# Software Requirements Specification (SRS) - Implementation Guide

## Project: Arealis Reconciliation & Data Platform

## Team: Data & DevOps

Lead: Kshitij

Version: 1.2 (Precision Implementation Guide)

Date: September 25, 2025

### 1. Introduction

#### 1.1 Purpose

This document provides a detailed specification for the components to be designed, built, and delivered by the Data & DevOps team within the 15-day sprint (Sep 24 - Oct 6). The scope includes data ingestion, reconciliation, journal export, testing simulators, and the foundational cloud environment. The primary objective is to deliver a functional, pilot-demoable system skeleton that meets the acceptance criteria for the October 6th demo.

#### 1.2 Scope

The components detailed in this SRS are:

* **Environment & CI/CD:** The cloud infrastructure and deployment pipelines.
* **Rail Statement Simulators:** Mock bank servers for end-to-end testing.
* **Ingest Pipeline:** The entry point for all customer payment data.
* **Reconciliation Engine (ARL):** The core logic for matching payments to statements.
* **Journal Generator:** The service for creating accounting-ready exports.
* **Evidence Store:** The secure, immutable storage for audit artifacts.
* **Observability:** The monitoring, logging, and alerting framework.

This document will structure the requirements into three development phases, aligned with the project's key gates, with added technical guidance for implementation.

### Phase 1: Foundation & Simulation (Target Date: Sep 27 - Gate 1)

**Goal:** Establish the core infrastructure and simulators to unblock all other teams and enable a basic, end-to-end "happy path" test.

#### 2. Component: Environment & Foundational CI/CD

##### 2.1 Introduction

This component covers the setup of the foundational cloud environments and the initial continuous integration/continuous deployment pipeline.

##### 2.2 Functional Requirements

* **FR-ENV-01:** The system shall have two separate, isolated environments: development and staging.
* **FR-ENV-02:** All cloud resources for these environments shall be provisioned using an Infrastructure as Code (IaC) tool (Terraform).
* **FR-SEC-01:** A centralized secrets management service (AWS Secrets Manager) shall be provisioned to store all application secrets, database credentials, and API keys.
* **FR-CICD-01:** A basic CI/CD pipeline shall be established that automatically deploys committed code to the development environment.

##### 2.3 Implementation Plan

* **Infrastructure (Terraform):**
  + **VPC:** Define a VPC with a CIDR block of 10.0.0.0/16.
  + **Subnets:** Create two public and two private subnets across two Availability Zones (e.g., ap-south-1a, ap-south-1b).
  + **NAT Gateway:** For cost-effectiveness in dev/staging, provision a single NAT Gateway in one AZ, with routes from both private subnets pointing to it.
  + **Modules:** Structure the Terraform code into modules for vpc, iam, s3, and security\_groups to promote reuse.
* **Secrets Management (AWS Secrets Manager):**
  + Create a single secret named arealis/dev/app-secrets. Store a placeholder JSON structure: {"DATABASE\_URL": "placeholder", "EXTERNAL\_API\_KEY": "placeholder"}.
  + Services will access this secret via an IAM Role with secretsmanager:GetSecretValue permission.
* **CI/CD Pipeline (GitHub Actions):**
  + Create a workflow file at .github/workflows/deploy-dev.yml.
  + **Trigger:** on: push: branches: [ develop ].
  + **Authentication:** Use GitHub's OIDC provider to securely authenticate with AWS. This avoids storing long-lived access keys as repository secrets.
  + **Workflow Steps:**
    1. actions/checkout@v3
    2. aws-actions/configure-aws-credentials@v2 (with OIDC role ARN)
    3. hashicorp/setup-terraform@v2
    4. terraform init
    5. terraform validate
    6. terraform plan -out=tfplan
    7. terraform apply -auto-approve tfplan

#### 3. Component: Rail Statement Simulators

##### 3.1 Introduction

This component provides mock bank servers that simulate the behavior of various payment rails. This is a critical dependency for the ACC, PDR, and Frontend teams to test their logic.

##### 3.2 Functional Requirements

* **FR-SIM-01:** A server-side API shall be created to act as the simulator.
* **FR-SIM-02:** The API shall expose distinct POST endpoints for each payment rail: /simulate/rtgs, /simulate/neft, /simulate/imps, and /simulate/upi.
* **FR-SIM-03:** Each endpoint shall accept a JSON payload containing transaction details.
* **FR-SIM-04:** The request payload must allow specifying a desired outcome (success or failure). If failure is chosen, a failureCode can be provided.
* **FR-SIM-05:** Upon receiving a request, the API shall return a JSON object mimicking a bank statement line item, including a unique transaction reference (UTR/RRN), status, and reason code.
* **FR-SIM-06:** The simulator shall support a predefined list of realistic failure reason codes.

##### 3.3 Implementation Plan

* **Technology:** A Node.js server using the Express.js framework.
* **API Schema:**
  + **Request (POST /simulate/rtgs):**  
    {  
     "amount": 250000.00,  
     "remitterAccount": "0123456789",  
     "beneficiaryAccount": "9876543210",  
     "beneficiaryIFSC": "HDFC0000001",  
     "outcome": "failure",  
     "failureCode": "AC04"  
    }
  + **Response (200 OK):**  
    {  
     "transactionId": "HDFC42025092700001234",  
     "transactionDate": "2025-09-27T10:30:00Z",  
     "status": "REJECTED",  
     "reasonCode": "AC04",  
     "reasonDescription": "Account specified is closed",  
     "amount": 250000.00  
    }
* **Logic (server.js):**
  + The UTR/RRN will be generated using the format: YYYYMMDD + BANK\_IFSC\_PREFIX + 9\_DIGIT\_RANDOM\_SEQUENCE.
  + Create a returnCodes.js module exporting an object:  
    export const codes = {  
     "AC04": "Account specified is closed",  
     "AM04": "Insufficient funds",  
     "BE01": "Beneficiary name mismatch"  
    };
  + The route handler will look up the failureCode in this module.
* **Deployment:**
  + Containerize the application using a multi-stage Dockerfile for a smaller production image.
  + Deploy as an AWS Fargate service with a configuration of cpu: 256 and memory: 512. Use an Application Load Balancer to expose the service.

### Phase 2: Core Data Processing & Reconciliation (Target Date: Sep 30 - Gate 2)

**Goal:** Build the main data pipeline, from ingestion to reconciliation and journal generation, and prove the core business logic can achieve ≥90% auto-reconciliation.

#### 4. Component: Ingest Pipeline

##### 4.1 Introduction

This component is the secure and validated entry point for customer payment instruction files.

##### 4.2 Functional Requirements

* **FR-ING-01:** An API endpoint shall be provided for uploading payment instruction files in CSV format.
* **FR-ING-02:** The system must validate every uploaded file against a predefined schema.
* **FR-ING-03:** Files that fail validation shall be rejected with a 400 Bad Request error.
* **FR-ING-04:** The system shall calculate a hash of the file's content to detect and reject exact duplicates.
* **FR-ING-05:** Successfully validated files shall be assigned a unique batch\_id and stored in a designated "processed" S3 bucket.
* **FR-ING-06:** Upon successful ingestion, a batch.created event shall be published to the system's Event Bus.

##### 4.3 Implementation Plan

* **Technology:** An AWS Lambda function (Node.js) triggered by an S3 PutObject event on the arealis-raw-uploads-dev bucket.
* **Validation Schema (schema.json):**  
  {  
   "columns": [  
   {"name": "Transaction ID", "type": "string", "required": true},  
   {"name": "Amount", "type": "number", "required": true},  
   {"name": "Beneficiary Account", "type": "string", "required": true}  
   ]  
  }
* **De-duplication (DynamoDB):** Use a PutItem operation with ConditionExpression: "attribute\_not\_exists(fileHash)" to perform an atomic check and insert, preventing race conditions.
* **Event Publishing (EventBridge):**
  + **Event Payload:**  
    {  
     "Source": "arealis.ingest",  
     "DetailType": "batch.created",  
     "Detail": {  
     "batchId": "batch-uuid-12345",  
     "s3Bucket": "arealis-processed-batches-dev",  
     "s3Key": "batch-uuid-12345.csv"  
     }  
    }
* **Step-by-Step Logic (Lambda Handler):**
  1. Receive S3 event for a new object in the 'raw' bucket.
  2. Open a read stream to the S3 object.
  3. Create a crypto hash stream (crypto.createHash('sha256')). Pipe the S3 read stream into the hash stream.
  4. Simultaneously, pipe the S3 read stream to papaparse for validation against schema.json.
  5. Once the hash stream finishes, perform the conditional PutItem to DynamoDB. If it fails (duplicate), delete the raw S3 object and exit.
  6. If validation fails, delete the raw S3 object and log the validation errors.
  7. If both succeed, generate a UUID for batch\_id, copy the object to the 'processed' bucket with the new name, and then delete the raw object.
  8. Publish the batch.created event to EventBridge.

#### 5. Component: Reconciliation Engine (ARL)

##### 5.1 Introduction

This component is the analytical core, responsible for automatically matching payment instructions against the (simulated) bank statement data.

##### 5.2 Functional Requirements

* **FR-REC-01:** The engine shall be event-driven, initiating a reconciliation job upon receiving the necessary events.
* **FR-REC-02:** The engine shall implement a multi-pass matching logic.
* **FR-REC-03:** All transactions that do not find a match shall be classified according to a defined Exception Taxonomy.
* **FR-REC-04:** The final output of the engine for each batch shall be a single Recon JSON file stored in S3.

##### 5.3 Implementation Plan

* **Technology:** A Python application using Pandas, deployed as a container on AWS Fargate.
* **Triggering:** An SQS queue subscribes to an EventBridge rule with the pattern: { "source": ["arealis.ingest"], "detail-type": ["batch.created"] } (and another for statement.available later). The Fargate service will poll this queue.
* **Logic (Python/Pandas):**
  1. Load the payment instruction CSV into payments\_df.
  2. Load the corresponding bank statement data into statements\_df.
  3. **Data Cleaning:** Standardize column names, convert date columns to datetime objects, and ensure amount columns are float.
  4. **Matching Pass 1:**  
     matched\_df = pd.merge(  
      payments\_df,  
      statements\_df,  
      how='inner',  
      left\_on=['UTR', 'Amount', 'Date'],  
      right\_on=['UTR', 'Amount', 'Date']  
     )
  5. **Exception Identification:**  
     unmatched\_payments = payments\_df[~payments\_df['UTR'].isin(matched\_df['UTR'])]  
     unmatched\_statements = statements\_df[~statements\_df['UTR'].isin(matched\_df['UTR'])]
  6. The unmatched\_payments DataFrame becomes the TIMEOUT exceptions.
  7. The unmatched\_statements DataFrame becomes the UNKNOWN\_REF exceptions.
  8. Convert these DataFrames into the required JSON structure and upload the final Recon JSON to S3.

#### 6. Component: Journal Generator

##### 6.1 Introduction

This service transforms the reconciled data into an import-ready format for accounting software.

##### 6.2 Functional Requirements

* **FR-JOU-01:** The service shall be automatically triggered by the creation of a new Recon JSON file in S3.
* **FR-JOU-02:** It shall parse the matched and exceptions arrays from the JSON file.
* **FR-JOU-03:** It shall generate a single CSV file, strictly adhering to the Tally v1 import format.

##### 6.3 Implementation Plan

* **Technology:** A lightweight AWS Lambda function (Python) triggered by an S3 PutObject event on the arealis-recon-output-dev bucket, filtered by suffix: ".json".
* **Logic (Lambda Handler):**
  1. Download and parse the triggering Recon JSON file.
  2. Initialize an in-memory CSV writer (io.StringIO and csv.writer).
  3. Write the header row: ['Date', 'Particulars', 'Vch Type', 'Vch No.', 'Debit Amount', 'Credit Amount', 'Narration'].
  4. Iterate through the matched array in the JSON. For each transaction:
     + **Debit Entry:** Write a row for the debit side (e.g., from the remitter account).
     + **Credit Entry:** Write a row for the credit side (e.g., to the beneficiary account).
  5. Upload the resulting CSV string to the arealis-evidence-store-dev bucket.

### Phase 3: Hardening, Security & Observability (Target Date: Oct 3 - Gate 3)

**Goal:** Secure the system, implement robust monitoring, and finalize all deliverables for the demo.

#### 7. Component: Evidence Store

##### 7.1 Introduction

This component ensures all critical artifacts are stored securely and verifiably for audit purposes.

##### 7.2 Functional Requirements

* **FR-EVI-01:** An S3 bucket shall be configured as the Evidence Store with immutability features (Object Lock) enabled.
* **FR-EVI-02:** The following artifacts shall be stored as evidence: original uploaded CSV, Recon JSON, and the generated Tally CSV.
* **FR-EVI-03:** A secure API endpoint shall be created to provide temporary, read-only, pre-signed URLs for accessing evidence artifacts.
* **FR-EVI-04:** The system must calculate and store a SHA256 hash of every file upon ingestion into the store.

##### 7.3 Implementation Plan

* **Technology:** S3, API Gateway (HTTP API), and a Lambda function (Node.js).
* **Immutability:** In Terraform, the aws\_s3\_bucket resource will include:  
  object\_lock\_configuration {  
   object\_lock\_enabled = "Enabled"  
  }  
  aws\_s3\_bucket\_versioning { ... }
* **Signed URL API (Lambda Logic):**  
  // Pseudo-code for handler  
  import { getSignedUrl } from "@aws-sdk/s3-request-presigner";  
  import { S3Client, GetObjectCommand } from "@aws-sdk/client-s3";  
    
  export const handler = async (event) => {  
   const s3Key = event.pathParameters.key;  
   const command = new GetObjectCommand({ Bucket: 'arealis-evidence-store-dev', Key: s3Key });  
   const url = await getSignedUrl(new S3Client({}), command, { expiresIn: 300 }); // 5 minute expiry  
   return {  
   statusCode: 302, // Redirect  
   headers: { "Location": url }  
   };  
  };
* **Hashing:** When services (Ingest, Recon, Journal) upload files to the store, they must calculate the SHA256 hash and include it as object metadata: Metadata: { "sha256-hash": "your-hash-here" }.

#### 8. Component: Observability

##### 8.1 Introduction

This component provides the necessary visibility into the system's health, performance, and operational status.

##### 8.2 Functional Requirements

* **FR-OBS-01:** All services shall output logs in a structured JSON format.
* **FR-OBS-02:** Every log entry must contain a consistent set of contextual identifiers.
* **FR-OBS-03:** The system shall collect and expose the key metrics: match%, exception aging, pack build time.
* **FR-OBS-04:** A metrics dashboard shall be created to visualize these KPIs in real-time.
* **FR-OBS-05:** Alerting rules shall be configured to notify the team of critical failures.

##### 8.3 Implementation Plan

* **Technology:** AWS CloudWatch (Logs, Metrics, Dashboards, Alarms).
* **Structured Logging Example:** A log line from the Recon Engine should look like:  
  {  
   "level": "INFO",  
   "timestamp": "2025-09-30T14:00:00Z",  
   "message": "Reconciliation job completed",  
   "batchId": "batch-uuid-12345",  
   "matchedCount": 95,  
   "exceptionsCount": 5,  
   "matchPercentage": 95.0  
  }
* **Custom Metrics:** The Recon Engine will make an API call to CloudWatch:  
  # boto3 pseudo-code  
  cloudwatch.put\_metric\_data(  
   Namespace='Arealis/Reconciliation',  
   MetricData=[{  
   'MetricName': 'AutoMatchPercentage',  
   'Dimensions': [{'Name': 'Environment', 'Value': 'development'}],  
   'Value': 95.0,  
   'Unit': 'Percent'  
   }]  
  )
* **Dashboard (Terraform):** Define an aws\_cloudwatch\_dashboard resource. The dashboard\_body will be a JSON string defining widgets for AutoMatchPercentage (Gauge), IngestCount (Number), and JournalGeneratedCount (Number).
* **Alarms (Terraform):**  
  resource "aws\_cloudwatch\_metric\_alarm" "recon\_match\_rate\_low" {  
   alarm\_name = "recon-match-rate-too-low-dev"  
   comparison\_operator = "LessThanThreshold"  
   evaluation\_periods = "2"  
   metric\_name = "AutoMatchPercentage"  
   namespace = "Arealis/Reconciliation"  
   period = "300"  
   statistic = "Average"  
   threshold = "90"  
   alarm\_description = "Reconciliation auto-match rate is below 90% for 10 minutes."  
  }
* **Final Security Check:** Before Gate 3, conduct a manual review of IAM roles to ensure they follow the principle of least privilege. Scan all application code for any hardcoded secrets.