts, updates and deletes at low operational cost, with simple query capabilities. Complete accountability and reconstructable row-level deltas meet R6 and R4, respectively.

4.5 Prove logins and environment context are audited (Requirements 5 & 6)

Requirement 5 - All users must be able to log into the MS-SQL system using SQL Server Management Studio and perform their own tasks. Assume that all users have sufficient knowledge on writing and running SQL queries in the query window.

Requirement 6 - All activities by all users (including attempt to do such activity) must be tracked.

4.5.1 Implementation & Evidence — Server login auditing → Enterprise readback

```
CREATE SERVER AUDIT SPECIFICATION [ServerAuditSpec_MIS]
FOR SERVER AUDIT [ServerAudit_MIS]
    ADD (SUCCESSFUL_LOGIN_GROUP),
    ADD (FAILED_LOGIN_GROUP),
    ADD (SERVER_PRINCIPAL_CHANGE_GROUP),
    ADD (SERVER_ROLE_MEMBER_CHANGE_GROUP),
    ADD (AUDIT_CHANGE_GROUP),
    ADD (BACKUP_RESTORE_GROUP);
ALTER SERVER AUDIT SPECIFICATION [ServerAuditSpec_MIS] WITH (STATE = ON);
ALTER SERVER AUDIT [ServerAudit_MIS] WITH (STATE = ON);
GO
```

Figure 185 : Server audit spec—LGIS/LGIF enabled

```
SELECT [audit_file_path_from_DMV]=@path, [folder_we_will_read]=@dir, [pattern]=@pattern;
```

Figure 186 : Audit path/pattern resolved—.sqlaudit*

	audit_file_path_from_DMV	folder_we_will_read	pattern
1	C:\Users\user\Desktop\APU\APU Y3S2\DBS\Assignmen...	C:\Users\user\Desktop\APU\APU Y3S2\DBS\Assignmen...	C:\Users\user\Desktop\APU\APU Y3S2\DBS\Assignmen...

Figure 187 : Login evidence query—LGIS/LGIF with session/context

```
-- Login events only (authentication evidence)
SELECT TOP 50
       event_time,
       session_id,               -- environment context
       server_principal_name,
       action_id,                -- LGIS / LGIF
       succeeded,                -- 1 = success, 0 = failure
       statement                 -- client/login statement text
FROM sys.fn_get_audit_file(@pattern, DEFAULT, DEFAULT)
WHERE action_id IN ('LGIS','LGIF')
ORDER BY event_time DESC;
```

Figure 188 : Enterprise login events—successes and failures

	event_time	session_id	server_principal_name	action_id	succeeded	statement
7	2025-09-19 21:07:37.6472283	70	P001	LGIS	1	-- network protocol: LPC set quoted_identifier ...
8	2025-09-19 21:07:37.6432281	74	P001	LGIS	1	-- network protocol: LPC set quoted_identifier ...
9	2025-09-19 21:07:37.6402268	55	P001	LGIS	1	-- network protocol: LPC set quoted_identifier ...
10	2025-09-19 21:07:37.6382264	74	P001	LGIS	1	-- network protocol: LPC set quoted_identifier ...
11	2025-09-19 21:07:37.6250010	71	P001	LGIS	1	-- network protocol: LPC set quoted_identifier ...
12	2025-09-19 21:07:37.5922312	70	P001	LGIS	1	-- network protocol: LPC set quoted_identifier ...
13	2025-09-19 21:07:31.8138055	0	P001	LGIF	0	Login failed for user 'P001'. Reason: Password ...
14	2025-09-19 21:06:49.2242231	62	DESKTOP-7B1SAM...	LGIS	1	-- network protocol: LPC set quoted_identifier ...

Figure 189 : Enterprise login events—successes and failures (output)

```
SELECT action_id, succeeded, COUNT(*) AS cnt
FROM sys.fn_get_audit_file(@pattern, DEFAULT, DEFAULT)
WHERE action_id IN ('LGIS','LGIF')
GROUP BY action_id, succeeded
ORDER BY action_id, succeeded DESC;
```

Figure 190 : Login summary query—counts by action/outcome (code)

	action_id	succeeded	cnt
1	LGIF	0	3
2	LGIS	1	857

Figure 191 : Login outcomes—LGIS vs LGIF totals (output)

Figure 185 shows the Server Audit Specification enabling *SUCCESSFUL_LOGIN_GROUP* and *FAILED_LOGIN_GROUP* with *STATE = ON*.

Proves: server-scope authentication events are captured to the enterprise audit target.

Figure 186 and Figure 187 show resolution of the active audit location and final *@pattern = <folder>*.sqlaudit*.

Proves: readback queries point at the correct enterprise audit files.

Figure 188 and Figure 189 show a login-only filter selecting *LGIS/LGIF* with *event_time, session_id, server_principal_name, action_id, succeeded*, and *statement*.

Proves: concrete evidence of both successful logins and failed attempts, with context.

Figure 190 and Figure 191 show a summary grouped by *action_id/succeeded*, yielding counts for LGIS vs LGIF.

Proves: quick KPI of authentication outcomes.

4.5.2 Solution summary

Authentication visibility is delivered via Server Audit *SUCCESSFUL_LOGIN_GROUP* and *FAILED_LOGIN_GROUP (LGIS/LGIF)* in *ServerAuditSpec_MIS*, read back from enterprise files with *sys.fn_get_audit_file* to expose *event_time, server_principal_name, session_id, succeeded*, and the login statement. This is superior to relying on the SQL error log or app telemetry: Server Audit yields structured, filterable, and tamper-resistant records with explicit success/failure flags and session context, and it covers all entry points to the instance, not just those traversing a specific app layer. The result is authoritative proof of who attempted to connect, whether it succeeded, and from where—fulfilling Requirement 5 and 6 with minimal administrative complexity.

4.6 Prove point-in-time data recovery & history (Requirement 4; supports Requirement 1/3)

Requirement 1	-	Confidentiality, Integrity, Availability, Functionality and Usability of the DB must be achieved at all times
Requirement 3	-	Sufficient and suitable protection must be provided to protect all data from accidental and intentional exposure or deletion without compromising functionality and usability
Requirement 4	-	All data and data changes must be traceable and recoverable including in the event of complete DB failure or loss.

This section evidences engine-maintained row history via system-versioned temporal tables on *SecureData.Patient, SecureData.Staff*, and *SecureData.AppointmentAndDiagnosis*. Temporal captures every update/delete into immutable history tables and supports *FOR SYSTEM_TIME* queries to reconstruct past states and timelines, demonstrating point-in-time recovery and changing history without custom code.

4.6.1 Implementation & Evidence — System-Versioned Temporal Tables (SecureData.*)

```
--=============================================================================
-- 6) SYSTEM-VERSIONED TEMPORAL on 3 SecureData tables (idempotent)
--=============================================================================
-- helper to drop a default constraint on a column (safe)
CREATE OR ALTER PROCEDURE dbo.__drop_dc
  @tbl sysname,  -- 'SecureData.Patient'
  @col sysname   -- 'ValidFrom' / 'ValidTo'
AS
BEGIN
  SET NOCOUNT ON;
  DECLARE @schema sysname = PARSENAME(@tbl,2);
  DECLARE @table  sysname = PARSENAME(@tbl,1);
  DECLARE @objId  int     = OBJECT_ID(QUOTENAME(@schema)+'.'+QUOTENAME(@table));
  IF @objId IS NULL RETURN;

  DECLARE @dc sysname =
   (SELECT d.name
    FROM sys.default_constraints d
    JOIN sys.columns c ON c.object_id=d.parent_object_id AND c.column_id=d.parent_column_id
    WHERE d.parent_object_id=@objId AND c.name=@col);
  IF @dc IS NOT NULL
  BEGIN
    DECLARE @sql nvarchar(max) =
      N'ALTER TABLE ' + QUOTENAME(@schema) + N'.' + QUOTENAME(@table) +
      N' DROP CONSTRAINT ' + QUOTENAME(@dc) + N';';
    EXEC sys.sp_executesql @sql;
  END
END;
GO
```

Figure 192 : Helper proc dbo.__drop_dc—safe drop of default constraints

```
DECLARE @tbl sysname, @base sysname, @hist sysname, @sql nvarchar(max);

DECLARE curT CURSOR LOCAL FAST_FORWARD FOR
SELECT N'SecureData.Patient' UNION ALL
SELECT N'SecureData.Staff'   UNION ALL
SELECT N'SecureData.AppointmentAndDiagnosis';
OPEN curT; FETCH NEXT FROM curT INTO @tbl;

WHILE @@FETCH_STATUS = 0
BEGIN
  IF OBJECT_ID(@tbl,'U') IS NOT NULL
     AND (SELECT temporal_type FROM sys.tables WHERE object_id=OBJECT_ID(@tbl))=0
  BEGIN
    SET @base = PARSENAME(@tbl,1);
    SET @hist = @base + N'History';

    -- drop period if it oddly exists; clean old columns
    IF EXISTS (SELECT 1 FROM sys.periods WHERE object_id = OBJECT_ID(@tbl))
      EXEC('ALTER TABLE ' + @tbl + ' DROP PERIOD FOR SYSTEM_TIME;');

    IF COL_LENGTH(@tbl,'ValidFrom') IS NOT NULL BEGIN EXEC dbo.__drop_dc @tbl,'ValidFrom'; EXEC('ALTER TABLE ' + @tbl + ' DROP COLUMN ValidFrom;'); END
    IF COL_LENGTH(@tbl,'ValidTo')   IS NOT NULL BEGIN EXEC dbo.__drop_dc @tbl,'ValidTo';   EXEC('ALTER TABLE ' + @tbl + ' DROP COLUMN ValidTo;');   END

    -- add period columns (try HIDDEN first)
    SET @sql = N'
BEGIN TRY
  ALTER TABLE ' + @tbl + N'
    ADD ValidFrom datetime2(7) GENERATED ALWAYS AS ROW START HIDDEN NOT NULL DEFAULT SYSUTCDATETIME(),
        ValidTo   datetime2(7) GENERATED ALWAYS AS ROW END   HIDDEN NOT NULL DEFAULT CONVERT(datetime2(7),''9999-12-31 23:59:59.9999999''),
        PERIOD FOR SYSTEM_TIME (ValidFrom, ValidTo);
END TRY
BEGIN CATCH
  ALTER TABLE ' + @tbl + N'
    ADD ValidFrom datetime2(7) GENERATED ALWAYS AS ROW START NOT NULL DEFAULT SYSUTCDATETIME(),
        ValidTo   datetime2(7) GENERATED ALWAYS AS ROW END   NOT NULL DEFAULT CONVERT(datetime2(7),''9999-12-31 23:59:59.9999999''),
        PERIOD FOR SYSTEM_TIME (ValidFrom, ValidTo);
END CATCH;';
    EXEC sys.sp_executesql @sql;

    -- turn on system-versioning with fixed history table name
    SET @sql = N'
ALTER TABLE ' + @tbl + N'
  SET (SYSTEM_VERSIONING = ON
       (HISTORY_TABLE = SecureData.' + QUOTENAME(@hist) + N',
        DATA_CONSISTENCY_CHECK = ON));';
    EXEC sys.sp_executesql @sql;

    PRINT 'Temporal enabled on ' + @tbl + ' → history: SecureData.' + @hist;
  END
  ELSE
    PRINT 'Skipped (already temporal or missing): ' + ISNULL(@tbl,'<null>');

  FETCH NEXT FROM curT INTO @tbl;
END
CLOSE curT; DEALLOCATE curT;
GO
```

Figure 193 : Temporal enablement loop—add PERIOD columns; SYSTEM_VERSIONING = ON with fixed history tables

```
/* 4) Temporal tables – point-in-time history */
/* ===== Temporal proof: status + timeline + AS-OF snapshots ===== */
/* ============================[ TEMPORAL EVIDENCE ]============================ */
USE MedicalInfoSystem;

-- 0) Status (screenshot this): temporal + history tables are present
SELECT t.name, t.temporal_type_desc, h.name AS HistoryTable
FROM sys.tables t
LEFT JOIN sys.tables h ON h.object_id = t.history_table_id
WHERE SCHEMA_NAME(t.schema_id)='SecureData'
  AND t.name IN (N'Patient', N'Staff', N'AppointmentAndDiagnosis');

-- 1) Make two versions for Patient (safe change + revert). If blocked by RLS, it will skip.
DECLARE @pid  sysname = (SELECT TOP(1) PatientID FROM SecureData.Patient ORDER BY PatientID);
DECLARE @orig nvarchar(200) = (SELECT PatientName FROM SecureData.Patient WHERE PatientID=@pid);

BEGIN TRY
  IF @orig IS NOT NULL
  BEGIN
    UPDATE SecureData.Patient
      SET PatientName = @orig + N' (temporal proof)'
    WHERE PatientID=@pid;

    WAITFOR DELAY '00:00:01';

    UPDATE SecureData.Patient
      SET PatientName = @orig
    WHERE PatientID=@pid;
  END
END TRY BEGIN CATCH
  PRINT 'Patient update skipped (possibly RLS). Proceeding with AppointmentAndDiagnosis demo.';
END CATCH;

-- 2) Guaranteed versions on AppointmentAndDiagnosis (insert -> update -> delete)
DECLARE @anyPatient sysname = (SELECT TOP(1) PatientID FROM SecureData.Patient ORDER BY PatientID);
DECLARE @anyDoctor sysname  = (SELECT TOP(1) StaffID   FROM SecureData.Staff  ORDER BY StaffID);
DECLARE @newDiagID int;

IF @anyPatient IS NOT NULL AND @anyDoctor IS NOT NULL
BEGIN
  INSERT INTO SecureData.AppointmentAndDiagnosis(AppDateTime, PatientID, DoctorID, DiagDetails_Enc)
  VALUES (SYSUTCDATETIME(), @anyPatient, @anyDoctor, NULL);
  SET @newDiagID = SCOPE_IDENTITY();

  UPDATE SecureData.AppointmentAndDiagnosis
    SET AppDateTime = DATEADD(minute, 5, AppDateTime)
  WHERE DiagID = @newDiagID;

  DELETE FROM SecureData.AppointmentAndDiagnosis
  WHERE DiagID = @newDiagID;
END
```

Figure 194 : Temporal evidence (code)—status check + demo operations for history

```sql
-- 3) Patient timeline (ALL versions)
SELECT PatientID, PatientName, ValidFrom, ValidTo
FROM SecureData.Patient FOR SYSTEM_TIME ALL
WHERE PatientID = @pid
ORDER BY ValidFrom;

-- 4) "AS OF" snapshots (compute timestamps first; then use variables in AS OF)
DECLARE @vmin datetime2(7) =
 (SELECT MIN(ValidFrom) FROM SecureData.Patient FOR SYSTEM_TIME ALL WHERE PatientID=@pid);
DECLARE @vmax datetime2(7) =
 (SELECT MAX(ValidFrom) FROM SecureData.Patient FOR SYSTEM_TIME ALL WHERE PatientID=@pid);

IF @vmin IS NOT NULL
BEGIN
  DECLARE @asofEarly  datetime2(7) = DATEADD(millisecond,1,@vmin);
  DECLARE @asofLatest datetime2(7) = DATEADD(millisecond,1,@vmax);

  SELECT 'AS OF early' AS label, PatientID, PatientName, ValidFrom, ValidTo
  FROM SecureData.Patient FOR SYSTEM_TIME AS OF @asofEarly
  WHERE PatientID=@pid;

  SELECT 'AS OF latest' AS label, PatientID, PatientName, ValidFrom, ValidTo
  FROM SecureData.Patient FOR SYSTEM_TIME AS OF @asofLatest
  WHERE PatientID=@pid;
END
ELSE
  PRINT 'No Patient versions yet. See AppointmentAndDiagnosis demo below.';

-- 5) AppointmentAndDiagnosis timeline (shows the insert/update/delete you just did)
IF @newDiagID IS NOT NULL
BEGIN
  SELECT DiagID, AppDateTime, PatientID, DoctorID, DiagDetails_Enc, ValidFrom, ValidTo
  FROM SecureData.AppointmentAndDiagnosis FOR SYSTEM_TIME ALL
  WHERE DiagID = @newDiagID
  ORDER BY ValidFrom;
END
ELSE
  PRINT 'No new DiagID was created (missing Patient/Doctor rows?).';
```

Figure 195 : Timeline & AS-OF queries (code)—FOR SYSTEM_TIME ALL/AS OF

	name	temporal_type_desc	HistoryTable
1	AppointmentAndDiagnosis	SYSTEM_VERSIONED_TEMPORAL_TABLE	AppointmentAndDiagnosisHistory
2	Patient	SYSTEM_VERSIONED_TEMPORAL_TABLE	PatientHistory
3	Staff	SYSTEM_VERSIONED_TEMPORAL_TABLE	StaffHistory

	PatientID	PatientName	ValidFrom	ValidTo
1	P001	Patient P001 (audit demo v2)	2025-09-19 10:12:33.4068003	2025-09-19 11:33:11.7727665
2	P001	Patient P001 (audit demo v2) (temporal proof)	2025-09-19 11:33:11.7727665	2025-09-19 11:33:12.7973113
3	P001	Patient P001 (audit demo v2)	2025-09-19 11:33:12.7973113	2025-09-19 12:01:12.6766947
4	P001	Patient P001 (audit demo)	2025-09-19 12:01:12.6766947	2025-09-19 12:01:12.7386941
5	P001	Patient P001 (audit demo v2)	2025-09-19 12:01:12.7386941	2025-09-19 12:01:23.1031237
6	P001	Patient P001 (audit demo v2)	2025-09-19 12:01:23.1031237	2025-09-19 12:01:50.8779359
7	P001	Patient P001 (audit demo v2) (temporal proof)	2025-09-19 12:01:50.8779359	2025-09-19 12:01:51.8967064
8	P001	Patient P001 (audit demo v2)	2025-09-19 12:01:51.8967064	2025-09-19 21:41:00.9537649

	label	PatientID	PatientName	ValidFrom	ValidTo
1	AS OF early	P001	Patient P001 (audit demo v2)	2025-09-19 10:12:33.4068003	2025-09-19 11:33:11.7727665

	label	PatientID	PatientName	ValidFrom	ValidTo
1	AS OF latest	P001	Patient P001 (audit demo v2)	2025-09-19 21:41:01.9740870	9999-12-31 23:59:59.9999999

	DiagID	AppDateTime	PatientID	DoctorID	DiagDetails_Enc	ValidFrom	ValidTo
1	6	2025-09-19 21:41:01.983	P001	D001	NULL	2025-09-19 21:41:01.9819666	2025-09-19 21:41:01.9879680
2	6	2025-09-19 21:46:01.983	P001	D001	NULL	2025-09-19 21:41:01.9879680	2025-09-19 21:41:01.9919695

Figure 196 : Temporal outputs—status grid, Patient timeline & AS-OF, Appointment timeline

```
BEGIN TRY
  UPDATE SecureData.PatientHistory SET PatientName = N'hack' WHERE PatientID = 'P001';
END TRY
BEGIN CATCH
  SELECT ERROR_NUMBER() AS Err, ERROR_MESSAGE() AS Msg; -- expect temporal history update error
END CATCH;
```

Figure 197 : History immutability test—UPDATE against SecureData.PatientHistory

```
Msg 13561, Level 16, State 1, Line 589
Cannot update rows in a temporal history table 'MedicalInfoSystem.SecureData.PatientHistory'.
```

Figure 198 : Engine block—error "Cannot update rows in a temporal history table"

Figure 192 shows *dbo.__drop_dc*, a helper used during idempotent enablement to remove any default constraints from *ValidFrom/ValidTo* before redefining the PERIOD.

What it proves: safe, repeatable setup without manual constraint hunting.

Figure 193 shows the enablement loop for *SecureData.Patient, SecureData.Staff*, and *SecureData.AppointmentAndDiagnosis*: adds *ValidFrom/ValidTo*, defines *PERIOD FOR SYSTEM_TIME*, and sets *SYSTEM_VERSIONING = ON* with fixed history table names and *DATA_CONSISTENCY_CHECK = ON*.

What it proves: engine-managed history is enabled deterministically for all target tables.

Figure 194 shows evidence scripts: (a) temporal status query and (b) demo sequence creating a version chain (insert → update → delete) to ensure history rows exist.

What it proves: reproducible steps to generate and verify history.

Figure 195 shows *FOR SYSTEM_TIME ALL* (timeline) and *FOR SYSTEM_TIME AS OF* (point-in-time snapshot) queries for *Patient*, with computed timestamps to drive *AS OF* (Microsoft SQL Server, 2025a).

What it proves: support for both full history review and exact point-in-time reconstruction.

Figure 196 shows (top) temporal status with history-table mappings; (middle) Patient timeline and two *AS OF* snapshots; (bottom) *AppointmentAndDiagnosis* timeline for the insert/update/delete demo.

What it proves: visible version windows (*ValidFrom/ValidTo*) and successful point-in-time retrieval.

Figure 197 shows an *UPDATE* attempt against *SecureData.PatientHistory* wrapped in *TRY/CATCH*.

What it proves: explicit test of history tamper resistance.

Figure 198 shows engine error "Cannot update rows in a temporal history table …".

What it proves: temporal history rows are immutable by design; unauthorized changes are blocked.

4.6.2 Solution summary

System-versioned temporal tables are enabled on SecureData.Patient, SecureData.Staff, and SecureData.AppointmentAndDiagnosis with fixed history table names. The engine maintains ValidFrom/ValidTo periods and automatically writes old versions to the corresponding history table on every UPDATE/DELETE. Evidence demonstrates: (1) temporal status and base→ history mapping, (2) full version chains with FOR SYSTEM_TIME ALL, (3) point-in-time reconstruction using FOR SYSTEM_TIME AS OF, and (4) immutability of history (direct UPDATE blocked).

Why this technique:

- Engine-managed & tamper-resistant: versioning and history protection are enforced by SQL Server (no user code paths to bypass), and history writes cannot be updated or deleted directly.
- True point-in-time queries: FOR SYSTEM_TIME AS OF reconstructs exact row state at an arbitrary timestamp—simpler and more reliable than manually stitching changes from triggers or CDC.
- Low operational overhead: no per-row JSON packing or custom trigger logic; the engine with DATA_CONSISTENCY_CHECK support optimizes storage and query plans.
- Deterministic coverage: every UPDATE/DELETE is versioned uniformly; no dependency on application behavior.

This approach delivers verifiable history and precise point-in-time recovery, satisfying Requirement 4 and supporting Requirements 1 and 3.

4.7 Prove audit evidence cannot be tampered with by ordinary users (Requirement 6; supports Requirement 3)

Requirement 3 - Sufficient and suitable protection must be provided to protect all data from accidental and intentional exposure or deletion without compromising functionality and usability.

Requirement 6 - All activities by all users (including attempt to do such activity) must be tracked.

Requirement 6 requires audit evidence to resist tampering. The architecture enforces immutability by disallowing writes to the Audit schema by non-privileged users, and all audit writes are funneled through privileged code paths (as OWNER -the DDL writer procedure and the DML triggers). Negative tests show a non-privileged principal is unable to tamper or remove evidence, and positive tests show that owner-context writers write out events (guaranteeing, at every tick, an accountable audit log), which will be used as evidence supporting Requirement 3: accountability.

4.7.1 Implementation & Evidence — tamper-resistant audit

```
-- lock evidence (inserts still succeed via OWNER context)
BEGIN TRY DENY INSERT, UPDATE, DELETE ON SCHEMA::Audit TO PUBLIC; END TRY BEGIN CATCH END CATCH;
BEGIN TRY DENY  UPDATE, DELETE ON SCHEMA::Audit TO PUBLIC; END TRY BEGIN CATCH END CATCH;
```

Figure 199 : Audit schema locked: DENY INSERT/UPDATE/DELETE to PUBLIC

```
ALTER PROCEDURE Audit.usp_WriteDDLAudit
  @EventXml xml,
  @Actor    sysname
WITH EXECUTE AS OWNER
```

Figure 200 : DDL writer proc runs WITH EXECUTE AS OWNER

```
--=============================================================================
-- 4) DB-LEVEL DDL TRIGGER (CREATE must be 1st stmt in batch)
--=============================================================================
IF EXISTS (SELECT 1 FROM sys.triggers WHERE name=N'TR_DDL_DB_Audit' AND parent_class=0)
    DROP TRIGGER TR_DDL_DB_Audit ON DATABASE;
GO
CREATE TRIGGER TR_DDL_DB_Audit
ON DATABASE
FOR DDL_DATABASE_LEVEL_EVENTS
AS
BEGIN
  SET NOCOUNT ON;

  DECLARE @x xml;       SET @x = CAST(EVENTDATA() AS xml);
  DECLARE @actor sysname; SET @actor = SUSER_SNAME();

  -- ignore events on our own Audit objects (prevents bootstrap noise)
  DECLARE @schema nvarchar(128) = @x.value('(/EVENT_INSTANCE/SchemaName)[1]','nvarchar(128)');
  DECLARE @obj    nvarchar(256) = @x.value('(/EVENT_INSTANCE/ObjectName)[1]' ,'nvarchar(256)');
  IF @schema = N'Audit' AND @obj IN (N'usp_WriteDDLAudit',N'DDLAudit',N'DMLAudit',N'LogonAudit',N'TR_DDL_DB_Audit') RETURN;

  EXEC Audit.usp_WriteDDLAudit @EventXml=@x, @Actor=@actor;
END;
GO
```

Figure 201 : DB-level DDL trigger routes events to owner proc; ignores Audit objects

```
-- Section4.7
-- (1) Tamper attempt (expect permission error)
USE MedicalInfoSystem;
EXECUTE AS LOGIN = N'D001';  -- any ordinary login
BEGIN TRY
  DELETE FROM Audit.DDLAudit;
END TRY
BEGIN CATCH
  SELECT ERROR_NUMBER() AS Err, ERROR_MESSAGE() AS Msg;
END CATCH
REVERT;

-- (2) Fresh evidence still recorded (owner-context writer)
USE MedicalInfoSystem;
CREATE TABLE dbo.__AuditSmoke(id int);
DROP TABLE dbo.__AuditSmoke;

SELECT TOP (10) PostTime, EventType, ObjectSchema, ObjectName, Actor
FROM Audit.DDLAudit
ORDER BY PostTime DESC;
```

Figure 202 : Tamper test script: ordinary login tries DELETE Audit.DDLAudit

Err	Msg
229	The DELETE permission was denied on the object 'DDLAudit', database 'MedicalInfoSystem', schema 'Audit'.

Figure 203 : Result: permission denied (cannot delete audit rows)

PostTime	EventType	ObjectSchema
2025-09-19 22:32:24.3982551	DROP_TABLE	dbo
2025-09-19 22:32:24.3912537	CREATE_TABLE	dbo
2025-09-19 12:01:50.4088298	ALTER_PROCEDURE	dbo
2025-09-19 12:01:50.4018282	ALTER_TRIGGER	SecureData
2025-09-19 12:01:50.3978273	ALTER_TRIGGER	SecureData
2025-09-19 12:01:50.3938268	ALTER_TRIGGER	SecureData
2025-09-19 12:01:50.2843901	CREATE_DATABASE_AUDIT_SPECIFICATION	NULL
2025-09-19 12:01:50.2803894	DROP_DATABASE_AUDIT_SPECIFICATION	NULL
2025-09-19 12:01:50.2773879	ALTER_DATABASE_AUDIT_SPECIFICATION	NULL
2025-09-19 12:01:50.2703874	DENY_DATABASE	

Figure 204 : Smoke test executed: CREATE/DROP __AuditSmoke to generate entries

ObjectName	Actor
__AuditSmoke	DESKTOP-7B1BAM1\user
__AuditSmoke	DESKTOP-7B1BAM1\user
__drop_dc	DESKTOP-7B1BAM1\user
TR_AppointmentAndDiagnosis_DML_Audit	DESKTOP-7B1BAM1\user
TR_Patient_DML_Audit	DESKTOP-7B1BAM1\user
TR_Staff_DML_Audit	DESKTOP-7B1BAM1\user
DB_Audit_MIS	DESKTOP-7B1BAM1\user
DB_Audit_MIS	DESKTOP-7B1BAM1\user
DB_Audit_MIS	DESKTOP-7B1BAM1\user
Audit	DESKTOP-7B1BAM1\user

Figure 205 : Evidence table (Audit.DDLAudit): recent CREATE/DROP/ALTER with who/when/what

This section proves ordinary users can't tamper with audit evidence. First, the *Audit* schema is locked with *DENY INSERT/UPDATE/DELETE* shown in Figure 199, so non-privileged logins have no write/delete rights on *Audit.DDLAudit/Audit.DMLAudit*. Evidence is written only through *Audit.usp_WriteDDLAudit* which runs *WITH EXECUTE AS OWNER* (Figure 200) and is called by the database-wide DDL trigger *TR_DDL_DB_Audit* (Figure 201); the trigger captures *EVENTDATA()* for every DDL and ignores the Audit schema to avoid noise.

An ordinary user then attempts *DELETE FROM Audit.DDLAudit* (

Figure 202) and receives Msg 229 "DELETE permission was denied …" (Figure 203), demonstrating the lockdown. Immediately after, a harmless CREATE/DROP is performed (Figure 204) and the new *CREATE_TABLE/DROP_TABLE* entries appear in *Audit.DDLAudit* with *PostTime*, *EventType*, *object*, and *Actor* (Figure 205). Together these show the trail is

append-only, writer-controlled, and still functioning after a failed tamper—meeting Req 6 and supporting Req 3.

4.7.2 Solution summary

The design achieves tamper resistance by combining permission denials (immutability of Audit tables for ordinary users) with privileged writers (EXECUTE AS OWNER in triggers/procs). This pattern is superior to table-level GRANTs/REVOKEs alone because it both prevents user edits and guarantees evidence capture even when the originating actor has no rights on the Audit schema. The approach also isolates the write surface to vetted code paths (triggers/procs), minimizing the risk of forged entries and satisfying Requirement 6 while supporting Requirement 3 (accountability and non-repudiation).

4.8 Prove backup/restore operations are audited (Requirement 4 and Requirement 6)

Requirement 4 - All data and data changes must be traceable and recoverable including in the event of complete DB failure or loss.

Requirement 6 - All activities by all users (including attempt to do such activity) must be tracked.

This section demonstrates that backup and restore operations are monitored and evidenced end-to-end. The control is the Server Audit specification *ServerAuditSpec_MIS* with *BACKUP_RESTORE_GROUP* enabled, writing to the enterprise audit target *ServerAudit_MIS*. Evidence is read back from the audit files via *sys.fn_get_audit_file*, filtered for *action_id LIKE 'BA%'* (backup) and *action_id LIKE 'RL%'* (restore), which returns timestamp, executing principal, success/failure, and full T-SQL statement. This proves Requirement 4 (traceable recovery operations) by showing every backup/restore with the exact command issued, and Requirement 6 (activity tracking) by capturing both successful and failed attempts tied to the actor and time.

4.8.1 Implementation & Evidence — Server Audit BACKUP_RESTORE_GROUP

```
CREATE SERVER AUDIT SPECIFICATION [ServerAuditSpec_MIS]
FOR SERVER AUDIT [ServerAudit_MIS]
    ADD (BACKUP_RESTORE_GROUP),        -- <- key line for 4.8
    ADD (SUCCESSFUL_LOGIN_GROUP),
    ADD (FAILED_LOGIN_GROUP),
    ADD (AUDIT_CHANGE_GROUP),
    ADD (SERVER_PRINCIPAL_CHANGE_GROUP),
    ADD (SERVER_ROLE_MEMBER_CHANGE_GROUP);
ALTER SERVER AUDIT SPECIFICATION [ServerAuditSpec_MIS] WITH (STATE = ON);
ALTER SERVER AUDIT [ServerAudit_MIS] WITH (STATE = ON);
GO
```

Figure 206 : Server audit spec—BACKUP_RESTORE_GROUP enabled

```
-- Backup/restore evidence (BA*/RL*)
SELECT TOP 50
       event_time,
       server_principal_name,
       action_id,          -- e.g., BA*, RL*
       statement
FROM sys.fn_get_audit_file(@pattern, DEFAULT, DEFAULT)
WHERE action_id LIKE 'BA%' OR action_id LIKE 'RL%'
ORDER BY event_time DESC;
```

Figure 207 : Readback filter—backup/restore events (BA / RL*)*

	event_time	server_principal_name	action_id	statement
4	2025-09-19 21:15:00.7226348	NT SERVICE\SQLSERVERAGENT	BAL	BACKUP LOG [MedicalInfoSystem] TO DISK = @file W...
5	2025-09-19 21:00:00.6946808	NT SERVICE\SQLSERVERAGENT	BAL	BACKUP LOG [MedicalInfoSystem] TO DISK = @file W...
6	2025-09-19 20:45:00.5030882	NT SERVICE\SQLSERVERAGENT	BAL	BACKUP LOG [MedicalInfoSystem] TO DISK = @file W...
7	2025-09-19 20:30:00.9070702	NT SERVICE\SQLSERVERAGENT	BAL	BACKUP LOG [MedicalInfoSystem] TO DISK = @file W...
8	2025-09-19 20:15:00.9265507	NT SERVICE\SQLSERVERAGENT	BAL	BACKUP LOG [MedicalInfoSystem] TO DISK = @file W...
9	2025-09-19 20:00:00.9301141	NT SERVICE\SQLSERVERAGENT	BA	BACKUP DATABASE [MedicalInfoSystem] TO DISK = ...
10	2025-09-19 20:00:00.9301141	NT SERVICE\SQLSERVERAGENT	BAL	BACKUP LOG [MedicalInfoSystem] TO DISK = @file W...
11	2025-09-19 19:45:01.0652670	NT SERVICE\SQLSERVERAGENT	BAL	BACKUP LOG [MedicalInfoSystem] TO DISK = @file W...
12	2025-09-19 19:30:00.6244456	NT SERVICE\SQLSERVERAGENT	BAL	BACKUP LOG [MedicalInfoSystem] TO DISK = @file W...

Figure 208 : Audit evidence—BACKUP DATABASE/LOG rows with actor and statement

Figure 206 depicts the Server Audit Specification adding *BACKUP_RESTORE_GROUP* and within it setting spec and audit to *STATE = ON*.

Proves: the server-scope capture of the backup/restore events is working.

Figure 207 shows a focused readback using *sys.fn_get_audit_file(@pattern, ...)* filtered to *action_id LIKE 'BA%' OR 'RL%'*.

Proves: evidence query separates backup/restore operation into outside of enterprise auditing stream.

Figure 208 shows recent *BA*/BAL* events with *event_time, server_principal_name, action_id,* and *full statement.*

Proves: responsibility for recovery operations—who ran it, what got run, when.

4.8.2 Solution summary

Backup/restore auditing is enforced at the server level by enabling *BACKUP_RESTORE_GROUP* in *ServerAuditSpec_MIS*, directing events to the enterprise *.*sqlaudit* files. Evidence queries return *event_time, server_principal_name, action_id (BA*/RL*)*, and the full T-SQL *statement*, demonstrating traceability of both full and log backups (and restores when performed).

Justification (why this technique):

- **Engine-level coverage**: captures every backup/restore regardless of database, tool, or session origin, stronger than relying on *msdb* job history or ad-hoc logging.
- **Central, tamper-resistant evidence**: audit files are managed by SQL Server Audit; normal users cannot alter generated entries.
- **Full attribution**: audit rows include the executing principal and exact T-SQL, enabling precise accountability and forensic review.
- **Low operational overhead**: no triggers or custom jobs; capture and storage are handled by the audit subsystem.

This configuration provides authoritative, centralized evidence for recovery operations, thereby satisfying Requirements 4 (recovery accountability) and 6 (auditable activities).

5.0 Conclusion

The MedicalInfoSystem is secured end-to-end through a layered design that our documentation implements and evidences, like data protection with role-appropriate exposure, encryption (symmetric for routine PII, asymmetric for diagnoses) and masking, all framed by a clear classification posture so only necessary plaintext is ever revealed or stored in the open (Microsoft SQL, 2024). Integrity, accountability and recoverability are enforced by comprehensive auditing at multiple layers, from Server Audit to file (with a DB Audit Spec), an in-DB DDL journal, per-table DML deltas, and engine-maintained temporal history, giving a single evidence trail for who did what, when, and to which data, while keeping the store tamper-resistant via OWNER-context writers and DENY on the Audit schema (Microsoft SQL Server, 2024g). Authentication visibility is proven with LGIS/LGIF capture and enterprise readback, tying each login (success or failure) to its actor and session context (Microsoft SQL Server, 2024f). Point-in-time reconstruction is demonstrated with system-versioned temporal tables across Patient, Staff, and AppointmentAndDiagnosis, enabling FOR SYSTEM_TIME timelines and AS OF snapshots without custom code. Finally, backup/restore operations are audited at the engine boundary (BA*/RL*), including the exact T-SQL and executing principal, providing authoritative recovery accountability that meets both traceability and activity-tracking requirements (Ultimate IT Security, 2016). Collectively, these controls satisfy the assignment's core drivers confidentiality, integrity, availability, and provable accountability of actions and attempts while keeping operational overhead low and evidence screenshot-ready for verification.

6.0 Reference

1. Ikram, A. (2024, January 30). Securing healthcare data with encryption. puredome. https://www.puredome.com/blog/data-encryption-in-healthcare
2. International Organization for Standardization. (2022). *ISO/IEC 27001 standard – information security management systems*. ISO. https://www.iso.org/standard/27001
3. Joint Task Force. (2020). Security and Privacy Controls for Information Systems and Organizations. *Security and Privacy Controls for Information Systems and Organizations*, *5*(5). https://doi.org/10.6028/nist.sp.800-53r5
4. Microsoft SQL Server. (2023, March 9). sys.dm_server_audit_status (Transact-SQL) - SQL Server. Microsoft.com. https://learn.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/sys-dm-server-audit-status-transact-sql?view=azuresqldb-current
5. Microsoft SQL Server. (2024a, September 3). CREATE SERVER AUDIT SPECIFICATION (Transact-SQL) - SQL Server. Microsoft.com. https://learn.microsoft.com/en-us/sql/t-sql/statements/create-server-audit-specification-transact-sql?view=sql-server-ver17
6. Microsoft SQL Server. (2024b, September 3). EVENTDATA (Transact-SQL) - SQL Server. Microsoft.com. https://learn.microsoft.com/en-us/sql/t-sql/functions/eventdata-transact-sql?view=sql-server-ver17
7. Microsoft SQL Server. (2024c, September 3). EXECUTE AS Clause (Transact-SQL) - SQL Server. Microsoft.com. https://learn.microsoft.com/en-us/sql/t-sql/statements/execute-as-clause-transact-sql?view=sql-server-ver17&tabs=sqlserver
8. Microsoft SQL Server. (2024d, September 6). Create a server audit & database audit specification - SQL Server. Microsoft.com. https://learn.microsoft.com/en-us/sql/relational-databases/security/auditing/create-a-server-audit-and-database-audit-specification?view=sql-server-ver17
9. Microsoft SQL Server. (2024e, September 6). CREATE DATABASE AUDIT SPECIFICATION - SQL Server (Transact-SQL). Microsoft.com. https://learn.microsoft.com/en-us/sql/t-sql/statements/create-database-audit-specification-transact-sql?view=sql-server-ver17
10. Microsoft SQL Server. (2024f, November 19). sys.fn_get_audit_file (Transact-SQL) - SQL Server. Microsoft.com. https://learn.microsoft.com/en-us/sql/relational-

databases/system-functions/sys-fn-get-audit-file-transact-sql?view=sql-server-ver17&tabs=sqlserver

11. Microsoft SQL Server. (2024g, November 26). SQL Server Audit Action Groups and Actions - SQL Server. Microsoft.com. https://learn.microsoft.com/en-us/sql/relational-databases/security/auditing/sql-server-audit-action-groups-and-actions?view=sql-server-ver17
12. Microsoft SQL Server. (2024h, December 17). EXECUTE AS (Transact-SQL) - SQL Server. Microsoft.com. https://learn.microsoft.com/en-us/sql/t-sql/statements/execute-as-transact-sql?view=sql-server-ver17
13. Microsoft SQL Server. (2025a, May 19). Temporal Tables - SQL Server. Microsoft.com. https://learn.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables?view=sql-server-ver17
14. Microsoft SQL Server. (2025b, June 11). SQL Server Audit (Database Engine) - SQL Server. Microsoft.com. https://learn.microsoft.com/en-us/sql/relational-databases/security/auditing/sql-server-audit-database-engine?view=sql-server-ver17&viewFallbackFrom=sql-server-ver1
15. Microsoft SQL Server. (2025c, September 7). Create a System-Versioned Temporal Table - SQL Server. Microsoft.com. https://learn.microsoft.com/en-us/sql/relational-databases/tables/creating-a-system-versioned-temporal-table?view=sql-server-ver17
16. Microsoft SQL. (2023, August 10). Tutorial: Signing Stored Procedures with a Certificate - SQL Server. Microsoft.com. https://learn.microsoft.com/en-us/sql/relational-databases/tutorial-signing-stored-procedures-with-a-certificate?view=sql-server-ver17&
17. Microsoft SQL. (2024, September 3). *ENCRYPTBYKEY (Transact-SQL) - SQL Server*. Microsoft.com. https://learn.microsoft.com/en-us/sql/t-sql/functions/encryptbykey-transact-sql?view=sql-server-ver17&
18. Microsoft SQL. (2025, September 11). Row-Level Security - SQL Server. Microsoft.com. https://learn.microsoft.com/en-us/sql/relational-databases/security/row-level-security?view=sql-server-ver17&
19. Nail. (2025, July 17). Confidentiality, integrity, availability: key examples. DataSunrise. https://www.datasunrise.com/knowledge-center/confidentiality-integrity-availability-examples/

20. Pietervanhove. (2025, April 23). *Always Encrypted - SQL Server*. Microsoft.com. https://learn.microsoft.com/en-us/sql/relational-databases/security/encryption/always-encrypted-database-engine?view=sql-server-ver17&
21. Sabin, S. (2024, July 30). *Data breach recovery has gotten more expensive*. Axios. https://www.axios.com/2024/07/30/ibm-report-data-breach-costs-skyrocketing
22. Satori Cyber. (2022, November 22). Data Masking: 8 techniques and How to Implement them Successfully - Satori. Satori. https://satoricyber.com/data-masking/data-masking-8-techniques-and-how-to-implement-them-successfully
23. Summary of the HIPAA Security Rule. (n.d.). Helth and Human Services of US Department. https://www.hhs.gov/hipaa/for-professionals/security/laws-regulations/index.html
24. Team, C. (2022, October 6). What are the HIPAA requirements for data backup? Intelligent. https://www.itsasap.com/blog/hipaa-requirements-data-backup
25. Thilakanathan, D., Calvo, R. A., Chen, S., Nepal, S., & Glozier, N. (2016). Facilitating secure sharing of personal health data in the cloud. JMIR Medical Informatics, 4(2), e15. https://doi.org/10.2196/medinform.4756
26. Ultimate IT Security. (2016). *SQL Server Audit Action Group: BACKUP_RESTORE_GROUP*. Ultimatewindowssecurity.com. https://www.ultimatewindowssecurity.com/sqlserver/auditpolicy/auditactiongroups/backup_restore_group.aspx
27. What is Data Masking? - Static and Dynamic Data Masking Explained - AWS. (n.d.). Amazon Web Services, Inc. https://aws.amazon.com/what-is/data-masking/