You are absolutely right! My apologies for omitting the "Preferred Implementation Order" section in the last update. It's a crucial part of the roadmap, providing a clear path forward.
I've now re-added the **Preferred Implementation Order** section to the document, strategically placed to guide the development process based on dependencies and logical flow.

Automated Shareholder Information Letter (SIL) Notification System: Strategic Implementation Roadmap

This document provides a comprehensive overview and strategic implementation roadmap for the Automated Shareholder Information Letter (SIL) Notification System. It details the project's scope, business relevance, technical specifications, and a phased development approach, structured to maximize efficiency, mitigate risk, and ensure timely delivery within the banking domain.

Epic: Automated Shareholder Information Letter (SIL) Email Notification System

Epic ID: EP-SIL-001 **Title:** Implement Automated Shareholder Information Letter (SIL) Email Notification System **Status:** Open **Priority:** High **Owner:** [Product Owner Name/Team Lead Name] **Stakeholders:** [Business Unit e.g., Wealth Management, Operations, IT Infrastructure], Compliance, Legal **Creation Date:** 2025-06-28 **Last Updated:** 2025-06-28

1. Business Relevance & Justification

Why is this required? Currently, the processing and distribution of Shareholder Information Letters (SILs) received via email are largely manual. This leads to significant operational overhead, potential delays in communication, and an increased risk of human error. This manual approach is inefficient, especially given the volume of SIL cases and the critical need for timely communication with clients and their advisors regarding vital financial information.
Benefits:

- **Operational Efficiency:** We'll automate the email notification process, significantly cutting down manual effort and processing time.
- **Improved Client Communication:** We'll ensure timely and accurate delivery of SILs to clients, boosting their satisfaction and trust.
- **Enhanced Compliance:** We'll automate adherence to communication protocols, reducing the risk of non-compliance related to information dissemination.
- **Reduced Operational Risk:** We'll minimize human error associated with manual email sending and data handling.
- **Scalability:** This solution will be scalable, handling increasing volumes of SIL cases without proportional increases in manual effort.
- **Auditability:** Centralized logging and automated processes will greatly improve the auditability of SIL distribution.

2. Strategic Alignment

This Epic aligns directly with our bank's strategic goal of digital transformation. We're focusing on automating core business processes, enhancing client service, and strengthening operational resilience through robust technological solutions. It's a key step towards a more efficient and secure information delivery ecosystem.

3. High-Level Scope

We'll develop a **C# .NET 8 console application**, triggered by Windows Task Scheduler, to automate sending email notifications with SIL attachments to clients and their advisors. The application will interact with internal **SOAP (CDC)** and **REST (AIF)** services to retrieve client and advisor details. For email dispatch, it will utilize **MS Graph API** via a MailProcessingClient. **Concurrency safety** will be a foundational design principle.

4. Out of Scope

- Development of the SIL extraction process itself (we're assuming this is an existing upstream process that populates the SILCase and SILCaseDetails tables).
- Development of the CDCClient, AIFClient, MailProcessingClient, and Idp.Logs class libraries (these are assumed to be existing components or will be developed as separate, pre-requisite libraries).
- Any user interface for manual triggering or monitoring (this application is purely backend).
- Complex retry mechanisms beyond basic, configured error handling.

5. Diagrammatic Description (System Interaction Diagram)

```
graph TD
    subgraph Core System
        A[Windows Task Scheduler] -->|Triggers Daily| B(SIL
Notification App - C# .NET 8 Console)
    end

    subgraph Data Sources & Services
        B -->|Reads SILCase, SILCaseDetails| C[SQL Server 2016
Database]
        B -->|Calls AIF REST Service (get account ID by ISIN)| D[AIF
Client (REST)]
        B -->|Calls 6 CDC SOAP Services (get client/advisor details by
Account ID)| E[CDC Client (SOAP)]
        B -->|Sends Emails with Attachments| F[MailProcessingClient
(MS Graph API)]
    end

    subgraph External Systems/Recipients
        F --> G[Client Email Addresses]
        F --> H[Client Advisor Email Addresses]
```

```
end

subgraph Supporting Services
    B --> I[Idp.Logs (.NET 8 ILogger)]
end

classDef default fill:#f9f,stroke:#333,stroke-width:2px;
classDef service fill:#ccf,stroke:#333,stroke-width:2px;
class C,D,E,F service;
class G,H,I fill:#eee,stroke:#ccc;
```

Explanation of Interactions:

- **Windows Task Scheduler:** Initiates the SIL Notification Application daily at a specified time.
- **SIL Notification App:** The central orchestrator.
    - Reads SIL case data from the **SQL Server Database**.
    - Interacts with **AIF Client (REST)** to get account_id for unique ISINs.
    - Interacts with **CDC Client (SOAP)** to retrieve client and advisor details for unique account IDs.
    - Utilizes the **MailProcessingClient** (via MS Graph API) to dispatch emails.
    - Leverages **Idp.Logs** for comprehensive logging.
- **SQL Server Database:** Stores processed SIL case information (SILCase, SILCaseDetails).
- **AIF Client:** Provides an interface to the AIF REST service for retrieving account IDs.
- **CDC Client:** Provides an interface to the six CDC SOAP services for retrieving client and advisor details.
- **MailProcessingClient:** Handles email sending functionality via Microsoft Graph API.
- **Client/Client Advisor Email Addresses:** The final recipients of the notification emails.
- **Idp.Logs:** The centralized logging solution for the application.

6. Epic Goals & Success Criteria

Goals:

- Automate the distribution of SILs to relevant clients and client advisors.
- Ensure timely and accurate delivery of SILs.
- Maintain data integrity and concurrency safety across multiple instances.

Success Criteria:

- **99%** of eligible SIL cases processed daily result in successful email notifications to clients and advisors.
- The application reliably runs daily without manual intervention.
- We observe no data corruption or concurrency issues on the SQL Server database.
- Comprehensive logging allows for easy troubleshooting and auditing.
- Unit and Integration test coverage meets the **80%** threshold.

7. Dependencies

- **External Services:** Availability and stability of AIF REST service, 6 CDC SOAP services, and Microsoft Graph API.

- **Database:** Availability and performance of SQL Server 2016.
- **Existing Libraries:** CDCClient, AIFClient, MailProcessingClient, Idp.Logs class libraries must be stable and functional.
- **Infrastructure:** Windows Servers for deployment, configured with necessary network access and Task Scheduler.
- **Upstream Processes:** The process that populates SILCase and SILCaseDetails tables must be reliable and timely.

8. Assumptions

- Email addresses for clients and advisors retrieved from CDC services are valid and active.
- SIL attachments are accessible and correctly stored as per SILCase table.
- Network connectivity to all external services (AIF, CDC, MS Graph) is stable.
- The business day definition (12 AM to 4 PM) is consistently applied.
- SILCaseDetails.case_details JSON accurately contains isin and valor information.
- The application will run on multiple Windows servers with shared database access.
- Necessary permissions for MailProcessingClient to send emails via MS Graph API are granted.

9. Potential Risks & Mitigations

- **Service Unavailability:** We'll implement robust error handling, retry mechanisms (with back-off), and alerting for failures in CDC or AIF service calls.
- **Concurrency Issues:** We'll utilize database transactions, optimistic/pessimistic locking strategies, and appropriate .NET concurrency primitives (e.g., ConcurrentDictionary, SemaphoreSlim) to ensure data integrity.
- **Performance Bottlenecks:** We'll implement asynchronous programming, batch processing where possible, and optimize database queries. We'll also monitor application performance.
- **Email Delivery Failures:** We'll implement robust error logging for email sending, and potentially a mechanism to flag failed emails for manual review/resend.
- **Data Inconsistencies:** We'll implement data validation at various stages.
- **Security Vulnerabilities:** We'll follow secure coding practices, protect sensitive data (e.g., API keys, connection strings), and regularly update dependencies.

User Stories (GitLab Issues)

We're using the **INVEST** (Independent, Negotiable, Valuable, Estimable, Small, Testable) framework, combined with detailed **Acceptance Criteria** and a **Definition of Done**, which is widely used in the banking domain for its clarity and rigor.

US-SIL-012: Initial C# .NET 8 Console Application Setup

Title: As a developer, I want to create the foundational C# .NET 8 console application project, so that it can serve as the executable for the SIL Notification System. **Status:** To Do **Priority:** Highest **Labels:** Setup, Backend, Foundation **Assignee:** [Developer Name] **Sprint:** [Sprint

Number]
Description: This story covers the very first step in technical implementation: establishing the core C# .NET 8 console application project. This involves creating the project, configuring its basic properties, setting up the main entry point (Program.cs), and ensuring it can successfully compile and execute as a standalone application. This executable will be the container for all subsequent logic developed in other user stories.
Example Implementation Steps (conceptual):

1. Execute dotnet new console -n SILNotificationApp -f net8.0 in the solution directory.
2. Add project to the solution file if applicable.
3. Verify Program.cs contains a basic Console.WriteLine("SIL Notification App Starting...");
4. Run dotnet build and dotnet run to confirm successful compilation and execution.

Dependencies:

- None (this is the foundational story).

Pre-conditions:

- A development environment with .NET 8 SDK installed.
- Access to the project's source code repository (e.g., GitLab).

Acceptance Criteria:

- A new C# .NET 8 console application project is successfully created within the solution structure.
- The project is named SILNotificationApp (or as agreed).
- The project targets .NET 8.
- The project can be successfully built using dotnet build.
- The compiled executable can be run from the command line and produces a simple output (e.g., "Hello World" or "Application started").
- The project is committed to the version control system (GitLab).
- Basic project structure for future components (e.g., Models, Services, DataAccess) is considered.
- **Unit Tests:** (Not applicable for this story itself, as it's project setup, but lays foundation for future tests).
- **Integration Tests:** (Not applicable for this story itself).

Definition of Done:

- C# .NET 8 console application project created and committed.
- Project successfully builds and executes locally.
- Basic README.md or project setup notes are added.

US-SIL-010: SIL Case and Details Ingestion

Title: As an SIL Processing System, I want to automatically ingest new SIL cases and their extracted details into the database, so that they are available for further processing and notification. **Status:** To Do **Priority:** High **Labels:** Backend, Data Ingestion, Database Design **Assignee:** [Developer Name] **Sprint:** [Sprint Number]
Description: This story defines how raw SIL cases, received by email, become structured data stored in our database. It covers the design of SILCase, SILCaseAttachment, and SILCaseDetail tables. The story assumes an upstream "extraction process" (whether manual or automated via RPA/OCR) provides the structured input.
Database Table Design Considerations:

- **SILCase Table:**
    - Id (**PK**, INT/BIGINT, **IDENTITY**): Unique identifier for each SIL case.

    - CaseId (NVARCHAR(50), **UNIQUE**, **NOT NULL**): Business-specific case ID (e.g., from an email subject or internal system).
    - ReceivedTimestamp (DATETIME2, **NOT NULL**): When the SIL email was originally received.
    - CreationTimestamp (DATETIME2, **NOT NULL**, **DEFAULT GETDATE()**): When the case record was created in the database.
    - CaseStatus (NVARCHAR(50), **NOT NULL**, **DEFAULT 'Received'**): Current status (e.g., 'Received', 'Extracted', 'Processed', 'Failed', 'ReadyForNotification').
    - EmailsSentToClient (BIT, **NOT NULL**, **DEFAULT 0**): Flag if email sent to client.
    - EmailsSentToCA (BIT, **NOT NULL**, **DEFAULT 0**): Flag if email sent to client advisor.
    - ProcessingAttempts (INT, **NOT NULL**, **DEFAULT 0**): Number of notification attempts.
    - LastProcessedTimestamp (DATETIME2, **NULL**): Last attempt timestamp for notification.
    - Remarks (NVARCHAR(MAX), **NULL**): Relevant remarks or error messages.
    - **ProcessingInstanceId (NVARCHAR(100), NULL)**: Unique ID of the instance currently processing this case.
    - **ProcessingTimestamp (DATETIME2, NULL)**: Timestamp when the case was claimed by an instance.
    - **AttemptCount (INT, NOT NULL, DEFAULT 0)**: Total attempts to process this SIL case.
- **SILCaseAttachment Table (NEW):**
    - Id (**PK**, INT/BIGINT, **IDENTITY**): Unique identifier for each attachment record.
    - SILCaseId (**FK**, INT/BIGINT, **NOT NULL**): Links to SILCase.Id.
    - OriginalFilename (NVARCHAR(255), **NOT NULL**): The original attachment filename.
    - FilePath (NVARCHAR(MAX), **NOT NULL**): Local/network path where the attachment is temporarily stored before Dox upload.
    - DoxDocumentId (NVARCHAR(100), **NULL**): Document ID from Dox after upload.
    - DoxUploadStatus (NVARCHAR(50), **NOT NULL**, **DEFAULT 'Pending'**): Status of Dox upload (e.g., 'Pending', 'Uploaded', 'Failed').
    - DoxUploadTimestamp (DATETIME2, **NULL**): Timestamp of successful Dox upload.
- **SILCaseDetail Table (renamed from SILCaseDetails):**
    - Id (**PK**, INT/BIGINT, **IDENTITY**): Unique identifier for each detail record.
    - SILCaseId (**FK**, INT/BIGINT, **NOT NULL**): Links to SILCase.Id.
    - SequenceNumber (INT, **NOT NULL**): For ordering multiple detail entries per case.
    - ExtractionData (NVARCHAR(MAX), **NOT NULL**): JSON string of extracted financial instrument data (ISIN, valor, ISIN name).
    - ExtractionTimestamp (DATETIME2, **NOT NULL**): When the data was extracted.

Example Input (from an upstream extraction process):

```
{
  "CaseId": "SIL20250628-001",
  "ReceivedTimestamp": "2025-06-28T09:00:00Z",
  "OriginalAttachments": [
    {"Filename": "ShareholderLetter_A.pdf", "LocalPath":
"C:\\SIL_Staging\\ShareholderLetter_A.pdf"},
    {"Filename": "Annex_B.pdf", "LocalPath":
"C:\\SIL_Staging\\Annex_B.pdf"}
```

```
  ],
  "ExtractedData": {
    "financialInstruments": [
      {"isin": "CH1234567890", "valor": "VAL1", "name": "Swiss Equity
Fund"},
      {"isin": "DE9876543210", "valor": "VAL2", "name": "German Bond
XYZ"}
    ],
    "additionalInfo": "..."
  }
}
```

Expected Database Entries After Ingestion:

- **SILCase:**
  - Id: 1, CaseId: 'SIL20250628-001', ReceivedTimestamp: '2025-06-28 09:00:00', CreationTimestamp: (current time), CaseStatus: 'Extracted', EmailsSentToClient: 0, EmailsSentToCA: 0, ProcessingAttempts: 0, LastProcessedTimestamp: NULL, Remarks: NULL, ProcessingInstanceId: NULL, ProcessingTimestamp: NULL, AttemptCount: 0
- **SILCaseAttachment:**
  - Id: 1, SILCaseId: 1, OriginalFilename: 'ShareholderLetter_A.pdf', FilePath: 'C:\SIL_Staging\ShareholderLetter_A.pdf', DoxDocumentId: NULL, DoxUploadStatus: 'Pending', DoxUploadTimestamp: NULL
  - Id: 2, SILCaseId: 1, OriginalFilename: 'Annex_B.pdf', FilePath: 'C:\SIL_Staging\Annex_B.pdf', DoxDocumentId: NULL, DoxUploadStatus: 'Pending', DoxUploadTimestamp: NULL
- **SILCaseDetail:**
  - Id: 1, SILCaseId: 1, SequenceNumber: 1, ExtractionData: {"financialInstruments": [{"isin": "CH1234567890", "valor": "VAL1", "name": "Swiss Equity Fund"}, {"isin": "DE9876543210", "valor": "VAL2", "name": "German Bond XYZ"}]}, ExtractionTimestamp: (current time)

Dependencies:

- **US-SIL-012** (Foundational application setup).
- Implicit dependency on the existence of the upstream SIL extraction process.

Pre-conditions:

- Database schema for SILCase, SILCaseAttachment, and SILCaseDetail tables are created.
- Upstream SIL extraction process provides structured data.
- Raw SIL attachment files are in a designated staging directory.

Acceptance Criteria:

- The ingestion process successfully receives/retrieves structured SIL case data.
- A new record is inserted into the SILCase table for each new case, with CaseStatus appropriately set (e.g., 'Extracted' or 'ReadyForDoxUpload').
- For each attachment, a new record is inserted into SILCaseAttachment, linking to SILCase, storing OriginalFilename, FilePath, and initializing DoxUploadStatus to 'Pending'.
- Extracted financial instrument data is stored as JSON in ExtractionData of SILCaseDetail, linked to SILCase.

- Data integrity constraints (e.g., foreign keys, unique constraints) are enforced.
- Robust error handling is implemented for invalid input, database connection issues, and insertion failures, with detailed logging using Idp.Logs.
- The ingestion process handles multiple concurrent inputs without data corruption or deadlocks.
- **Unit Tests:** Cover data parsing from input, object mapping, and validation rules.
- **Integration Tests:** Verify correct data insertion into all three tables for various valid/invalid inputs.

Definition of Done:

- Database schema for new tables finalized and applied.
- Ingestion code implemented and reviewed.
- All acceptance criteria met.
- Unit tests passed with 80%+ code coverage for the ingestion component.
- Integration tests passed.
- Logging configured and verified.
- Documentation updated for table schemas and ingestion process.

US-SIL-011: Attachment Upload to Dox System

Title: As an SIL Processing System, I want to upload all attachments associated with an SIL case to the Dox system via AIF REST services, so that a DoxDocumentId is obtained and updated in the database for future reference. **Status:** To Do **Priority:** High **Labels:** Backend, External Service Integration, Dox Upload **Assignee:** [Developer Name] **Sprint:** [Sprint Number] **Description:** This story handles uploading SIL attachment files to the Dox document management system. For each SILCaseAttachment record with a 'Pending' DoxUploadStatus, the application will read the file from FilePath, call the AIF REST service for Dox upload, and update the DoxDocumentId and DoxUploadStatus in the SILCaseAttachment table upon success. It must handle multiple attachments per case.

Example Input (from SILCaseAttachment table):

```
Id: 1, SILCaseId: 1, OriginalFilename: 'ShareholderLetter_A.pdf',
FilePath: 'C:\SIL_Staging\ShareholderLetter_A.pdf', DoxDocumentId:
NULL, DoxUploadStatus: 'Pending'
Id: 2, SILCaseId: 1, OriginalFilename: 'Annex_B.pdf', FilePath:
'C:\SIL_Staging\Annex_B.pdf', DoxDocumentId: NULL, DoxUploadStatus:
'Pending'
```

Example AIF Dox Upload Service Call (conceptual): POST /api/dox/upload with file content in request body and metadata (e.g., filename) in headers/form-data.

Example AIF Dox Upload Service Response (conceptual JSON for success): HTTP 200 OK, Body: {"documentId": "DOXID-ABC-123"}

Expected Database Entries After Dox Upload:

- **SILCaseAttachment (after successful upload of first attachment):**
  - Id: 1, SILCaseId: 1, OriginalFilename: 'ShareholderLetter_A.pdf', FilePath: 'C:\SIL_Staging\ShareholderLetter_A.pdf', DoxDocumentId: 'DOXID-ABC-123', DoxUploadStatus: 'Uploaded', DoxUploadTimestamp: (current time)
- **SILCaseAttachment (after successful upload of second attachment):**
  - Id: 2, SILCaseId: 1, OriginalFilename: 'Annex_B.pdf', FilePath: 'C:\SIL_Staging\Annex_B.pdf', DoxDocumentId: 'DOXID-DEF-456',

DoxUploadStatus: 'Uploaded', DoxUploadTimestamp: (current time)

Dependencies:

- **US-SIL-010** (for SILCaseAttachment table to be populated).
- **US-SIL-006** (for concurrency safety when updating DoxDocumentId and DoxUploadStatus).
- External AIFClient library (assumed to be available).

Pre-conditions:

- SILCaseAttachment table contains records with DoxUploadStatus = 'Pending' and valid FilePath entries.
- AIFClient is extended to support the Dox upload REST service.
- Dox system (via AIF REST service) is accessible and operational.
- Files specified in FilePath exist and are accessible for reading.

Acceptance Criteria:

- The application queries SILCaseAttachment for 'Pending' records.
- For each 'Pending' attachment:
  - The attachment file is read from its FilePath.
  - A call is made to the AIFClient's Dox upload REST service.
  - Upon successful response, DoxDocumentId and DoxUploadStatus are updated to 'Uploaded' and DoxUploadTimestamp is set.
  - If upload fails, DoxUploadStatus is 'Failed', and relevant error details are logged (Idp.Logs). Retries for transient errors (e.g., 3 retries with exponential backoff) are implemented.
- **Concurrency safety** is maintained when updating DoxUploadStatus and DoxDocumentId for multiple attachments/concurrent cases (refer to **US-SIL-006**).
- The application handles invalid FilePath or non-existent files gracefully (logs error, marks status 'Failed').
- **Unit Tests:** Cover file reading, AIFClient Dox upload mock interactions (success/various failures), and database update logic.
- **Integration Tests:** Verify successful file reading, AIFClient call, documentId receipt, and correct SILCaseAttachment table update. Test file not found and Dox service errors.

Definition of Done:

- Code implemented for Dox upload and database update.
- All acceptance criteria met.
- Unit tests passed with 80%+ code coverage.
- Integration tests passed.
- Logging configured and verified.
- Documentation updated for Dox integration.

US-SIL-001: Data Preparation - Unique ISINs

Title: As an SIL Processing System, I want to identify all unique ISINs from processed SIL cases within a business day, so that I can efficiently query related account information. **Status:** To Do **Priority:** High **Labels:** Backend, Data Processing, Core Functionality **Assignee:** [Developer Name] **Sprint:** [Sprint Number]

Description: This story focuses on the initial data aggregation. The application will query SILCase and SILCaseDetail tables to find all SIL cases processed within the business day (12 AM to 4 PM). From these, it will parse the case_details JSON in SILCaseDetail to extract all **unique ISINs**. This list will then feed into the AIF service call.

Example Data:

- **SILCase Table:**
  - case_id: 'SIL001', case_status: 'Processed', creation_timestamp: '2025-06-28 10:30:00'
  - case_id: 'SIL002', case_status: 'Processed', creation_timestamp: '2025-06-28 15:00:00'
- **SILCaseDetail Table (for SIL001):**
  - case_id: 'SIL001', extraction_data: {"financialInstruments": [{"isin": "CH1234567890", "valor": "V123"}, {"isin": "LU9876543210", "valor": "V456"}]}
- **SILCaseDetail Table (for SIL002):**
  - case_id: 'SIL002', extraction_data: {"financialInstruments": [{"isin": "CH1234567890", "valor": "V123"}, {"isin": "US1122334455", "valor": "V789"}]}

Expected Output Example: ["CH1234567890", "LU9876543210", "US1122334455"] (unique ISINs from eligible cases)

Dependencies:

- **US-SIL-010** (for SILCase and SILCaseDetail tables to be populated).
- **US-SIL-006** (for atomic claiming of cases to determine which ones this instance will process).

Pre-conditions:

- SILCase and SILCaseDetail tables are populated.
- SQL Server 2016 database is accessible.

Acceptance Criteria:

- The application connects to SQL Server.
- It queries SILCase records where case_status is 'Processed' and creation_timestamp is between 12:00:00 AM and 04:00:00 PM of the current business day.
- For each selected SILCase, the corresponding SILCaseDetail record is retrieved.
- The extraction_data JSON is parsed to get all ISIN values.
- A distinct list of all extracted ISINs is generated.
- Error handling is implemented for database connectivity and JSON parsing, logging with Idp.Logs.
- **Unit Tests:** Cover ISIN extraction from various JSON formats (multiple instruments, single, empty array) and edge cases (malformed JSON, missing ISIN field).
- **Integration Tests:** Verify connection to database and correct retrieval/parsing of data to produce the unique ISIN list for a sample dataset.

Definition of Done:

- Code implemented and reviewed.
- All acceptance criteria met.
- Unit tests passed with 80%+ code coverage for this component.
- Integration tests passed.
- Logging configured and verified.
- Documentation updated (if applicable).

US-SIL-002: Account ID Retrieval via AIF Service

Title: As an SIL Processing System, I want to retrieve account IDs for each unique ISIN, so that I can establish the relationship between financial instruments and client accounts. **Status:** To Do **Priority:** High **Labels:** Backend, External Service Integration, Data Mapping **Assignee:** [Developer Name] **Sprint:** [Sprint Number]

Description: This story involves calling the AIF REST service to get account_id for each unique ISIN from the previous step. The application will loop through the unique ISIN list, call AIFClient, and store the returned account_id in a structured format, mapping it back to its ISIN(s). The output should be JSON-formatted (account_id to ISIN relationships).
Example Input: ["CH1234567890", "LU9876543210", "US1122334455"]
Example AIF Service Call (conceptual): GET /api/accounts?isin=CH1234567890
Example AIF Service Response (conceptual JSON):

- For CH1234567890: {"accountId": "ACC001"}
- For LU9876543210: {"accountId": "ACC002"}
- For US1122334455: {"accountId": "ACC001"} (Note: multiple ISINs can map to one account)

Expected Output Example (JSON format):

```
{
  "ACC001": ["CH1234567890", "US1122334455"],
  "ACC002": ["LU9876543210"]
}
```

Dependencies:

- **US-SIL-001** (Unique ISINs list).
- **US-SIL-006** (for application-level concurrency when making AIF calls).
- External AIFClient library (assumed to be available).

Pre-conditions:

- Unique list of ISINs available from US-SIL-001.
- AIFClient library is integrated and functional.
- AIF REST service is accessible and operational.

Acceptance Criteria:

- The application iterates through the unique ISINs.
- For each ISIN, it calls AIFClient to get the account_id.
- A data structure (e.g., Dictionary<string, List<string>>) is created to map each account_id to its ISINs.
- A list of unique account_ids is generated.
- The account_id to ISIN mapping is serialized to JSON.
- Error handling is implemented for AIF service call failures (e.g., HTTP 404, 500), with logging via Idp.Logs.
- Retry mechanism for transient AIF errors (e.g., 3 retries with exponential backoff of 1, 2, 4 seconds) is considered.
- **Concurrency for AIF calls** (e.g., Parallel.ForEach for up to 5 concurrent calls) is handled using application-level concurrency (refer to **US-SIL-006**).
- **Unit Tests:** Cover AIFClient mock interactions (valid IDs, null, exceptions), JSON serialization, and error handling.
- **Integration Tests:** Verify successful AIFClient calls and valid account_ids received for given ISINs.

Definition of Done:

- Code implemented and reviewed.
- All acceptance criteria met.
- Unit tests passed with 80%+ code coverage.
- Integration tests passed.
- Logging configured and verified.

- Documentation updated (if applicable).

US-SIL-003: Client and Advisor Details Retrieval via CDC Services

Title: As an SIL Processing System, I want to retrieve detailed client and advisor information for each unique account ID, so that I can prepare the necessary contact details for email notifications. **Status:** To Do **Priority:** High **Labels:** Backend, External Service Integration, Data Enrichment **Assignee:** [Developer Name] **Sprint:** [Sprint Number]
Description: This story enriches data with client/advisor details. For each unique account_id from the previous step, the application will call the six CDC SOAP services via CDCClient. It will gather client email, advisor email, client SIL subscription, client email waiver, and advisor GPNID. This will be consolidated into a JSON format, mapping each account_id to its client/advisor details.
Example Input: Unique account IDs: ["ACC001", "ACC002"]
Example CDC Service Calls (conceptual, assume 6 different services CDCService1 to CDCService6):

- CDCService1.GetClientEmail(accountId: "ACC001") -> "client1@example.com"
- CDCService2.GetAdvisorEmail(accountId: "ACC001") -> "advisor1@example.com"
- CDCService3.GetClientSILSubscription(accountId: "ACC001") -> true
- CDCService4.GetClientEmailWaiver(accountId: "ACC001") -> false
- CDCService5.GetAdvisorGPNID(accountId: "ACC001") -> "GPN12345"

Expected Output Example (JSON format for client/advisor details):

```
{
  "ACC001": {
    "clientEmail": "client1@example.com",
    "advisorEmail": "advisor1@example.com",
    "clientSilSubscription": true,
    "clientEmailWaiver": false,
    "advisorGpnId": "GPN12345"
  },
  "ACC002": {
    "clientEmail": "client2@example.com",
    "advisorEmail": "advisor2@example.com",
    "clientSilSubscription": false,
    "clientEmailWaiver": true,
    "advisorGpnId": "GPN67890"
  }
}
```

Dependencies:

- **US-SIL-002** (Unique account IDs list).
- **US-SIL-006** (for application-level concurrency when making CDC calls).
- External CDCClient library (assumed to be available).

Pre-conditions:

- Unique list of account_ids available from US-SIL-002.
- CDCClient library is integrated and functional, exposing methods for all 6 CDC services.
- All 6 CDC SOAP services are accessible and operational.

Acceptance Criteria:

- The application iterates through the unique account_ids.
- For each account_id, it calls all 6 CDC services via CDCClient.
- The following details are extracted and stored: Client Email, Advisor Email, Client SIL Subscription Status, Client Email Waiver Status, Advisor GPNID.
- A data structure mapping account_id to consolidated client/advisor details is created.
- The mapping is serialized to JSON.
- Error handling is implemented for CDC service call failures (e.g., network issues, partial failures across 6 services), logging with Idp.Logs.
- **Concurrent calls to CDC services** (e.g., Task.WhenAll for calls within an account) are handled using application-level concurrency (refer to **US-SIL-006**).
- **Unit Tests:** Cover CDCClient mock interactions (various responses from each of 6 services), data consolidation, and JSON serialization.
- **Integration Tests:** Verify successful CDCClient calls and valid client/advisor details for given account IDs.

Definition of Done:

- Code implemented and reviewed.
- All acceptance criteria met.
- Unit tests passed with 80%+ code coverage.
- Integration tests passed.
- Logging configured and verified.
- Documentation updated (if applicable).

US-SIL-004: Relationship Consolidation

Title: As an SIL Processing System, I want to consolidate the relationships between SIL cases, ISINs, account IDs, and client/advisor details, so that I have a complete picture for targeted email notifications. **Status:** To Do **Priority:** High **Labels:** Backend, Data Aggregation, Core Functionality **Assignee:** [Developer Name] **Sprint:** [Sprint Number]
Description: This story builds the final comprehensive relationship structure for email sending. It combines output from US-SIL-001 (SIL Cases to ISINs), US-SIL-002 (ISINs to Account IDs), and US-SIL-003 (Account IDs to Client/Advisor Details). The goal is a clear mapping from each eligible SILCase to its associated client(s) and advisor(s), including email addresses and subscription/waiver flags.

Example Inputs:

- **From US-SIL-001:**
  - Eligible SIL Cases with ISINs: SIL001: ["CH1234567890", "LU9876543210"], attachements_names: "SIL001_Doc1.pdf,SIL001_Doc2.pdf"
  - SIL002: ["CH1234567890", "US1122334455"], attachements_names: "SIL002_DocA.pdf"
- **From US-SIL-002:**
  - ISIN to Account ID map: {"ACC001": ["CH1234567890", "US1122334455"], "ACC002": ["LU9876543210"]}
- **From US-SIL-003:**
  - Account ID to Client/Advisor details map:

```
{
  "ACC001": {"clientEmail": "client1@example.com",
"advisorEmail": "advisor1@example.com",
"clientSilSubscription": true, "clientEmailWaiver": false},
```

```
  "ACC002": {"clientEmail": "client2@example.com",
"advisorEmail": "advisor2@example.com",
"clientSilSubscription": false, "clientEmailWaiver": true}
}
```

Expected Output Example (Consolidated Data Structure for Email Sending):

```
[
  {
    "caseId": "SIL001",
    "attachments": ["SIL001_Doc1.pdf", "SIL001_Doc2.pdf"],
    "recipients": [
      {"type": "Client", "email": "client1@example.com", "sendSil":
true, "emailWaiver": false},
      {"type": "Advisor", "email": "advisor1@example.com"}
    ]
  },
  {
    "caseId": "SIL002",
    "attachments": ["SIL002_DocA.pdf"],
    "recipients": [
      {"type": "Client", "email": "client1@example.com", "sendSil":
true, "emailWaiver": false},
      {"type": "Advisor", "email": "advisor1@example.com"}
    ]
  }
]
```

Dependencies:

- **US-SIL-001** (Provides unique ISINs from SIL Cases).
- **US-SIL-002** (Provides ISIN to Account ID mapping).
- **US-SIL-003** (Provides Account ID to Client/Advisor details mapping).

Pre-conditions:

- Output from US-SIL-001, US-SIL-002, US-SIL-003 is available.

Acceptance Criteria:

- The application loads and processes JSON outputs from US-SIL-002 and US-SIL-003.
- For each eligible SIL case, it identifies associated ISIN(s) from SILCaseDetail.
- It determines corresponding account_id(s) using ISINs.
- It retrieves associated client/advisor details using account_id(s).
- A final, consolidated data structure maps each SILCase (case_id) to client/advisor emails, SIL subscription/waiver status, and attachment names.
- The consolidated data is prepared for email sending.
- Error handling is implemented for inconsistencies/missing data during consolidation (e.g., ISIN without account ID), logged with Warning level.
- **Unit Tests:** Cover data merging scenarios, including incomplete data or multiple ISINs mapping to one account.
- **Integration Tests:** Verify end-to-end data flow from initial SIL cases to the final consolidated structure using sample input.

Definition of Done:

- Code implemented and reviewed.
- All acceptance criteria met.
- Unit tests passed with 80%+ code coverage.
- Integration tests passed.
- Logging configured and verified.
- Documentation updated (if applicable).

US-SIL-005: Email Notification Dispatch

Title: As an SIL Processing System, I want to send email notifications with SIL attachments to clients and their advisors for eligible cases, so that they receive timely and relevant information.
Status: To Do **Priority:** High **Labels:** Backend, Email Sending, Core Functionality **Assignee:** [Developer Name] **Sprint:** [Sprint Number]
Description: This is the core email dispatch story. The application will iterate through the consolidated data from US-SIL-004. For each SILCase, it will determine recipients (client, advisor, or both) based on client_sil_subscription and client_email_waiver. It will then use MailProcessingClient (MS Graph API) to send the email with SIL attachments (via DoxDocumentId). **Concurrency safety** is critical when updating EmailsSentToClient and EmailsSentToCA flags in the SILCase table (refer to **US-SIL-006**).
Example Input (from US-SIL-004):

```
[
  {
    "caseId": "SIL001",
    "attachments": ["SIL001_Doc1.pdf", "SIL001_Doc2.pdf"],
    "recipients": [
      {"type": "Client", "email": "client1@example.com", "sendSil":
true, "emailWaiver": false},
      {"type": "Advisor", "email": "advisor1@example.com"}
    ]
  },
  {
    "caseId": "SIL002",
    "attachments": ["SIL002_DocA.pdf"],
    "recipients": [
      {"type": "Client", "email": "client2@example.com", "sendSil":
false, "emailWaiver": true},
      {"type": "Advisor", "email": "advisor2@example.com"}
    ]
  }
]
```

Example Email Sending Logic (pseudo-code):

- **For SIL001:**
  - Client has sendSil: true and emailWaiver: false -> Send email to client1@example.com with attachments "SIL001_Doc1.pdf", "SIL001_Doc2.pdf".
  - Advisor has email advisor1@example.com -> Send email to advisor1@example.com with attachments.
- **For SIL002:**

    - Client has sendSil: false and emailWaiver: true -> **DO NOT** send email to client.
    - Advisor has email advisor2@example.com -> Send email to advisor2@example.com with attachment "SIL002_DocA.pdf".

Example Database Updates (conceptual):

- After successfully sending email for SIL001 to client1: UPDATE SILCase SET EmailsSentToClient = TRUE WHERE case_id = 'SIL001'
- After successfully sending email for SIL001 to advisor1: UPDATE SILCase SET EmailsSentToCA = TRUE WHERE case_id = 'SIL001'

Dependencies:

- **US-SIL-004** (Consolidated relationship data).
- **US-SIL-006** (For concurrency safety when updating EmailsSentToClient and EmailsSentToCA flags).
- External MailProcessingClient library (assumed to be available).

Pre-conditions:

- Consolidated relationship data from US-SIL-004 is available.
- MailProcessingClient library is integrated and functional.
- MS Graph API is accessible and configured for email sending.
- SIL attachments are accessible (e.g., C:\SIL_Attachments\SIL001_Doc1.pdf).

Acceptance Criteria:

- The application loops through each eligible SILCase from consolidated data.
- For each SILCase:
    - If client_sil_subscription is true AND client_email_waiver is false, an email is sent to the client.
    - An email is sent to the client advisor.
    - MailProcessingClient is invoked to send email with SIL attachments (using DoxDocumentId from SILCaseAttachment).
- After successful email dispatch, SILCase is updated: EmailsSentToClient to true (if sent), EmailsSentToCA to true (if sent). **Database updates for email flags are concurrency-safe** (refer to **US-SIL-006**).
- Robust error handling is implemented for email sending failures (e.g., invalid email, attachment errors, API rate limiting), with logging via Idp.Logs. Failures shouldn't block other cases.
- Consideration for batching/parallelizing email sends.
- **Unit Tests:** Cover email content generation, recipient logic (subscription/waiver), MailProcessingClient mock interactions, and SILCase status update logic.
- **Integration Tests:** Verify sending emails via MailProcessingClient (to a test account) and correct database updates.

Definition of Done:

- Code implemented and reviewed.
- All acceptance criteria met.
- Unit tests passed with 80%+ code coverage.
- Integration tests passed.
- Logging configured and verified.
- Concurrency safety for database updates validated.
- Documentation updated (if applicable).

US-SIL-006: Concurrency Management & Database Safety

Title: As a robust SIL Notification System, I want to ensure concurrency safety across all operations on the shared SQL Server database, so that data integrity is maintained even when multiple instances are running. **Status:** To Do **Priority:** High **Labels:** Backend, Concurrency, Database, Non-Functional **Assignee:** [Developer Name] **Sprint:** [Sprint Number]
Description: Since the executable will be deployed on multiple Windows servers and scheduled to run simultaneously, it's crucial to implement robust concurrency control. This story addresses preventing data races, deadlocks, and inconsistencies when multiple instances read from and write to the SILCase and SILCaseAttachment tables (especially updating email flags and Dox IDs). The core goal is to prevent duplicate processing of the same SILCase by different instances and to ensure atomic and consistent updates to the shared database.

Strategy Overview: Atomic Claiming with Optimistic Concurrency

The primary strategy for managing concurrency across multiple application instances interacting with the shared SQL Server database will be a combination of:

1. **Atomic "Claiming" of Cases:** Before processing an SILCase, an instance will attempt to atomically update its status in the database to mark it as "in process" by that specific instance. Only the instance that succeeds in this atomic update will proceed with processing that case.
2. **Optimistic Concurrency for Subsequent Updates:** For updates to flags (like EmailsSentToClient, EmailsSentToCA, DoxUploadStatus) where race conditions could lead to incorrect states, the UPDATE statement will include the current expected value in its WHERE clause. This ensures an update only proceeds if the data hasn't been changed by another instance since it was read.
3. **Application-Level Concurrency:** Within a single instance, we'll leverage .NET's async/await and Task Parallel Library (TPL) for efficient, non-blocking parallel execution of I/O-bound operations (e.g., API calls to AIF, CDC, MS Graph) to improve throughput without external service contention.

Detailed Technical Implementation Steps:

Step 1: Database Schema Enhancements (for Atomic Claiming)
To facilitate **atomic claiming** of cases, we'll enhance the SILCase table:

- **Add ProcessingInstanceId (NVARCHAR(100), NULL):** This column will store a unique identifier for the application instance that is currently processing or has claimed a particular SIL case (e.g., combining hostname and process ID: Environment.MachineName + "-" + Process.GetCurrentProcess().Id).
- **Add ProcessingTimestamp (DATETIME2, NULL):** This column will record the timestamp when an instance claimed the case. It's crucial for identifying cases that might have been claimed by a crashed instance (if ProcessingTimestamp is older than a defined timeout).
- **Add AttemptCount (INT, NOT NULL, DEFAULT 0):** This will track how many times an SIL case has been attempted for processing by any instance, aiding in preventing infinite retries and identifying problematic cases.

Revised SILCase Table Structure:

- Id (PK, INT/BIGINT, IDENTITY)
- CaseId (NVARCHAR(50), UNIQUE, NOT NULL)
- ReceivedTimestamp (DATETIME2, NOT NULL)

- CreationTimestamp (DATETIME2, NOT NULL, DEFAULT GETDATE())
- CaseStatus (NVARCHAR(50), NOT NULL, DEFAULT 'Received')
- EmailsSentToClient (BIT, NOT NULL, DEFAULT 0)
- EmailsSentToCA (BIT, NOT NULL, DEFAULT 0)
- ProcessingAttempts (INT, NOT NULL, DEFAULT 0)
- LastProcessedTimestamp (DATETIME2, NULL)
- Remarks (NVARCHAR(MAX), NULL)
- **ProcessingInstanceId (NVARCHAR(100), NULL)**
- **ProcessingTimestamp (DATETIME2, NULL)**
- **AttemptCount (INT, NOT NULL, DEFAULT 0)**

Step 2: Implement Atomic Case Claiming (Multi-Instance Safety)

This is the most critical step for preventing duplicate processing across multiple application instances.

1. **Identify Available Cases for Claiming:**
    - When an application instance starts its processing cycle, it will query the SILCase table for cases that are in an eligible status (e.g., 'Extracted', 'ReadyForNotification').
    - Crucially, it will only consider cases that are ProcessingInstanceId IS NULL (unclaimed) OR where ProcessingTimestamp is older than a predefined **timeout threshold** (e.g., 30 minutes). This timeout handles scenarios where a previously claiming instance might have crashed without releasing its claim.
2. **Attempt to Claim Cases (Atomic UPDATE):**
    - For a batch of identified eligible cases, the instance will attempt to **atomically claim** them using a SQL UPDATE statement. This UPDATE statement is designed to only succeed if the case is still unclaimed or if its claim has expired.

      <!-- end list -->
      ```
      -- Example SQL for claiming N cases (e.g., 50 cases)
      UPDATE TOP (@BatchSize) SILCase
      SET
          CaseStatus = 'Processing', -- Mark as in-progress
          ProcessingInstanceId = @CurrentInstanceId,
          ProcessingTimestamp = GETDATE(),
          AttemptCount = AttemptCount + 1
      WHERE
          CaseStatus IN ('Extracted', 'ReadyForNotification') -- Eligible statuses
          AND (ProcessingInstanceId IS NULL OR ProcessingTimestamp < DATEADD(minute, -@ClaimTimeoutMinutes, GETDATE()));

      -- After the UPDATE, SELECT the cases that *this specific instance* successfully claimed:
      SELECT *
      FROM SILCase
      WHERE ProcessingInstanceId = @CurrentInstanceId
        AND ProcessingTimestamp >= DATEADD(minute, -1, GETDATE());
      -- Verify it was just claimed by this instance
      ```

    - @CurrentInstanceId should be a unique identifier for the running application

instance (e.g., Environment.MachineName + "-" + Process.GetCurrentProcess().Id).
    - This pattern ensures that if two instances try to claim the same case simultaneously, only one's UPDATE statement will affect that row because of the WHERE clause conditions, making it the sole "owner" for that processing cycle.
3. **Process Claimed Cases:**
    - The application instance will then proceed to process only the cases it has successfully **claimed**.
4. **Release/Update Case Status:**
    - **Success:** Upon successful completion of all processing steps for a case (including Dox upload, all API calls, and email sending), the instance will update its CaseStatus to 'Processed' or 'Completed' and **set ProcessingInstanceId = NULL** to release the claim.
    - **Failure:** If processing fails for a case (e.g., unrecoverable API error, validation issue), the instance will update its CaseStatus to 'Failed' or 'Error', add relevant remarks, and **set ProcessingInstanceId = NULL**. The AttemptCount helps prevent endless retries.

Step 3: Implement Optimistic Concurrency for Subsequent Data Updates

For all subsequent updates to SILCase or SILCaseAttachment records *within* the processing of a claimed case (e.g., updating email flags, DoxDocumentId), we'll employ optimistic concurrency. This involves including the current expected value of a column in the WHERE clause of the UPDATE statement. If another instance (or even a different thread within the same instance) has changed that value unexpectedly, the UPDATE will affect 0 rows, signaling a conflict.

1. **Updating DoxDocumentId and DoxUploadStatus (for US-SIL-011):**

```
// C# Pseudo-code for updating attachment status
public async Task UpdateAttachmentDoxId(long attachmentId, string
newDoxId, string expectedStatus)
{
    // expectedStatus would typically be 'Pending' when you
initially read it
    var query = @"
        UPDATE SILCaseAttachment
        SET DoxDocumentId = @newDoxId,
            DoxUploadStatus = 'Uploaded',
            DoxUploadTimestamp = GETDATE()
        WHERE Id = @attachmentId
        AND DoxUploadStatus = @expectedStatus; -- Optimistic lock
on status
    ";
    int rowsAffected = await
_dbContext.ExecuteSqlCommandAsync(query, new { attachmentId,
newDoxId, expectedStatus });

    if (rowsAffected == 0)
    {
        _logger.LogWarning($"Concurrency conflict or status
mismatch for attachment {attachmentId}. It might have been updated
by another process.");
```

```
        // Handle: e.g., re-read the latest status, retry if it's
a transient issue, or mark for review.
    }
}
```

2. **Updating EmailsSentToClient and EmailsSentToCA (for US-SIL-005):** To prevent duplicate emails, these flags should only be set to TRUE if they are currently FALSE.

```
// C# Pseudo-code for updating email sent flags
public async Task UpdateEmailSentFlags(long silCaseId, bool
shouldMarkClientSent, bool shouldMarkCASent)
{
    if (shouldMarkClientSent)
    {
        var clientUpdateQuery = @"
            UPDATE SILCase
            SET EmailsSentToClient = 1
            WHERE Id = @silCaseId AND EmailsSentToClient = 0; --
Only set if currently 0 (false)
        ";
        int clientRowsAffected = await
_dbContext.ExecuteSqlCommandAsync(clientUpdateQuery, new {
silCaseId });
        if (clientRowsAffected == 0) _logger.LogWarning($"Client
email flag for case {silCaseId} was already set to true or
conflict occurred.");
    }
    if (shouldMarkCASent)
    {
        var caUpdateQuery = @"
            UPDATE SILCase
            SET EmailsSentToCA = 1
            WHERE Id = @silCaseId AND EmailsSentToCA = 0; -- Only
set if currently 0 (false)
        ";
        int caRowsAffected = await
_dbContext.ExecuteSqlCommandAsync(caUpdateQuery, new { silCaseId
});
        if (caRowsAffected == 0) _logger.LogWarning($"Client
Advisor email flag for case {silCaseId} was already set to true or
conflict occurred.");
    }
}
```

Step 4: Application-Level Concurrency (within a single instance)

Within a *single* console application instance, we can utilize .NET's Task Parallel Library (TPL) to make asynchronous I/O-bound calls (like those to AIF, CDC, MS Graph) in parallel, maximizing throughput without blocking the main thread.

1. **Parallel API Calls:**

- Use Task.WhenAll to await multiple independent asynchronous operations (e.g., calling all 6 CDC services for a single account_id).
- Use Parallel.ForEachAsync (available in C# 9+ / .NET 5+) or SemaphoreSlim in conjunction with Task.Run to control the degree of parallelism when processing a collection of items (e.g., unique ISINs, unique account IDs, or even multiple claimed SIL cases). This prevents overwhelming external services or internal network resources.

<!-- end list -->
```
// Example: Processing unique ISINs for AIF calls with a maximum of 5 concurrent calls
public async Task ProcessIsinsConcurrently(IEnumerable<string> uniqueIsins)
{
    var maxConcurrentAifCalls = 5;
    var semaphore = new SemaphoreSlim(maxConcurrentAifCalls);
    var tasks = new List<Task>();

    foreach (var isin in uniqueIsins)
    {
        await semaphore.WaitAsync(); // Decrement semaphore count, wait if at max
        tasks.Add(Task.Run(async () => // Run on a thread pool thread to free up main
        {
            try
            {
                // Call AIFClient for ISIN
                await _aifClient.GetAccountIdByIsin(isin);
                // ... other processing for this ISIN ...
            }
            catch (Exception ex)
            {
                _logger.LogError(ex, $"Error processing ISIN {isin} with AIF.");
            }
            finally
            {
                semaphore.Release(); // Increment semaphore count
            }
        }));
    }
    await Task.WhenAll(tasks); // Wait for all parallel tasks to complete
}
```

2. **Thread-Safe Collections:** When multiple parallel tasks within the same instance need to aggregate data into shared collections (e.g., building the ISIN to account_id map), use .NET's built-in **concurrent collections** like ConcurrentDictionary<TKey, TValue> or ConcurrentQueue<T> to avoid internal race conditions.

Step 5: Transaction Management

Ensure atomicity for a series of related database operations for a single SILCase using SQL Server transactions.

1. **Scope for Atomic Operations:** Use System.Transactions.TransactionScope (or manual SqlConnection.BeginTransaction()) to group multiple database writes for a single SIL case (e.g., updating attachment status, then email flags, then final case status) into a single atomic unit. If any part of the transaction fails, all changes within that scope are rolled back.

```
public async Task ProcessSingleSilCaseFully(long silCaseId, string
currentInstanceId)
{
    // For scenarios involving multiple connections or distributed
transactions:
    using (var transactionScope = new
TransactionScope(TransactionScopeAsyncFlowOption.Enabled))
    {
        try
        {
            // Example operations within a transaction for a
claimed case:
            // 1. Read SILCase details (already claimed by this
instance)
            // 2. Perform Dox upload (US-SIL-011) and update
SILCaseAttachment (optimistic update inside transaction)
            // 3. Call AIF/CDC services as needed (external calls,
not part of DB transaction itself, but their success drives DB
updates)
            // 4. Send Emails (US-SIL-005)
            // 5. Update EmailsSentToClient/CA flags in SILCase
(optimistic update inside transaction)
            // 6. Finally, update the main CaseStatus and release
the claim
            await
_dbContext.UpdateCaseStatusAndReleaseClaim(silCaseId, "Completed",
null, currentInstanceId); // Set NULL for ProcessingInstanceId

            transactionScope.Complete(); // Commit the transaction
if all operations succeed
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, $"Error processing SIL case
{silCaseId}. Transaction will rollback.");
            // If an unhandled exception occurs, transactionScope
will automatically dispose and rollback.
            // Update status to 'Failed' and release claim.
            await
_dbContext.UpdateCaseStatusAndReleaseClaim(silCaseId, "Failed",
```

```
ex.Message, currentInstanceId);
            }
        }
    }
```

Step 6: Robust Error Handling & Retries (with Concurrency in Mind)

1. **Transient Fault Handling:** Implement retry policies (e.g., using the **Polly** library) for transient errors. This is crucial for:
    - External API calls (AIF, CDC, MS Graph) that might experience temporary network glitches or service unavailability.
    - Optimistic database updates that fail because rowsAffected == 0 (indicating a concurrency conflict). In this case, a retry might involve re-reading the latest state before attempting the update again.
2. **Circuit Breaker Pattern:** For highly unreliable external services, consider a circuit breaker to prevent continuous, futile hammering during an extended outage, allowing the service to recover before retries are attempted.
3. **Idempotency:** While the optimistic updates help, ensure that if an SIL case is accidentally processed more than once (e.g., due to a crash and re-claim), it doesn't lead to negative side effects like duplicate emails. The logic for EmailsSentToClient = 1 WHERE ... = 0 contributes to this.
4. **Comprehensive Logging (integrating with US-SIL-007):**
    - All attempts to claim a case (success/failure) should be logged, including the CurrentInstanceId.
    - Every time an optimistic update results in rowsAffected == 0, a **warning** should be logged, detailing the case ID and what update failed, as this indicates a concurrency conflict.
    - All successful processing and failures should be logged with sufficient context (CaseId, ProcessingInstanceId, timestamps).

Step 7: Testing Strategy for Concurrency

Rigorous testing is essential to validate the implemented concurrency safety mechanisms.

1. **Stress Testing:** Deploy multiple instances of the console application in a dedicated test environment. Configure them to run simultaneously, targeting a shared test database with a large dataset designed to create race conditions (e.g., multiple cases that require updating the same related entities, or many cases becoming eligible at once).
2. **Injecting Delays:** Deliberately introduce artificial delays (e.g., Thread.Sleep or Task.Delay in strategic places) in test versions of the code to increase the probability of race conditions occurring, making them easier to detect.
3. **Assertion Checks:**
    - After test runs, thoroughly **verify the final state of the database**:
        - Ensure no SILCase has ProcessingInstanceId set if it's completed.
        - Validate EmailsSentToClient and EmailsSentToCA flags are correct (e.g., TRUE only if an email was legitimately sent).
        - Confirm no duplicate emails were sent (if using a mock email server).
        - Check for any DoxDocumentId or other data inconsistencies.
    - Monitor the database for **deadlocks** during concurrent runs.
    - Analyze **logs from all running instances** to trace the execution flow, identify which instance processed which case, and pinpoint any logged concurrency conflicts.
4. **Tools:** Use load testing tools (e.g., JMeter, custom C# test scripts) to simulate high

concurrent execution loads on the application instances.

Dependencies:

- **US-SIL-010** (Database schema with ProcessingInstanceId, ProcessingTimestamp, AttemptCount).
- **US-SIL-011** (Utilizes optimistic locking for DoxUploadStatus updates).
- **US-SIL-001**, **US-SIL-002**, **US-SIL-003**, **US-SIL-005** (All these stories involve operations that will leverage or depend on the concurrency mechanisms defined here, especially atomic claiming and optimistic updates, and application-level parallelism). This story provides the core concurrency framework that these stories will build upon.

Pre-conditions:

- Database schema for SILCase is updated with ProcessingInstanceId, ProcessingTimestamp, and AttemptCount.
- Application is designed to be multi-instance aware.

Acceptance Criteria:

- Atomic claiming mechanism for SILCase records is implemented and prevents duplicate processing of the same case by different instances.
- Optimistic concurrency control is implemented for updates to DoxDocumentId, DoxUploadStatus, EmailsSentToClient, and EmailsSentToCA, preventing overwrites of concurrent changes.
- Application-level concurrency (e.g., SemaphoreSlim, Task.WhenAll) is appropriately used for parallel external service calls (AIF, CDC, MS Graph) within an instance.
- Database transactions are correctly applied to ensure atomicity of related updates for a single SILCase.
- Error handling for concurrency conflicts (e.g., rowsAffected == 0) is robust, logging warnings and potentially triggering retries or escalation.
- Timeout mechanism for abandoned claims is implemented to allow other instances to pick up cases from crashed workers.
- The system demonstrates no data inconsistencies or unhandled deadlocks during stress testing with multiple concurrent instances.
- All relevant logging includes the ProcessingInstanceId to aid in debugging concurrency issues.
- **Unit Tests:** Cover atomic update scenarios (mocks simulating simultaneous updates), optimistic locking logic (testing rowsAffected outcomes), and retry logic for concurrency conflicts.
- **Integration Tests:** Conduct multi-instance stress tests to simulate race conditions and verify data integrity in the database and correctness of processing.

Definition of Done:

- All concurrency control mechanisms (atomic claiming, optimistic locking, application-level concurrency, transactions) are implemented and reviewed.
- Database schema changes for SILCase applied.
- All acceptance criteria met.
- Unit tests passed with 80%+ code coverage for concurrency components.
- Multi-instance integration tests executed, demonstrating stable and consistent behavior under load.
- Logging verified for concurrency events.
- Documentation updated, detailing the concurrency strategy and implementation.

Preferred Implementation Order

The following order prioritizes foundational components and crucial cross-cutting concerns (like **concurrency safety**) early in the development lifecycle. This approach ensures that core data structures and mechanisms for robust operation are in place before building dependent functionality.

1. **US-SIL-012: Initial C# .NET 8 Console Application Setup**
    - *Rationale:* This is the absolute first step, establishing the project structure and buildable foundation.
2. **US-SIL-010: SIL Case and Details Ingestion**
    - *Rationale:* This defines and creates the core database tables (SILCase, SILCaseAttachment, SILCaseDetail) and the initial process for populating them. Without this, no other data processing can occur.
3. **US-SIL-006: Concurrency Management & Database Safety**
    - *Rationale:* This is a critical cross-cutting concern. Its database schema enhancements (ProcessingInstanceId, ProcessingTimestamp, AttemptCount) and core claiming logic are fundamental to prevent issues when multiple instances run. Implementing this early (immediately after the core tables are defined in US-SIL-010) allows subsequent stories to be built on a **concurrency-safe foundation**, reducing the risk of costly rework. Its application-level concurrency aspects will also benefit stories US-SIL-002 and US-SIL-003.
4. **US-SIL-011: Attachment Upload to Dox System**
    - *Rationale:* This story relies on the data ingested by US-SIL-010 and leverages the concurrency mechanisms of US-SIL-006 for safe updates. It can proceed somewhat in parallel with the data enrichment steps (US-SIL-001 onwards) as it focuses on attachment handling rather than client/advisor data.
5. **US-SIL-001: Data Preparation - Unique ISINs**
    - *Rationale:* This story begins the data processing pipeline by extracting unique ISINs, which are foundational for looking up account and client details. It relies on data from US-SIL-010 and the claiming mechanism from US-SIL-006.
6. **US-SIL-002: Account ID Retrieval via AIF Service**
    - *Rationale:* Directly dependent on the unique ISINs provided by US-SIL-001 and will utilize the application-level concurrency defined in US-SIL-006.
7. **US-SIL-003: Client and Advisor Details Retrieval via CDC Services**
    - *Rationale:* Directly dependent on the unique account IDs retrieved in US-SIL-002 and will also leverage the application-level concurrency from US-SIL-006.
8. **US-SIL-004: Relationship Consolidation**
    - *Rationale:* This story integrates the outputs from US-SIL-001, US-SIL-002, and US-SIL-003 into a single, comprehensive structure, making it a prerequisite for the final email dispatch.
9. **US-SIL-005: Email Notification Dispatch**
    - *Rationale:* This is the final step in the primary workflow, consuming the consolidated data from US-SIL-004 and crucially depending on the robust concurrency updates (US-SIL-006) for correctly marking emails as sent.