Enterprise Transportation Management System (TMS): Architectural Analysis and Engineering Blueprint

1. Executive Introduction: The Paradigm Shift in Logistics Engineering

The modern supply chain ecosystem is witnessing a fundamental structural transformation, migrating from static, record-keeping systems to dynamic, event-driven architectures capable of real-time decision-making. The traditional Transportation Management System (TMS) — historically a monolithic repository for orders and invoices — is evolving into a "Control Tower," a centralized digital twin of the physical supply chain. This report presents a rigorous technical analysis and implementation blueprint for an enterprise-grade TMS utilizing a hybrid technology stack: Django for robust transactional logic, React (via Vite) for high-performance reactive interfaces, and Traccar for scalable telemetry ingestion.

The primary engineering challenge addressed in this analysis is the reconciliation of two disparate data velocities: the high-frequency, ephemeral stream of geospatial telemetry generated by IoT assets, and the low-frequency, high-integrity transactional data required for financial settlement and freight planning. Conventional architectures often fail at enterprise scale because they attempt to force high-velocity telemetry into relational structures designed for transactional integrity, leading to database lock contention and latency.

This blueprint proposes a decoupled "Hexagonal" architecture. It leverages Traccar not as a monolithic application, but as a specialized protocol gateway, offloading persistence to a stream-processing pipeline underpinned by Apache Kafka and Redis. This design ensures that the ingestion layer achieves horizontal scalability independent of the business logic layer. Furthermore, by integrating Google OR-Tools for algorithmic optimization and TimescaleDB for time-series persistence, the system transitions from a passive tracking tool into a prescriptive planning engine. The analysis that follows details the engineering decisions, data structures, and infrastructure patterns required to deploy this system for fleets exceeding

50,000 assets, ensuring strict multi-tenant isolation and millisecond-level responsiveness.

2. Architectural Philosophy and System Design

2.1 The Hexagonal Control Tower Pattern

To satisfy the requirements of high availability and modularity, the system architecture adopts the Ports and Adapters (Hexagonal) pattern. This ensures that the core business logic—encompassed within the Django domain—remains isolated from external interfaces such as GPS hardware, ERP integrations, and mobile applications. This separation allows the application to remain agnostic to the specific communication protocols used by tracking devices, treating telemetry as just one of many input ports.

The architecture is stratified into four distinct planes, each optimized for specific data characteristics:

1. **Ingestion Plane (The Telemetry Gateway):** This layer is responsible for maintaining persistent TCP/UDP connections with hardware trackers. It handles the raw byte-stream manipulation, handshake protocols, and keep-alive heartbeats. By utilizing Traccar in a stateless configuration, this plane can scale horizontally across multiple availability zones.
2. **Data Streaming Plane (The Nervous System):** Acting as the asynchronous buffer between ingestion and processing, this plane utilizes Apache Kafka. It decouples the write-heavy telemetry load from the read-heavy analytical and transactional workloads, preventing back-pressure from the database from impacting device connectivity.[1]
3. **Business Logic Plane (The Transactional Core):** Built on Django, this layer manages the state of the business entities—Orders, Shipments, Loads, and Drivers. It consumes processed events from the streaming plane to trigger business rules, such as geofence entry alerts or automatic status updates.
4. **Presentation Plane (The Reactive Surface):** A Single Page Application (SPA) built with React and Vite provides the operator interface. It consumes REST APIs for CRUD operations and connects to a dedicated WebSocket gateway for real-time state synchronization, utilizing WebGL for high-density map rendering.

2.2 Microservices vs. Macro-services Strategy

While microservices offer granular scalability, they introduce significant operational complexity regarding distributed transactions and data consistency. For a TMS, where the relationship between an Order, a Load, and a Driver is tightly coupled, a distributed transaction spanning three services creates unnecessary overhead.

Therefore, this blueprint advocates for a **Macro-service** architecture. The "Core TMS" runs as a modular monolith within Django, ensuring referential integrity and simplifying complex queries. However, distinct operational domains with different scaling characteristics are extracted as separate services:

- **Telemetry Service:** (Traccar) Scales on CPU/Network I/O.
- **Optimization Worker:** (Celery + OR-Tools) Scales on CPU/Memory for algorithmic solving.
- **Notification Service:** Scales on I/O for email/SMS/Push delivery.

This hybrid approach retains the development velocity of a monolith while isolating the specific components that are prone to bottlenecks.

2.3 Event-Driven Data Flow

The lifecycle of a data point within this architecture illustrates the decoupled nature of the design:

1. **Transmission:** A GPS device transmits a binary packet via TCP to the ingestion endpoint.
2. **Decoding:** The Traccar gateway decodes the binary payload into a standardized JSON object containing coordinates, timestamp, and attributes (ignition, fuel level).
3. **Buffering:** Instead of writing to a database, Traccar publishes the JSON object to a Kafka topic named telemetry_raw.[3]
4. **Stream Processing:** A stream processor (e.g., a Faust agent or Python consumer) subscribes to the topic. It performs a "split" operation:
   - **Hot Path:** The latest position is pushed to a Redis Key-Value store with a TTL (Time-to-Live), serving as the "Current State" cache for the frontend API.[4]
   - **Cold Path:** The position is batched and written to the TimescaleDB hypertable for historical archival.[5]
5. **Event Detection:** The processor checks the position against active geofences active in the Redis cache. If a boundary is crossed, an Event message is published to the business_events topic, which the Django backend consumes to update the Shipment

status.

3. The Ingestion Engine: Traccar Optimization and Configuration

Traccar is selected for the ingestion plane due to its support for over 1,500 protocols and its Netty-based asynchronous I/O core. However, the default configuration of Traccar is designed for standalone operation, which is unsuitable for enterprise scale.

3.1 Netty Pipeline and Protocol Decoding

At the core of Traccar is the Netty framework, which handles the non-blocking I/O. The pipeline consists of a FrameDecoder, which splits the incoming TCP stream into discrete message frames, and a ProtocolDecoder, which parses the frame into a semantic Position object.[6]

In an enterprise environment, the FrameDecoder is critical. Mobile networks often fragment packets or concatenate multiple messages into a single TCP segment. The decoding layer must be robust enough to handle these anomalies without dropping data. For custom hardware, implementing a specific ProtocolDecoder in Java requires extending the BaseProtocolDecoder class.

Optimization for High Concurrency:
To support 50,000+ devices, the Linux kernel and JVM parameters must be tuned. The standard file descriptor limit is often too low for the number of persistent TCP connections required.

- **File Descriptors:** Increase the limit using ulimit -n 100000 to prevent Too many open files errors.[8]
- **Heap Memory:** While Traccar is efficient, the sheer volume of object creation for incoming packets requires significant heap space. A configuration of -Xmx4G is recommended for the JVM, monitoring Garbage Collection (GC) pauses via JMX exporters.[9]

3.2 Configuration for Stateless Operation

To enable horizontal scaling, Traccar must treat each device connection as ephemeral and stateless regarding persistence.

- **Disabling Internal Database Writes:** In high-throughput scenarios, the database write lock becomes the primary bottleneck. The configuration should minimize direct database interaction for positions.
- **Kafka Forwarding:** The traccar.xml configuration should be set to forward positions directly to Kafka. This transforms Traccar into a "Pass-Through" gateway.
  XML
  ```xml
  <entry key='forward.type'>kafka</entry>
  <entry key='forward.url'>kafka-broker:9092</entry>
  <entry key='forward.topic'>telemetry_raw</entry>
  ```

  This configuration ensures that the ingestion latency is determined by the network bandwidth and Kafka acknowledgment speed, rather than disk I/O on a database server.[3]

3.3 Handling Connection Staling and Buffering

A common issue in large fleets is "ghost connections," where a device disconnects ungracefully (e.g., signal loss), leaving the TCP socket open on the server. This consumes resources.

- **Timeout Strategy:** The server.timeout parameter defines the inactivity period before the server forcibly closes the connection. This should be set slightly higher than the device's heartbeat interval (e.g., 120 seconds for a 60-second reporting interval) to aggressively prune stale connections.[8]
- **Buffering:** To handle out-of-order delivery common with UDP protocols, the server.buffering.threshold parameter enables an internal buffer (e.g., 3000ms). This allows the server to reorder packets based on their sequence number or timestamp before processing, ensuring path integrity.[10]

3.4 Synchronization with Redis

When running multiple Traccar instances behind a load balancer, a user sending a command (e.g., "Unlock Door") via the API might hit Server A, while the device is connected to Server B.

- **Redis Broadcast:** Traccar supports a broadcast mechanism to synchronize state across nodes. By configuring broadcast.type to redis, instances subscribe to a shared channel. When Server A receives the command, it publishes it to Redis. Server B, recognizing the device ID in its active session list, picks up the message and transmits the command to the device.[11]

4. Data Engineering: Persistence and Schema Design

The data layer is the foundation of the TMS, requiring a schema that supports complex logistics relationships while handling the massive volume of time-series telemetry.

4.1 Geospatial Time-Series with TimescaleDB

Standard PostgreSQL tables (B-Tree indexed) suffer from performance degradation as table size increases, particularly for time-series data which is insert-heavy and mostly ordered. **TimescaleDB** is the requisite solution.

- **Hypertables:** The VehiclePosition table is converted into a hypertable. This abstraction automatically partitions the data into "chunks" based on a time interval (e.g., 1 day).
  SQL
  ```sql
  SELECT create_hypertable('vehicle_position', 'timestamp', chunk_time_interval => INTERVAL '1 day');
  ```

  This structure allows for "constant-time" inserts because the database engine writes to the active memory-resident chunk, avoiding the overhead of rebalancing a massive index for every write.[5]
- **Compression:** TimescaleDB's columnar compression can reduce storage requirements by 90% for historical data. Older chunks can be compressed automatically, making long-term retention (e.g., for audit compliance) cost-effective.
- **Continuous Aggregates:** For reporting dashboards that show "Daily Mileage" or "Idle Time," calculating these metrics from raw points on-the-fly is prohibitive. TimescaleDB's Continuous Aggregates maintain a pre-calculated materialized view that updates incrementally as new data arrives, providing millisecond-level query responses for analytical dashboards.[5]

4.2 Logistics Data Model

The relational schema must capture the nuance of multi-leg, multi-mode transportation.

Core Entities and Relationships:

- **Order (The Demand):** Represents the commercial agreement with the shipper. It contains the CustomerID, Origin, Destination, and requested DeliveryWindow.
- **Shipment (The Plan):** The operational unit derived from an Order. A single Order (e.g., 100 pallets) might be split into multiple Shipments (LTL), or multiple Orders might be consolidated into one Shipment. This entity holds the BillOfLading number.
- **Load (The Execution):** The assignment of a Shipment to a physical asset (Vehicle/Trailer) and a human resource (Driver). A Load can contain multiple Shipments (Cross-docking/Milk-run).
- **Stop/Leg (The Movement):** The atomic unit of travel. A Load consists of a sequence of Stops (Pickup, Drop-off, Layover, Fuel Stop). Each Stop has a ScheduledTime, EstimatedTime (ETA), and ActualTime.

Schema Optimization Table:

Entity	Field Recommendations	Rationale
All Entities	id (UUIDv4)	Prevents enumeration attacks; facilitates sharding and data migration.[14]
Shipment	attributes (JSONB)	Stores variable accessorials (e.g., "Liftgate Required", "Hazmat") without rigid schema changes.[15]
Address	location (Geography)	PostGIS geography type ensures accurate distance calculations on the geoid (spheroid) rather than flat plane geometry.[12]
Rates	effective_date (Range)	Allows temporal versioning

		of tariff contracts, crucial for audit and historical billing accuracy.

4.3 Multi-Tenancy Strategy

For a SaaS TMS serving multiple carrier companies, data isolation is paramount. The **Schema-per-Tenant** architecture using django-tenants is the optimal approach.[16]

- **Mechanism:** Every client organization has a dedicated PostgreSQL schema (e.g., client_a, client_b). The public schema contains shared data like User accounts (if users span tenants) and system-wide configurations.
- **Request Routing:** Django middleware intercepts every HTTP request. It inspects the subdomain (e.g., client-a.tms.platform.com) and sets the PostgreSQL SEARCH_PATH to client_a, public.
- **Security Implication:** This ensures logical isolation at the database level. Even if a developer writes a broad query (SELECT * FROM orders), the database only returns records from the active schema, preventing accidental data leakage between competitors.

5. Algorithmic Intelligence: Route Optimization

A modern TMS must do more than track; it must optimize. Integrating **Google OR-Tools** allows the system to solve the Vehicle Routing Problem (VRP) and its variants.

5.1 The Optimization Pipeline

The optimization process is computationally intensive and must be decoupled from the request-response cycle using Celery.[18]

Step 1: Matrix Generation
The VRP solver requires a distance/time matrix between all locations (Depot, Pickups, Drop-offs).

- **Cost Management:** Relying solely on Google Maps Distance Matrix API for N x N matrices is cost-prohibitive at scale.
- **Solution:** Integrate a self-hosted OSRM (Open Source Routing Machine) or Valhalla instance. The system queries this local service to build the matrix efficiently. Google Maps is reserved for the final, customer-facing ETA display to ensure high accuracy.[19]

Step 2: Constraint Modeling
The OR-Tools constraint solver is configured with business rules:

- **Capacity Constraint (CVRP):** The total weight/volume of shipments on the truck cannot exceed its limit.
- **Time Window Constraint (VRPTW):** Deliveries must occur within the customer's receiving hours.
- **Precedence Constraint:** A Pickup stop must always precede its corresponding Drop-off stop.[20]
- **Driver HoS:** Constraints are added to model legal driving limits (e.g., max 11 hours driving), potentially forcing a "Rest Stop" insertion.

Step 3: Solver Execution
The solver iterates through thousands of permutations using metaheuristics (e.g., Guided Local Search) to minimize the objective function (usually Total Distance or Total Cost). The output is a sequence of stop IDs, which the Django worker translates back into Load and Stop database records.

5.2 Asynchronous Task Management with Celery

Different optimization tasks have different latency requirements.

- **Instant Quote:** Needs sub-second response. This task goes to a high-priority Celery queue backed by Redis.
- **Nightly Fleet Planner:** Takes minutes to run for hundreds of vehicles. This goes to a batch-processing queue.
- **Routing:** CELERY_TASK_ROUTES in Django settings ensures that these compute-heavy tasks are routed to worker nodes with high-CPU instance types, keeping the web-serving nodes lightweight.[18]

6. Financial Engineering: Rating and Audit

The financial module handles the complexity of freight rating (calculating the cost) and auditing (verifying the carrier's invoice).

6.1 The Strategy-Based Rating Engine

Freight rates are highly variable. A shipment might be rated per mile, per hundredweight (CWT), per pallet, or via a flat zone-to-zone matrix.

- **Design Pattern:** The Rating Engine implements the **Strategy Pattern**. A RatingProfile model determines which strategy (Python class) to invoke.[21]
- **Data Structures:**
  - RateTariff: The master agreement.
  - RateBreak: Stores tiered pricing (e.g., 0-500lbs: $50, 501-1000lbs: $45).
  - AccessorialMaster: Definitions of extra charges.
- **Logic:** When rating a shipment, the engine first calculates the Base Rate using the selected strategy and the shipment's weight/distance. It then iterates through applicable Accessorials.

6.2 Handling Fuel Surcharges

Fuel surcharges are dynamic, based on fluctuating national averages (e.g., DOE index).

- **Implementation:** A FuelIndex model stores weekly average prices. The RateTariff links to a specific index and defines a formula (e.g., "For every $0.05 increase above $2.50, add $0.01 per mile").
- **Automation:** A periodic Celery task scrapes the DOE website weekly to update the FuelIndex, ensuring that all loads rated that week automatically reflect the correct surcharge without manual intervention.[22]

6.3 Automated Freight Audit and Payment (FAP)

Automating the reconciliation of carrier invoices against rated loads allows for "Management by Exception."

- **Matching Algorithm:** When an invoice arrives (EDI 210), the system attempts to match it to a Shipment based on the Bill of Lading (BOL) number.

- **Tolerance Logic:** A "Tolerance Rule" is applied (e.g., Invoice Total vs. Rated Amount).
    - If the variance is within tolerance (e.g., < $10.00), the invoice is auto-approved for payment.
    - If outside tolerance, it creates a Dispute record and flags it for a human auditor. This drastically reduces the labor required for accounts payable.[24]

7. Frontend Engineering: React and Visualization

The frontend is the operator's cockpit. It must remain responsive even when rendering thousands of moving assets.

7.1 High-Performance Visualization with Deck.gl

DOM-based mapping libraries like Leaflet struggle with high entity counts because each marker is a separate HTML element. For an enterprise fleet map, **Deck.gl** is the required solution.

- **WebGL Rendering:** Deck.gl renders layers on the HTML5 Canvas using WebGL. This offloads the rendering work to the client's GPU, allowing for the smooth visualization of 100,000+ points.[26]
- **Layer Strategy:**
    - IconLayer: Renders vehicle positions using sprites. It supports efficient updates of position and rotation (heading) by modifying a binary data buffer rather than the DOM.
    - TripsLayer: A specialized layer for visualizing historical paths. It takes a massive array of coordinates and timestamps and renders a "fading trail" animation that represents the vehicle's movement over a time window. This provides immediate visual context on speed and direction that static lines cannot convey.[28]

7.2 Reactive State Management

The application state changes rapidly as telemetry streams in.

- **WebSocket Integration:** The React app maintains a persistent WebSocket connection to

the Django Channels server.
- **Throttling:** A raw stream of 50 updates/second is too fast for the human eye and the React render cycle. A buffering hook (e.g., useThrottledMessage) collects updates and applies them to the global state store in batches, synchronized with the browser's refresh rate (requestAnimationFrame) to prevent UI freezing.[30]
- **State Library: Zustand** is recommended over Redux for its minimalist API and transient state handling. It allows the map component to subscribe strictly to the vehiclePositions slice of state, preventing unnecessary re-renders of the navigation bar or side panels when a truck moves.

7.3 Historical Playback Component

Auditability requires the ability to replay a route.

- **Implementation:** The component fetches the full GeoJSON path and a series of timestamped waypoints.
- **Slider Logic:** A React slider controls a currentTime state.
- **Interpolation:** The frontend performs linear interpolation between the two waypoints surrounding the currentTime to determine the exact screen position of the marker. This ensures smooth movement of the icon even if the GPS reporting interval was coarse (e.g., every 5 minutes).[31]

8. Security, Compliance, and Enterprise Readiness

8.1 Role-Based Access Control (RBAC) with Object-Level Permissions

Standard Django permissions (Global Create/Read/Update/Delete) are insufficient. An enterprise client needs to restrict a Dispatcher to seeing only *their* region's loads.

- **Django Guardian:** This library enables Object-Level Permissions (OLP). It adds a table mapping User + Permission + ObjectInstance.
- **Implementation:** When a Load is created and assigned to the "Northeast" region, the system programmatically grants view_load permission to the "Northeast Dispatchers" group.

- **View Integration:** Django REST Framework views use the DjangoObjectPermissions class to automatically filter querysets based on these assignments, ensuring the API never returns unauthorized data.[32]

8.2 Immutable Audit Logging

For compliance (e.g., pharmaceutical or hazmat transport), every change must be logged.

- **Library:** django-auditlog.
- **Mechanism:** It listens to Django's post_save signals. When a Shipment status changes from "Pending" to "In-Transit," it writes a log entry recording the Old Value, New Value, Actor (User), and Timestamp.
- **Middleware:** Custom middleware captures the user ID from the JWT token and attaches it to the thread local storage so the signal handler can attribute the change to the correct API user.[34]

8.3 Authentication with JWT and Multi-Tenancy

- **JWT Customization:** The SimpleJWT library is extended. The token payload must include the schema_name (tenant identifier).
- **Middleware:** A custom authentication middleware parses the JWT. Before the request reaches the view, it reads the schema_name and configures the django-tenants context. This ensures that a valid token for Tenant A cannot be used to access API endpoints for Tenant B, providing a second layer of defense beyond database schemas.[36]

9. Infrastructure and Scalability Strategies

9.1 Kubernetes Deployment Architecture

The system is containerized and orchestrated via Kubernetes (K8s) to ensure high availability.

- **Ingress Controller:** NGINX Ingress is used to manage external access.
  - **UDP Support:** A critical configuration for Traccar. Standard Ingress is HTTP-only. To support GPS devices, the NGINX controller's ConfigMap is patched to expose the necessary TCP/UDP ports (e.g., 5055 for OsmAnd protocol) and map them to the Traccar service.[37]
  - **Cloud Load Balancer:** On AWS, a Network Load Balancer (NLB) is provisioned with annotations (service.beta.kubernetes.io/aws-load-balancer-type: nlb) to handle the UDP traffic pass-through.[39]

9.2 Scalability Drivers

- **Horizontal Autoscaling:**
  - **Stateless:** The React frontend and Traccar gateway scale based on CPU/Memory usage using Horizontal Pod Autoscalers (HPA).
  - **Stateful:** The Database scales vertically (bigger instance) initially, then horizontally via Read Replicas.
- **Celery Scaling:** The worker pools are scaled based on **Queue Depth** (custom metric in KEDA - Kubernetes Event-driven Autoscaling). If the "Optimization" queue builds up backlog, KEDA spins up more worker pods automatically.[40]

9.3 Observability

- **Metrics:** A Prometheus sidecar scrapes metrics from the Django application (request latency, error rates) and the Traccar JVM (heap usage, thread count).
- **Tracing:** Distributed tracing (e.g., Jaeger or OpenTelemetry) is implemented to track a request from the React frontend, through the Django API, into the Celery worker, and finally to the database, allowing for precise bottleneck identification.[41]

10. Conclusion

This research delineates a robust path for engineering an enterprise TMS. By rejecting the monolithic "all-in-one" server approach in favor of a decoupled, event-driven architecture, the system gains the resilience required for mission-critical logistics. The strategic use of

Traccar as a raw gateway, **Kafka** as a shock absorber, and **Django** as the transactional authority creates a balanced ecosystem. This backend, paired with the high-fidelity visualization capabilities of **Deck.gl** and **React**, delivers a user experience that empowers operators to manage complex supply chains with clarity and precision. The inclusion of algorithmic optimization and strict multi-tenant security ensures the platform is not just a recording tool, but a competitive asset for enterprise logistics providers.

Works cited

1. Processing Time-Series Data with Redis and Apache Kafka, accessed December 2, 2025, https://redis.io/blog/processing-time-series-data-with-redis-and-apache-kafka/
2. Building High-Throughput Data Ingestion Pipelines with Kafka on OpenMetal, accessed December 2, 2025, https://openmetal.io/resources/blog/building-high-throughput-data-ingestion-pipelines-with-kafka-on-openmetal/
3. Forwarding - Traccar, accessed December 2, 2025, https://www.traccar.org/forward/
4. High availability and load balancing - Traccar, accessed December 2, 2025, https://www.traccar.org/forums/topic/high-availability-and-load-balancing/
5. [Managing Time-Series Data: Why TimescaleDB Beats PostgreSQL] - Mad Devs, accessed December 2, 2025, https://maddevs.io/writeups/time-series-data-management-with-timescaledb/
6. Traccar Architecture, accessed December 2, 2025, https://www.traccar.org/architecture/
7. Implementing New Protocol - Traccar, accessed December 2, 2025, https://www.traccar.org/implement-protocol/
8. Optimization - Traccar, accessed December 2, 2025, https://www.traccar.org/optimization/
9. JVM Metrics | Grafana Labs, accessed December 2, 2025, https://grafana.com/grafana/dashboards/15392-jvm-metrics/
10. Configuration File - Traccar, accessed December 2, 2025, https://www.traccar.org/configuration-file/
11. Redis Broadcast Issues and Configuration with 6.9.1 - Traccar, accessed December 2, 2025, https://www.traccar.org/forums/topic/redis-broadcast-issues-and-configuration-with-691/
12. PostgreSQL Extensions: Using PostGIS for Geospatial and Time-Series Data, accessed December 2, 2025, https://dev.to/tigerdata/postgresql-extensions-using-postgis-for-geospatial-and-time-series-data-ja5
13. Should I use TimescaleDB or partitioning is enough? : r/PostgreSQL - Reddit, accessed December 2, 2025, https://www.reddit.com/r/PostgreSQL/comments/t6pbqa/should_i_use_timescaledb_or_partitioning_is_enough/

14. Designing an Efficient Database Schema for Logistics Operations: Tracking Shipment Statuses, Driver Assignments, and Delivery Times with Real-Time Updates and Historical Analysis - Zigpoll, accessed December 2, 2025, https://www.zigpoll.com/content/how-can-i-design-a-database-schema-to-efficiently-track-shipment-statuses-driver-assignments-and-delivery-times-while-ensuring-realtime-updates-and-historical-data-analysis-for-a-logistics-company's-operations
15. Dimensional Data Modeling for Logistics: A Step-by-Step Guide | by Saham Siddiqui, accessed December 2, 2025, https://medium.com/@sahamsiddiqui/dimensional-data-modeling-for-logistics-a-step-by-step-guide-f5a5e833517c
16. Building Secure Multi-Tenant Apps with django_tenants - Mindbowser, accessed December 2, 2025, https://www.mindbowser.com/multitenant-apps-django-tenants/
17. Evolving django-multitenant to build scalable SaaS apps on Postgres & Citus, accessed December 2, 2025, https://www.citusdata.com/blog/2023/05/09/evolving-django-multitenant-to-build-scalable-saas-apps-on-postgres-and-citus/
18. How to route tasks to different queues with Celery and Django - Stack Overflow, accessed December 2, 2025, https://stackoverflow.com/questions/51631455/how-to-route-tasks-to-different-queues-with-celery-and-django
19. Vehicle Routing Problem | OR-Tools | Google for Developers, accessed December 2, 2025, https://developers.google.com/optimization/routing/vrp
20. Vehicle Routing Problem Pickup and Delivery in a certain order #2628 - GitHub, accessed December 2, 2025, https://github.com/google/or-tools/discussions/2628
21. Design pattern for rating engine - Stack Overflow, accessed December 2, 2025, https://stackoverflow.com/questions/62848724/design-pattern-for-rating-engine
22. Using AI to Manage Accessorial Charges in Logistics - iCaptur, accessed December 2, 2025, https://icaptur.ai/using-ai-to-manage-accessorial-charges/
23. dynamics-365-unified-operations-public/articles/supply-chain/transportation/tasks/associate-fuel-index-carrier-accessorial-charge.md at main - GitHub, accessed December 2, 2025, https://github.com/MicrosoftDocs/Dynamics-365-Unified-Operations-Public/blob/live/articles/supply-chain/transportation/tasks/associate-fuel-index-carrier-accessorial-charge.md
24. Simplifying the Freight Audit Process: Auditing & Payment Guide, accessed December 2, 2025, https://www.zdscs.com/blog/freight-audit-pay-avoiding-fees/
25. Reconcile freight in transportation management - Supply Chain Management | Dynamics 365 | Microsoft Learn, accessed December 2, 2025, https://learn.microsoft.com/en-us/dynamics365/supply-chain/transportation/reconcile-freight-transportation-management
26. Performance Optimization | deck.gl, accessed December 2, 2025, https://deck.gl/docs/developer-guide/performance
27. Questions on Leaflet · visgl deck.gl · Discussion #8941 - GitHub, accessed

December 2, 2025, https://github.com/visgl/deck.gl/discussions/8941
28. Google Maps and deck.gl Trips Layer | Maps JavaScript API - Google for Developers, accessed December 2, 2025, https://developers.google.com/maps/documentation/javascript/examples/deckgl-tripslayer
29. TripsLayer | deck.gl, accessed December 2, 2025, https://deck.gl/docs/api-reference/geo-layers/trips-layer
30. The complete guide to WebSockets with React - Ably, accessed December 2, 2025, https://ably.com/blog/websockets-react-tutorial
31. linghuam/Leaflet.TrackPlayBack: a leaflet track-playback plugin - GitHub, accessed December 2, 2025, https://github.com/linghuam/Leaflet.TrackPlayBack
32. Assign Object Permissions - Django Guardian, accessed December 2, 2025, https://django-guardian.readthedocs.io/en/3.0.2/userguide/assign/
33. Using django permissions with django guardian (object level) permissions - Django Forum, accessed December 2, 2025, https://forum.djangoproject.com/t/using-django-permissions-with-django-guardian-object-level-permissions/39243
34. django-auditlog documentation — django-auditlog 3.3.0.post8+g7d13fd4ba documentation, accessed December 2, 2025, https://django-auditlog.readthedocs.io/
35. City-of-Helsinki/django-auditlog-extra - GitHub, accessed December 2, 2025, https://github.com/City-of-Helsinki/django-auditlog-extra
36. simplejwt token work for all tenants for django-tenants - Stack Overflow, accessed December 2, 2025, https://stackoverflow.com/questions/77312448/simplejwt-token-work-for-all-tenants-for-django-tenants
37. Load Balancing TCP and UDP Traffic in Kubernetes with NGINX - F5, accessed December 2, 2025, https://www.f5.com/company/blog/nginx/load-balancing-tcp-and-udp-traffic-in-kubernetes-with-nginx
38. Ingress nginx for TCP and UDP services | minikube - Kubernetes, accessed December 2, 2025, https://minikube.sigs.k8s.io/docs/tutorials/nginx_tcp_udp_ingress/
39. Target groups for your Network Load Balancers - AWS Documentation, accessed December 2, 2025, https://docs.aws.amazon.com/elasticloadbalancing/latest/network/load-balancer-target-groups.html
40. Mastering Asynchronous Tasks: Celery with Django Rest Framework – Part 1 - Mindbowser, accessed December 2, 2025, https://www.mindbowser.com/celery-django-rest-framework/
41. Java Virtual Machine (JVM) monitoring made easy | Grafana Labs, accessed December 2, 2025, https://grafana.com/solutions/java-virtual-machine-jvm/monitor/