Borla Tracker (Waste Management System) Backend Documentation

Architecture Overview

The system leverages a robust, scalable stack centered on Django for backend services, ensuring production-grade stability with Django and Django REST Framework for modular API development, including Swagger/Redoc integration for automated documentation.

Technology Stack

Frameworks

Django v5.2.7: Stable LTS release optimized for production deployments, handling core business logic and security.

Django REST Framework (DRF) v3.16.1: Delivers RESTful APIs with serializers, role-based access control, and seamless Swagger/Redoc support for interactive documentation.

Database

PostgreSQL: Primary relational database with PostGIS for geospatial features like GPS validation and route optimization.

SQLite: Limited to local development and testing environments.

Caching

Redis: Manages sessions, request/response caching, and real-time GPS updates with advanced data structures.

Memcached available as a lightweight alternative if needed.

Task Queue

Celery: Processes asynchronous tasks such as incentive calculations, notifications (SMS/email/push), and route optimization.

Broker: Redis (shared with caching layer) for reliability; RQ considered but Celery selected for ecosystem maturity.

File Storage

Local Storage: Development-only for simplicity.

Amazon S3: Production storage for evidence photos and client uploads.

Deployment

Docker: Containerizes all services for consistent portability.

Kubernetes (K8s): Orchestrates scaling, auto-scaling for GPS microservices, and 99.9% uptime across multi-zone deployments.

CI/CD: GitHub Actions automates builds, testing, and rollouts to Docker/K8s environments.

API Documentation: Automated Swagger/Redoc generation

Development Setup

Establish a consistent local environment using Python 3.12.3 to align with Django 5.2.7 and DRF 3.16.7 requirements, ensuring compatibility across the waste management backend stack.

Prerequisites

- **Python v3.11+**: Core runtime for Django and DRF
- **PostgreSQL with PostGIS**: Relational database with geospatial extensions for route optimization
- **Redis**: Caching and Celery broker
- **Git**: Version control
- **Docker**: Optional for containerized workflows

Installation Steps

```
# Clone repository
git clone https://github.com/Kyei-Ernest/wms.git
cd waste-management-api

# Create virtual environment
python3 -m venv venv
source venv/bin/activate  # Linux/Mac
# venv\Scripts\activate  # Windows
```

```
# Install dependencies
pip install -r requirements.txt
```

Environment Configuration

Create .env file in project root:

```
# Django
DJANGO_SECRET_KEY=your-secret-key
DJANGO_DEBUG=True

# Allowed Hosts
DJANGO_ALLOWED_HOSTS=localhost,127.0.0.1,your-domain.com

# CORS & CSRF
CORS_ALLOW_ALL_ORIGINS=True
CSRF_TRUSTED_ORIGINS=https://your-domain.com

# Database
DB_NAME=your_database_name
DB_USER=your_database_user
DB_PASSWORD=your_database_password
DB_HOST=localhost
DB_PORT=5432

# JWT
ACCESS_TOKEN_LIFETIME_DAYS=1
REFRESH_TOKEN_LIFETIME_DAYS=7
```

Database & Server Startup

```
# Apply migrations
python manage.py makemigrations
python manage.py migrate

# Enable PostGIS (in PostgreSQL)
CREATE EXTENSION postgis;

# Start development server
```

python manage.py runserver

Project Structure

The project adopts a domain-driven Django architecture with modular apps organized by actor responsibilities and business domains, ensuring clear separation of concerns and scalability.

Directory Layout

```
borla_master/                # Django project root
│
├── manage.py                # Entry point for CLI commands
├── requirements.txt         # Python dependencies
├── .env                     # Environment variables
│
├── borla_master/            # Global config
│   ├── settings.py          # Environment-aware settings
│   ├── urls.py              # Root URL routing
│   ├── wsgi.py              # WSGI entrypoint
│   └── asgi.py              # ASGI entrypoint (for WebSockets, async)
│
├── accounts/                # Authentication, user roles, RBAC
├── client/                  # Client registration, request creation, wallet
├── collector/               # Collector onboarding, assignment, evidence
├── supervisor/              # Route generation, dashboards, performance
├── company/                 # Supervisor creation, collector approval
├── on_demand/               # OnDemandRequest model, views, serializers
├── scheduled_request/       # ScheduledRequest model, recurrence logic
├── routes/                  # Route + RouteStop models, optimization
├── collection_management/   # CollectionRecord, audit logging, compliance
├── zones/                   # GIS zones, boundaries, geospatial logic
├── documentation/           # Swagger, Redoc, diagrams, onboarding docs
└── waste_management/        # Shared utilities, base models, permissions
```

Domain Apps Overview

- **accounts**: User authentication, role-based access control, permissions
- **client**: Request creation, wallet management, real-time tracking
- **collector**: Onboarding, GPS evidence submission, assignment handling

- **supervisor**: Route generation, performance dashboards, compliance review
- **company**: Supervisor management, collector approval workflows
- **on_demand/scheduled_request**: Type-specific models, serializers, business rules
- **routes**: RouteStop models, geospatial optimization with PostGIS
- **wallet**: Dynamic pricing, incentive/penalty calculations
- **collection_management**: Immutable audit trails, compliance scoring
- **zones**: GIS boundary definitions, spatial query logic

Database Schema

The database schema follows a normalized PostgreSQL structure with PostGIS extensions, modeling core waste management entities through Django ORM for scalability and geospatial capabilities.

Entity Relationships

Users → Clients, Collectors, Supervisors, Company (inheritance)
├── Clients → OnDemandRequest, ScheduledRequest
├── Collectors → Routes (execution), CollectionRecords (logging)
├── Supervisors → Routes (oversight)
├── Routes → RouteStops (composition)
└── CollectionRecord → Payment, AuditTrail (immutable records)

This ERD captures hierarchical user roles, request-to-route workflows, and audit-ready collection data with foreign key constraints ensuring referential integrity.

🔗 VIEW ERD DIAGRAM

Migration Strategy

Development

python manage.py makemigrations
python manage.py migrate

Staging/Production

- Create PostgreSQL backup before applying changes
- Deploy migrations incrementally with zero-downtime techniques (e.g., Django's atomic transactions)

Version Control

- Commit all migration files to Git for reproducible schema evolution
- Maintain migration consistency across environments via Dockerized PostgreSQL instances and CI/CD pipelines

Business Logic Implementation

Pricing Algorithm

The pricing algorithm calculates service fees based on multiple factors to ensure fair and zone-appropriate pricing.

Base Price Calculation

The base price is determined by either bag count or bin size (measured in liters), depending on the service type requested by the client.

Pricing Multipliers

The system applies two categories of multipliers to the base price:

- **Waste Type Multiplier**: Hazardous waste (2.0×), recyclables (0.8×), general waste (1.0×)
- **Zone Factor**: High-density zones (1.2×), low-density zones (1.0×)

Pricing Formula

Final Price = (Base Rate × Bag Count or Bin Size) × Waste Type Multiplier × Zone Factor

Discount Application

Discounts are applied after the final price calculation for clients with active subscriptions or qualifying loyalty tier status.

Route Optimization Algorithm

The route optimization system generates efficient collection routes by processing pickup requests and geographic data.

Input Parameters

The algorithm receives a consolidated list of requests including both on-demand and scheduled pickups, each containing GPS coordinates and time preferences.

Processing Steps

1. Group requests by geographic zone to minimize travel distance
2. Apply nearest-neighbor heuristic to sequence stops within each zone
3. Adjust routing based on time slot requirements (morning versus afternoon pickups)
4. Allow supervisor override or manual reordering of stops when necessary

Output Deliverables

The system generates an optimized route containing ordered stops, estimated duration, and total distance for each collection run.

Auto-Assignment Logic

The auto-assignment system matches collection requests with appropriate collectors based on multiple criteria.

Assignment Criteria

- **Availability**: Collector must not have overlapping route assignments
- **Proximity**: Collector must be within acceptable GPS radius of the service area
- **Rating**: Collector must meet minimum threshold requirements for sensitive waste types
- **Classification**: Distinction between company collectors (requiring supervisor approval) and private contractors
- **Mandatory Assignment**: Collector can accept or decline but is mandatorily assigned

Escalation Protocol

When no collector meets the assignment criteria, the request is escalated to the supervisor for manual assignment and review.

Incentive and Penalty Calculation

The incentive system rewards performance while maintaining service quality standards.

Incentive Structure

- Base incentive per completed request
- Bonus multipliers for:

    - Segregation compliance ≥ 90%
    - Punctuality (arrival within ±10 minutes of scheduled time)
    - High client ratings

Penalty Structure

- GPS mismatch (collector not within 100 meters of designated stop)
- Missed stops without documented valid reason
- Segregation compliance < 70%

Calculation Formula

Net Incentive = Base Incentive + Compliance Bonus − Penalty Deductions

GPS Proximity Validation

The system validates collector location to ensure service delivery at the correct address.

Standard Parameters

- **Acceptable Radius**: 100 meters from client's registered pickup location
- **Validation Process**: Compare collector's live GPS coordinates with request coordinates
- **Violation Protocol**: If outside acceptable radius, flag violation and require supervisor review

Special Cases

- **Poor GPS Signal**: Tolerance extended to 150 meters under documented signal degradation
- **Multi-Unit Complexes**: Supervisor override available for large properties with multiple pickup points

Segregation Scoring

The segregation scoring system evaluates waste separation quality through photographic evidence and category assessment.

Evidence Submission

Collectors submit photographic documentation of segregated waste at each pickup location.

Scoring Methodology

1. Identify correctly segregated bags by category (organic, recyclable, hazardous)

2. Calculate percentage compliance for each category
3. Generate weighted average to produce final segregation score

Performance Thresholds

- **≥ 90%**: Qualifies for incentive bonus
- **70–89%**: Neutral (no bonus or penalty)
- **< 70%**: Penalty applied to collector's compensation

Background Tasks

Celery orchestrates scheduled and asynchronous operations across priority queues, leveraging Redis as broker for high-throughput waste management workflows.

Scheduled Tasks

- **Daily Incentive/Penalty Calculation**: Nightly batch processing of compliance scores and collector bonuses via wallet app
- **Weekly Route Generation**: Sunday evening pre-computation of supervisor routes using routes app optimization
- **Wallet Settlement**: Daily payment reconciliation and balance updates
- **Cleanup Jobs**: Removes expired sessions, stale GPS logs, and soft-deleted records

Async Operations

- **Notification Dispatch**: Non-blocking SMS, email, and push notifications
- **Evidence Processing**: S3/Cloudinary uploads for photos with background segregation scoring
- **Audit Logging**: Immutable CollectionRecords created without API blocking

Queue & Retry Policies

Priority Levels

- **High Priority**: Notifications, GPS validation, auto-assignment
- **Medium Priority**: Incentives, segregation scoring
- **Low Priority**: Analytics, cleanup

Workers subscribe to queues with Flower monitoring; retries use exponential backoff (max 3), idempotency for settlements, and circuit breakers for failures.

Testing Standards

Coverage: Minimum 80% across models, serializers, views, and critical paths (pricing, GPS, assignment).

Unit Tests

```
python manage.py test
# or
pytest --cov=apps/
```

Integration: Django TestCase with PostGIS fixtures simulating full workflows; mocks for Redis, S3, payments, notifications.

Fixtures

```
python manage.py loaddata test_clients.json
```

Code Standards

- **Style**: PEP 8 via Black/Flake8
- **Naming**: Models (singular), Serializers/ViewSets (suffixed), URLs (plural)
- **Docs**: Google-style docstrings, annotated Swagger/Redoc endpoints

Git Workflow

- **Branches**: main (prod)
- **Commits**: Conventional (feat:, fix:, docs:)
- **Reviews**: PRs require 1+ approval, passing CI/CD tests focusing on readability and business rules

API Implementation Details

The Borla Tracker API serves as the unified backend infrastructure for a comprehensive waste management system, providing digital transformation and operational efficiency for private collectors, company-employed collectors, supervisors, and clients. Built on a Django-powered architecture, the API delivers enterprise-grade functionality across all operational workflows.

Core Capabilities

Collector Assignment

The system efficiently assigns collectors to both on-demand and scheduled collection requests through Django REST Framework endpoints. Assignment logic incorporates geospatial PostgreSQL queries to optimize routing and minimize travel time, ensuring cost-effective operations and timely service delivery.

Supervisor Oversight

Supervisors maintain real-time visibility into route execution, collector progress, and performance metrics. The platform leverages Celery-driven asynchronous updates combined with Redis caching to provide live GPS tracking capabilities, enabling proactive management and rapid response to operational issues.

Client Interaction

Clients interact with the system through secure, role-based API access for request creation, collection tracking, and wallet management. The platform provides automated API documentation through Swagger and Redoc interfaces, ensuring developer accessibility and integration ease for third-party systems.

Business Automation

The system automates incentive distribution, penalty assessment, and compliance monitoring through background Celery tasks. This architecture ensures scalable processing capabilities within Kubernetes-orchestrated environments, maintaining performance under varying operational loads.

Audit and Reporting

All operational activities are recorded in immutable, audit-ready records within PostgreSQL, providing complete operational transparency and regulatory compliance. Evidence files, including photographs and document uploads, are securely stored in S3-backed infrastructure with appropriate retention policies.

System Actors

- **Client**: End-user who creates requests (on-demand or scheduled), pays for services, and tracks status
- **Collector**: Worker who performs waste collection. Can be:
  - Private collector (self-registered)
  - Company collector (created/approved by a company and supervised)
- **Supervisor**: Oversees company collectors, monitors routes, approves registrations, and ensures compliance

- **Company**: Organizational entity that creates supervisors, manages collectors, and owns routes
- **Admin**: System-level actor who manages roles, monitors statistics, and ensures platform integrity

🔗 **VIEW USE CASE DIAGRAM**

Domain Models

Model Definitions

User

- Fields: username, phone_number, email, role, profile_photo, address, is_verified, is_active, is_staff, created_at
- Methods: save() (auto-generate username), **str**()

Client

- Fields: first_name, last_name, alternate_phone, wallet_balance, segregation_compliance_percent, registration_date
- Methods: **str**()

Collector

- Fields: first_name, last_name, company, supervisor, is_private_collector, vehicle_number, vehicle_type, assigned_area_zone, daily_wage_or_incentive_rate, bank_account_details, average_rating, total_collections, last_known_latitude, last_known_longitude
- Methods: full_name(), **str**()

Company

- Fields: name, registration_number, address, created_at

Supervisor

- Fields: first_name, last_name, assigned_zone, created_at

OnDemandRequest

- Fields: pickup_date, pickup_time_slot, address_line1, landmark, area_zone, city, latitude, longitude, location, bag_count, bin_size_liters, waste_type, special_instructions, waste_image, quoted_price, final_price, payment_status, request_status, requested_at, accepted_at, completed_at, cancelled_at, cancellation_reason, created_at, updated_at
- Methods: calculate_quoted_price(), save() (auto-populate fields), **str**()

ScheduledRequest

- Fields: pickup_date, pickup_time_slot, recurrence, request_status, created_at, updated_at
- Methods: **str**()

Route

- Fields: route_date, start_time, end_time, actual_start, actual_end, completion_percent, status, total_distance_km, estimated_duration, created_at, updated_at
- Methods: update_distance_and_duration(), update_completion_status(), save(), **str**()

RouteStop

- Fields: order, expected_minutes, actual_start, actual_end, status, notes, created_at, updated_at
- Methods: **str**()

CollectionRecord

- Fields: collection_id, payment_method, amount_paid, collection_type, scheduled_date, collection_start, collection_end, collected_at, bag_count, bin_size_liters, estimated_volume_liters, waste_type, photo_before, photo_after, segregation_score, status, latitude, longitude, notes, created_at, updated_at, duration_minutes
- Methods: save() (timestamps/duration), get_volume_description(), verify_location(), **str**()

Relationships

- **Inheritance**: Client, Collector, Company, Supervisor all extend User
- **Company** → **Supervisor/Collector**: Company creates supervisors and collectors
- **Client** → **Requests**: Client creates OnDemandRequest and ScheduledRequest
- **Collector** → **Requests/Records**: Collector executes requests and logs CollectionRecords
- **Supervisor** → **Routes**: Supervisor oversees routes and collectors
- **Route** → **RouteStop**: Route contains stops linked to requests
- **CollectionRecord** → **Client/Collector/RouteStop**: Immutable evidence tying together actors and operations

🔗 VIEW UML DIAGRAM

Authentication & Authorization

This API uses **JSON Web Tokens (JWT)** to provide secure, stateless authentication across all protected endpoints.

Upon successful login, the system issues two tokens:

- **Access Token**: Short-lived token used to authenticate API requests
- **Refresh Token**: Long-lived token used to obtain a new access token when the current one expires

Clients must include the access token in the Authorization header for all authenticated requests:

Authorization: Bearer <access_token>

Token Lifecycle

- Access tokens expire automatically after a defined duration
- Refresh tokens are used to generate new access tokens without re-authentication
- During logout, refresh tokens are invalidated to prevent reuse

Role-Based Authorization

Access to resources and actions is strictly controlled based on user roles:

- **Clients**: Can create and track their own requests and manage payments
- **Collectors**: Can view and update only their assigned requests
- **Supervisors**: Can assign collectors, monitor routes, and view company-level data
- **Companies**: Can manage supervisors, collectors, and routes
- **Admins**: Have system-wide visibility and control

Authorization rules are enforced at the API level, ensuring users cannot access or modify unauthorized data.

Security Considerations

The system applies multiple security controls to protect data integrity and operational trust:

- User credentials are securely hashed and never stored in plain text
- JWT tokens are validated on every protected request
- Role-based access control prevents unauthorized operations

- GPS proximity validation reduces fraudulent collection confirmations
- Collection records are immutable to support auditing and accountability

These measures ensure the platform remains secure, reliable, and resistant to misuse.

Error Handling & Status Codes

The API follows standard HTTP status codes to communicate request outcomes clearly:

Status Code	Description
200	Request completed successfully
201	Resource created successfully
400	Invalid request or validation error
401	Unauthorized (missing or invalid token)
403	Forbidden (insufficient permissions)
404	Resource not found
500	Internal server error

Error responses are returned in a consistent JSON format to support predictable client-side handling.

Access Control Model

Authentication is handled globally using JWT, while authorization is enforced through role-aware permissions and data filtering.

- Users can only access resources relevant to their role
- Query results are automatically filtered to prevent data leakage
- Sensitive actions require elevated privileges (e.g., supervisor or admin roles)

This layered approach ensures both security and operational correctness.

Non-Functional Requirements

This section outlines the key non-functional requirements that guide the expected performance, scalability, security, maintainability, and reliability of the system. These requirements represent design targets rather than strict guarantees and reflect realistic expectations for an academic project.

Performance

Response Time: The system is designed to respond to API requests within approximately two (2) seconds under typical operating conditions.

Scalability

Concurrent Requests: The system is intended to support up to 1,000 active requests per day per zone without significant performance degradation.

Collector Tracking: Real-time GPS tracking is designed to scale to approximately 1,000 concurrent collectors per city, assuming normal usage patterns.

Security

Role-Based Access Control (RBAC): Access to system features is controlled based on user roles, including Client, Collector, Supervisor, Company, and Admin, and is enforced at the API level.

Data Protection: Sensitive data such as wallet information, payment details, and GPS coordinates is protected through encryption during data transmission and storage.

Audit Logging: Collection records are treated as immutable once created, helping to support transparency and accountability.

Maintainability

Modular Design: The system follows a modular design approach, with Django applications organized by domain (e.g., Requests, Routes, Collectors, Companies) to simplify development and future changes.

Documentation: API documentation generated through Swagger UI and ReDoc is maintained to reflect the current state of the system and support frontend development and testing.

Reliability

Availability: The system aims to achieve high availability for core request handling and assignment features during normal operational periods.

API Endpoints

This API exposes its endpoints through **auto-generated OpenAPI documentation**, which serves as the authoritative reference for all available resources, request/response schemas, and authentication requirements.

Swagger UI (Interactive Documentation)

Swagger UI provides an interactive interface for exploring and testing API endpoints in real time.

- Allows developers to browse all available endpoints grouped by resource
- Displays required headers, parameters, and request body schemas
- Shows example responses and HTTP status codes
- Supports authenticated testing using access tokens (where enabled)

Access Swagger UI:

🔗 https://kyei-ernest.github.io/wms/documentation/swagger/

ReDoc (Reference Documentation)

ReDoc provides a clean, read-only reference view of the API specification, optimized for clarity and long-form reading.

- Presents endpoints in a structured, easy-to-navigate format
- Clearly documents request and response models
- Highlights authentication and authorization requirements
- Suitable for external partners and non-technical stakeholders

Access ReDoc:

🔗 https://kyei-ernest.github.io/wms/documentation/redoc/

Both Swagger UI and ReDoc expose the **complete OpenAPI schema**, including:

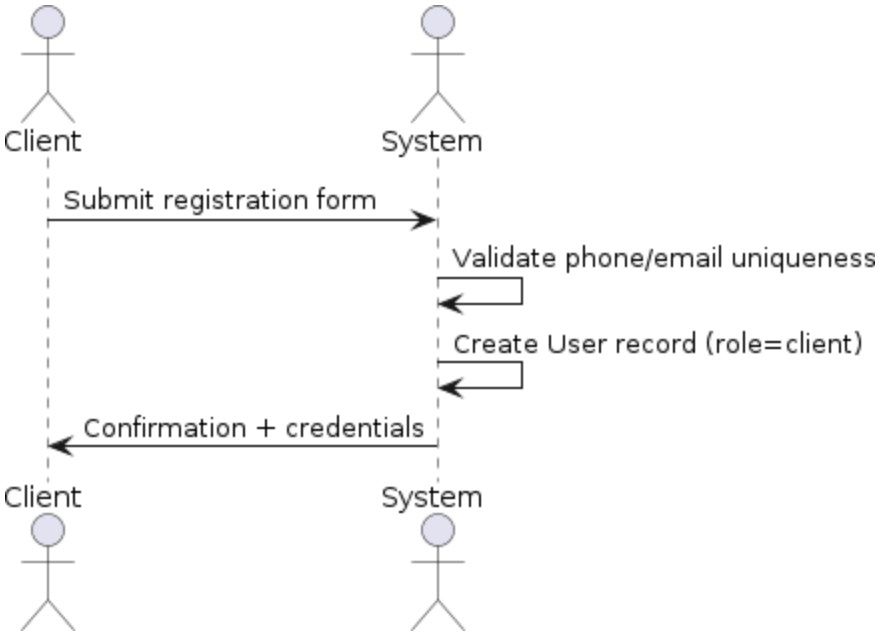
- All available API endpoints
- HTTP methods (GET, POST, PUT, PATCH, DELETE)
- Request parameters and body schemas
- Response payloads and error formats
- Authentication mechanisms (e.g., JWT, API keys)
- Role-based access constraints (e.g., Admin, Supervisor, Collector)

These documentation links should be considered the **single source of truth** for endpoint definitions and usage.

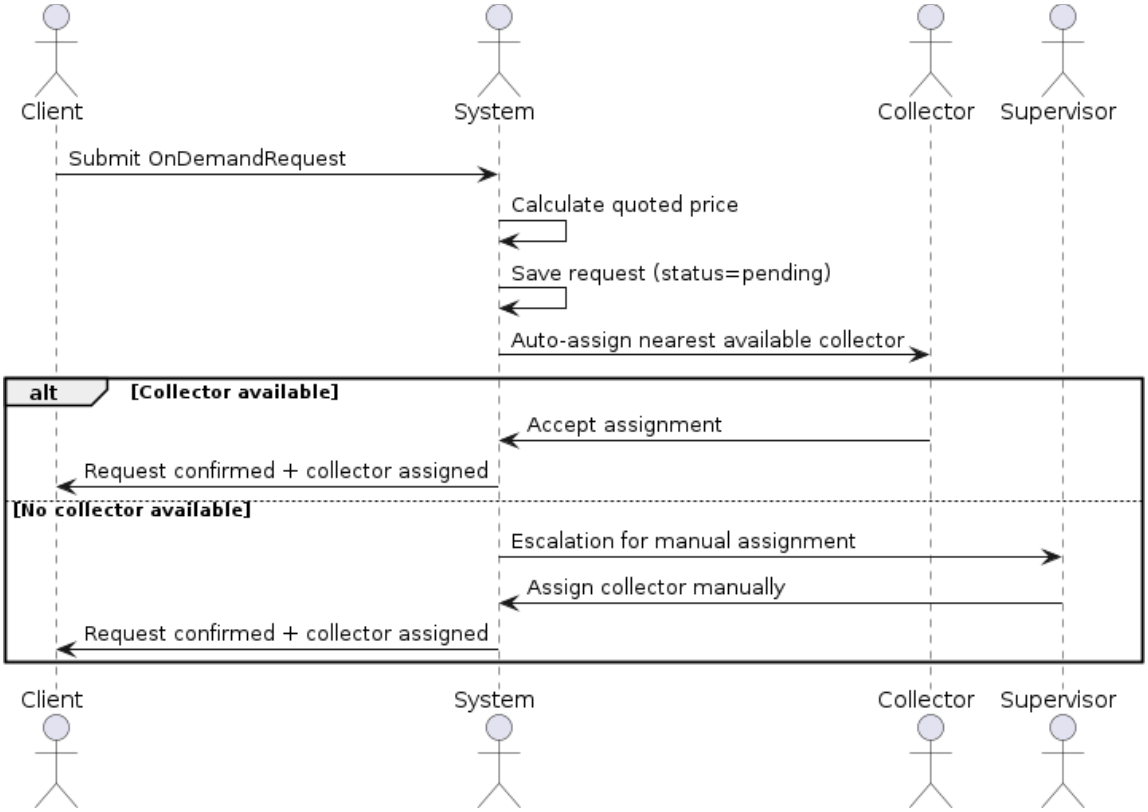
Operational Workflows

1. Client Workflows

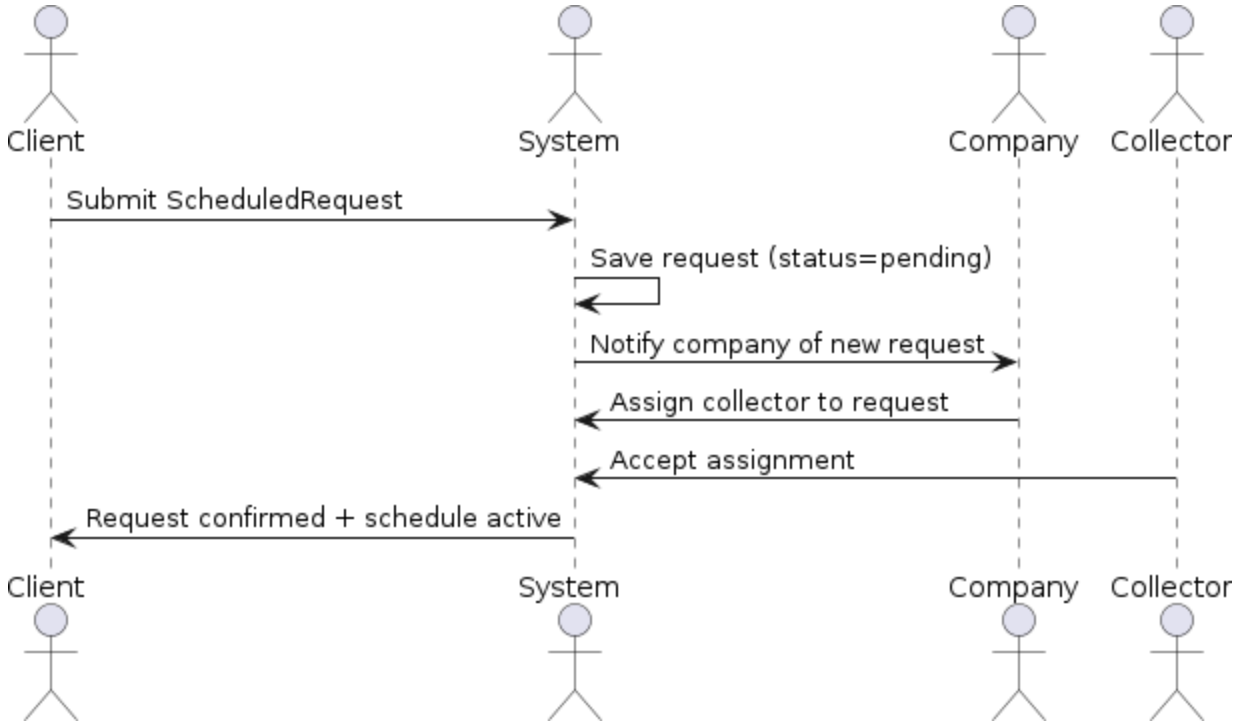
- **Account Registration**: Client creates a user account.



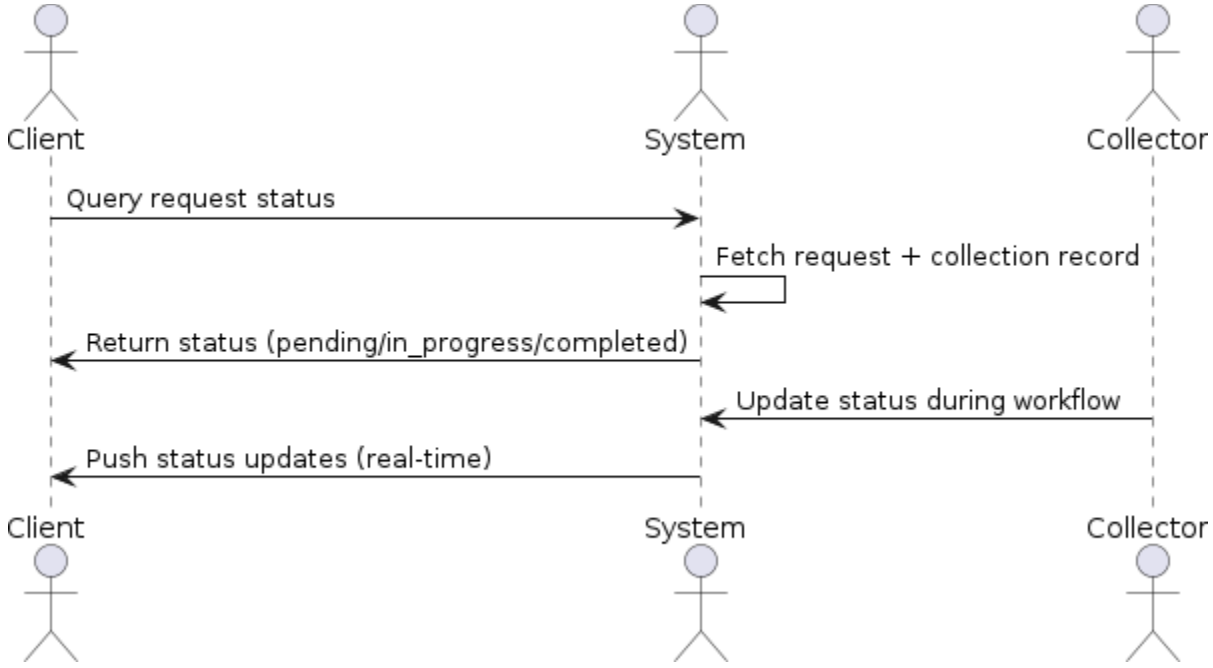
- **Create On-Demand Request**: Client submits one-time pickup request.



- **Create Scheduled Request**: Client sets up recurring pickups.

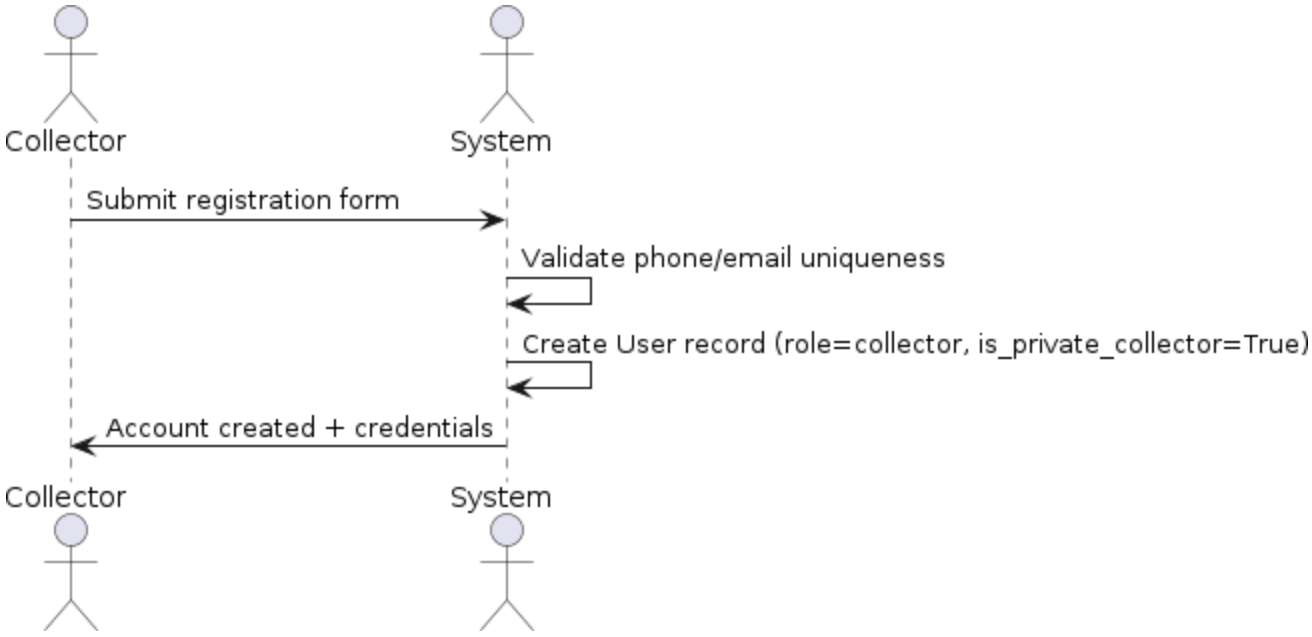


- **Track Request Status**: Client monitors progress of their requests.

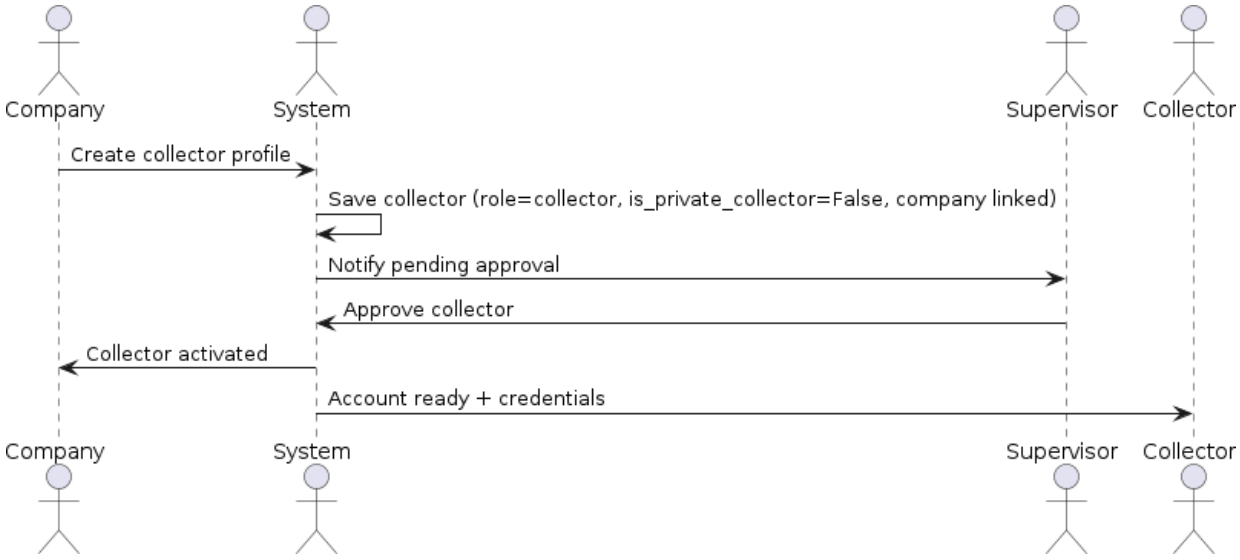


2. Collector Workflows

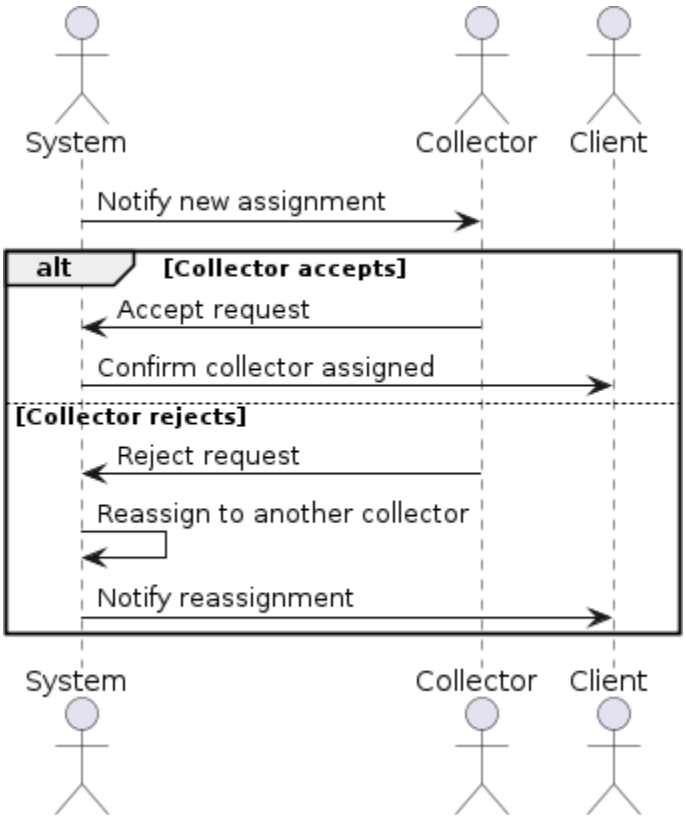
- **Private Collector Registration**: Collector self-registers as independent.



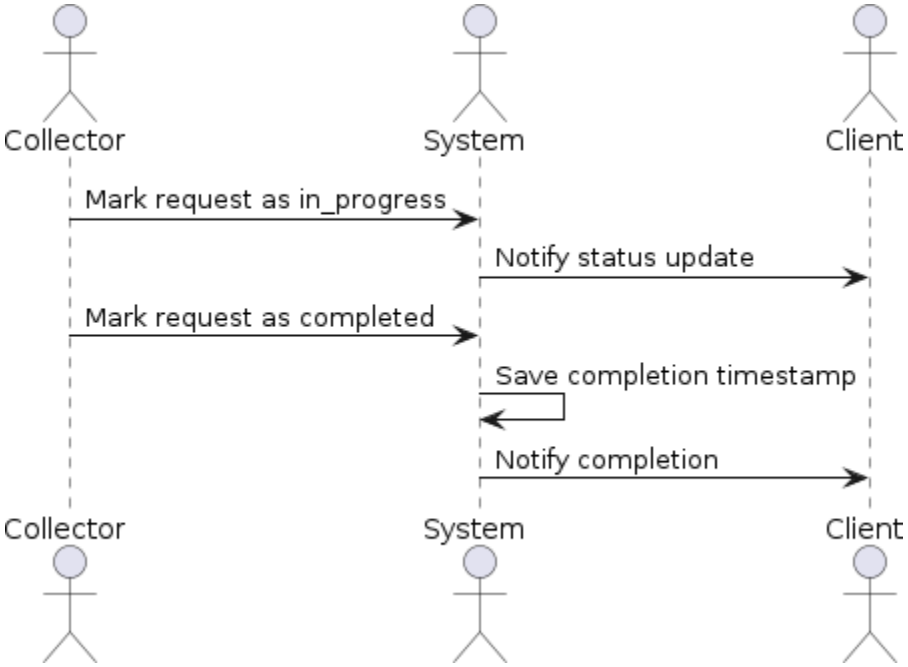
- **Company Collector Onboarding**: Collector created by company, approved by supervisor.



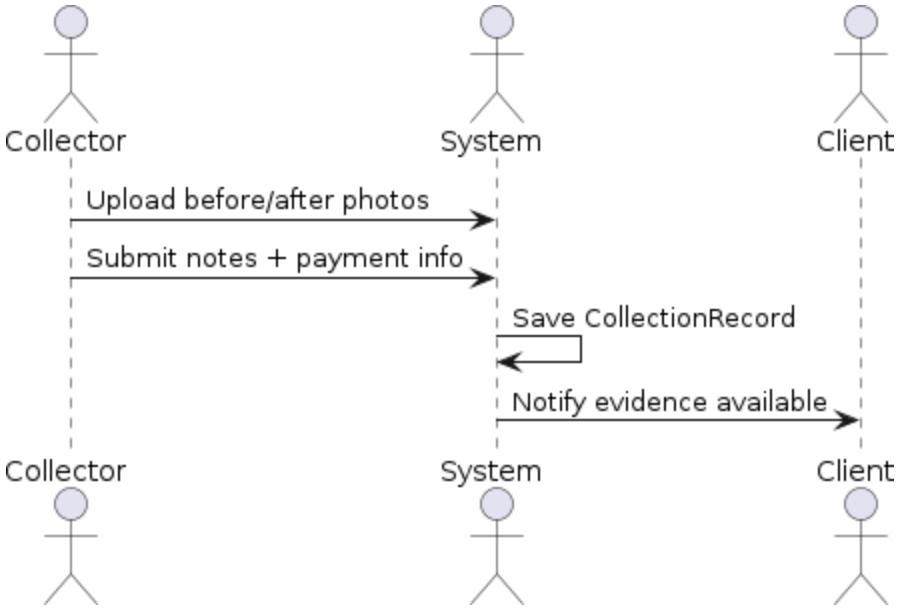
- **Accept Assigned Request**: Collector accepts on-demand or scheduled assignment.



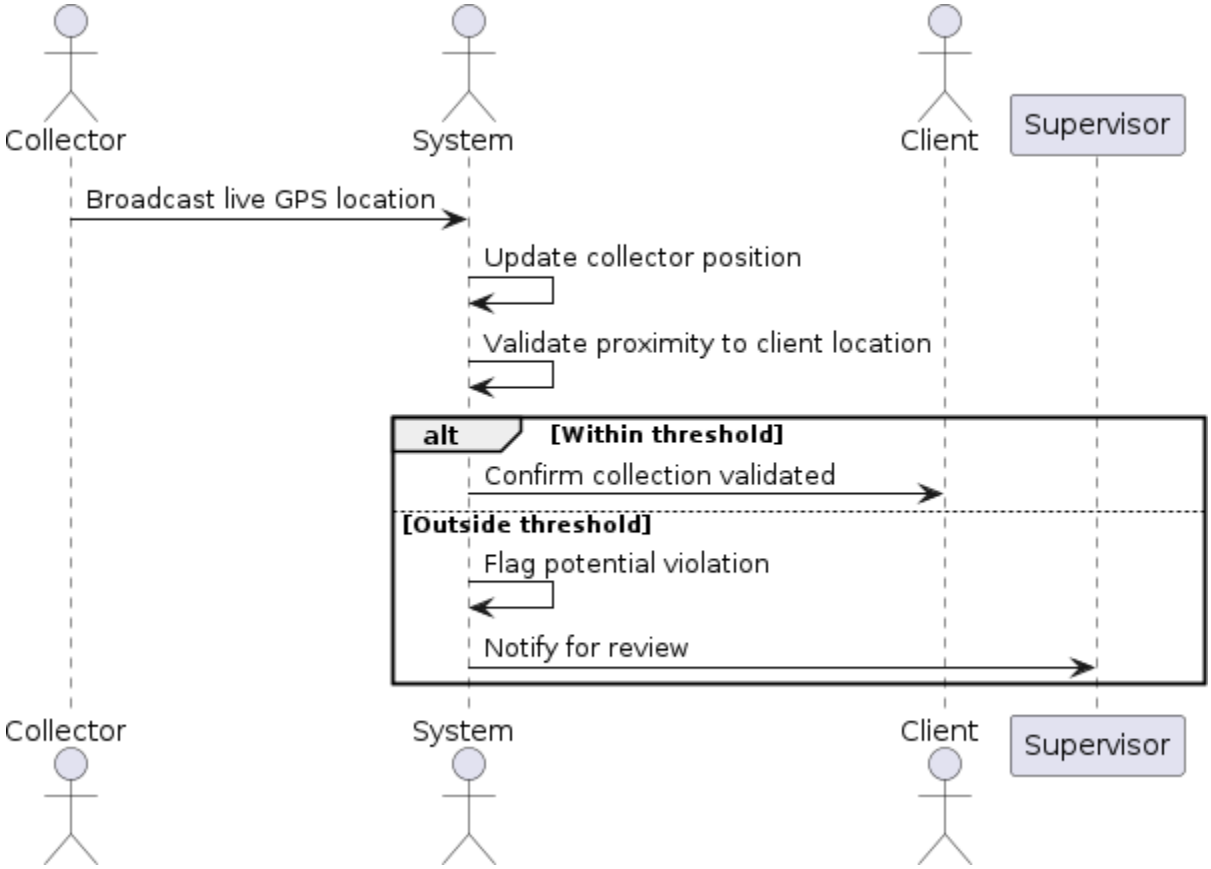
- **Update Collection Status**: Collector marks request as in-progress, completed, skipped, etc.



- **Upload Collection Evidence**: Collector submits before/after photos, notes, payment info.

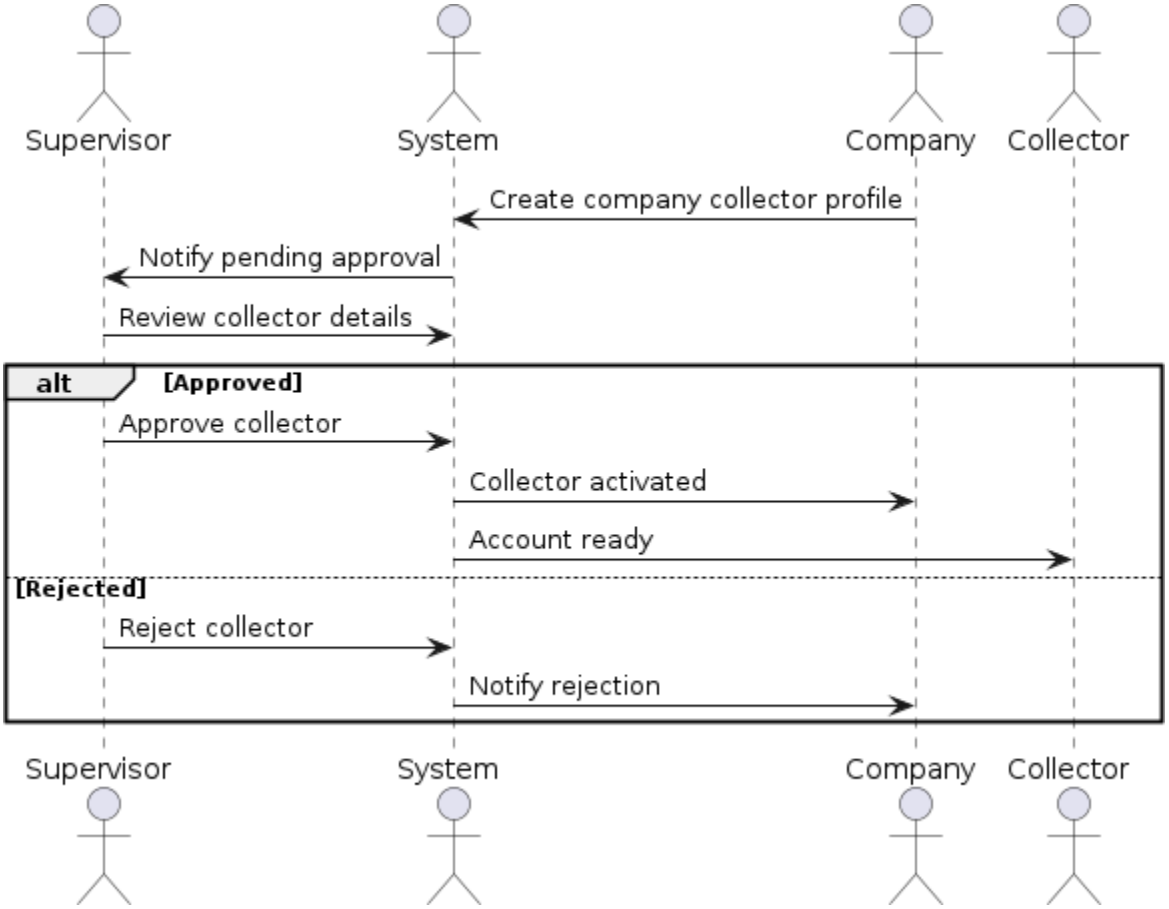


- **GPS Broadcast/Validation**: Collector shares live location; system validates completion proximity.

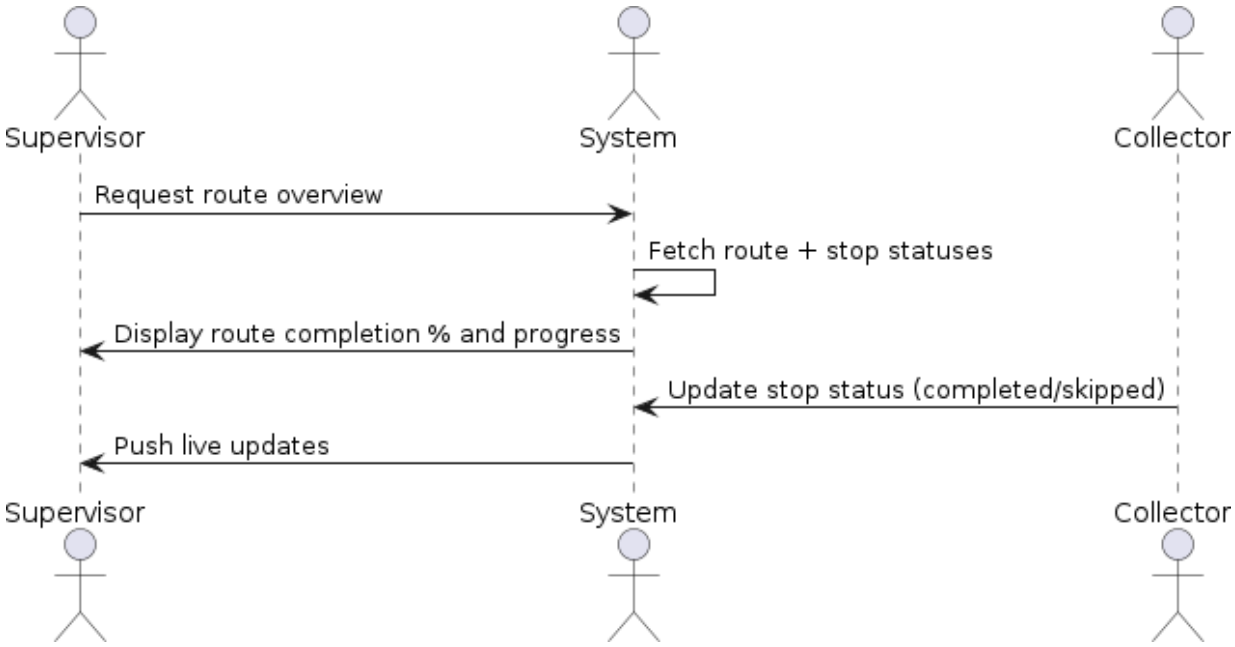


3. Supervisor Workflows

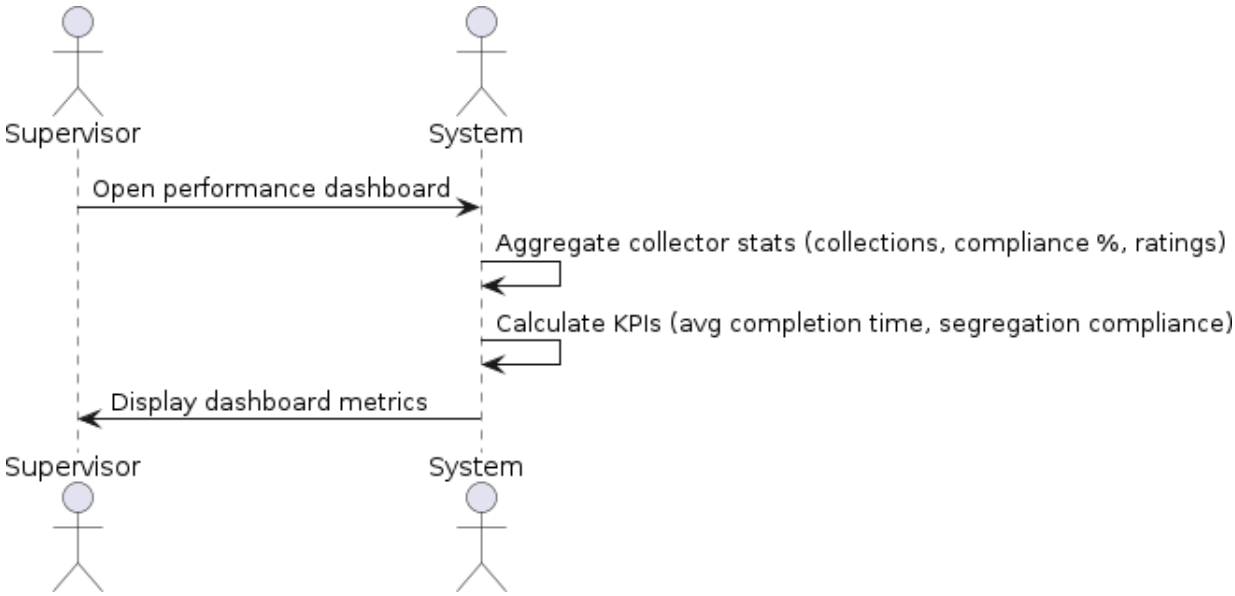
- **Approve Company Collector**: Supervisor validates new company collector accounts.



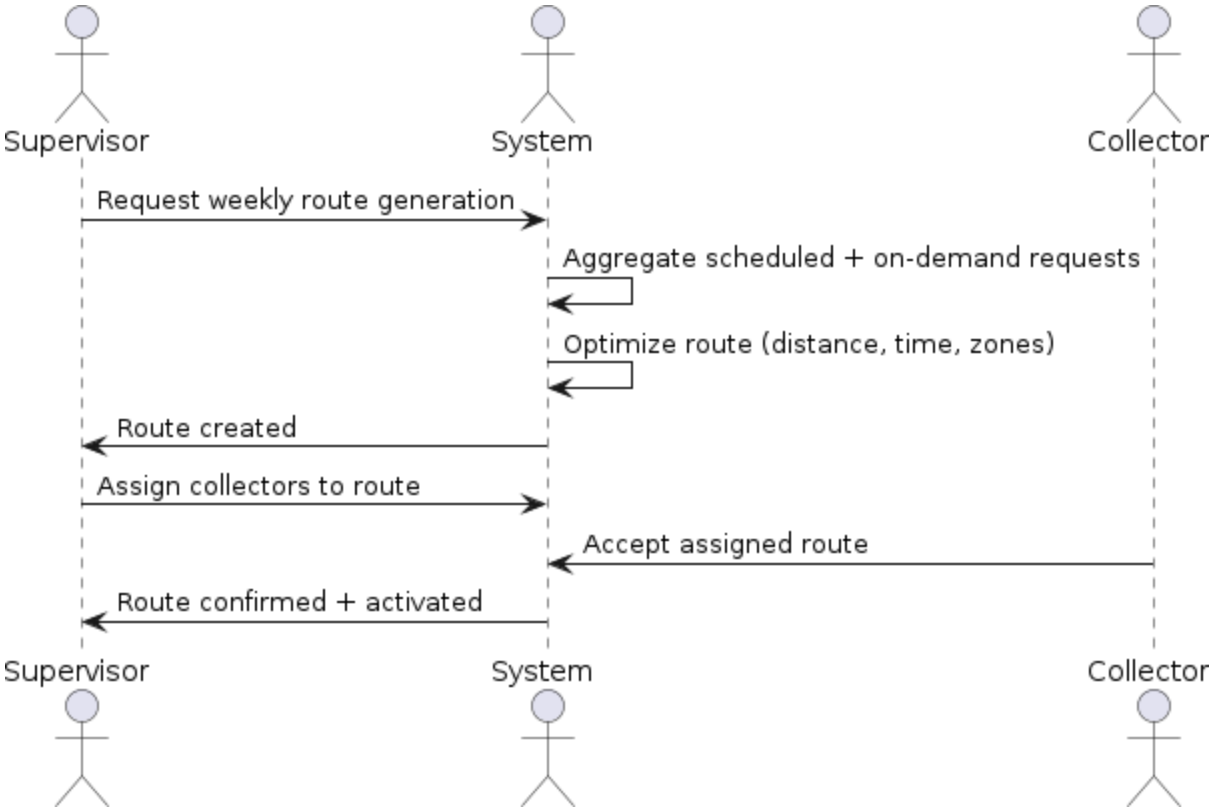
- **Monitor Routes**: Supervisor tracks route progress and completion percentage.



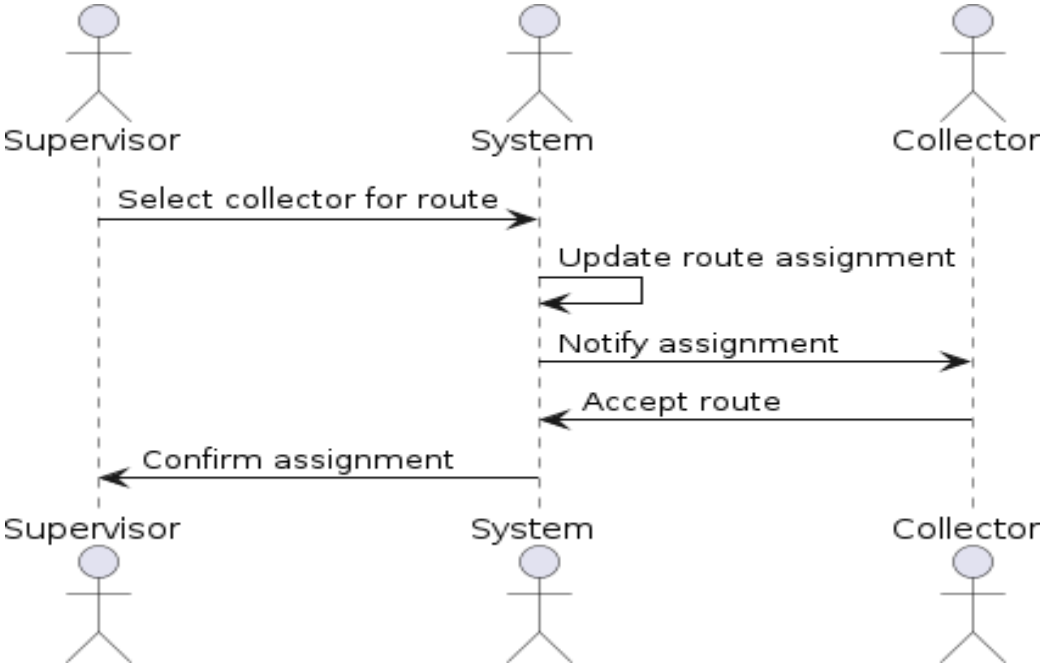
- **View Performance Dashboard**: Supervisor reviews collector stats, compliance, and KPIs.



- **Generate Weekly Routes**: Company generates routes with stops linked to requests.



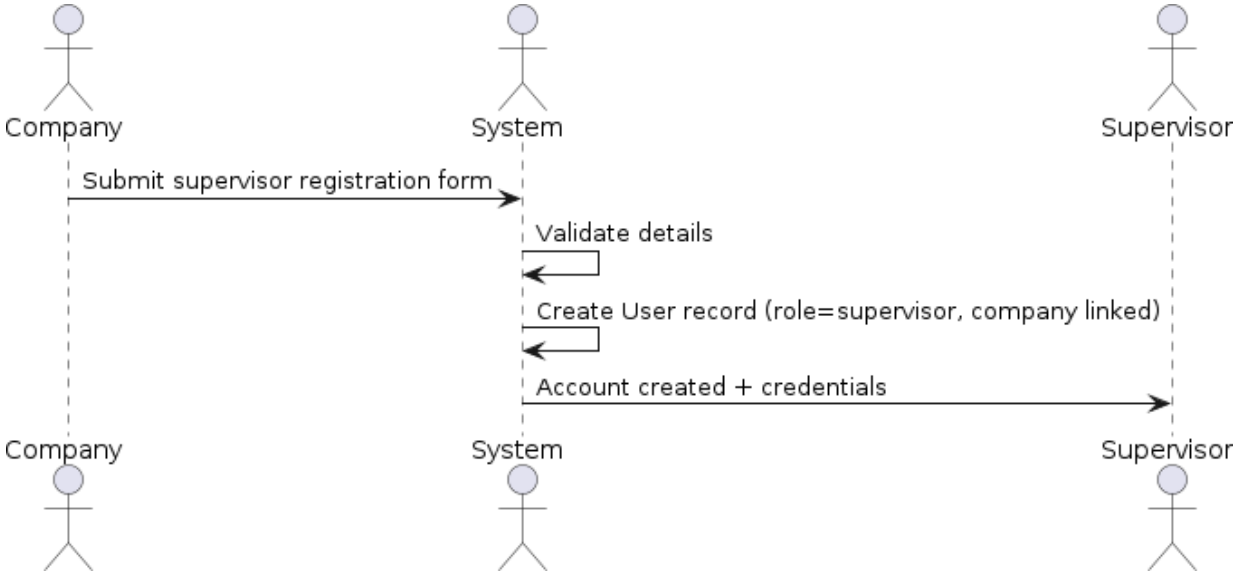
- **Assign Collectors to Routes**: Company assigns collectors to specific routes.



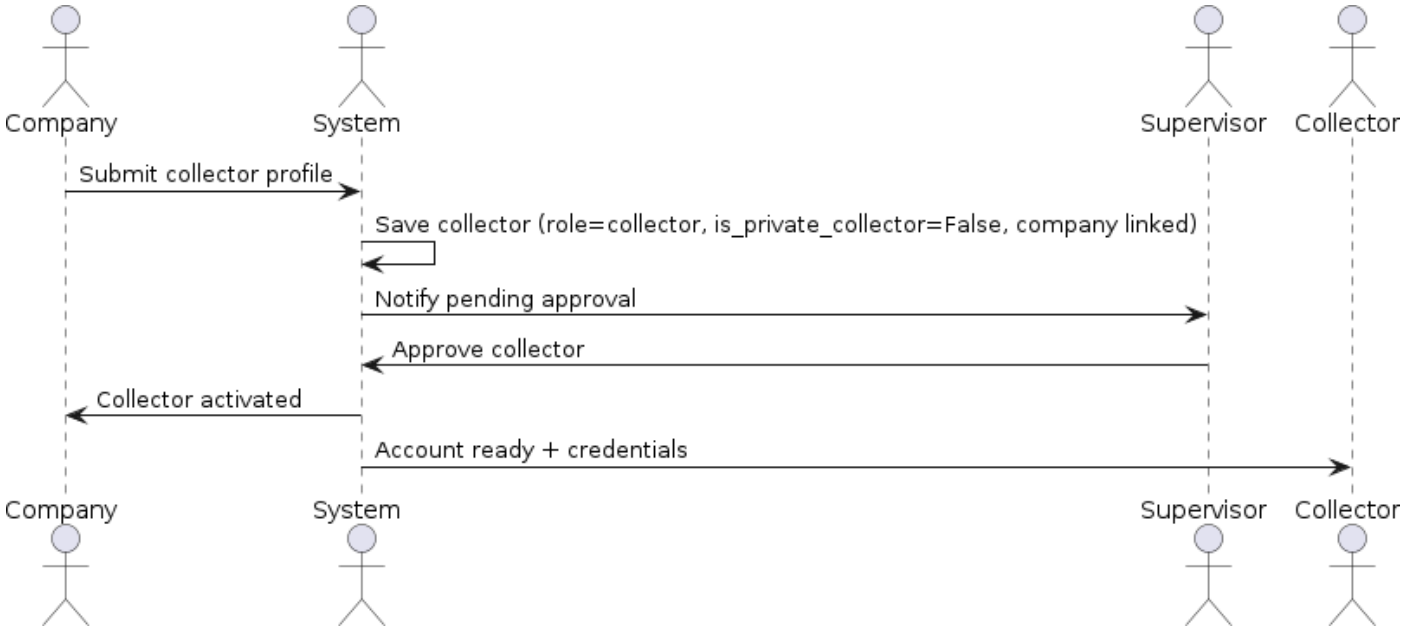
- **Handle Escalations**: Supervisor assigns requests manually if auto-assignment fails.

4. Company Workflows

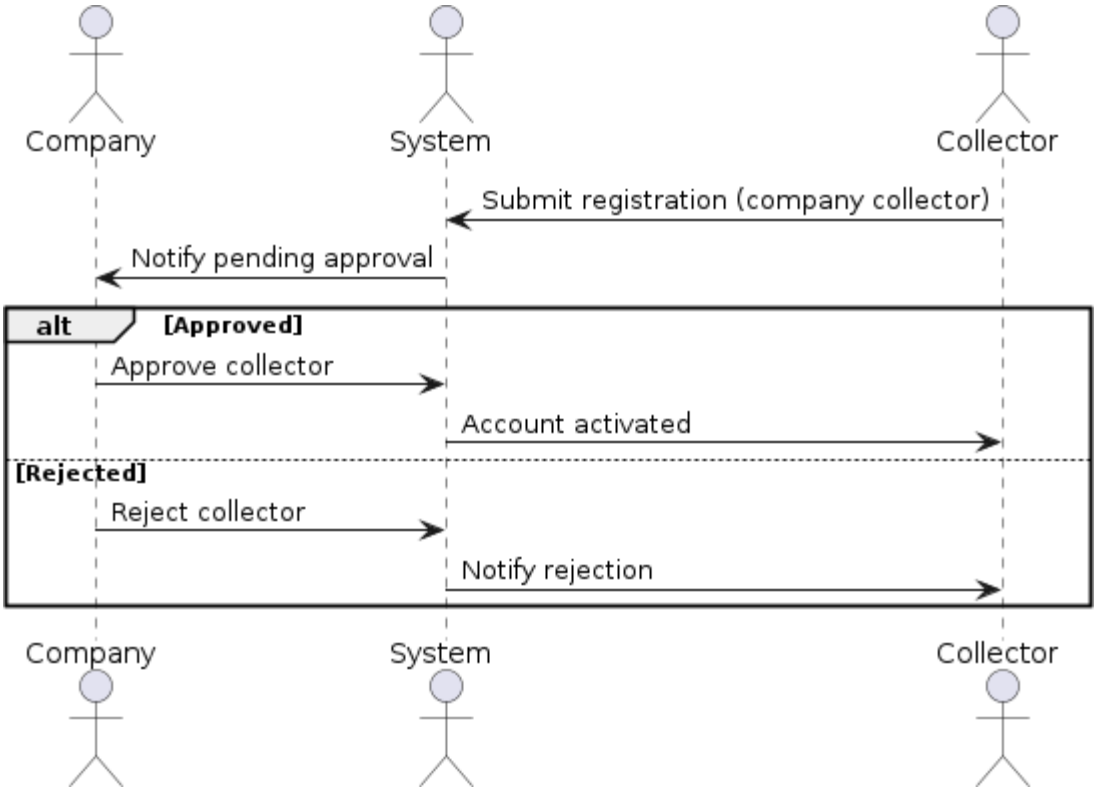
- **Create Supervisor**: Company registers supervisors.



- **Create Company Collector**: Company creates collector accounts

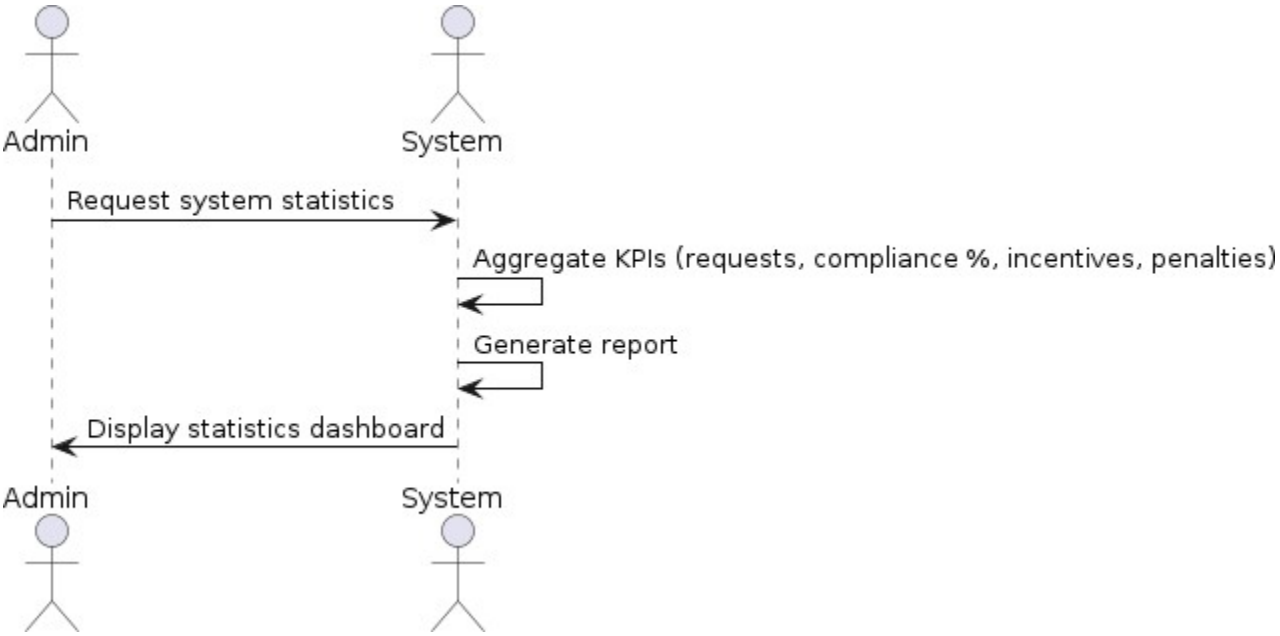


- **Approve Collector Registration**: Company validates pending collector accounts.



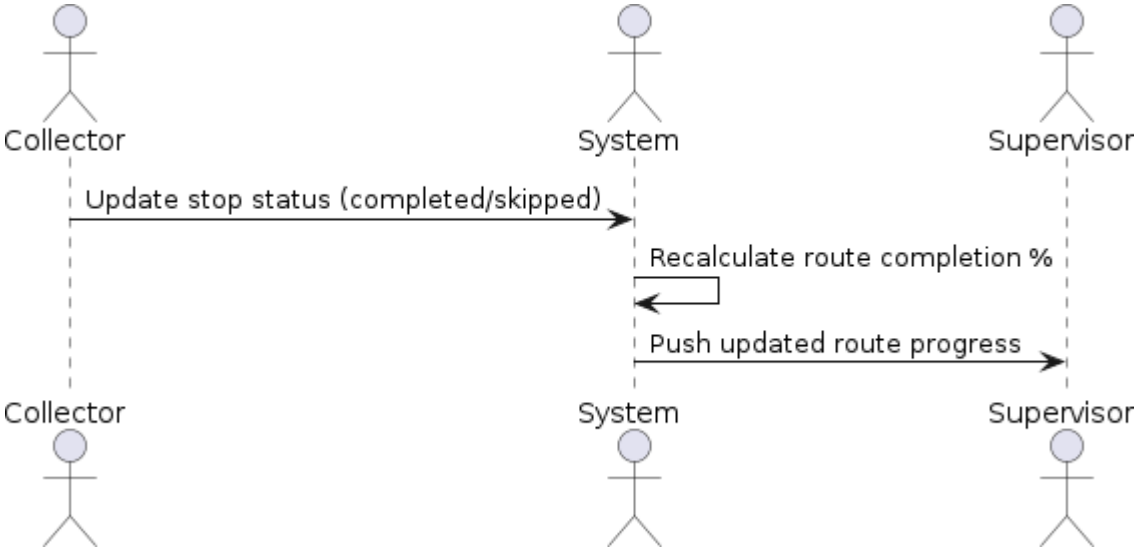
5. Admin Workflows

View System Statistics: Admin monitors system-wide KPIs and operational metrics.

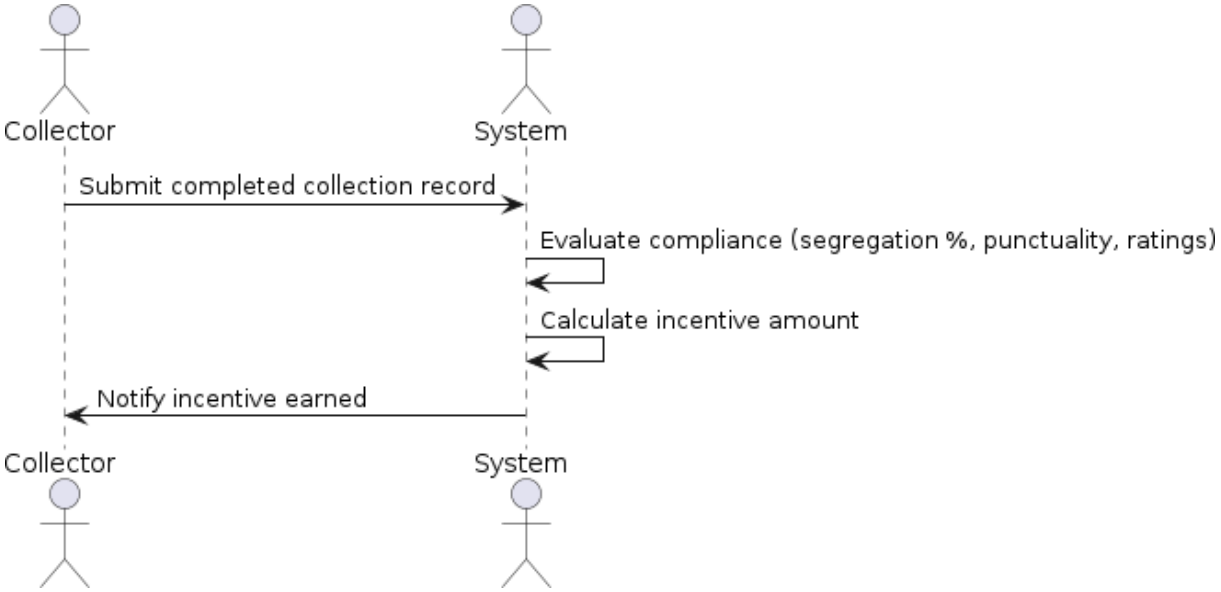


6. System Workflows (Automated)

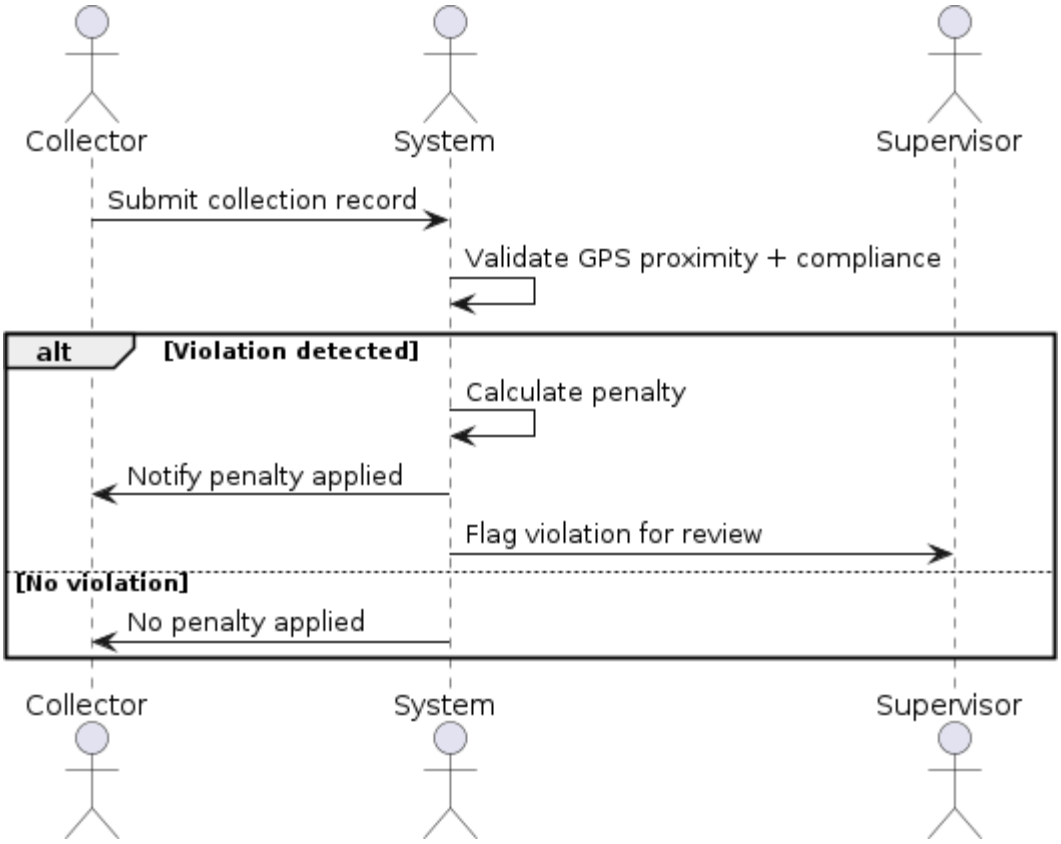
- **Route Completion Update**: System recalculates completion percentage based on stop statuses.



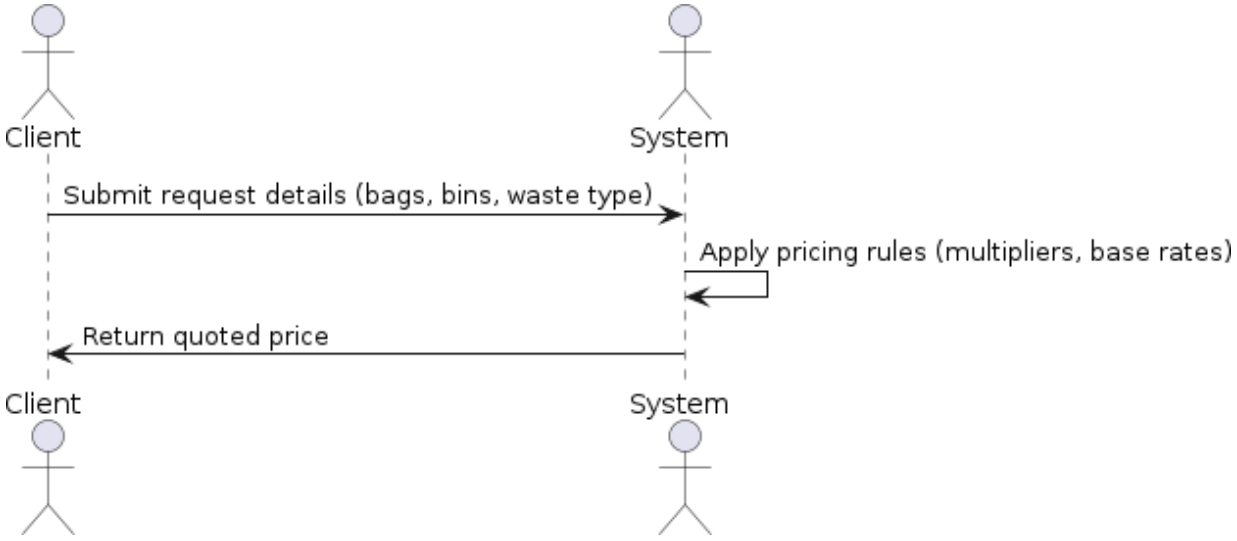
- **Incentive Calculation**: System rewards collectors based on compliance, punctuality, and ratings.



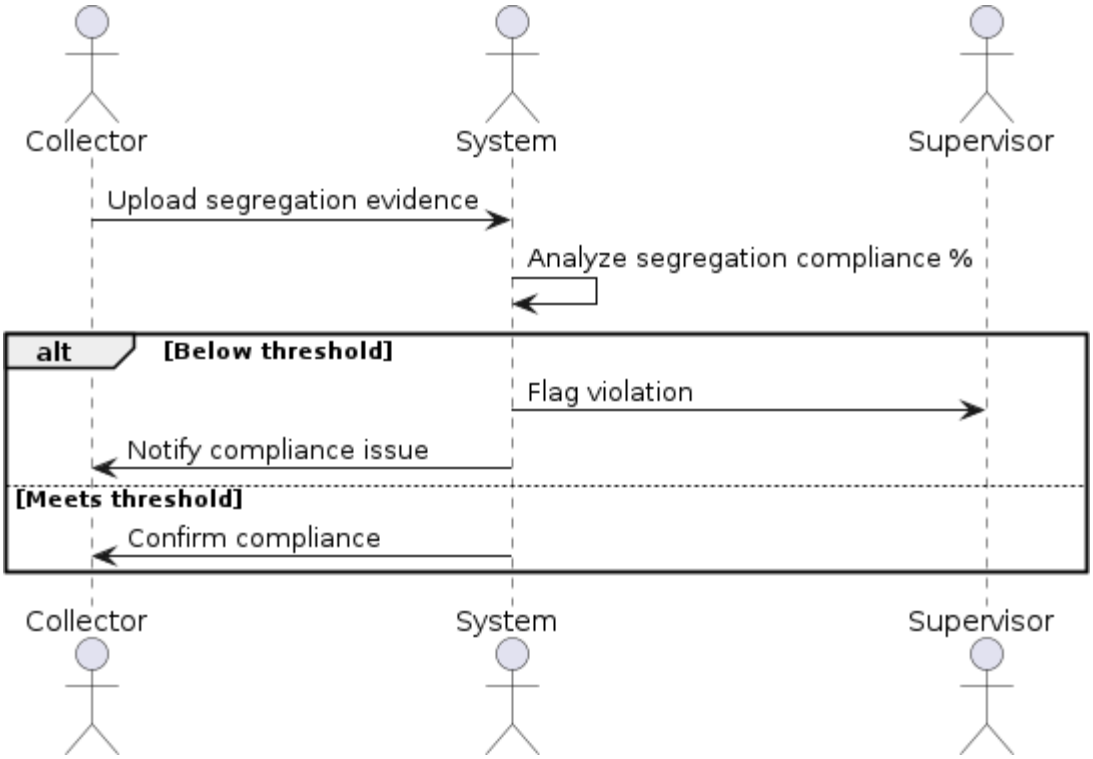
- **Penalty Calculation**: System penalizes collectors for missed stops, low compliance, or GPS mismatch.



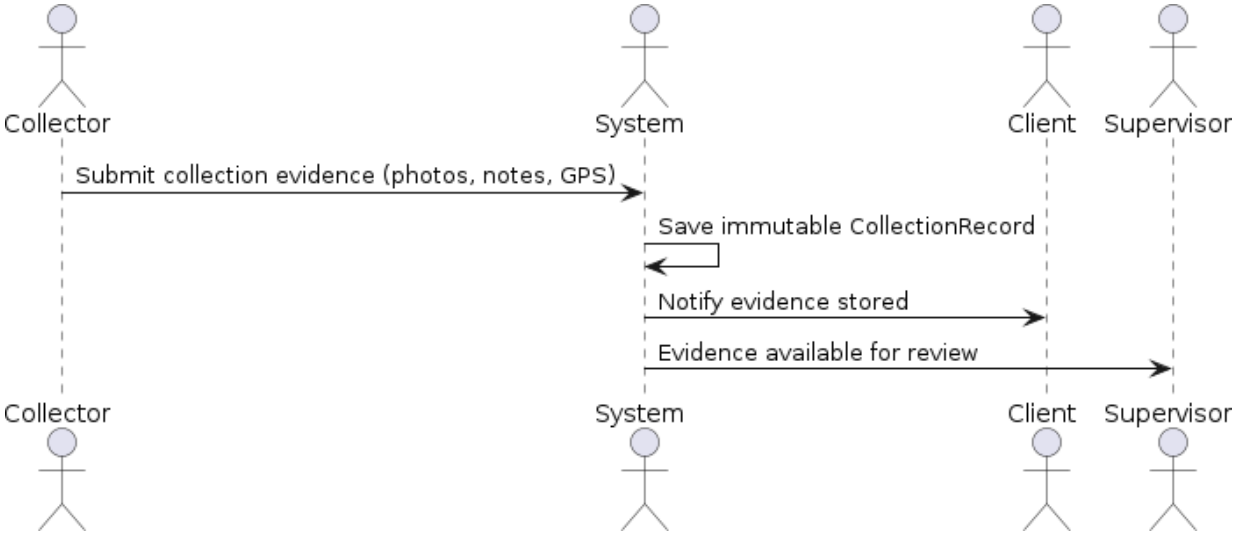
- **Pricing Calculation**: System computes quoted price based on bag/bin size and waste type multipliers.



- **Compliance Validation**: System checks segregation compliance and flags violations.



- **Audit Logging**: System records immutable collection evidence in CollectionRecord.



Future Work / Not Yet Achieved

This section outlines features and improvements that are **planned but not yet implemented**. These items represent potential extensions of the system

beyond the current project scope and are intended to guide future development.

Features Pending Implementation

Dynamic Pricing: The current system applies fixed pricing multipliers. A future enhancement would introduce demand-based pricing that adjusts costs dynamically based on factors such as time, location, and request volume.

Real-Time Incentive Calculation: Collector incentives are currently calculated in batch processes. A future goal is to compute incentives immediately after each completed collection.

Advanced Route Optimization: Route planning is currently zone-based. Future improvements may include AI-driven optimization across multiple zones or cities to improve efficiency.

Collector Ratings and Feedback Integration: Collector ratings are stored but not yet used in assignment logic. Future versions could incorporate ratings into automated assignment decisions.

Segregation Compliance Automation: Waste segregation evidence is manually reviewed. A future enhancement would involve automated image analysis to assist with compliance scoring.

Technical Enhancements

Scalability Improvements: The current architecture is designed to support approximately 10,000 requests per day per zone. Future iterations may target significantly higher volumes using distributed or microservice-based architectures.

Hybrid Technology Stack: While Django currently handles all business logic, future versions may introduce Go-based microservices for high-frequency operations such as real-time GPS tracking.

Offline Support: Collectors currently require continuous connectivity. A future improvement would allow offline data capture with synchronization once connectivity is restored.

Operational Improvements

Supervisor Dashboards: Current dashboards provide basic performance indicators. Future enhancements may include predictive analytics for missed stops, compliance trends, and workload forecasting.

Company Insights: Companies can currently view collector lists. Future versions may provide aggregated company-wide analytics and performance reports.

Client Experience Enhancements: While a wallet system exists, future improvements may include loyalty programs, subscription tiers, and push notifications.

End of Documentation