You're absolutely right! My apologies. The previous response outlined the strategy and order but inadvertently omitted the detailed user stories themselves. That was an oversight on my part. Let's get this fixed. Below is the complete, stitched document, incorporating the detailed Epic, all nine user stories with their examples, and the preferred implementation sequence.

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) 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-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').
  - EmailIsSentToClient (BIT, **NOT NULL**, **DEFAULT 0**): Flag if email sent to client.
  - EmailIsSentToCA (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.
- **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
- **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)

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)

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.
- 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 for Dox upload.
- 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)
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"]
}
```

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 considered.
- **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"
  }
}
```

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 are considered if performance is a bottleneck (e.g., Task.WhenAll for calls within an account).
- **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"}
    ]
  }
]
```

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.
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'

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 (e.g., UPDATE SILCase SET EmailsSentToClient = TRUE WHERE case_id = 'SIL001' AND EmailsSentToClient = FALSE).
- 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). This may involve database-level locking, application-level synchronization, or optimistic concurrency.
Example Scenario (illustrating potential issue):

1. **Instance A** reads SILCase 'SIL005', EmailsSentToClient is FALSE.
2. **Instance B** also reads SILCase 'SIL005', EmailsSentToClient is FALSE.
3. **Instance A** sends client email for 'SIL005', then attempts to UPDATE SILCase SET EmailsSentToClient = TRUE WHERE case_id = 'SIL005'.
4. **Instance B** also sends client email for 'SIL005' (duplicate email!), then attempts to UPDATE SILCase SET EmailsSentToClient = TRUE WHERE case_id = 'SIL005'. Without proper concurrency control, both updates might succeed, or one might overwrite the other without acknowledging the concurrent attempt, leading to duplicate emails and race conditions.

Example Solution Strategies (to prevent above issue):

- **Optimistic Concurrency:** When selecting cases, include EmailsSentToClient and EmailsSentToCA in the WHERE clause of the UPDATE statement.
    - UPDATE SILCase SET EmailsSentToClient = TRUE WHERE case_id = 'SIL005' AND EmailsSentToClient = FALSE (Ensures only the first successful sender updates the flag if it was false).

- **Distributed Lock:** Implement a distributed locking mechanism (e.g., using a dedicated database table for locks, Redis lock) to ensure only one instance processes a specific case_id at a time.
- **"Processing" Status:** Introduce a case_status value like 'Processing' with atomic updates:
  - Instance A: UPDATE SILCase SET case_status = 'Processing' WHERE case_id = 'SIL005' AND case_status = 'Extracted'; (Only one instance succeeds atomically).

Pre-conditions:

- All data access layers are designed to allow for concurrency control.
- Understanding of SQL Server 2016 locking mechanisms.

Acceptance Criteria:

- The application implements a strategy to prevent duplicate processing of the same SILCase by multiple instances.
- Updates to SILCase.EmailsSentToClient, SILCase.EmailsSentToCA, and SILCaseAttachment.DoxUploadStatus/DoxDocumentId are atomic and thread-safe.
- No data corruption or unexpected behavior occurs when multiple instances run concurrently and update the same records.
- Database transactions are correctly used for multi-step operations (e.g., reading, sending email, updating status).
- Thorough testing with multiple concurrent instances demonstrates no concurrency issues (e.g., processing 100 cases with 3 instances yields 100 client emails, not more, and flags are correct).
- Logging includes concurrency control events (e.g., acquiring/releasing locks, retry attempts due to optimistic concurrency failures).
- **Unit Tests:** Simulate concurrent access to shared resources and mocked database updates with race conditions.
- **Integration Tests:** Deploy multiple instances in test environments concurrently against a shared database to verify data integrity.

Definition of Done:

- Concurrency control strategy defined and implemented.
- Code reviewed for concurrency issues.
- All acceptance criteria met.
- Unit tests passed with 80%+ code coverage for concurrency logic.
- Integration tests specifically designed for concurrency passed.
- Monitoring shows no deadlocks or excessive contention.
- Documentation updated on concurrency strategy.

US-SIL-007: Logging & Monitoring

Title: As a maintainable SIL Notification System, I want comprehensive logging across the solution, so that operational issues can be easily diagnosed and audited. **Status:** To Do **Priority:** Medium **Labels:** Backend, Non-Functional, Observability **Assignee:** [Developer Name] **Sprint:** [Sprint Number]

Description: This story ensures the application provides sufficient logging for troubleshooting, performance monitoring, and audit requirements. All significant events, errors, warnings, and informational messages will be logged using the Idp.Logs library (built on ILogger). This includes service call details, database operations, error specifics, and critical application state changes.

Example Log Entries (conceptual):

- **Information:**
  - [2025-06-28 09:00:00.123 INFO] SIL_Processor - Application started. Business day: 2025-06-28. Instance: ServerA.
  - [2025-06-28 09:00:05.456 INFO] DataPrep - Found 50 eligible SIL cases for processing.
  - [2025-06-28 09:00:30.111 INFO] Email_Sender - Preparing email for CaseID 'SIL001'. Recipients: Client (client1@example.com), Advisor (advisor1@example.com). Attachments: SIL001_Doc1.pdf, SIL001_Doc2.pdf.
- **Warning:**
  - [2025-06-28 09:00:20.333 WARN] DataConsolidation - ISIN 'NONEXISTENT123' from SILCase 'SIL005' did not yield an account ID. Case will be processed with incomplete data for this ISIN.
  - [2025-06-28 09:00:45.678 WARN] Email_Sender - Client 'client2@example.com' for CaseID 'SIL002' has email waiver. Email to client skipped.
- **Error:**
  - [2025-06-28 09:00:16.789 ERROR] AIF_Client - Error calling AIF for ISIN 'INVALIDISIN'. Error: HTTP 500 Internal Server Error. StackTrace: ...
  - [2025-06-28 09:00:33.777 ERROR] DB_Update - Concurrency conflict updating SILCase 'SIL001'. Retrying. Exception: System.Data.DBConcurrencyException. StackTrace: ...

Pre-conditions:

- Idp.Logs library is integrated and functional.
- Logging configuration (e.g., file paths, log levels) is defined.

Acceptance Criteria:

- All external service calls (AIF, CDC, MailProcessingClient) are logged at appropriate levels (start, end, success, failure, response details).
- Database interactions (reads, writes, updates) are logged.
- Critical application flow steps (e.g., start of processing, ISIN extraction, email dispatch decisions) are logged as informational messages.
- All exceptions and errors are caught and logged with full details (stack trace, error message, relevant context like case_id or isin).
- Logging levels (Debug, Information, Warning, Error, Critical) are appropriately used.
- Sensitive information is not logged (e.g., full email content, raw credentials).
- Log files are generated (e.g., daily rolling files in C:\Logs\SILNotification\) and accessible.
- **Unit Tests:** Verify logging calls are correctly placed and messages contain expected information.
- **Integration Tests:** Verify application generates logs as expected during various operational scenarios (success and error conditions), checking log file content/format.

Definition of Done:

- Logging implemented and reviewed across the entire application.
- All acceptance criteria met.
- Unit tests passed.
- Integration tests confirm logging behavior.
- Log configuration verified.
- Documentation updated (if applicable).

US-SIL-008: Unit and Integration Testing

Title: As a high-quality SIL Notification System, I want comprehensive unit and integration tests, so that the application is robust, reliable, and maintainable. **Status:** To Do **Priority:** High **Labels:** Quality, Testing, Non-Functional **Assignee:** [Developer Name/QA Engineer] **Sprint:** [Sprint Number]
Description: This story mandates creating a comprehensive test suite. It includes writing **unit tests** for individual components and functions, targeting at least **80% line code coverage**. Additionally, **integration tests** will verify correct interaction between modules and external dependencies (mocked or real).
Example Test Scenarios:

- **Unit Test Example (for ISIN extraction logic):**
  - **Test Case:** ExtractUniqueIsins_ValidJson_ReturnsDistinctList
  - **Input:** SILCaseDetail with extraction_data JSON containing {"financialInstruments": [{"isin": "ISIN1"}, {"isin": "ISIN2"}, {"isin": "ISIN1"}]}
  - **Expected Output:** ["ISIN1", "ISIN2"]
- **Unit Test Example (for Email recipient logic):**
  - **Test Case:** DetermineRecipients_ClientSubscribedNoWaiver_SendsToClientAndAdvisor
  - **Input:** Consolidated recipient data: clientSilSubscription: true, clientEmailWaiver: false, advisorEmail: "adv@example.com"
  - **Expected Behavior:** Calls MailProcessingClient.SendEmail for both client and advisor.
- **Integration Test Example (Database to AIF to CDC flow):**
  - **Test Setup:** Populate SILCase/SILCaseDetail with test data. Configure AIFClient and CDCClient to use local test endpoints or mocks returning predefined data.
  - **Test Action:** Run the application (or a specific flow segment) against this setup.
  - **Assertion:** Verify intermediate data structures (ISIN-Account map, Account-Client/Advisor map) are correctly populated.
- **Integration Test Example (End-to-End with Mock Email):**
  - **Test Setup:** Populate database. Mock AIF/CDC. Configure MailProcessingClient to use a local SMTP server or capture sent emails.
  - **Test Action:** Run the full application.
  - **Assertion:** Verify SILCase email flags are updated correctly. Verify expected emails (count, subject, attachments) were captured by the mock.

Pre-conditions:

- Test frameworks (e.g., NUnit, xUnit, Moq) are set up.
- Test environment is available for integration testing.

Acceptance Criteria:

- **Unit Testing:**
  - Unit tests are written for all core logic components.
  - Dependencies are properly mocked/stubbed.
  - Minimum **80% line code coverage** is achieved and verified by a code coverage tool (e.g., Coverlet).
  - All unit tests pass consistently.
- **Integration Testing:**
  - Integration tests are written to verify interaction between the application and

    CDCClient, AIFClient, MailProcessingClient, and SQL Server.
    - Tests cover sequential service calls.
    - Tests include positive and negative scenarios for external service calls.
    - Tests verify end-to-end data flow (database read to email send/database update).
    - All integration tests pass consistently in the test environment.
- Test reports are generated and reviewed.
- Test cases cover major error handling paths.
- **Unit Tests:** Verify component isolation and correct logic.
- **Integration Tests:** Verify system-wide functionality and component interactions.

Definition of Done:
- All required unit and integration tests are written and pass.
- Code coverage report confirms 80%+ line coverage.
- Automated test pipeline (CI/CD) includes these tests.
- Test documentation updated.

US-SIL-009: CI/CD Pipeline Definition

Title: As a development team, we want a defined GitLab CI/CD pipeline, so that the build, test, and deployment process for the SIL Notification System is automated and consistent. **Status:** To Do **Priority:** Medium **Labels:** DevOps, CI/CD, Non-Functional **Assignee:** [DevOps Engineer] **Sprint:** [Sprint Number]
Description: This story covers defining and implementing the CI/CD pipeline using a gitlab-ci.yaml file. The pipeline will automate the build, testing, and potentially deployment steps for the C# .NET 8 console application, ensuring consistent and reliable releases.
Example gitlab-ci.yaml Snippets (conceptual):
```
stages:
  - build
  - test
  - deploy

variables:
  DOTNET_SDK_VERSION: "8.0.x" # Specify .NET SDK version
  BUILD_CONFIG: "Release"
  PROJECT_PATH: "src/SILNotificationApp/SILNotificationApp.csproj" #
Path to your main console app
  TEST_PROJECT_PATH:
"tests/SILNotificationApp.Tests/SILNotificationApp.Tests.csproj" #
Path to your test project
  ARTIFACT_PATH: "publish"

build_job:
  stage: build
  image: mcr.microsoft.com/dotnet/sdk:${DOTNET_SDK_VERSION}
  script:
    - dotnet restore ${PROJECT_PATH}
    - dotnet build ${PROJECT_PATH} --configuration ${BUILD_CONFIG}
--no-restore
  artifacts:
    paths:
      - ./src/SILNotificationApp/bin/${BUILD_CONFIG}/net8.0/
    expire_in: 1 day # Cache built artifacts for subsequent stages

test_job:
  stage: test
  image: mcr.microsoft.com/dotnet/sdk:${DOTNET_SDK_VERSION}
  dependencies: # Ensure build artifacts are available
    - build_job
  script:
    - dotnet test ${TEST_PROJECT_PATH} --configuration ${BUILD_CONFIG}
--no-build --logger trx --results-directory "TestResults"
/p:CollectCoverage=true /p:CoverletOutputFormat=opencover
/p:CoverletOutput="coverage.xml"
    - dotnet tool install --global dotnet-reportgenerator-globaltool
--version 5.2.1 # For converting coverage to readable format
    - reportgenerator -reports:"TestResults/**/coverage.xml"
-targetdir:"coverage-report" -reporttypes:Html
  artifacts:
    paths:
      - TestResults/
      - coverage-report/
    expire_in: 1 week
  coverage: '/Total\s+coverage:\s(\d+\.\d+)%/' # Regex to extract
coverage percentage

deploy_job:
  stage: deploy
  image: mcr.microsoft.com/dotnet/sdk:${DOTNET_SDK_VERSION} # Or a
custom image with deployment tools
  dependencies:
    - test_job
  script:
    - dotnet publish ${PROJECT_PATH} --configuration ${BUILD_CONFIG}
--no-build -o ${ARTIFACT_PATH}
    # Example deployment script - replace with actual deployment logic
(e.g., SCP, PowerShell Remoting, Octopus Deploy CLI)
    - echo "Simulating deployment to Windows Server..."
    - echo "Copying files from ./${ARTIFACT_PATH} to network share or
target server."
    # Example: Copy-Item -Path "${ARTIFACT_PATH}\*" -Destination
"\\YourServer\C$\DeploymentFolder" -Recurse
    # For Windows Task Scheduler, you might just copy the executable
and update the task.
  only:
    - main # Only deploy from the main branch
  environment:
    name: Production
    url: http://your-application-endpoint.com # Optional, for
environment tracking
```

Pre-conditions:

- GitLab repository is set up with the .NET 8 project.
- GitLab Runners are available and configured to run .NET 8 builds.
- Deployment targets (Windows Servers) are accessible for automated deployment (if applicable).

Acceptance Criteria:

- A gitlab-ci.yaml file is created in the repository root with appropriate stages and jobs.
- The pipeline includes stages for:
    - **Build:** Compiles the .NET 8 application and its class libraries.
    - **Test:** Executes all unit/integration tests and generates test reports.
    - **Code Coverage:** Calculates code coverage and fails if below 80%. A readable report is generated.
    - **Artifact Generation:** Publishes the console application and its dependencies into a deployable artifact.
    - **Deployment (Optional/Manual Trigger):** Has a defined deployment step (e.g., copies artifact to servers). This stage should ideally be manually triggered or on specific branches.
- The pipeline triggers automatically on code pushes to specific branches (e.g., main, develop).
- Pipeline status is clearly visible in GitLab.
- Build artifacts and test reports are stored and retrievable.
- **Unit Tests:** Verify gitlab-ci.yaml syntax and basic job execution for build/test stages.
- **Integration Tests:** Verify the entire pipeline runs successfully, including artifact creation and (simulated) deployment, checking the deployed app starts and functions.

Definition of Done:

- gitlab-ci.yaml file is committed and functional.
- All pipeline stages execute successfully.
- Automated tests are integrated into the pipeline.
- Build artifacts are correctly generated.
- Documentation on pipeline usage is available.

Preferred Implementation Order for SIL Notification System User Stories

This section outlines the proposed implementation sequence for the user stories. This order is chosen to optimize development flow, mitigate technical risks, and ensure early validation of critical integrations and foundational components.

Phase 1: Core Data Ingress and Document Management Foundation

Objective: Establish the foundational mechanisms for acquiring, storing, and managing raw SIL case data and its associated attachments within the system, ensuring data integrity and preparing documents for long-term storage.

1. **US-SIL-010: SIL Case and Details Ingestion**

- **Rationale:** This is the absolute starting point. Before any processing or notification can happen, the system needs to have the raw SIL case data, its details (ISINs), and attachment metadata in its database. This story also dictates the foundational database table designs, which are crucial for all subsequent stories.
   - **Dependencies:** None (initial data input for the system).
2. **US-SIL-011: Attachment Upload to Dox System**
   - **Rationale:** Although attachments are sent later in emails, uploading them to Dox is an *ingestion-time* activity that enriches the SILCaseAttachment table with DoxDocumentIds. It's best to handle this immediately after case ingestion, making the DoxDocumentId available for when emails are constructed. This is a critical early integration with an external service (AIF for Dox).
   - **Dependencies:** US-SIL-010 (requires SILCaseAttachment table populated with FilePath).

Phase 2: Data Enrichment and Relationship Mapping

Objective: To enrich the ingested SIL case data by retrieving critical financial instrument, client, and advisor details from external core banking systems, establishing the necessary relationships for targeted communication.

1. **US-SIL-001: Data Preparation - Unique ISINs**
   - **Rationale:** Once cases are ingested (US-SIL-010), the first processing step for the notification system is to identify the unique ISINs from the newly ingested SILCaseDetail records. This provides the input for the next AIF call.
   - **Dependencies:** US-SIL-010 (relies on SILCase and SILCaseDetail tables being populated).
2. **US-SIL-002: Account ID Retrieval via AIF Service**
   - **Rationale:** With unique ISINs identified, the next logical step is to call the AIF service to get account IDs. This establishes the critical link between the financial instruments and the client accounts. This is a key external service integration.
   - **Dependencies:** US-SIL-001 (requires the list of unique ISINs).
3. **US-SIL-003: Client and Advisor Details Retrieval via CDC Services**
   - **Rationale:** Once account IDs are known, the application can proceed to call the various CDC services to gather the client and advisor contact and preference details. This is another major external service integration, dependent on the account_id from the previous step.
   - **Dependencies:** US-SIL-002 (requires the list of unique account IDs).

Phase 3: Notification Orchestration and Dispatch

Objective: To consolidate all collected data and execute the primary business function of sending automated email notifications with SIL attachments to the identified clients and advisors.

1. **US-SIL-004: Relationship Consolidation**
   - **Rationale:** Before sending emails, all the disparate pieces of information (SIL cases, ISINs, Account IDs, Client/Advisor details) need to be brought together into a coherent structure. This story creates the final data model for the email dispatch logic.
   - **Dependencies:** US-SIL-001, US-SIL-002, US-SIL-003, US-SIL-011 (requires

output from all previous data processing and service calls).

2. **US-SIL-005: Email Notification Dispatch**
    - **Rationale:** This is the primary functional goal of the entire application. It consumes the consolidated data and performs the actual email sending, leveraging the MailProcessingClient and the DoxDocumentId from US-SIL-011 for attachments.
    - **Dependencies:** US-SIL-004 (requires the consolidated data for email preparation), US-SIL-011 (requires DoxDocumentId in SILCaseAttachment for attachments).

Phase 4: Cross-Cutting Quality and Operational Enablement

Objective: To ensure the system's robustness, maintainability, auditability, and efficient deployment by embedding critical non-functional requirements throughout the development lifecycle. These stories are implemented iteratively alongside the functional ones.

1. **US-SIL-007: Logging & Monitoring**
    - **Rationale:** Essential for debugging during development and critical for operational visibility. Logging should be integrated from the very beginning of development of each component.
    - **Dependencies:** Should be integrated incrementally with all functional stories from Phase 1.
2. **US-SIL-006: Concurrency Management & Database Safety**
    - **Rationale:** Since the application will run on multiple servers, concurrency is a fundamental architectural concern. While the specific implementation might evolve, the design considerations and preliminary strategies for concurrency safety should be woven into the development of all stories that interact with the shared database (especially US-SIL-005 and US-SIL-010). Testing for concurrency should happen as soon as components interact with the database.
    - **Dependencies:** Implicitly impacts US-SIL-010, US-SIL-011, US-SIL-005, and any other database write operations.
3. **US-SIL-008: Unit and Integration Testing**
    - **Rationale:** Testing is continuous. Unit tests are written as each component is developed. Integration tests are built as modules are integrated. This story represents the overarching commitment to quality and the specific coverage targets.
    - **Dependencies:** All functional stories (tests are written *for* them).
4. **US-SIL-009: CI/CD Pipeline Definition**
    - **Rationale:** Automating the build and test process is crucial for efficient development and quality assurance. The pipeline should be set up early to run builds and tests regularly. Deployment aspects will follow as the application matures.
    - **Dependencies:** US-SIL-008 (pipeline runs tests), and the successfully built application.