Transport Request Service API Design

System Design Team

September 24, 2025

Contents

1 Introduction

The Transport Request Service (TRS) is a core microservice in the logistics platform, responsible for managing transport requests from creation to fulfillment. It ensures smooth interaction between clients, transport providers, and supporting services, enabling efficient and reliable logistics operations in an event-driven architecture.

1.1 Purpose of the Document

This document serves as a comprehensive guide to the design and implementation of the Transport Request Service. It provides detailed specifications for API endpoints, data models, integrations, error handling strategies, and non-functional requirements. The primary objectives of this document are:

- To define clear API contracts for synchronous and asynchronous communication.
- To specify data models, validation rules, and business logic.
- To describe integrations with external systems and services.
- To establish guidelines for error handling, security, and performance.
- To ensure extensibility, scalability, and maintainability of the service.

1.2 Scope

The Transport Request Service is responsible for the following key functionalities:

- **Request Creation:** Enabling clients to create transport requests with details such as pickup and delivery locations, cargo specifications, and preferred transport type.
- **Request Validation:** Validating request inputs, such as geospatial constraints, cargo limits, and user permissions.
- **Provider Matching:** Facilitating the assignment of transport providers to requests, in collaboration with the Matching Service.
- **State Management:** Tracking the lifecycle of a transport request, from creation to cancellation or fulfillment.
- **Asynchronous Communication:** Publishing and subscribing to Kafka events to integrate with other services in the logistics platform.

1.3 Target Audience

This document is intended for the following stakeholders:

- **Backend Engineers:** To implement and maintain the Transport Request Service.
- **Frontend Developers:** To integrate with the synchronous API endpoints.
- **System Architects:** To ensure compliance with the platform's architectural principles.

- **QA Engineers:** To develop and execute test cases based on the API specifications.
- **Business Stakeholders:** To understand the scope and functionality of the service.

2 API Contracts

This section provides a detailed overview of the Transport Request Service's API, emphasizing clear design, modularity, and ease of use. The TRS employs a hybrid communication model, with synchronous REST APIs for direct interactions and asynchronous Kafka events for state changes and event propagation.

2.1 Synchronous API Endpoints

The Transport Request Service provides a RESTful API for managing transport requests, including creation, retrieval, status updates, and cancellation. These endpoints are designed to facilitate the seamless lifecycle of transport requests while ensuring scalability, reliability, and backward compatibility. Below is the detailed API contract.

Table 1: Transport Request Service REST API Endpoints

Endpoint	**HTTP Method**	**Description**	**Request/Response Format**
`/v1/transport/requests`	POST	Create a new transport request	Request: JSON payload with pickup/drop-off locations, item details, and time window. Response: Request ID, status, and estimated cost.
`/v1/transport/requests/{id}`	GET	Retrieve details of a transport request	Request: Path parameter with request ID. Response: JSON payload containing full request details.
`/v1/transport/requests/{id}/status`	PATCH	Update the status of a transport request	Request: JSON payload with updated status (e.g., ”in progress”, ”completed”). Response: Confirmation of the update.
`/v1/transport/requests/{id}`	DELETE	Cancel a transport request	Request: Path parameter with request ID. Response: Confirmation of cancellation.

2.1.1 Endpoint Specifications

`/v1/transport/requests` - Create a Transport Request **Description:** This endpoint allows users to create a new transport request by providing all necessary details, including pickup and drop-off locations, item details, and time preferences. It calculates an estimated cost for the request.

Request Body

Listing 1: Request Body Example

```
{
    "pickupLocation": {
        "address": "123 Main St, Cityville",
        "latitude": 37.7749,
        "longitude": -122.4194
    },
    "dropoffLocation": {
        "address": "456 Elm St, Townsville",
        "latitude": 37.8044,
        "longitude": -122.2712
    },
    "itemDetails": {
        "description": "5 boxes, fragile",
        "weight": 25.0,
        "dimensions": "50x40x30 cm"
    },
    "preferredTimeWindow": {
        "start": "2024-12-06T09:00:00",
        "end": "2024-12-06T12:00:00"
    }
}
```

Response Body

Listing 2: Response Body Example

```
{
    "requestId": "98765",
    "status": "CREATED",
    "estimatedCost": 150.00
}
```

—

`/v1/transport/requests/{id}` - Retrieve Transport Request Details **Description:** Fetches detailed information about a specific transport request using its unique identifier. This includes locations, item details, time window, and the current status of the request.

Response Body

Listing 3: Response Body Example

```
{
    "requestId": "98765",
    "pickupLocation": {
        "address": "123 Main St, Cityville",
        "latitude": 37.7749,
        "longitude": -122.4194
    },
    "dropoffLocation": {
        "address": "456 Elm St, Townsville",
        "latitude": 37.8044,
        "longitude": -122.2712
    },
    "itemDetails": {
        "description": "5 boxes, fragile",
        "weight": 25.0,
        "dimensions": "50x40x30 cm"
    },
    "status": "IN_PROGRESS",
    "estimatedCost": 150.00,
    "timeWindow": {
        "start": "2024-12-06T09:00:00",
        "end": "2024-12-06T12:00:00"
    }
}
```

—

`/v1/transport/requests/{id}/status` - Update Request Status **Description:** Allows updates to the current status of a transport request. Common statuses include ”IN_PROGRESS”, ”COMPLETED”, ”CANCELED”, etc. This helps in tracking the request lifecycle.

Request Body

Listing 4: Request Body Example

```
{
    "status": "IN_PROGRESS"
}
```

Response Body

Listing 5: Response Body Example

```
{
    "message": "Status updated successfully."
}
```

—

`/v1/transport/requests/{id}` - Cancel a Transport Request **Description:** Cancels an existing transport request by its unique identifier. This operation updates the request status to "CANCELED".

Response Body

Listing 6: Response Body Example

```
{
    "message": "Transport request canceled successfully."
}
```

2.2 Asynchronous Event Contracts

The Transport Request Service uses Kafka for asynchronous communication to propagate state changes, notify other services, and ensure decoupled workflows. Below are the event definitions categorized by their respective topics, with detailed payload structures for each event type.

Table 2: Transport Request Service Kafka Topics

Topic Name	**Description**
`TransportEvents.Request.Created`	Published when a new transport request is created, notifying downstream services (e.g., Provider Matching, Route Optimization).
`TransportEvents.Request.Updated`	Published when an existing transport request is updated (e.g., status change, location updates).
`TransportEvents.Request.Cancelled`	Published when a transport request is canceled.
`TransportEvents.Capacity.Verified`	Published when capacity checks for the requested transport are completed.
`TransportEvents.Route.Optimized`	Published when the optimal route for the transport request is determined.

2.2.1 Event Specifications

`TransportEvents.Request.Created` **Description:** This event is published whenever a new transport request is successfully created. It informs downstream services about the new request so they can initiate necessary workflows, such as provider matching or route planning.

Event Payload

Listing 7: Event Payload Example for Request Created

```json
{
    "eventType": "TransportEvents.Request.Created",
    "timestamp": "2024-12-06T08:00:00Z",
    "requestId": "98765",
    "pickupLocation": {
        "address": "123 Main St, Cityville",
        "latitude": 37.7749,
        "longitude": -122.4194
    },
    "dropoffLocation": {
        "address": "456 Elm St, Townsville",
        "latitude": 37.8044,
        "longitude": -122.2712
    },
    "itemDetails": {
        "description": "5 boxes, fragile",
        "weight": 25.0,
        "dimensions": "50x40x30 cm"
    },
    "preferredTimeWindow": {
        "start": "2024-12-06T09:00:00",
        "end": "2024-12-06T12:00:00"
    },
    "estimatedCost": 150.00,
    "status": "CREATED"
}
```

—

`TransportEvents.Request.Updated` **Description:** This event is published when a transport request is updated, such as changes in the status, time window, or location details.

Event Payload

Listing 8: Event Payload Example for Request Updated

```
{
    "eventType": "TransportEvents.Request.Updated",
    "timestamp": "2024-12-06T10:00:00Z",
    "requestId": "98765",
    "updatedFields": {
        "status": "IN_PROGRESS",
        "timeWindow": {
            "start": "2024-12-06T10:00:00",
            "end": "2024-12-06T13:00:00"
        }
    }
}
```

—

`TransportEvents.Request.Cancelled` **Description:** This event is published when a transport request is canceled. It notifies downstream services to stop processing workflows related to the request.

Event Payload

Listing 9: Event Payload Example for Request Cancelled

```
{
    "eventType": "TransportEvents.Request.Cancelled",
    "timestamp": "2024-12-06T11:00:00Z",
    "requestId": "98765",
    "reason": "User requested cancellation."
}
```

—

`TransportEvents.Capacity.Verified` **Description:** This event is published when capacity checks (e.g., weight, dimensions) for the requested transport are successfully completed.

Event Payload

Listing 10: Event Payload Example for Capacity Verified

```
{
    "eventType": "TransportEvents.Capacity.Verified",
    "timestamp": "2024-12-06T12:00:00Z",
    "requestId": "98765",
    "capacityStatus": "VERIFIED",
    "details": {
        "availableCapacity": {
            "weight": 100.0,
            "dimensions": "100x80x60 cm"
        }
    }
}
```

—

`TransportEvents.Route.Optimized` **Description:** This event is published when an optimal route for the transport request is determined, providing information about the planned route, estimated time of arrival (ETA), and any alternative routes.

Event Payload

Listing 11: Event Payload Example for Route Optimized

```
{
    "eventType": "TransportEvents.Route.Optimized",
    "timestamp": "2024-12-06T12:30:00Z",
    "requestId": "98765",
    "optimizedRoute": {
        "eta": "2024-12-06T13:30:00Z",
        "distance": "15 km",
        "duration": "30 minutes"
    },
    "alternativeRoutes": [
        {
            "eta": "2024-12-06T13:45:00Z",
            "distance": "17 km",
            "duration": "35 minutes"
        }
    ]
}
```

3 Data Models

The Transport Request Service (TRS) utilizes several key data models to represent the transport requests and their associated details. These models encapsulate the essential attributes and relationships needed for the effective functioning of the service.

3.1 Transport Request Model

The Transport Request model represents an individual transport request initiated by a user. It includes essential details about the request, such as pickup and drop-off locations, item details, and timestamps.

Listing 12: Transport Request Model

```
{
    "requestId": "98765",
    "userId": "12345",
    "pickupLocation": {
        "address": "123 Main St, City, Country",
        "coordinates": {
            "latitude": 40.7128,
            "longitude": -74.0060
        }
    },
    "dropoffLocation": {
        "address": "456 Elm St, City, Country",
        "coordinates": {
            "latitude": 40.7328,
            "longitude": -74.0160
        }
    },
    "itemDetails": {
        "type": "Electronics",
        "weight": 1.5,
        "dimensions": {
            "length": 10,
            "width": 5,
            "height": 2
        }
    },
    "timeWindow": {
        "start": "2024-12-01T10:00:00Z",
        "end": "2024-12-01T12:00:00Z"
    },
    "status": "CREATED",
    "estimatedCost": 150.00,
    "createdAt": "2024-12-01T09:00:00Z",
    "updatedAt": "2024-12-01T09:00:00Z"
}
```

3.2 User Model

The User model represents the user who initiates the transport request. It contains basic user information and links to their associated transport requests.

Listing 13: User Model

```
{
    "userId": "12345",
    "name": "John Doe",
    "email": "johndoe@example.com",
    "phone": "+1234567890",
    "registeredAt": "2024-01-01T09:00:00Z",
    "requests": [
        "98765",
        "98766"
    ]
}
```

3.3 Item Details Model

The Item Details model provides specifics about the items being transported, including type, weight, and dimensions. This model helps in managing transport capacity and regulatory compliance.

Listing 14: Item Details Model

```
{
    "itemId": "item-001",
    "type": "Electronics",
    "weight": 1.5,
    "dimensions": {
        "length": 10,
        "width": 5,
        "height": 2
    }
}
```

3.4 Location Model

The Location model encapsulates the address and geographic coordinates for pickup and drop-off locations. This model is crucial for routing and geospatial analysis.

Listing 15: Location Model

```
{
    "address": "123 Main St, City, Country",
    "coordinates": {
        "latitude": 40.7128,
        "longitude": -74.0060
    }
}
```

3.5 Status Model

The Status model defines the possible states a transport request can be in throughout its lifecycle. This model ensures clarity in workflow management.

Listing 16: Status Model

```
{
    "status": [
        "CREATED",
        "ASSIGNED",
        "IN_TRANSIT",
        "COMPLETED",
        "CANCELLED"
    ]
}
```

3.6 Relationships

The relationships between these models are as follows: - A **User** can have multiple **Transport Requests**. - Each **Transport Request** contains details of one **Item** (or multiple items) and links to one **Pickup Location** and one **Drop-off Location**. - The **Status** of a transport request changes as it progresses through its lifecycle, reflecting updates to its state.

This structure enables the TRS to manage transport requests efficiently and maintain an accurate representation of user and item details throughout the request's lifecycle.

4. ValidationResult

- **Description**: Stores the results of validation checks for the transport request.
- **Attributes**:

Listing 17: ValidationResult Entity Schema

```
{
    "requestId": "string",                // ID of the transport
        request
    "geospatialValidation": "boolean",   // Whether geospatial
       constraints are satisfied
    "cargoValidation": "boolean",        // Whether cargo
       constraints are satisfied
    "capacityValidation": "boolean",     // Whether capacity
       constraints are satisfied
    "errorMessages": ["string"]          // List of validation
       error messages, if any
}
```

3.7 Validation Constraints

This section defines validation rules applied to the fields of the core entities.

1. **`TransportRequest`**

- **requestId**: Must be a unique alphanumeric string.
- **userId**: Must correspond to an existing, verified user in the User Management Service.
- **pickupLocation / deliveryLocation**:
  - Latitude and Longitude: Must be valid geographic coordinates (latitude between -90 and 90, longitude between -180 and 180).
  - Address: Must be a non-empty string.
- **cargoDetails**:
  - **weight**:
    * Minimum: 0.1 kg.
    * Maximum: 10,000 kg (or configurable system limit based on vehicle type).
  - **volume**:
    * Minimum: 0.01 m^3.
    * Maximum: 50 m^3 (or configurable system limit based on vehicle type).
  - **description**: Must be a string with a maximum of 250 characters.
- **preferredTransportType**: Must be one of the supported vehicle types (e.g., ”Truck”, ”Van”).
- **status**: Must be a valid enum value (”Pending”, ”InProgress”, ”Completed”, ”Cancelled”).

2. **`Location`**

- Latitude: Must be a float between -90 and 90.
- Longitude: Must be a float between -180 and 180.
- Address: Must be a valid string.

3. **`ValidationResult`**

- **geospatialValidation**: Must be `true` if both pickup and delivery locations are within serviceable regions.
- **cargoValidation**: Must be `true` if cargo weight and volume are within supported limits.
- **capacityValidation**: Must be `true` if transport capacity is available for the requested type.
- **errorMessages**: Should provide human-readable error descriptions if any validations fail.

Additional Notes

- **Geospatial Constraints**: The system should validate that the pickup and delivery locations fall within predefined serviceable regions. Locations outside these boundaries should trigger an error.
- **Weight and Volume Limits**: These limits are configurable and based on the vehicle type selected.
- **Enum Validation**: Fields like `status` and `preferredTransportType` should strictly adhere to predefined enum values to ensure consistency.

3.8 Validation Constraints

This section outlines the validation rules for each field within transport requests, ensuring that all data is correctly formatted and adheres to specified limits.

- **Pickup and Dropoff Locations:**
  - **Address:** Must be a non-empty string.
  - **Latitude:** Must be a float within the range of -90 to 90.
  - **Longitude:** Must be a float within the range of -180 to 180.
- **Item Details:**
  - **Description:** Must be a non-empty string with a maximum length of 255 characters.
  - **Weight:** Must be a positive float, with a maximum weight limit of 500.0 kg.
  - **Dimensions:** Must be a string formatted as ”length x width x height” (e.g., ”50x40x30 cm”), with each dimension being a positive float.
- **Preferred Time Window:**
  - **Start Time:** Must be a valid datetime in ISO 8601 format.
  - **End Time:** Must be a valid datetime in ISO 8601 format and must be later than the start time.
- **Estimated Cost:**
  - **Cost:** Must be a positive float, with a maximum allowable cost of 1000.00.
- **Status:**
  - Must be one of the following values: `CREATED`, `IN_PROGRESS`, `COMPLETED`, `CANCELLED`.

4 Integrations

The Transport Request Service (TRS) integrates with various external systems and services through event-driven communication. This section outlines the topics the service subscribes to, as well as the events it publishes to facilitate seamless interactions across the system.

4.1 Subscribed Topics

The TRS subscribes to several external topics to receive relevant updates that affect transport requests. These events allow TRS to react to changes in related services and maintain an accurate state of transport requests.

Table 3: Subscribed Topics for Transport Request Service

Topic Name	**Description**
`MatchingEvents.ProviderAssigned`	This event is received when a provider is assigned to a transport request. It allows TRS to update the request status and notify users of the assigned provider details.
`GeospatialEvents.Location.Updated`	This event is subscribed to for real-time updates on location data, enabling TRS to optimize routing and provide accurate ETAs based on the latest geospatial information.
`CapacityEvents.Capacity.Updated`	This event provides updates on capacity changes for providers, allowing TRS to adjust requests accordingly and ensure the feasibility of transport operations.

4.2 Published Topics

The TRS publishes events to inform other services and systems about significant changes and actions related to transport requests. These events are crucial for maintaining a cohesive and responsive ecosystem.

Table 4: Published Topics by Transport Request Service

Topic Name	Description
`TransportEvents.Request.Created`	Published when a new transport request is created. This event includes all relevant details of the request and is consumed by services involved in provider matching and route optimization.
`TransportEvents.Request.Updated`	This event is published when there are updates to an existing transport request, such as changes in status or item details. Other services can use this information to react appropriately to the request's current state.
`TransportEvents.Request.Cancelled`	This event is published when a transport request is canceled, allowing downstream services to cease any processing related to that request.
`TransportEvents.Capacity.Verified`	This event indicates that capacity checks for a transport request have been completed successfully. Other services can utilize this event to manage their workflows accordingly.
`TransportEvents.Route.Optimized`	Published when an optimal route for a transport request has been determined, providing crucial information for delivery planning and coordination.

5 Error Scenario Management

Effective error scenario management is critical for maintaining the reliability and user experience of the Transport Request Service (TRS). This section outlines potential error scenarios that can occur within the service, their handling mechanisms, and strategies for recovery and prevention.

5.1 Common Error Scenarios

- **Invalid Input Data**
  - **Description:** Users may submit invalid data for transport requests (e.g., incorrect latitude/longitude, negative weights).
  - **Handling:** The service will validate inputs before processing requests. If invalid data is detected, the service will respond with a `400 Bad Request` status code, along with a detailed error message specifying the nature of the validation failure.
  - **Example:**

Listing 18: Error Response Example

```
{
    "error": {
        "code": "INVALID_INPUT",
        "message": "Latitude must be between -90 and 90.",
        "field": "pickupLocation.latitude"
    }
}
```

- **Transport Provider Not Available**
  - **Description:** When attempting to assign a transport provider, no available providers match the request criteria.
  - **Handling:** The service will return a `404 Not Found` status code with a message indicating that no suitable providers are available at the moment. This will trigger a retry mechanism based on a backoff strategy.
  - **Example:**

Listing 19: Error Response Example

```
{
    "error": {
        "code": "PROVIDER_NOT_FOUND",
        "message": "No transport providers are currently available
           for the requested route."
    }
}
```

- **Service Unavailability**
  - **Description:** The Transport Request Service may experience downtime due to maintenance or unexpected failures.
  - **Handling:** The service will respond with a `503 Service Unavailable` status code, indicating that the service is temporarily unavailable. Clients should implement exponential backoff retries to avoid overwhelming the service when it comes back online.
  - **Example:**

Listing 20: Error Response Example

```
{
    "error": {
        "code": "SERVICE_UNAVAILABLE",
        "message": "The Transport Request Service is currently
           unavailable. Please try again later."
    }
}
```

- **Database Connection Issues**
  - **Description:** Issues connecting to the database can arise due to network problems, configuration errors, or the database being down.
  - **Handling:** The service will log the error and respond with a `500 Internal Server Error` status code. Admins should be alerted for immediate investigation.
  - **Example:**

Listing 21: Error Response Example

```
{
    "error": {
        "code": "DB_CONNECTION_ERROR",
        "message": "Unable to connect to the database. Please
           contact support."
    }
}
```

- **Kafka Message Processing Failure**
  - **Description:** Failure in processing Kafka messages, which may lead to loss of events or delayed processing.
  - **Handling:** Messages will be sent to a Dead Letter Queue (DLQ) for further analysis. The system should implement alerting for monitoring message processing failures.

5.2 Error Handling Strategies

To ensure resilience and a better user experience, the following error handling strategies should be implemented:

- **Input Validation:** Comprehensive validation checks at the API level to catch errors before processing.
- **Graceful Degradation:** If certain features are unavailable, the service should still provide essential functionalities to users where possible.

- **Retry Mechanisms:** Implementing retries with exponential backoff for transient errors, especially for service unavailability and provider assignment issues.
- **Monitoring and Alerts:** Setting up monitoring on service health, database connections, and message processing to proactively identify and address issues.
- **User Notifications:** Informing users about errors in a user-friendly manner and providing guidance on the next steps.

5.3 Recovery and Prevention

To further enhance the robustness of the TRS, the following practices should be adopted:

- **Automated Testing:** Regularly test the service under various failure scenarios to ensure that error handling works as intended.
- **Load Testing:** Simulate high-load conditions to identify potential bottlenecks and ensure service stability during peak usage.
- **Logging and Auditing:** Maintain comprehensive logs of errors for post-mortem analysis, which can inform future improvements.
- **Documentation Updates:** Regularly update the API documentation to reflect error scenarios and handling procedures to assist developers in understanding the service behavior.

6 Security Requirements

The Transport Request Service (TRS) must adhere to stringent security requirements to protect sensitive data, ensure authorized access, and mitigate threats. This section outlines the key security measures implemented in the service.

6.1 Authentication and Authorization

- **Authentication:**
  - TRS uses JSON Web Tokens (JWT) for authentication.
  - Tokens are signed using the RS256 algorithm to ensure integrity and prevent tampering.
  - Expired or invalid tokens are rejected with a `401 Unauthorized` status code.
- **Authorization:**
  - Role-Based Access Control (RBAC) ensures users and services have access only to the resources necessary for their roles.
  - Permissions are defined at the API endpoint level, and unauthorized access attempts result in a `403 Forbidden` status code.
  - System roles include:
    - **User:** Can create, view, and cancel transport requests.

    * **Transport Provider:** Can update request status and view assigned requests.
    * **Admin:** Has full access to all resources and can manage system configurations.

- **Token Validation:**
  - Tokens are validated for:
    * Signature integrity.
    * Expiry (`exp` claim).
    * Audience (`aud` claim).
    * Issuer (`iss` claim).
  - A token revocation list (TRL) is maintained to invalidate tokens in case of security breaches or user logout.

6.2 Data Security

- **Data in Transit:**
  - All communication between clients and the TRS is encrypted using TLS 1.3.
  - Secure HTTPS endpoints are enforced with strong cipher suites.
  - Mutual TLS (mTLS) is used for service-to-service communication to prevent unauthorized access to internal APIs.
- **Data at Rest:**
  - Sensitive data, including user details and transport request information, is encrypted using AES-256.
  - Database encryption keys are managed using a secure Key Management Service (KMS).
  - Access to storage systems is restricted using strict IAM policies.
- **Personally Identifiable Information (PII):**
  - PII is anonymized or pseudonymized where applicable to minimize risks in case of a breach.
  - Data retention policies ensure that PII is only stored for the duration necessary for business purposes.

6.3 Threat Mitigation

The TRS implements a range of measures to mitigate security threats, including but not limited to:

- **Rate Limiting:**
  - API rate limits are enforced to prevent abuse, such as brute force attacks or denial-of-service attempts.

  - Clients exceeding rate limits are temporarily blocked and receive a `429 Too Many Requests` status code.

- **Input Validation:**
  - All user inputs are validated against strict schemas to prevent injection attacks (e.g., SQL injection, XSS).
  - Invalid inputs are rejected with descriptive error messages and appropriate status codes.

- **Web Application Firewall (WAF):**
  - A WAF is used to monitor and filter incoming traffic for known attack patterns, such as injection attempts and DDoS attacks.
  - Suspicious activity is logged and reported for further analysis.

- **Audit Logs:**
  - Detailed audit logs are maintained for all critical actions, such as transport request creation, status updates, and authentication attempts.
  - Logs are stored in a secure, tamper-proof system to ensure traceability and compliance.

- **Secure Coding Practices:**
  - The TRS codebase follows OWASP Secure Coding Guidelines to minimize vulnerabilities.
  - Regular static and dynamic application security testing (SAST/DAST) is conducted to identify and fix vulnerabilities.

- **Incident Response:**
  - A well-documented incident response plan ensures rapid containment and resolution of security breaches.
  - Alerts and automated responses are triggered for unusual activity, such as repeated failed login attempts or suspicious API usage.

6.4 Compliance and Certification

To ensure compliance with industry standards and regulations, the TRS adheres to the following:

- GDPR/CCPA for handling PII and user consent.
- ISO 27001 for information security management.
- PCI-DSS for secure payment processing (if applicable).

7 Non-Functional Requirements Specification

The Transport Request Service (TRS) must meet stringent non-functional requirements to ensure reliability, scalability, and observability. This section outlines the key non-functional requirements, including performance benchmarks, resilience mechanisms, and monitoring capabilities.

7.1 Performance

Performance requirements are critical to providing a seamless user experience and maintaining system efficiency under varying loads.

- **Latency:**
  - API endpoints must respond within **200 milliseconds (ms)** for 95% of requests under normal load conditions.
  - For heavy computational operations (e.g., cost estimations), the response time should not exceed **500 ms**.
- **Throughput:**
  - The service must handle a minimum of **1,000 requests per second (RPS)** under peak conditions, with the ability to scale to **5,000 RPS** as demand grows.
- **Service Level Agreement (SLA):**
  - The system must maintain **99.9% uptime** monthly, allowing no more than **43 minutes of downtime per month**.
  - Planned maintenance should be communicated at least **48 hours in advance**.

7.2 Resilience and Fault Tolerance

The TRS must remain resilient and maintain availability during failures or adverse conditions.

- **Retry Mechanisms:**
  - Implement exponential backoff with jitter for retrying failed requests to downstream services.
  - Retries should be capped at **three attempts** to avoid overwhelming dependencies.
- **Circuit Breakers:**
  - Use circuit breakers to prevent cascading failures when a dependent service is unresponsive.
  - Circuit breakers should move to an **open state** after **three consecutive failures**, rejecting further calls for a **cool-off period of 30 seconds**.
- **Eventual Consistency:**

  - Employ eventual consistency for asynchronous operations (e.g., status updates), ensuring data synchronization across components without blocking workflows.
  - Use message deduplication to avoid processing duplicate events caused by retries.

- **High Availability:**
  - Deploy the TRS in a multi-region setup to ensure availability even during regional failures.
  - Use load balancers to distribute traffic and detect unhealthy instances for immediate rerouting.

7.3 Monitoring and Observability

Comprehensive monitoring and observability are essential to detect issues, understand system behavior, and optimize performance.

- **Metrics to Track:**
  - **Request Metrics:**
    - Request latency (P50, P90, P95, P99)
    - Throughput (requests per second)
    - API error rates (4xx and 5xx errors)
  - **System Metrics:**
    - CPU and memory utilization
    - Disk I/O and network usage
  - **Business Metrics:**
    - Number of transport requests created, updated, and canceled.
    - Average time to fulfill a transport request.
- **Distributed Tracing:**
  - Implement distributed tracing to track requests as they traverse multiple services, identifying bottlenecks or failures.
  - Use trace correlation IDs to connect logs and traces for the same request.
- **Tools and Technologies:**
  - Use tools like **Prometheus** and **Grafana** for metrics collection and visualization.
  - Employ **Jaeger** or **OpenTelemetry** for distributed tracing.
  - Integrate a centralized logging system such as **ELK Stack** (Elasticsearch, Logstash, Kibana) or **Datadog** for log aggregation and analysis.
- **Alerting and Incident Management:**
  - Define alerts for critical metrics (e.g., high error rates, latency spikes).

  - Integrate alerting with incident management tools like **PagerDuty** or **OpsGenie**.
  - Set up runbooks to guide the resolution of common issues.

8 Implementation Guidelines

- Use OpenAPI Specification for documenting the API.
- Implement idempotency for state-changing operations.
- Validate inputs on both client and server sides.

9 Conclusion

Summarize the API design, integrations, and operational requirements. Highlight next steps for implementation or iteration.