# Task 1

## Understanding the Problem

The objective is to design a Payment Instruction Processing Service responsible for accepting, validating, persisting, and tracking payment instructions in a regulated enterprise environment.

This service acts as a system of record for payment instructions, ensuring durability, traceability, and correctness. It does not perform fund settlement but is designed to integrate with downstream processing systems in the future.

## Key Assumptions

### Transaction volume and peak behaviour

- The expected transaction volume is low to moderate (hundreds to thousands per day).
- During peak period, traffic may be bursty
- The system will prioritise durability and correctness over low latency.
- Horizontal scaling is not immediately required but should be achievable if demands increases.

### Ordering, idempotency, duplicate handling

- Upstream systems may retry submissions due to timeouts or transient failures.
- Each instruction includes a globally unique Instruction ID, treated as the idempotency key.
- Handling duplicate submissions:
    - Same Instruction ID + same payload > treated as idempotent retry, returns existing record.
    - Same Instruction ID + different payload > rejected with conflict to prevent data corruption.
    - Global ordering across instructions is not required.
    - Instructions are processed independently.

### Data integrity, audit, and compliance requirements

- Payment instructions are treated as immutable business records.
- Instructions must not be deleted once persisted.
- All state changes must be:
    - Explicit

    - Timestamped
    - Traceable
- The system must support:
    - Audit and investigation
    - Operational troubleshooting
    - Reconciliation with downstream systems
- Instruction state changes occur through controlled transitions.

### Operational Expectations

- The service is expected to be highly available, but not real-time critical.
- The database acts as the source of truth.
- The system must tolerate:
    - Client retries
    - Service restarts
    - Transient infrastructure failures
- Failures should be:
    - Detectable through logs and metrics
    - Recoverable without data loss
    - Safe for clients to retry
- Manual reconciliation and operational intervention are acceptable recovery mechanisms.

## Clarifying questions

### Product / Business Stakeholders

- Can payment instructions be amended or cancelled after submission?
- What instruction lifecycle statuses are required from a business perspective?
- Are execution dates allowed to be in the past?
- Are partial payments or split payments expected in future versions?
- What SLA is expected for instruction acceptance and processing?

### Enterprise Architecture

- Is there an existing standardized payment or account model that must be aligned with?
- Is the Instruction ID guaranteed to be globally unique across all source systems?

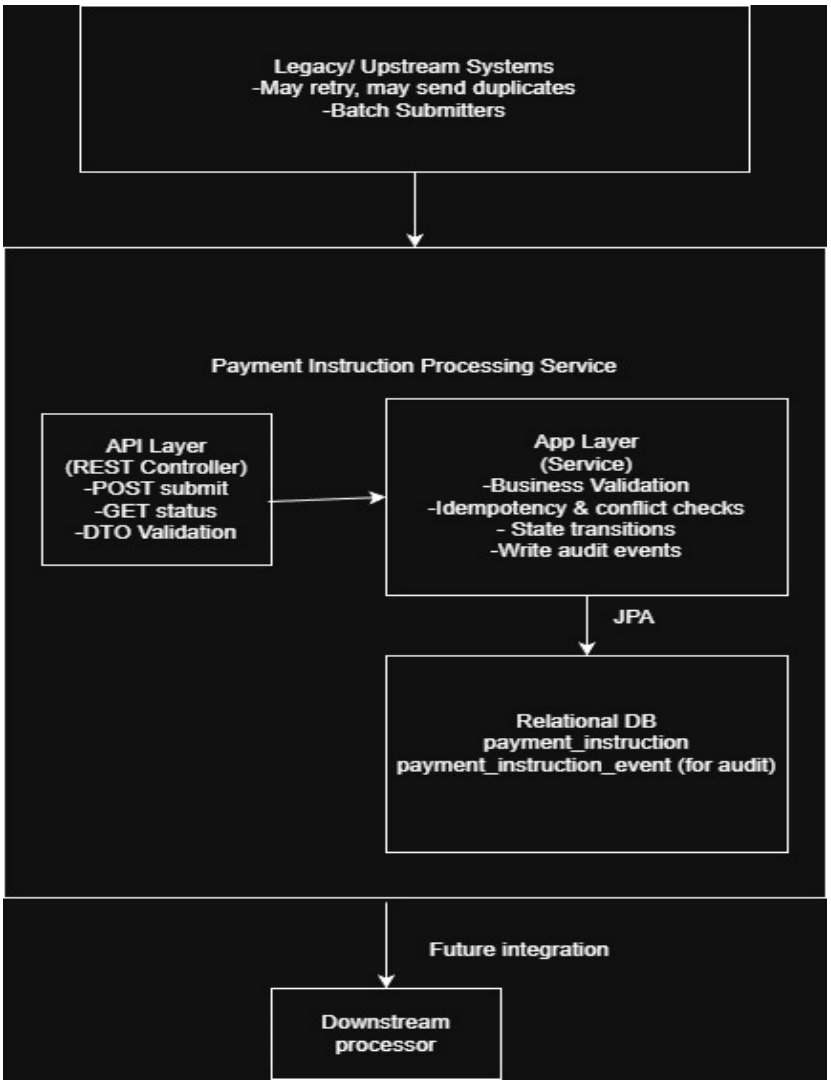
- What downstream settlement or processing systems must this service integrate with?
- Are there requirements for event publishing or message-based integration?

## Platform, infrastructure, or security teams

- What authentication and authorisation mechanisms are required (e.g. OAuth, JWT)?
- Are there compliance requirements for:
    - Encryption at rest
    - Sensitive data masking in logs?
- What data retention and archival policies must be followed?
- What environments will the service run in (on-premise, cloud, hybrid)?
- What monitoring and observability standards must be adhered to?

# Task 3

## Architecture Diagram




This system is a greenfield instruction intake service that must integrate with legacy upstream and downstream systems. It is designed as a durable record system for payment instructions. It accepts instructions, validates them, persists them, and exposes status query APIs.

It is intentionally not responsible for actual fund movement / settlement. That responsibility belongs to downstream processors. The separation ensures that instruction capture remains reliable even if downstream processing is slow or unavailable.

## Key Components and Responsibilities

## API Layer (REST)

Responsibilities:

- Accept POST /payment-instructions to submit instructions
- Accept GET /payment-instructions/{id} to retrieve current status
- Apply request level validation (required fields, formats)
- Return consistent error responses (4xx for client errors, 5xx for server faults)

Design intent:

- Keep controllers thin (mapping + validation + HTTP semantics)
- Delegate all business logic to the service layer

## Service Layer

Responsibilities:

- Apply business rules (e.g: payer != payee, execution date rules)
- Enforce idempotency using instructionId
- Handle duplicates safely:
    - Same instructionId + same payload > idempotent success
    - Same instructionId + different payload > conflict (protects data integrity)
- Persist instruction and write audit events in the same transaction
- Maintain lifecycle status (currently minimal, it is designed to expand with downstream updates)

Design intent:

- Correctness first: protect against corruption caused by legacy retries/duplicates
- Make behaviour deterministic so upstream systems can safely retry
-

## Relational Database (System of Record)

Responsibilities:

- Store the instruction record (payment_instruction)
- Store the append only audit trail (payment_instruction_event)
- Enforce uniqueness of instruction IDs (primary key)

- Support operational queries (by ID, status, time window)

Design intent:

- Treat DB as the single source of truth
- Allow reconciliation and investigations without relying on upstream logs

## Upstream Systems

Expected characteristics:

- May send duplicate requests due to retry logic or timeouts
- May be batch job and resubmit in bulk
- May have limited observability and inconsistent error handling

How the service supports this:

- Idempotent API design using a stable instruction ID
- Deterministic responses for retries
- Strict boundary validation so ‘bad data’ is rejected early

## Downstream Systems (Future integration)

Examples:

- Payment execution engine
- Clearing/settlement
- Ledger / accounting / reconciliation systems

Integration model (recommended):

- Asynchronous consumption of persisted instructions
- Downstream updates instruction status via:
    - a callback API (PATCH /payment-instructions/{id}/status), or
    - polling/reading from the database (less ideal but common in legacy), or
    - an event stream

Design intent:

- Instruction capture remains reliable even when downstream systems fail
- Downstream processing can be retried without resubmitting instructions

## Data Flow

### *Submit Instruction Flow (POST)*

1. Upstream sends instruction to POST /payment-instructions
2. API performs DTO validation (required fields, formats)
3. Service checks for existing instructionId:
    - If exists and payload matches > return existing record (idempotent)
    - If exists and payload differs > return 409 conflict
4. Service applies business rule validation
5. Valid instruction persisted to DB
6. Audit event recorded
7. Response returned (201 for new, 200 for idempotent retry)

### *Retrieve Status Flow (GET)*

1. Client calls GET /payment-instructions/{instructionId}
2. Service reads record from DB
3. If not found > 404
4. If found > return current status and timestamps

## Failure Scenarios and Recovery

### *Client Retries / Duplicate Requests*

- Safe retries are expected and supported
- Idempotency ensures duplicate submissions do not create multiple records

### *Service Restart / Crash*

- Service is stateless, no in-memory state required
- DB preserves all persisted instructions and audit events
- On restart, service resumes normally

### *DB Failure / Unavailable*

- Submission fails; instruction not persisted
- Caller can retry safely later (idempotency remains valid)

### *Downstream Failures (Future)*

- Instructions remain durable in DB even if downstream cannot execute
- Reconciliation jobs can identify “stuck” instructions by status/age window
- Audit trail helps determine last known transitions

# Task 4

This service is designed to ensure that payment instructions are:

- Correct (validated at the boundary)
- Non-duplicated (idempotent retries)
- Non-corruptible (conflicting duplicates rejected)
- Traceable (audit trail)
- Durable (DB as system-of-record)

These properties are critical in an enterprise environment where upstream callers may be legacy systems with imperfect retry and error-handling behaviour.

## Idempotency Strategy

### Idempotency Key

- instructionId is treated as the idempotency key.
- The database primary key enforces uniqueness.

### Duplicate Submission Behaviour

When a request arrives with an existing instructionId:

- Same payload > return existing record (200 OK)
  This supports safe retries when legacy systems time out and resubmit.
- Different payload → reject with 409 Conflict
  This prevents silent corruption (e.g: legacy system accidentally reusing an ID for a different instruction).

### Why this is safe

- Guarantees exactly one instruction record per instructionId.
- Makes client retry behaviour deterministic.
- Prevents double processing caused by retries.

# Validation & Integrity Controls

## Boundary Validation

The service applies:

- Request schema validation (required fields, formats)
- Business rule validation (e.g: payer != payee, supported currency, execution date constraints)

Invalid business rules return 400 Bad Request, and the instruction is not persisted.

## Rationale

- Prevents invalid data from polluting the system-of-record.
- Keeps reconciliation workflows focused on legitimate instructions.

# Auditability & Traceability

## Audit Trail

The system records an audit trail in payment_instruction_event, including:

- instructionId
- event type (e.g: created, conflict)
- timestamp
- details

This supports:

- Root cause investigations
- Operational troubleshooting (eg: why a request was rejected)
- Compliance audits

## Audit Event Examples

- Instruction created successfully → CREATED
- Conflicting duplicate submission → DUPLICATE_CONFLICT

With additional time / future scope, the same mechanism can record downstream status transitions (e.g: “EXECUTED”, “FAILED”).

# Reconciliation Approach

**5.1 What reconciliation means here**

## Queries that enable reconciliation

The model supports reconciliation through:

- Lookup by instructionId (authoritative record)
- Query by status + updatedAt window (find “stuck” or old instructions)
- Audit events history (explain anomalies and conflicts)

**5.3 Typical operational workflow**

1. Support receives a report: “Payment missing”
2. Support queries by instructionId
3. If found, check status + timestamps
4. If not found, upstream never persisted it (caller should retry)
5. If conflict occurred, audit event explains mismatch (“same ID, different payload”)

# Operability: Monitoring & Alerting

## Key Metrics

**Traffic & throughput**

- payment_instruction.submit.count
- payment_instruction.get.count

**Quality & integrity**

- payment_instruction.validation_failed.count (400)
- payment_instruction.conflict.count (409)
- duplicate rate = conflict + idempotent retries

**Reliability**

- error rate (5xx)
- latency for POST/GET
- DB connection pool saturation (if applicable)

## Logs (minimum useful fields)

Structured logs should include:

- instructionId
- sourceSystem
- outcome (CREATED, IDEMPOTENT_OK, CONFLICT, BAD_REQUEST)

This makes troubleshooting across multiple systems feasible.

## Test Coverage to Protect Integrity

The test suite validates the most important integrity behaviours:

- 201 on new valid instruction
- 200 on idempotent retry (same payload)
- 409 on conflict retry (different payload same ID)
- 400 on business rule violation
- 404 when fetching missing instruction
- 200 when fetching existing instruction

This ensures future changes do not break idempotency or validation semantics.

## Operational Support: Issue Detection, Investigation, and Resolution

### *Issue Detection*

Key detection signals include:

- Spike in 5xx errors > possible database or service failure
- Increase in 409 conflict rates > potential upstream ID reuse or data mismatch
- Increase in 400 validation failures > upstream sending malformed or invalid instructions
- Drop in the successful submission rate > upstream connectivity or integration issue
- Instructions stuck in same status for extended time > downstream processing issue
- Increased latency > infrastructure or database performance degradation

Alerts can be configured on thresholds such as:

- Error rate above baseline

- Conflict rate above expected retry level
- Unusual drop in throughput

## Investigation Process

When an issue is detected, support teams can investigate using the system-of-record and audit trail.

An example of an investigation workflow:

1. Identify affected instruction
    - Use instructionId from customer or upstream logs
2. Check instruction record
    - If not found > instruction never persisted > upstream must retry
    - If found > check status and timestamps
3. Inspect audit events
    - Determine what happened (created, rejected, conflict, etc)
    - Identify timing and sequence of events
4. Review logs
    - Correlate using instructionId and request correlation ID
    - Identify validation failure, conflict reason, or infrastructure error
5. Check system health metrics
    - Determine if issue is isolated or systemic (DB outage, spike in conflicts, etc.)

## Issue Resolution

Resolution depends on the type of issue:

Case 1 - Instruction Not Found

Cause:

- Upstream request failed before persistence

Resolution:

- Upstream retries submission safely (idempotent)

Case 2 - Conflict (Duplicate with Different Payload)

Cause:

- Upstream reused same instructionId incorrectly

Resolution:

- Support informs upstream to correct identifier usage
- No manual correction needed; system prevents corruption

Case 3 - Validation Failure

Cause:

- Invalid data submitted by upstream

Resolution:

- Upstream corrects payload and resubmits
- Support provides rejection reason from logs / error response

Case 4 — Service or Database Outage

Cause:

- Infrastructure failure

Resolution:

- Restore service / DB
- Upstream retries submissions (safe due to idempotency)

Case 5 - Downstream Processing Delay or Failure (Future)

Cause:

- Execution engine unavailable or failed

Resolution:

- Identify stuck instructions via status + time window
- Retry downstream processing or reconcile manually
- Audit trail confirms last known transition

## Future Enhancements (If Scope Expands)

In the minimal design, some recovery scenarios rely on upstream retries. In a production enterprise deployment, I would reduce upstream dependency by introducing asynchronous internal processing (queue/worker) and using the Outbox pattern to reliably publish processing events after persisting instructions. This enables internal retries, DLQ handling without requiring resubmission by legacy clients.

If downstream execution is added, the following improvements are natural extensions:

- Additional statuses: RECEIVED, PROCESSING, EXECUTED, FAILED
- Controlled state transition rules
- Async integration via queue/event stream
- Dead-letter handling + retry policies
- Automated reconciliation jobs and dashboards

## Trade-offs

The current implementation intentionally prioritises simplicity, correctness, and auditability over throughput and full lifecycle automation.

Key trade-offs include:

- Synchronous processing keeps the system simple but some recovery scenarios depend on upstream retries.
- Minimal lifecycle model is implemented to keep scope focused; downstream execution handling is designed but not fully implemented.
- Relational database as system-of-record ensures strong consistency but may limit horizontal scaling at very high throughput.
- No asynchronous queue included to avoid unnecessary complexity in this phase.
- H2 used for simplicity. a production deployment would use a durable database such as PostgreSQL or MySQL.

## Deployment & Observability

The service is cloud-agnostic and can be deployed using standard container approaches. In an AWS environment, a typical deployment could be:

- Compute: ECS (Fargate) or EC2 running the containerised service
- Logging: Application logs streamed to CloudWatch Logs with structured fields (instructionId, sourceSystem, outcome, correlationId)
- Metrics: CloudWatch metrics for request rate, latency, 4xx/5xx, validation failures, and conflict rate
- Alerting: CloudWatch alarms on elevated error rate, abnormal spikes in conflicts/validation failures, or sustained latency

These observability controls support rapid detection and investigation by operations teams.