# Software Requirements Specification (SRS) - Implementation Guide

## Project: Arealis Reconciliation & Data Platform

## Team: Data & DevOps

Lead: Kshitij
Version: 1.3 (Local-First 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 local development environment. The primary objective is to deliver a functional, pilot-demoable system skeleton that runs entirely on a local machine.

### 1.2 Scope

The components detailed in this SRS are:

- **Local Environment & CI/CD:** The local containerized environment and validation 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 a local-first implementation.

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

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

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

#### 2.1 Introduction

This component covers the setup of the foundational local, container-based environments and the initial continuous integration pipeline for code validation.

#### 2.2 Functional Requirements

- **FR-ENV-01:** The entire multi-service application shall be defined and run locally using Docker Compose.
- **FR-ENV-02:** A single docker-compose up command shall start all required services for a development environment.
- **FR-SEC-01:** A local .env file shall be used to manage all application secrets, database credentials, and API keys. This file will be excluded from version control.
- **FR-CICD-01:** A basic CI pipeline shall be established to build Docker images and run tests, ensuring code quality before merging.

#### 2.3 Implementation Plan

- **Environment (Docker Compose):**
  - Create a docker-compose.yml file at the root of the project.
  - This file will define all services (e.g., simulator, ingest-api, recon-engine), their container images, exposed ports, and shared volumes.
  - Create a shared Docker volume (e.g., arealis-data) to be mounted into services that need to exchange files.
- **Secrets Management (.env file):**
  - Create a .env file at the project root.
  - Example content: DATABASE_URL="placeholder"
  - The docker-compose.yml file will use the env_file directive to load these variables into the containers.
  - Ensure .env is added to your .gitignore file.
- **CI Pipeline (GitHub Actions):**
  - Create a workflow file at .github/workflows/build-and-test.yml.
  - **Trigger:** on: push: branches: [ develop ].
  - **Workflow Steps:**
    1. actions/checkout@v3
    2. For each service, navigate to its directory.
    3. Run docker build . to ensure the Docker image can be built successfully.
    4. Run unit tests within the container context if applicable. (Note: End-to-end deployment is not part of CI in this local-only setup).

## 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.
- **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:** (Unchanged from previous version)
- **Logic (server.js):** (Unchanged from previous version)
- **Deployment (Docker Compose):**
  - Define a service named simulator in your docker-compose.yml.
  - Use the build: ./simulator-service directive to build its Docker image.
  - Map ports to allow access from the host machine: ports: - "3000:3000".

## 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 shared local directory.
- **FR-ING-06:** Upon successful ingestion, a batch.created event shall be published to a local message broker.

#### 4.3 Implementation Plan

- **Technology:** A Node.js/Express API service.
- **Storage:** Files will be saved to a local directory (e.g., ./data/processed) which is mounted

as a shared Docker volume.

- **De-duplication:** Use a Redis container (defined in Docker Compose) as a key-value store. The SHA256 hash of the file will be the key. Use the SETNX (SET if Not eXists) command for an atomic check.
- **Event Publishing:** Use Redis Pub/Sub. After successful ingestion, the service will publish a JSON message to a channel named arealis:events.
- **Step-by-Step Logic (API Handler for POST /upload):**
  1. Receive the file upload.
  2. Stream the file content to calculate its SHA256 hash.
  3. Simultaneously, validate the CSV structure using papaparse.
  4. Attempt to SETNX filehash:your-hash "processed" in Redis. If the command returns 0, the key already exists (duplicate); return an error.
  5. If validation or de-duplication fails, return a 400 error.
  6. If both succeed, generate a UUID for batch_id and save the file to the shared volume (e.g., /data/processed/batch-uuid-12345.csv).
  7. Publish the batch.created event message to the arealis:events Redis channel.

## 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, listening to the local message broker.
- **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 shall be a single Recon JSON file stored in the shared local directory.

#### 5.3 Implementation Plan

- **Technology:** A Python application using Pandas, defined as a service in Docker Compose.
- **Triggering:** The Python script will connect to the Redis container and subscribe to the arealis:events channel. It will filter for batch.created messages.
- **Logic (Python/Pandas):**
  1. On receiving a batch.created message, extract the file path.
  2. Load the payment instruction CSV from the shared volume (/data/processed/...) into payments_df.
  3. (For now) Load a corresponding sample bank statement file from the volume.
  4. **Data Cleaning & Matching:** (Logic unchanged from previous version).
  5. Convert the results into the required JSON structure and save the Recon JSON file to a sub-directory in the shared volume (e.g.,

/data/recon-output/batch-uuid-12345.json).

### 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 after a reconciliation job is complete.
- **FR-JOU-02:** It shall parse the Recon JSON file.
- **FR-JOU-03:** It shall generate a Tally v1 compliant CSV file in the shared local directory.

#### 6.3 Implementation Plan

- **Technology:** A lightweight Python script, running as a separate service in Docker Compose.
- **Triggering:** After the Reconciliation Engine saves its Recon JSON, it will publish a recon.completed event to the arealis:events Redis channel. The Journal Generator service will listen for this event.
- **Logic (Python Script):**
  1. On receiving a recon.completed message, extract the path to the Recon JSON file.
  2. Download and parse the JSON file from the shared volume.
  3. Initialize an in-memory CSV writer.
  4. Write the header and iterate through the matched array to create debit/credit entries.
  5. Save the resulting CSV string to the evidence store directory in the shared volume (e.g., /data/evidence/batch-uuid-12345.csv).

## 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 Implementation Plan

- **Technology:** A simple Node.js/Express static file server, defined as a service in Docker Compose.
- **Storage:** The service will be configured to serve files from the /data/evidence directory

within the shared Docker volume.

- **Immutability:** For local development, true immutability is not feasible. The process will rely on convention: services only *write* to this directory; they do not modify or delete.
- **Access:** The service will expose a port (e.g., 8080). The API will return a simple local URL to access evidence: http://localhost:8080/batch-uuid-12345.csv.
- **Hashing:** The logic for calculating and storing the SHA256 hash remains the same, but instead of S3 metadata, the hash could be stored in a Redis key or alongside the file. For simplicity, we'll log the hash upon creation.

## 8. Component: Observability

#### 8.1 Introduction

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

#### 8.2 Implementation Plan

- **Technology:** Prometheus for metrics collection and Grafana for dashboarding.
- **Structured Logging:** All services will log structured JSON to stdout. Developers will use docker-compose logs -f <service_name> to view real-time logs.
- **Metrics:**
  - Each service (Node.js/Python) will include a lightweight metrics library (e.g., prom-client for Node, prometheus-client for Python).
  - They will expose a /metrics endpoint that the Prometheus server can scrape.
- **Dashboard & Alarms:**
  - Add prometheus and grafana services to the docker-compose.yml file using their official images.
  - Create a prometheus.yml configuration file to define scrape targets (e.g., simulator:3000, recon-engine:8001).
  - Configure Grafana with a provisioning file (grafana.ini) to automatically add Prometheus as a data source.
  - Create a JSON file defining a basic Grafana dashboard to be provisioned on startup, showing key metrics like AutoMatchPercentage.
- **Final Security Check:** Before the final demo, ensure the .env file is not checked into version control.