Technical Analysis: User Migration Platform

1. Introduction and Architectural Goals

The implemented solution is an asynchronous distributed architecture designed to manage user data migration from a Legacy system ("OLD") to a new platform ("NEW") under constraints of data integrity, limited concurrency, and administrative traceability.

Guiding Principles

1. **Separation of Concerns (SoC) / Single Responsibility Principle (SRP):** The solution is strictly divided into layers (API, Core Logic, Data Access, Worker), ensuring each component has only one reason to change.
2. **Reliability and Integrity (SAGA Pattern):** The migration process, which involves two primary databases (source/target) and an audit database, is managed as a limited distributed transaction (SAGA) with explicit compensation (Rollback) mechanisms.
3. **Concurrency Management (Throttling):** The use of the SlotManager based on SemaphoreSlim enforces a functional/hardware limit on simultaneous migrations, protecting the NEW system from overload.

2. Architectural Structure and Components

The solution is organized into four main layers communicating via Dependency Injection (DI) and a Message Broker (simulated by IMessageProducer).

2.1 Presentation Layer (Migration.API)

This layer exposes the JWT-protected RESTful APIs.

Controller	Responsibility	Technical Details
AuthController	Token Generation (Testing)	Creates mock JWT tokens for Admin and User roles, reading user details (FullName, LegacyUserId) directly from the respective Repositories to enrich the Claims. Roles will not be developed in this prototype (any user can be admin).

AdminController	Administrative Functions	Global monitoring (MigrationManager) and concurrency control status (SlotManager). Includes the endpoint for **Forced Migration**.
UserOldController	Legacy User Interaction	Allows the OLD user to request migration (AcceptMigration) only if slots are available and the user is migratable.
UserNewController	NEW User Interaction	Provides access to migrated user data.

- **Strength:** Business logic (e.g., InitiateMigration, UserOldCanBeMigrated) is correctly delegated to IMigrationManager in Migration.Core, keeping Controllers thin.

2.2 Business Logic Layer (Migration.Core)

Contains the core logic, independent of the framework or database implementation.

Component	Core Responsibility	Technical Details
MigrationProcessor	Asynchronous SAGA Coordinator	Executes the entire migration process, handling validation, slot acquisition, and rollback logic (see Section 3).
IDataTransformer	Mapping & Normalization	Implements data transformation and validation (e.g., Normalized Email, DocumentType Mapping). Throws ValidationException on dirty data.
ISlotManager	Concurrency Control	Uses SemaphoreSlim to limit the number of simultaneous migrations (Throttling). Implemented

		as a **Singleton**.
IMigrationManager	API Orchestration	Gating logic (e.g., UserOldCanBeMigrated) and interaction with the Message Broker (IMessageProducer).

2.3 Data Access Layer (Migration.DataAccess)

Adheres to the Repository Pattern, isolating the DbContext and Entity Framework Core (EF Core) from the business logic.

- **Separate Repositories:** Three distinct DbContexts and Repository sets exist, each responsible for one database (OLD, NEW, Audit), ensuring physical data separation.
  - IUserOldRepository
  - IUserNewRepository
  - IMigrationStatusRepository
- **Data Seeding:** The UserOldDbContext includes mock data (HasData) to facilitate the testing environment and simulate the Legacy database.

2.4 Asynchronous Worker Layer (Migration.Worker)

Simulates an asynchronous message consumer that processes migration requests.

- **Infrastructure:** Uses the .NET IHost to set up the environment and resolve dependencies for the MigrationProcessor.
- **Deployment:** In a production environment, the core MigrationProcessor logic would be wrapped in a BackgroundService or triggered by an actual message queue listener.

3. Integrity Management (SAGA and Rollback)

The most critical stage is the ProcessMigrationAsync function within the MigrationProcessor.

SAGA Logic (Distributed Transaction)

Since three different databases are involved, a 3-phase SAGA coordinated by the MigrationStatus is implemented:

Phase	Involved Component	Action	Atomicity / Role
1. Acquisition & Validation	MigrationProcessor,	Verifies OLD existence, validates	**Optimization:** Executed *before*

	IUserOldRepository, IDataTransformer	data, and acquires the slot.	slot acquisition to prevent resource waste.
2. NEW Write & Log Update	IUserNewRepository, IMigrationStatusRepository	Creates the user in the NEW DB, then updates the audit log to **SUCCESS** (source of truth).	**Consistency:** This step acts as the SAGA's *Commit*. If it fails, the NEW user is compensated (deleted).
3. OLD Finalization	IUserOldRepository	Updates the OLD user to IsMigrated = true.	**Idempotency:** If this step fails (e.g., transient DB issue), the user is still marked as SUCCESS in the Log and can be reconciled later.

Compensation Mechanism (Rollback)

The MigrationProcessor handles the fallback logic upon critical failure:

1. **Critical Failure (Phase 2 - NEW Write):** If the process fails *after* transformation but *before* final completion (e.g., DB failure during NEW user creation or log update):
    - The catch block is triggered.
    - Compensation is called: _newUserRepository.DeleteNewUser(newUserId.Value).
    - The audit log is updated to **FAILED** for administrative traceability.

This design ensures no orphaned data exists in the NEW DB and the state of the OLD/Audit system remains consistent.