Technical Assessment MyFuel System

PREPARED BY

Hossein Ashrafipoor

9/4/2025

Table of Contents

1. Introduction

1.1 Purpose of the Document

The purpose of this document is to present the system design and partial implementation of the **MyFuel Transaction Processing Service**. The document translates business requirements into a technical design that ensures scalability, reliability, and extensibility. It also provides details of one implemented core service using NestJS, demonstrating clean architecture principles and readiness for production deployment.

1.2 Scope and Assumptions

This document focuses on:

- Designing the high-level system architecture including flow diagrams, ERD, and component interactions.
- Implementing the **Webhook Transaction Service** that validates and processes fuel purchase requests.
- Outlining technical considerations such as concurrency handling, idempotency, and caching strategies.
- Demonstrating best practices for CI/CD pipelines and documentation.

Assumptions:

- All transactions are submitted via secure HTTPS webhooks from trusted petrol stations.
- The system relies on a relational database as the system of record.
- Daily and monthly card limits reset automatically at the start of each new period (UTC).
- Redis may be used as a performance optimization but the source of truth remains in the database.

1.3 Audience

This document is intended for:

- **Engineering Leads and Architects** reviewing design quality and scalability.
- **Backend Developers** responsible for implementation and maintenance.
- **DevOps Engineers** supporting CI/CD, monitoring, and operations.

2. Business Requirements

2.1 Overview of MyFuel Service

MyFuel is a digital fleet management service that enables organizations to control and monitor fuel expenses. Each organization maintains a **prepaid balance account**, from which all fuel transactions are deducted in real time. Organizations can issue multiple **fuel cards**, and every card is subject to both daily and monthly spending limits.

2.2 Organizational Accounts and Prepaid Balances

- Each organization holds a prepaid balance account.
- All approved transactions are deducted immediately from this account.
- Insufficient balance results in rejection of the transaction.

2.3 Fuel Card Limits (Daily / Monthly)

- Each card has a defined **daily spending limit** and **monthly spending limit**.
- At the start of each new day and month, counters reset to zero automatically.
- Transactions that would cause these limits to be exceeded must be rejected.

2.4 Transaction Flow (from Petrol Station Webhook)

1. A petrol station initiates a transaction via a webhook call.
2. The system identifies the card and the associated organization.
3. The system validates:
    - Whether the organization's balance covers the requested amount.
    - Whether the card's daily limit allows the transaction.
    - Whether the card's monthly limit allows the transaction.
4. If validation passes:

    - The transaction is approved.
    - The organization’s balance is reduced.
    - The card’s daily and monthly usage counters are updated.
5. If validation fails:
    - The transaction is rejected.
    - The petrol station receives a failure response with the proper error code.

3. Flow of Events

3.1 Transaction Lifecycle

The lifecycle of a fuel purchase transaction includes the following steps:

1. **Transaction Initiation**
   - A fuel purchase is initiated at a petrol station.
   - The station sends the transaction data to the MyFuel system through a secure webhook.
2. **Card and Organization Identification**
   - The system locates the card based on the provided card number.
   - The corresponding organization is determined from the card information.
3. **Validation Phase**
   - **Balance Check:** Verify that the organization's prepaid account has sufficient funds.
   - **Daily Limit Check:** Confirm that the card's daily spending limit has not been exceeded.
   - **Monthly Limit Check:** Confirm that the card's monthly spending limit has not been exceeded.
4. **Processing**
   - If all validations pass:
     - The transaction is **approved**.
     - The transaction amount is deducted from the organization's balance.
     - The card's daily and monthly counters are incremented accordingly.

    - If any validation fails:
        - The transaction is **rejected**.
        - An appropriate error code and message are returned to the petrol station.

5. **Logging and Auditing**
    - Every transaction, whether approved or rejected, is logged in the system.
    - Logs include transaction ID, card number, organization ID, timestamp, amount, status, and reason code (if rejected).

3.2 Validation Rules

- **Atomicity:** Balance deduction and counter update must occur within a single atomic operation.
- **Idempotency:** Duplicate webhook requests with the same transaction ID must not create duplicate entries.
- **Time Zone Standardization:** All date and time calculations are performed in UTC to ensure consistency.

3.3 Approval and Rejection Scenarios

- **Approval:**
    - Balance sufficient AND limits not exceeded.
    - The system returns a success response (HTTP 200) with transaction details.
- **Rejection:**
    - Balance insufficient → Error code: `INSUFFICIENT_FUNDS`.
    - Daily or monthly limit exceeded → Error code: `LIMIT_EXCEEDED`.
    - Invalid or unknown card → Error code: `INVALID_CARD`.

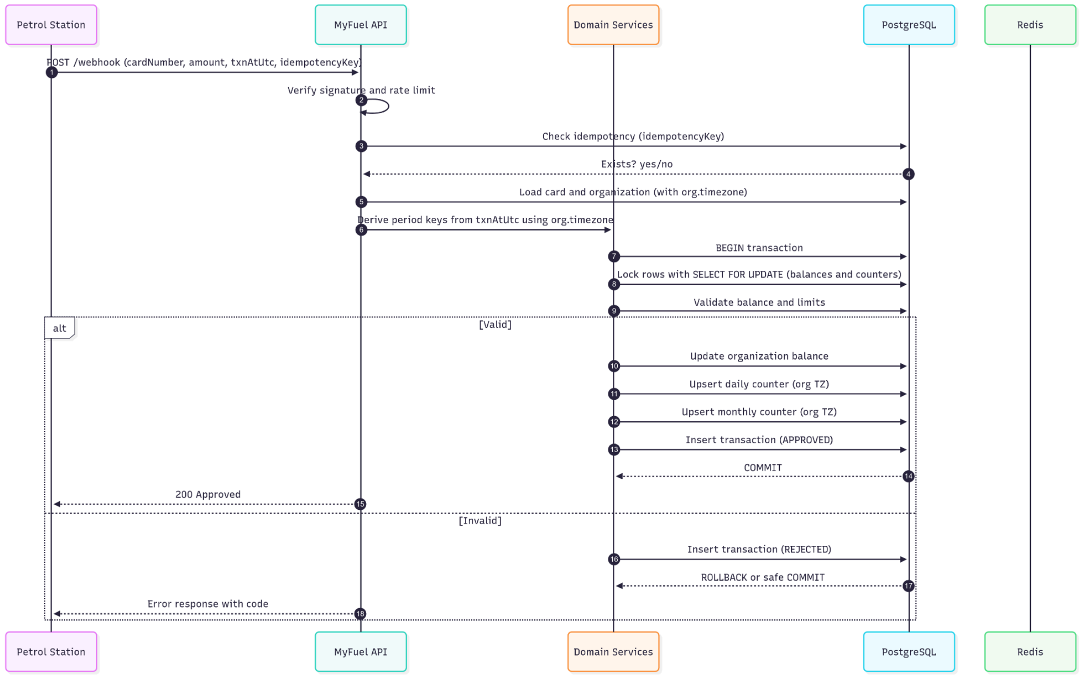
4. System Design

4.1 Flow Diagram (Webhook to Persistence)

Narrative.
A petrol station posts a webhook to the MyFuel API. The system authenticates the request, enforces idempotency, looks up the card and organization, **derives daily/monthly period keys using the organization's time zone** while all timestamps remain in UTC, validates balance and limits, applies changes atomically, emits events, and returns a response.

Key Steps.

1. Receive webhook (`POST /webhook/fuel-transactions`) with `cardNumber`, `amount`, `txnAtUtc`, `stationId`, `idempotencyKey`.
2. Authenticate & verify signature / allowlist.
3. Enforce idempotency (check `idempotencyKey`).
4. Resolve `card → organization` and fetch `organization.timezone`.
5. Convert `txnAtUtc` **(UTC)** → `txnLocal` using org IANA TZ.
6. Derive `dailyKey = yyyy-MM-dd(txnLocal)` and `monthlyKey = yyyy-MM(txnLocal)`.
7. Begin DB transaction; lock relevant rows.
8. Validate: org balance, daily limit, monthly limit.
9. Apply: deduct balance, increment counters (daily & monthly), write transaction row.
10. Commit, emit `TransactionApproved` or `TransactionRejected` event.
11. Return 200 (approved) or appropriate error code.



4.2 Entity-Relationship Diagram (ERD)

Entities & Relationships.

- **organizations** (1) —— (N) **cards**
- **cards** (1) —— (N) **transactions**
- **organizations** (1) —— (N) **limit_counters** (scoped by org+card+period)
- **organizations** (1) —— (N) **balance_ledger**; plus a current **org_balances** snapshot

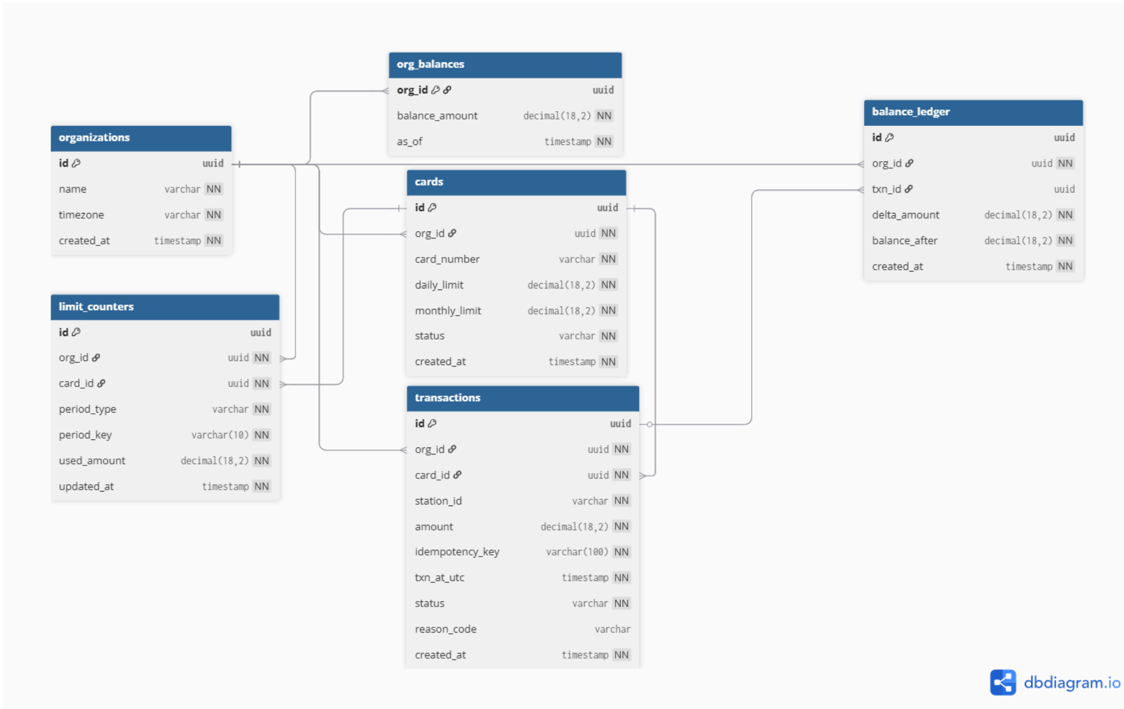
Tables (essential fields).

- `organizations`
  - `id (PK)`, `name`, `timezone (IANA string, e.g., 'Asia/Tehran')`, `created_at (UTC)`
- `cards`

  - `id (PK)`, `org_id (FK)`, `card_number (unique)`, `daily_limit`, `monthly_limit`, `status`, `created_at (UTC)`
- `org_balances` (current snapshot)
  - `org_id (PK/FK)`, `balance_amount`, `as_of (UTC)`
- `balance_ledger` (immutable history)
  - `id (PK)`, `org_id (FK)`, `txn_id (nullable FK)`, `delta_amount`, `balance_after`, `created_at (UTC)`
- `limit_counters`
  - `id (PK)`, `org_id (FK)`, `card_id (FK)`, `period_type ENUM('DAILY','MONTHLY')`, `period_key VARCHAR` (e.g., `2025-09-03` or `2025-09`), `used_amount`, `updated_at (UTC)`
  - **Unique index:** `(org_id, card_id, period_type, period_key)`
- `transactions`
  - `id (PK)`, `org_id (FK)`, `card_id (FK)`, `station_id`, `amount`, `idempotency_key (unique)`, `txn_at_utc (UTC)`, `status ENUM('APPROVED','REJECTED')`, `reason_code`, `created_at (UTC)`

Indexing (examples).

- `cards(card_number)` unique
- `transactions(idempotency_key)` unique
- `limit_counters(org_id, card_id, period_type, period_key)` unique
- `transactions(card_id, txn_at_utc)` for history queries



4.3 High-Level System Architecture

Components.

- **API Gateway / Webhook Controller (NestJS):** Receives webhook, validates schema, verifies signature, enforces rate limits.
- **Idempotency Guard:** Checks `idempotency_key` (unique constraint + fast lookup).
- **Card & Org Resolver:** Finds `card`, `organization`, loads `organization.timezone`.
- **Period Key Deriver (TZ-Aware):** Converts `txn_at_utc` → local time using org TZ; derives `dailyKey` and `monthlyKey`.
- **Validation Service:** Balance sufficiency, daily limit, monthly limit.
- **Ledger & Counter Service:** Atomic DB transaction to deduct balance, increment counters, write ledger & transaction rows.
- **Cache (Redis, optional):** Read-through cache for hot `limit_counters` and card profiles; DB remains source of truth.

- **Observability:** Structured logs (with `requestId`), metrics (QPS/latency/errors), tracing.
- **Event Bus (optional):** Publish `TransactionApproved/Rejected` for async consumers (notifications, analytics).

Non-Functional Properties.

- **Consistency & Atomicity:** Single DB transaction with row-level locks (`SELECT … FOR UPDATE`) on balance and counter rows.
- **Scalability:** Stateless API instances; horizontal scale. Use connection pooling; cache hot reads.
- **Fault Tolerance:** Idempotent writes; retries safe with `idempotency_key`.
- **Security:** HTTPS + signed webhook, IP allowlist, input validation, rate limiting, least-privilege DB user.
- **Time & TZ:** All persisted timestamps are UTC. **Reset windows derive from `organization.timezone`** using IANA database, handling DST correctly.

4.4 Scalability & Extensibility

- **Additional Rules:** Weekly limits, vehicle-based limits, and org-wide aggregate caps can be added by introducing new `period_type` rows (e.g., `WEEKLY`) and corresponding `period_key` formats (`YYYY-Www`) without schema changes elsewhere.
- **Sharding Options:** Partition `transactions` and `limit_counters` by `org_id` or by time for very high volumes.
- **API Evolution:** Version endpoints (`/v1/webhook/...`) and maintain OpenAPI 3.1 for contract tests.
- **Housekeeping:** Scheduled jobs to archive/prune old `limit_counters` and `transactions` based on retention policies.

5. Implementation Task

5.1 Selected Core Service

Webhook Transaction Service (NestJS):
The goal of this task is to implement the `POST /v1/webhook/fuel-transactions` endpoint, which processes fuel purchase requests from petrol stations. The service will validate the transaction details (balance, daily/monthly limits), process the transaction atomically, and respond idempotently.

5.2 API Specification

- **Endpoint:** `POST /v1/webhook/fuel-transactions`
- **Headers:**
    - Content-Type: application/json
    - Idempotency-Key: <uuid> *(required)*
    - X-Signature: <hex> *(required, HMAC-SHA256 over body)*
    - X-Signature-Timestamp: <epoch-millis> *(required, within 5 minutes)*
- **Request (JSON):**

```json
{
  "cardNumber": "4111-2222-3333-4444",
  "amount": 47.50,
  "txnAtUtc": "2025-09-03T20:30:00Z",
  "stationId": "ST-92810"
}
```

- **Response (200):**

```json
{
  "status": "APPROVED",
  "transactionId": "c1a7f4d0-2d8d-4c9f-9e1a-3c1a8c8f7b2e",
```

```
  "orgId": "...",
  "cardId": "...",
  "balanceAfter": 1250.35,
  "period": { "dailyKey": "2025-09-04", "monthlyKey":
"2025-09" },
  "requestId": "..."
}
```

- Errors:

  `400` INVALID_REQUEST

  `402` INSUFFICIENT_FUNDS

  `409` IDEMPOTENCY_MISMATCH (same key, different payload)

  `429` RATE_LIMITED

5.3 Processing Logic (Domain & Infrastructure Split)

1. Webhook Controller (`webhook.controller.ts`):
   - Receives the webhook request and validates headers (signature, idempotency key).
   - Calls `TransactionService` (domain) to process the transaction.
   - Returns a response to the petrol station (either success or error).
2. Transaction Service (`transactions.service.ts`) (Domain Logic):
   - Receives the transaction details (amount, card, station).
   - Derives period keys (dailyKey, monthlyKey) based on `organization.timezone`.
   - Validates: sufficient balance, daily and monthly limits.
   - Handles the business logic for updating balances and counters.
3. Balance and Limit Counter Update (`balances.service.ts`, `limits.service.ts`) (Domain Logic):

   - Deducts the amount from `org_balance` in a single atomic operation.
   - Updates the `limit_counters` for daily and monthly limits, ensuring consistency.

4. Idempotency (`idempotency.interceptor.ts`, `idempotency.service.ts`) (Infrastructure):
   - Ensures that repeated requests with the same `Idempotency-Key` produce the same result.
   - If the same key and payload are received, returns the previously stored response.

5. Security (`security.guard.ts`) (Infrastructure):
   - Validates the HMAC signature in the request headers to ensure authenticity.
   - Implements rate-limiting based on client IP or `stationId` to prevent abuse.

5.4 Error Handling & Idempotency

- Error handling: Each error type (e.g., `INSUFFICIENT_FUNDS`, `LIMIT_EXCEEDED`) is handled by the appropriate service and returned as part of the standardized API response.
- Idempotency:
   - If the same `idempotency_key` with the same payload is received, the same response is returned, without executing the logic again.
   - If a different payload with the same `idempotency_key` is received, a `409 Conflict` is returned to signal a mismatch.
   - This ensures that duplicate requests due to retries do not result in double charges or incorrect updates.

5.5 Validation & Limit Reset Strategy (Domain Logic)

- Limit Reset:

    - All timestamps are stored in UTC in the database.
    - The system derives the `dailyKey` and `monthlyKey` using the organization's timezone.
    - The limits for each period (daily, monthly) are automatically "reset" when the period key changes.
    - This ensures consistency in resetting limits based on the correct local time, while UTC is used for all data storage.

- Period Key Calculation:

```
// limits.service.ts (Domain Logic)
derivePeriodKeys(txnAtUtc: string, orgTz: string) {
  const localTime = toZonedTime(new Date(txnAtUtc), orgTz);  //
Convert UTC to org's local time
  const dailyKey = format(localTime, 'yyyy-MM-dd', { timeZone: orgTz
});  // Calculate daily period key
  const monthlyKey = format(localTime, 'yyyy-MM', { timeZone: orgTz
});  // Calculate monthly period key
  return { dailyKey, monthlyKey };
}
```

5.6 Persistence & Transactions (Infrastructure)

- Transaction Handling:
    - Atomicity is ensured by using row-level locks (`SELECT ... FOR UPDATE`) in a single DB transaction to prevent race conditions.
    - The `transactionService` is responsible for initiating the transaction, deducting balances, and updating counters.
- Database Entities:
    - `transactions`: Stores the transaction details with `idempotency_key`.
    - `org_balances`: Stores the current balance of the organization.

- `limit_counters`: Stores the daily/monthly limit counters with `period_key` and `used_amount`.

5.7 Security (Infrastructure)

- Signature Validation:
  - The HMAC signature is validated against the request body to ensure that the request is coming from a trusted source.
  - The `X-Signature` header contains the hash of the request body and timestamp, verified using a shared secret.
- Rate Limiting:
  - Each client (identified by IP or `stationId`) is rate-limited to prevent excessive requests and abuse of the system.
- IP Allowlist:
  - Only requests from allowed IPs (petrol stations) are processed.

5.8 Example cURL

```
curl -X POST https://api.myfuel.example/v1/webhook/fuel-transactions
\
  -H "Content-Type: application/json" \
  -H "Idempotency-Key: 1c9c7d0e-6a1f-4a9a-9e6a-5b8d72a3e0aa" \
  -H "X-Signature: 5ee4..." \
  -H "X-Signature-Timestamp: 1756921800000" \
  -d '{
    "cardNumber":"4111-2222-3333-4444",
    "amount":47.50,
    "txnAtUtc":"2025-09-03T20:30:00Z",
    "stationId":"ST-92810"
  }'
```

5.9 Minimal OpenAPI (excerpt)

```yaml
openapi: 3.1.0
info: { title: MyFuel Webhook API, version: 1.0.0 }
paths:
  /v1/webhook/fuel-transactions:
    post:
      summary: Process a fuel transaction webhook
      parameters:
        - in: header; name: Idempotency-Key; required: true; schema: { type: string, format: uuid }
        - in: header; name: X-Signature; required: true; schema: { type: string }
        - in: header; name: X-Signature-Timestamp; required: true; schema: { type: string }
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              required: [cardNumber, amount, txnAtUtc, stationId]
              properties:
                cardNumber: { type: string }
                amount: { type: number }
                txnAtUtc: { type: string, format: date-time, description: "UTC ISO-8601" }
                stationId: { type: string }
      responses:
        "200": { description: Approved }
        "400": { description: Invalid request }
        "401": { description: Auth failed }
        "402": { description: Insufficient funds }
        "409": { description: Idempotency mismatch }
        "429": { description: Rate limited }
```

6. Technical Considerations

6.1 Data Model & Persistence

- Database: PostgreSQL as the system of record; normalized schema with explicit foreign keys and unique constraints.
- Balances:
  - `org_balances` holds the current snapshot for fast reads.
  - `balance_ledger` is an immutable audit trail of all balance changes, including the post-change `balance_after`.
- Counters: `limit_counters` keeps daily/monthly aggregates using a composite uniqueness on `(org_id, card_id, period_type, period_key)`. Counters "reset" by encountering a new `period_key`.
- Transactions: `transactions` stores each attempt with `idempotency_key`, `txn_at_utc` (UTC), `status`, and `reason_code`.
- Precision: Monetary fields use `DECIMAL(18,2)` (or your org standard) with explicit rounding rules at boundaries.
- Indexes (examples):
  - `cards(card_number)` unique;
  - `transactions(idempotency_key)` unique;
  - `limit_counters(org_id, card_id, period_type, period_key)` unique;
  - `transactions(card_id, txn_at_utc)` for queries by card/time.

6.2 Concurrency & Atomicity

- Single atomic write path: Deduct balance + increment counters + write transaction occur in one DB transaction.

- Row-level locking: `SELECT … FOR UPDATE` on the `org_balances` row and the two `limit_counters` rows (daily & monthly) derived for the request.
- Isolation level: `READ COMMITTED` is typically sufficient with explicit row locks; consider `REPEATABLE READ` if you observe anomalies under peak load.
- Lock ordering: Always lock in a deterministic order (`org_balances` → daily counter → monthly counter) to minimize deadlocks.
- Alternative (high throughput): Per `(org_id, card_id)` serializer/queue to process updates sequentially, reducing lock contention at very high QPS.

6.3 Idempotency

- Definition: Multiple identical requests must have the same effect and response as a single request.
- Client contract: Clients send a unique `Idempotency-Key` (UUID) per logical transaction.
- Server behavior:
  - On first processing, persist the canonical result and return it.
  - On replay with the same key + same payload, return HTTP 200 with the stored result.
  - On replay with the same key + different payload, return 422 Unprocessable Entity (or `409 Conflict`) to signal mismatch.
- Storage options:
  - (Simple) Unique constraint on `transactions.idempotency_key`;
  - (Advanced) Dedicated `idempotency_keys` table storing payload hash + response body for strict matching.
- Crash safety: Track in-progress states so a node crash does not lead to double application on retry.

6.4 Validation & Error Handling

- Input schema: Enforce types, ranges, and formats (card number, amount ≥ 0, timestamp in ISO-8601 UTC).
- Clock skew: Accept a reasonable time window for `txn_at_utc` (e.g., ±10 minutes) and reject outliers or flag them for review.
- Error taxonomy:
  - `400 Bad Request` → schema/format error.
  - `401/403` → authentication/signature or authorization failed.
  - `402 Payment Required` → `INSUFFICIENT_FUNDS`.
  - `409 Conflict` → duplicate/idempotency conflict.
  - `429 Too Many Requests` → rate limit exceeded.
- Error body (contract):

```
{ "code": "LIMIT_EXCEEDED", "message": "Daily limit
exceeded", "requestId": "..." }
```

- Localization: Errors are system-facing; keep codes stable and messages developer-friendly.

6.5 Caching (Redis)

- What to cache: Hot reads such as card profiles and current `limit_counters` for today/month.
- Keys:
  - `card:{card_id}`;
  - `lc:{org_id}:{card_id}:D:{YYYY-MM-DD}` and `lc:{org_id}:{card_id}:M:{YYYY-MM}`.
- Policy: Read-through cache only; database remains the source of truth.
- Invalidation: On successful commit, invalidate or update the relevant cache keys.

- TTL: Short TTLs (e.g., 60–300s) to reduce staleness; avoid caching negative lookups for long.

6.6 Time & Time Zone (UTC storage, org-specific resets)

- Storage: All timestamps are persisted in UTC.
- Derivation: For each request, convert `txn_at_utc` to local time using `organization.timezone` (IANA) and derive:
    - `dailyKey = format(localDate, 'YYYY-MM-DD')`,
    - `monthlyKey = format(localMonth, 'YYYY-MM')`.
- DST handling: Use a TZ-aware library (IANA database). Overlapping/short days are handled by local calendar boundaries.
- Reference: See *Limits Reset Strategy* for the full algorithm and pseudo-code.

6.7 Security

- Transport & origin: HTTPS only; signed webhooks (HMAC with shared secret) and/or IP allowlist for petrol stations.
- Replay protection: Combine `Idempotency-Key` with a short-lived signature timestamp.
- Rate limiting: Per station and per organization to protect against abuse.
- Secrets management: Store secrets in a secure vault; rotate regularly.
- Least privilege: Separate DB users for read/write; narrow grants to required tables.
- PII minimization: Store only what is necessary; mask in logs.

6.8 Observability

- Structured logging: Include `requestId`, `idempotencyKey`, `org_id`, `card_id`, `status`, `amount`, latency. JSON logs preferred.
- Metrics (SLIs):
  - Throughput (QPS), p95/p99 latency, error rate, approval vs. rejection counts, idempotent replays.
- Dashboards: Separate views per org/station; live counters and balance deltas.
- Tracing: Distributed tracing around webhook path (ingress → DB ops → event emission).
- Alerting: SLO violations, sudden spikes in `LIMIT_EXCEEDED` or `INSUFFICIENT_FUNDS`, DB errors, cache timeouts.

6.9 Performance & Scalability

- Connection pooling: Tune pool sizes; use circuit breakers for downstreams.
- Indexes & plans: Monitor slow queries; add partial indexes if a few period_keys dominate.
- Partitioning (when needed): Time-based or by `org_id` for `transactions` and `balance_ledger`.
- Back-pressure: Return safe errors under overload; prefer queue-based serialization for hotspots.
- Batching: Housekeeping and archival in batches to avoid long locks.

6.10 Testing Strategy

- Unit tests: Domain logic (balance calc, limit checks, period key derivation).
- Integration tests: With a real DB container (migrations, locks, unique constraints).

- **Contract tests: OpenAPI-backed request/response validation and error codes.**
- **Idempotency tests: Replays (same key/same payload), mismatch (same key/different payload).**
- **TZ/DST tests: Boundary cases at local midnight, DST transitions, month ends.**
- **Load tests: Sustained QPS with realistic retry patterns.**

6.11 Data Retention & Archival

- **Transactions & ledger: Retain per compliance (e.g., ≥7 years).**
- **Counters: Keep 12–24 months for analytics; archive older periods.**
- **Purge jobs: Scheduled jobs with safeguards; maintain referential integrity.**

6.12 Configuration & Operations

- **12-factor: Config via environment variables; immutable builds; stateless services.**
- **Migrations: Versioned migrations (e.g., Flyway/Liquibase) executed on deploy.**
- **Health checks: Liveness/readiness endpoints; DB and cache ping checks.**
- **Feature flags: Gradual rollout of new validation rules or limits.**
- **Runbooks: Incident checklists for DB failover, cache outage, and message bus delays.**

7. CI/CD Pipeline

7.1 Development Workflow and Branching

- Branching Strategy:
    - `main` or `master`: Code that is ready for deployment to production.
    - `develop`: Code that is complete but still in testing or preparing for release.
    - `feature/*`: Each new feature is developed in its own branch.
    - `hotfix/*`: Critical fixes that need to be applied immediately to production.
- Workflow:
    - Changes are first developed in a `feature/*` branch.
    - After the code is reviewed, the changes are merged into `develop`.
    - Before deployment to production, changes in `develop` are merged into `main`.

7.2 Automated Testing Strategy

- Unit Tests:
  Write unit tests for every module and service (such as `TransactionService`, `BalanceService`, `WebhookController`) to cover all possible scenarios, including insufficient balance, daily/monthly limit exceeded, etc.
- Integration Tests:
  Perform integration tests to ensure that modules work together properly (e.g., the transaction process from receiving the webhook to updating balances and counters).

- Contract Tests:
  Use OpenAPI tools to ensure the API request and response match the documented contract.

7.3 Deployment Pipeline (GitHub Actions)

- CI Workflow:
  - Linting: Run linters to ensure code style consistency (prettier, eslint).
  - Unit Tests: Run unit tests using Jest or Mocha.
  - Build: Automatically build the project.
  - Deployment: Once all steps pass successfully, deploy the code to production or staging environments.

Sample GitHub Actions Workflow:

```yaml
name: CI/CD Pipeline

on:
  push:
    branches:
      - develop
      - main

jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Set up Node.js
        uses: actions/setup-node@v2
        with:
          node-version: '14'
      - name: Install dependencies
        run: npm install
      - name: Lint code
        run: npm run lint
```

```yaml
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Set up Node.js
        uses: actions/setup-node@v2
        with:
          node-version: '14'
      - name: Install dependencies
        run: npm install
      - name: Run tests
        run: npm test

  deploy:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Set up Node.js
        uses: actions/setup-node@v2
        with:
          node-version: '14'
      - name: Install dependencies
        run: npm install
      - name: Deploy to Production
        run: |
          npm run build
          # Add deployment steps here (e.g., deploy to Heroku, AWS,
etc.)
```

8. Plus Points

8.1 Unit Testing and Coverage

- Unit Tests:
  Write unit tests for each service, controller, and business logic layer (e.g., `TransactionService`, `BalanceService`, `WebhookController`).
    - Use tools like Jest for unit tests.
    - Code coverage should be at least 80%.
    - Example tests:
        - Approved transaction with sufficient balance.
        - Rejected transaction due to daily limit.
        - Tests for Idempotency-Key and X-Signature headers.

8.2 API Documentation (OpenAPI)

- Use Swagger or OpenAPI 3.1 for automatic API documentation.
- Use Swagger UI for interactive API testing.
- OpenAPI Example:

```yaml
openapi: 3.1.0
info:
  title: MyFuel Webhook API
  version: 1.0.0
paths:
  /v1/webhook/fuel-transactions:
    post:
      summary: Process a fuel transaction webhook
      requestBody:
```

```yaml
        required: true
        content:
          application/json:
            schema:
              type: object
              required: [cardNumber, amount, txnAtUtc, stationId]
              properties:
                cardNumber: { type: string }
                amount: { type: number }
                txnAtUtc: { type: string, format: date-time }
                stationId: { type: string }
      responses:
        "200":
          description: Approved transaction
        "400":
          description: Invalid request
        "402":
          description: Insufficient funds
```

8.3 Events and Handlers for Future Extensions

- For future capabilities, such as adding new limits (e.g., weekly or vehicle-based limits), events and handlers should be used.
  - Use Event-driven Architecture to emit events like `TransactionApproved`, `TransactionRejected`.
  - These events can be used in microservices or for notifying other systems.

9. Conclusion

9.1 Summary of Design Choices

- The system design is based on Domain-Driven Design and Modular Architecture.
- We store all timestamps in UTC and perform the daily/monthly limit reset based on the organization's timezone.
- Idempotency is fully implemented to prevent duplicate transactions.

9.2 Future Improvements

- Adding features like weekly limits or vehicle-based limits.
- Further optimization in Caching and leveraging Redis for faster responses.
- Improved scalability using Microservices and/or Event-driven Architecture for transaction processing.

---

10. Appendices

A. API Contract (OpenAPI Specification)

- All the detailed specifications of the API are provided in the OpenAPI Specification.

B. Example API Requests and Responses

- Example requests and responses for the API, including error codes.

C. README (Runbook for Setup & Execution)

- **Complete guide for setting up the project in testing and production environments.**

- **Instructions for Docker setup and how to run the tests.**

D. Architecture Decision Records (ADR)

- **Documenting all architectural decisions (like choosing UTC and Idempotency).**