

Samsara Asset Tag Location Estimation System

Design Document

Author: Principal Engineer, Smart Trailers & Connected Equipment

Date: October 2, 2025

Status: Initial Design

Reviewers: Engineering Leadership, Product Management

1. Executive Summary

This document presents the architecture for Samsara's Asset Tag tracking platform. The system ingests BLE observations from millions of gateways and produces accurate, near-real-time asset locations with uncertainty estimates every ~1 minute. Beyond core location estimation, the platform enables operational workflows (maintenance, compliance, job costing), intelligent features (anomaly detection, predictive alerts), and ecosystem integrations (APIs, ERP/CMMS, marketplace).

Key Design Decisions:

- **Stream processing architecture** using Kafka for high-throughput observation ingestion
- **Time-window based aggregation** to batch observations for location calculation
- **Trilateration with RSSI-based distance estimation** as primary location algorithm
- **Microservices architecture** with domain-driven design for workflows
- **Event-driven integration** pattern for ecosystem extensibility
- **99.9% uptime target** with multi-region deployment and graceful degradation

System Scope:

- **Core (this doc focus):** Location estimation pipeline, geofencing, historical queries
 - **Extended:** Workflow services, analytics, integrations (architecture overview provided)
 - **Out of scope:** Hardware/firmware design, mobile app UI implementation details
-

2. Functional Requirements

2.1 Core Location Estimation (Objective 1: Source of Truth)

FR1: Observation Ingestion

- Ingest BLE observations from millions of Vehicle Gateways (VGs) and Asset Gateways (AGs)
- Each observation contains: `{asset_tag_id, gateway_id, timestamp, rssi, battery_level}`
- Support peak ingestion rate of 100K observations/second
- Handle out-of-order messages with up to 5-minute clock skew

FR2: Gateway Location Tracking

- Consume GPS streams from gateways: `{gateway_id, timestamp, lat, lng, speed, accuracy}`
- Update gateway locations every few seconds
- Maintain last-known location for up to 24 hours

FR3: Location Estimation

- Compute asset location every 60 seconds (± 10 s tolerance)
- Produce output: `{asset_tag_id, timestamp, lat, lng, uncertainty_radius_meters, confidence_score}`
- Handle scenarios: 3+ gateways (trilateration), 2 gateways (midpoint), single gateway, no recent observations

FR4: Location Quality Assessment

- Calculate uncertainty radius based on:
 - Number of observing gateways
 - RSSI signal strength distribution
 - Gateway GPS accuracy
 - Time since last observation
 - Environment type (indoor/outdoor)
- Provide confidence score (0-100) with algorithm transparency

FR5: Battery Life Management

- Monitor battery levels from Asset Tag telemetry
- Calculate estimated end-of-life based on usage patterns
- Generate proactive alerts when battery < 20% or < 30 days remaining
- Support 4-year battery lifecycle tracking

FR6: System Reliability

- 99.9% uptime for location estimation service
- <5s P99 latency from observation to location update
- Zero data loss (at-least-once delivery guarantee)
- Automatic failover and recovery

FR7: Security & Privacy

- Validate encrypted BLE signals with rotating device IDs
- Prevent spoofing and unauthorized tracking
- Role-based access control (RBAC) per customer organization
- Audit logging for all location access

- Zero major security incidents target

2.2 Geofencing & Alerting (Objective 3: Intelligence)

FR8: Geofence Management

- Support customer-defined geofences (circular, polygonal, custom boundaries)
- Enable multiple geofence types: authorized zones, restricted zones, job sites
- Real-time geofence evaluation (entry/exit detection within 2 minutes)
- Geofence rule configuration (alerts, notifications, workflows)

FR9: Anomaly Detection

- Detect off-hour movement (movement outside business hours)
- Identify unusual movement patterns (abnormal speed, unexpected routes)
- Flag prolonged idling or inactivity anomalies
- Detect unauthorized zone entry
- ML-based baseline learning per asset

FR10: Predictive Alerts

- Generate theft risk alerts based on anomaly detection
- Provide "lost contact" notifications when asset goes offline unexpectedly
- Send predictive maintenance alerts based on usage patterns
- Deliver underutilized asset recommendations (assets idle >30 days)

FR11: Alert Delivery

- Multi-channel notifications (push, SMS, email, webhook)
- Configurable alert priorities (critical, high, medium, low)

- Alert throttling and deduplication
- Customer-configurable alert rules and thresholds

2.3 Operational Workflows (Objective 2: Operations)

FR12: Asset Onboarding & Identity

- Support asset registration with metadata (type, owner, group, project)
- Asset type classification (tools, trailers, heavy equipment, containers)
- Bulk onboarding via CSV import or API
- ERP-driven auto-onboarding (sync with CMMS/ERP systems)
- Barcode/RFID dual-tag support for inventory verification

FR13: Asset Finding (Mobile)

- "Find Nearby" feature using BLE signal strength visualization
- AR Precision Finding with augmented reality guidance on mobile
- Audible beaconing (trigger tag to emit sound/vibration for indoor finding)
- Distance and direction indicators to guide users to assets
- Works offline with cached gateway locations

FR14: Vehicle-Asset Pairing

- Automatic detection when asset loaded onto vehicle (tag-gateway proximity)
- Load-out verification (confirm all required assets present before departure)
- Automatic unpairing when asset separated from vehicle
- Load manifest generation for drivers
- Integration with vehicle telematics data

FR15: Asset Check-in/Check-out

- Digital check-out workflow (assign asset to user/project)
- Check-in workflow with condition assessment
- Custody chain tracking (who has which asset)
- Transfer recommendations between users/locations
- Overdue asset notifications

FR16: Maintenance Workflows

- Predictive maintenance triggers based on usage hours/cycles
- Maintenance scheduling tied to asset tags
- Digital inspection forms linked to assets
- Maintenance history and work order integration
- Service due reminders based on time or usage thresholds

FR17: Compliance Workflows

- Track inspection dates and certification expiry
- Compliance audit reports (proof of asset location at time T)
- Regulatory inspection reminders (OSHA, DOT, etc.)
- Digital compliance logs with tamper-proof timestamps
- Asset-specific compliance rules (e.g., safety equipment inspections)

FR18: Job Costing & Utilization

- Track asset time-on-site per project/job
- Automatic project allocation based on geofence entry
- Utilization reporting (active vs. idle time)

- Cost allocation for asset usage (rental equivalent calculations)
- Idle time identification and recommendations

2.4 Insights & Reporting (Objectives 2 & 3)

FR19: Historical Data Access

- Store location history for minimum 90 days (with tiered retention)
- Query APIs: location at time T, path between T1-T2, dwell time analysis
- Location playback with timeline scrubbing
- Export to CSV/GeoJSON for external analysis

FR20: Inventory & Utilization Reports

- Real-time inventory audit (all assets, locations, statuses)
- Asset utilization metrics (% time active, idle, in-transit)
- Underutilized asset recommendations (ML-driven)
- Asset movement frequency analysis
- Ghost asset detection (tags not seen in X days)

FR21: ROI Dashboard

- Theft/loss prevention savings calculation
- Labor time saved (reduced search time)
- Utilization improvement metrics
- Insurance cost reduction tracking
- Total cost of ownership (TCO) analysis

FR22: Advanced Analytics

- Faceted search (filter by owner, group, region, geofence, status, asset type)
- Saved views and custom dashboards
- Heatmaps of asset density and movement patterns
- Root-cause analysis for theft/loss events
- Predictive analytics for asset lifecycle

FR23: Insurance & Compliance Reports

- Theft/loss incident reports (location history, timeline, evidence)
- Asset security documentation for insurance claims
- Audit trails for regulatory compliance
- Tamper-proof location logs
- Chain of custody reports

2.5 Integration & Ecosystem (Objective 4: Ecosystem)

FR24: APIs & SDKs

- RESTful APIs for all asset data operations (CRUD, queries, alerts)
- GraphQL API for flexible data queries
- SDKs in JavaScript/TypeScript, Python, and Java
- Comprehensive API documentation with code examples
- API rate limiting and authentication (OAuth 2.0)

FR25: Webhooks & Event Streaming

- Real-time webhooks for key events:
 - Location updates
 - Geofence entry/exit

- Battery low alerts
- Lost mode triggered
- Custody changes
- Configurable event filters and delivery guarantees
- Webhook retry logic with exponential backoff

FR26: ERP/CMMS Integrations

- Pre-built integrations with major systems:
 - SAP (inventory sync, work orders)
 - ServiceNow (CMMS integration)
 - Procore (construction management)
 - Oracle (ERP, financials)
 - Microsoft Dynamics 365
 - Fiix (maintenance management)
- Bi-directional data sync
- Field mapping customization

FR27: Partner Marketplace

- Curated marketplace with certified partner apps
- App discovery, installation, and configuration UI
- Partner revenue sharing model
- App sandboxing and security review
- Rating and review system

FR28: Third-Party Gateway Support

- Support for non-Samsara BLE gateways (Geotab, Zebra, generic BLE scanners)
- Gateway registration and certification process
- Protocol adapters for different BLE standards
- Quality scoring for third-party observations

2.6 User Experience (Objective 5: Delight)

FR29: Mobile App Features

- Native mobile apps (iOS, Android) for field users
- Offline mode with cached asset locations
- Voice search ("Find the generator")
- Turn-by-turn navigation to asset location
- Quick actions (check-out, report issue, trigger beacon)

FR30: AR Precision Finding

- Camera-based AR overlay showing asset direction and distance
- Works with standard smartphone cameras (ARKit/ARCore)
- Visual indicators (arrows, distance counter, proximity alerts)
- Accuracy within 5 meters for close-range finding
- Haptic feedback as user approaches asset

FR31: Performance & Usability

- Reduce average time-to-locate by 60% (baseline: 10 min → target: 4 min)
- Mobile app load time <2 seconds
- Map rendering <1 second for 10,000 assets
- 80%+ positive rating on mobile UX (in-app surveys)

- Accessibility compliance (WCAG 2.1 Level AA)

FR32: Notifications & Communication

- In-app notifications with action buttons
- Smart notification batching (avoid alert fatigue)
- Do-not-disturb schedules
- Notification history and archive
- Team collaboration features (comments, @mentions)

2.7 Administration & Configuration

FR33: Multi-Tenancy & Permissions

- Organization-level isolation (customer data segregation)
- Role-based permissions (admin, manager, operator, viewer)
- Fine-grained access control (per asset, geofence, report)
- SSO integration (SAML, OIDC)
- Audit logs for all administrative actions

FR34: System Configuration

- Configurable location update frequency (30s, 60s, 120s)
 - RSSI-to-distance calibration per environment
 - Alert threshold customization per customer
 - Retention policy configuration (storage cost management)
 - API rate limits per organization
-

3. Non-Functional Requirements

3.1 Scalability (All Objectives)

NFR1: Volume

- Support 5M active Asset Tags (10M peak)
- Support 10M active gateways (20M peak)
- Handle 100K observations/second sustained, 500K peak
- Support 100K concurrent API requests
- Linear horizontal scaling for all services

NFR2: Data Growth

- 10% month-over-month customer growth
- 100TB+ total storage capacity
- Automatic storage tiering (hot/warm/cold)
- Data lifecycle management with configurable retention

NFR3: Geographic Distribution

- Multi-region deployment (US, EU, APAC)
- <100ms latency for API queries within region
- Cross-region data replication for disaster recovery
- Region-aware data residency for compliance (GDPR)

3.2 Availability & Reliability (Objective 1)

NFR4: Uptime

- 99.9% uptime SLA (43.8 min/month downtime budget)

- 99.99% uptime for critical path (location estimation)
- Zero data loss for observations and locations
- Automatic failover within 30 seconds

NFR5: Latency

- Location update: P50 <2s, P99 <5s from observation to database
- API queries: P95 <500ms for current location
- Historical queries: P95 <2s for 24h of data, P99 <10s for 90d
- Geofence evaluation: <2 minutes from location to alert
- Dashboard load: P95 <3s for 10K assets on map

NFR6: Fault Tolerance

- Graceful degradation when subsystems fail
- Circuit breakers on all external dependencies
- Retry logic with exponential backoff
- Dead letter queues for failed message processing
- Automatic rollback on deployment failures

3.3 Performance (Objective 5)

NFR7: Throughput

- Location estimation: 5M assets every 60 seconds = 83K/sec sustained
- API serving: 10K requests/sec per region
- Webhook delivery: 50K/sec event publishing
- Report generation: 1000 concurrent reports

NFR8: Resource Efficiency

- CPU utilization target: 60-70% average (headroom for spikes)
- Memory utilization: <80% to prevent OOM
- Database connections: connection pooling, <5K connections/instance
- Cache hit ratio: >90% for hot data (recent locations)

3.4 Data Integrity & Security (Objectives 1 & 7)

NFR9: Data Quality

- Accurate timestamps (NTP synchronized, <100ms drift)
- Handle out-of-order observations (up to 5 min clock skew)
- Idempotent processing for retry scenarios
- Data validation at ingestion (schema enforcement)

NFR10: Security

- Encrypted BLE signals with rotating device IDs (prevent replay attacks)
- TLS 1.3 for all data in transit
- AES-256 encryption for data at rest
- API authentication via OAuth 2.0 / JWT
- Regular security audits and penetration testing
- Zero-trust network architecture

NFR11: Privacy & Compliance

- GDPR compliance (data portability, right to erasure)
- SOC 2 Type II certification

- HIPAA compliance for healthcare customers (optional)
- Data residency controls per jurisdiction
- PII masking in logs and non-production environments

3.5 Cost Efficiency (All Objectives)

NFR12: Target Costs

- \$0.10/asset/month for compute + storage (at 5M scale)
- 70% of compute on spot instances or autoscaling
- Tiered storage reducing cost by 60% for cold data
- Optimized data compression (70-80% reduction for time-series)

NFR13: Resource Optimization

- Autoscaling based on demand (time-of-day, customer onboarding)
- Batch processing for non-critical workflows (nightly reports)
- Caching to reduce database load (90%+ cache hit rate)
- Query optimization (indexed queries, materialized views)

3.6 Observability & Operations

NFR14: Monitoring

- Real-time metrics (latency, throughput, error rates)
- Distributed tracing (request flow across services)
- Centralized logging (indexed, searchable)
- Alerting on SLO violations (PagerDuty/OpsGenie)
- Business metrics dashboards (asset count, location updates/min)

NFR15: Maintainability

- <4 hour MTTR (mean time to recovery) for incidents
 - <2 hour deployment cycle (CI/CD pipeline)
 - Zero-downtime deployments (blue-green or canary)
 - Automated rollback on error rate spikes
 - Weekly release cadence
-

4. Resource Estimation

4.1 Storage Requirements

Observation Data (Raw, 7-day retention):

- Observations per asset: 72/hour (assuming 10% gateway coverage)
- Record size: 100 bytes
- Daily per asset: $72 \times 24 \times 100 \text{ bytes} = \mathbf{173 \text{ KB}}$
- For 5M assets: **865 GB/day = 6 TB** (7-day retention)

Computed Locations (90-day retention):

- Locations per asset: 1440/day (1 per minute)
- Record size: 150 bytes
- Daily per asset: $1440 \times 150 = \mathbf{216 \text{ KB}}$
- For 5M assets: **1.08 TB/day = 97 TB** (90 days, with tiering)

Gateway Locations (24-hour retention):

- $10\text{M gateways} \times 1 \text{ location}/3\text{s} = 3.33\text{M}/\text{sec}$

- Daily: $3.33\text{M} \times 120 \text{ bytes} \times 86400 = \mathbf{34.5 \text{ TB/day}}$
- With compression: $\sim \mathbf{7 \text{ TB}}$ (time-series DB)

Workflow & Metadata:

- Asset metadata: $5\text{M} \times 2 \text{ KB} = \mathbf{10 \text{ GB}}$
- Geofence definitions: $100\text{K geofences} \times 10 \text{ KB} = \mathbf{1 \text{ GB}}$
- Maintenance records: $5\text{M} \times 10 \text{ records} \times 1 \text{ KB} = \mathbf{50 \text{ GB}}$
- Check-in/out logs: $1\text{M events/day} \times 500 \text{ bytes} \times 90\text{d} = \mathbf{45 \text{ GB}}$

Analytics & Aggregations:

- Materialized views (daily/hourly rollups): $\mathbf{5 \text{ TB}}$
- ML model data (anomaly detection baselines): $\mathbf{2 \text{ TB}}$

Total Storage: $\sim \mathbf{120 \text{ TB}}$

- Hot (0-7d): 20 TB (SSD)
- Warm (7-30d): 40 TB (SSD/HDD)
- Cold (30-90d): 60 TB (S3)

4.2 Bandwidth Requirements

Ingress:

- Gateway observations: $100\text{K obs/sec} \times 100 \text{ bytes} = \mathbf{10 \text{ MB/s}} = 80 \text{ Mbps}$
- Gateway GPS: $3.33\text{M/sec} \times 120 \text{ bytes} = \mathbf{400 \text{ MB/s}} = 3.2 \text{ Gbps}$
- API writes (asset updates, workflows): $\mathbf{50 \text{ MB/s}} = 400 \text{ Mbps}$
- **Total ingress: $\sim \mathbf{3.7 \text{ Gbps}}$** (5 Gbps with overhead)

Egress:

- Location writes (Kinesis, DB): $83\text{K/sec} \times 150 \text{ bytes} = \mathbf{12.5 \text{ MB/s}} = 100 \text{ Mbps}$
- API queries: $10\text{K req/sec} \times 10 \text{ KB avg} = \mathbf{100 \text{ MB/s}} = 800 \text{ Mbps}$
- Webhook events: $10\text{K events/sec} \times 1 \text{ KB} = \mathbf{10 \text{ MB/s}} = 80 \text{ Mbps}$
- Dashboard streaming: $\mathbf{50 \text{ MB/s}} = 400 \text{ Mbps}$
- **Total egress: ~1.4 Gbps**

4.3 Compute Requirements

Core Location Services:

- Location Estimation: $83\text{K assets/sec}, 1\text{K assets/min/worker} \rightarrow \mathbf{1000 \text{ instances}}$ (c6i.2xlarge, 8 vCPU)
- Observation Aggregation: $100\text{K obs/sec}, 50\text{K/worker} \rightarrow \mathbf{5 \text{ instances}}$ (c6i.4xlarge)
- Gateway Location Cache: $\mathbf{5 \text{ nodes}} \times 64 \text{ GB Redis cluster}$

Intelligence Services:

- Anomaly Detection: $83\text{K locations/min} \rightarrow \mathbf{50 \text{ instances}}$ (ML inference)
- Geofence Evaluation: $83\text{K locations/min} \times 10 \text{ geofences avg} \rightarrow \mathbf{100 \text{ instances}}$

Workflow Services:

- Maintenance Service: $\mathbf{10 \text{ instances}}$
- Check-in/out Service: $\mathbf{10 \text{ instances}}$
- Job Costing Service: $\mathbf{10 \text{ instances}}$

Integration Services:

- API Gateway: $\mathbf{50 \text{ instances}}$ (load balanced)
- Webhook Delivery: $\mathbf{20 \text{ instances}}$
- ERP Sync Workers: $\mathbf{20 \text{ instances}}$

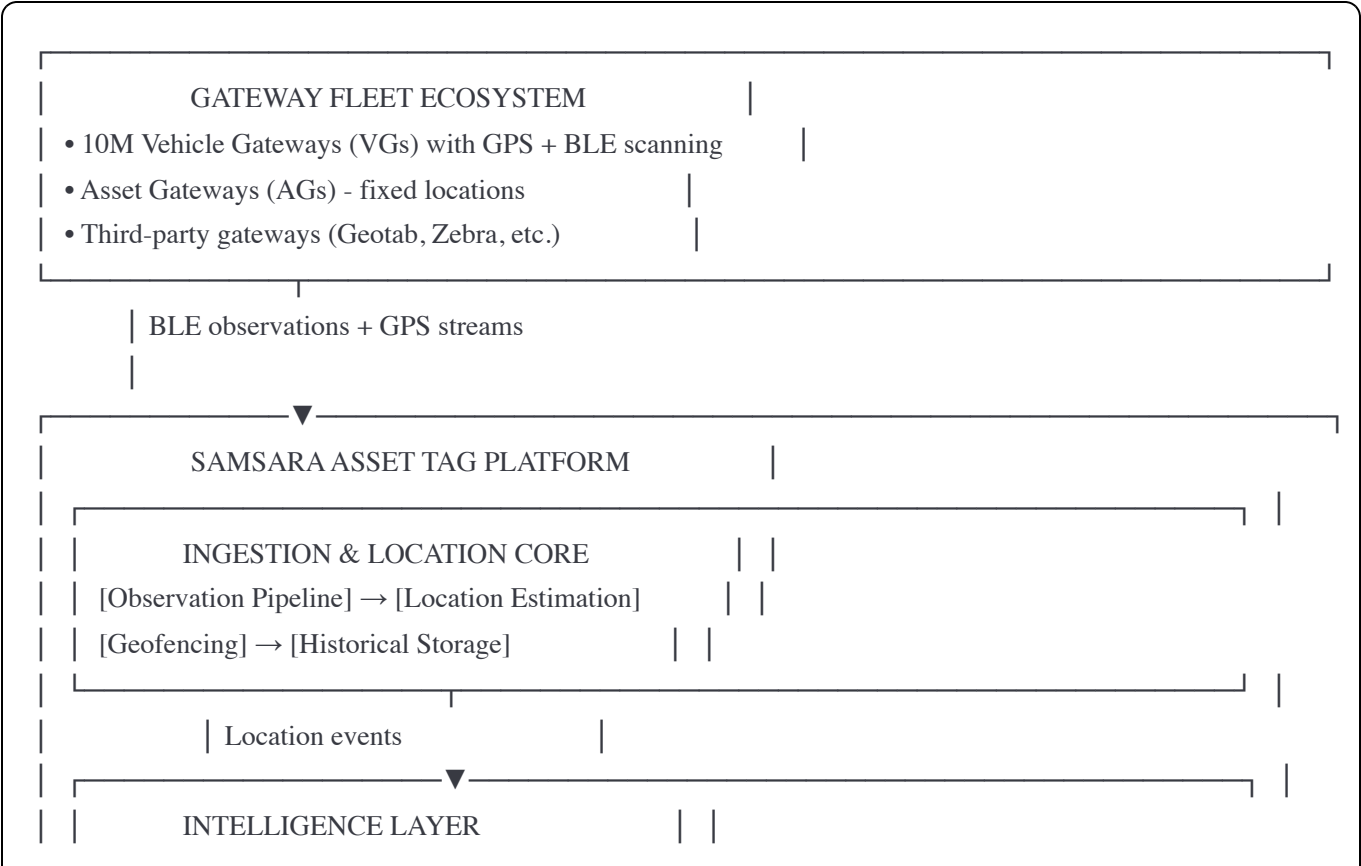
Data & Storage:

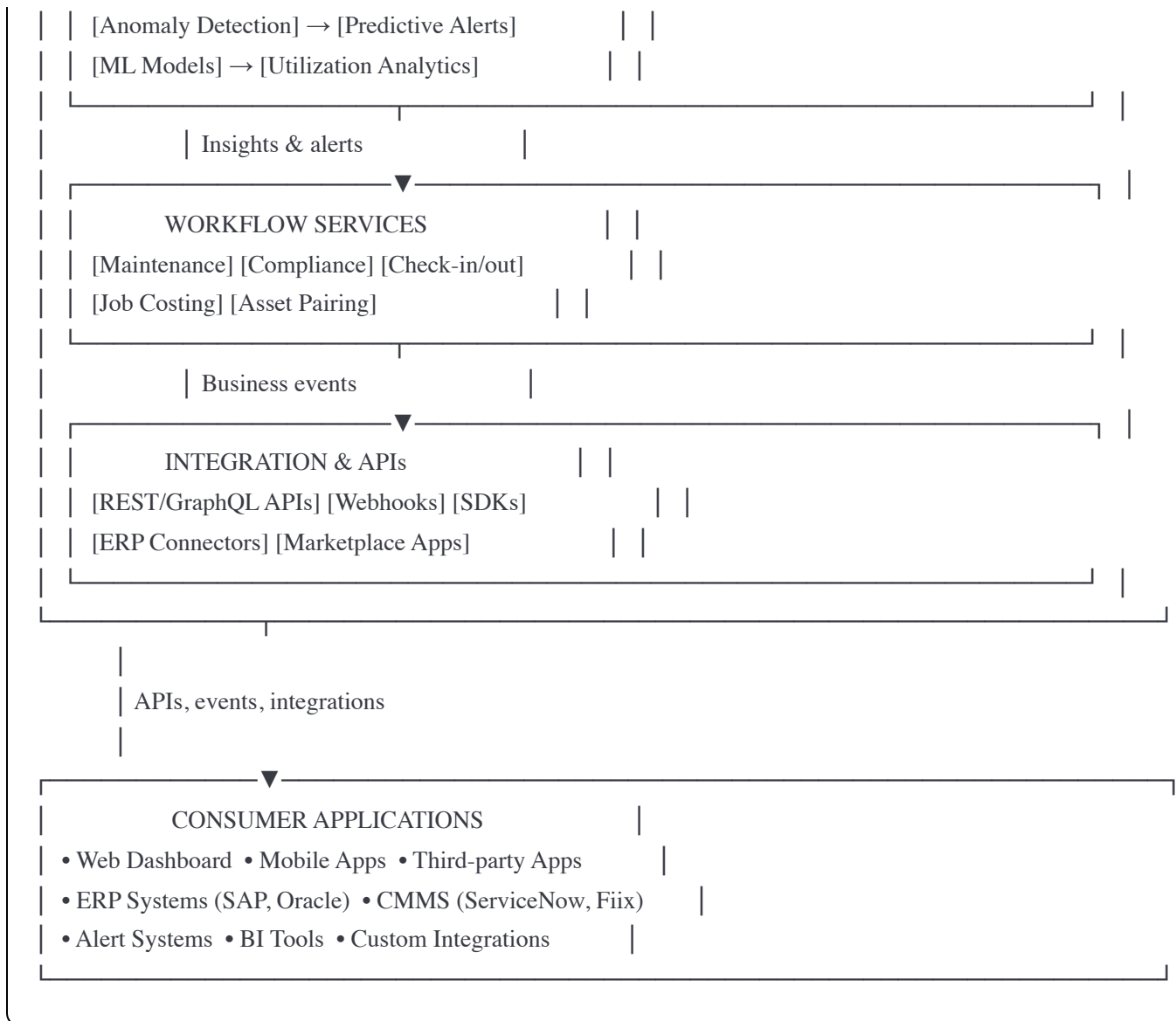
- TimescaleDB: **15 nodes** (primary + replicas)
- PostgreSQL (metadata): **5 nodes**
- Redis Cluster: **20 nodes** (distributed cache)
- Kafka: **30 brokers** (multi-region)

Total: ~1400 compute instances (mix of instance types) **Peak capacity (2x): ~2800 instances**

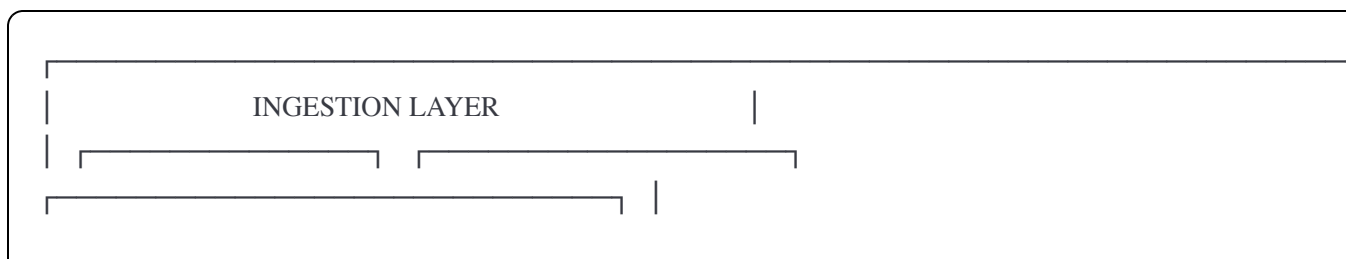
5. High-Level Architecture

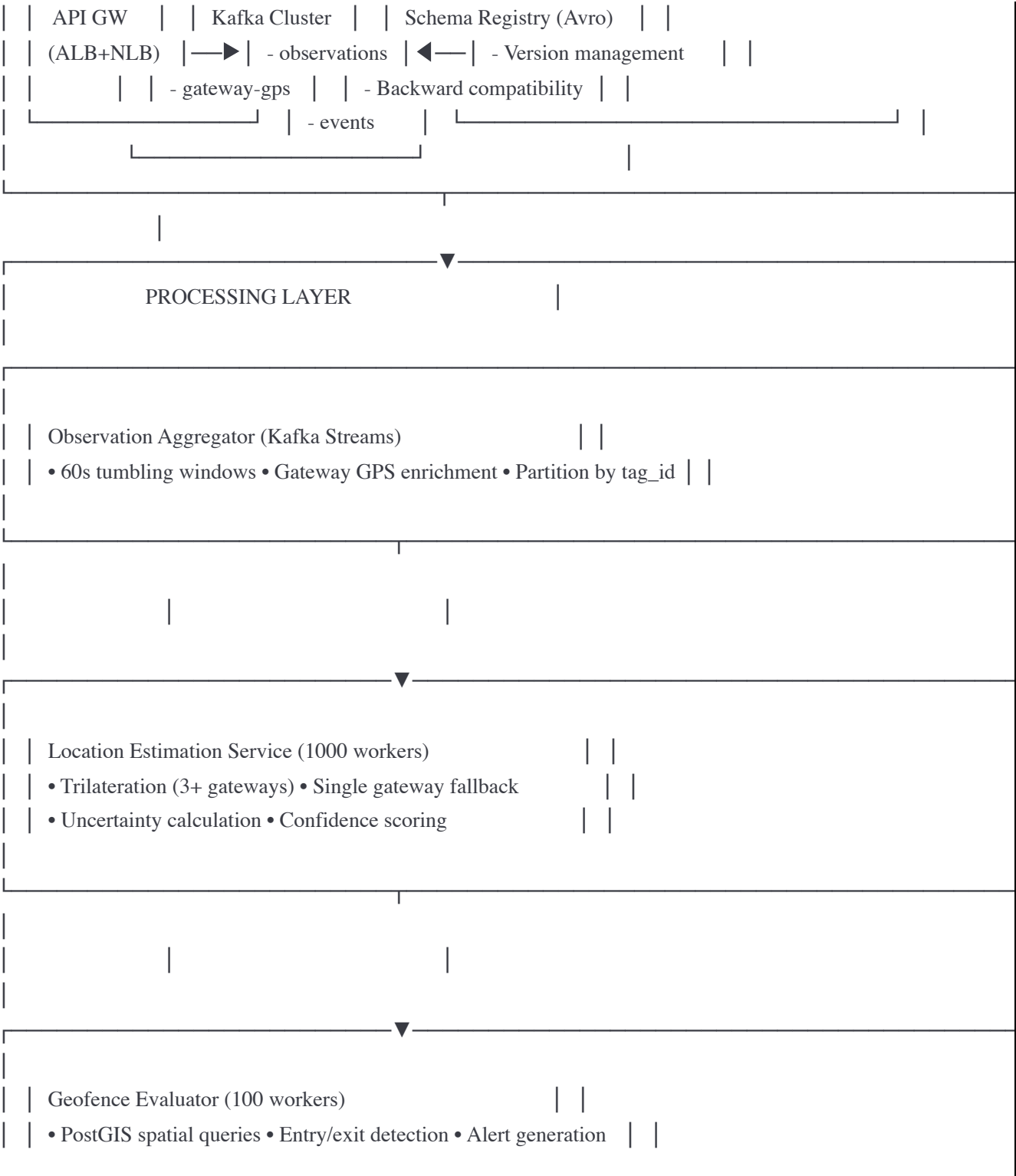
5.1 System Context Diagram

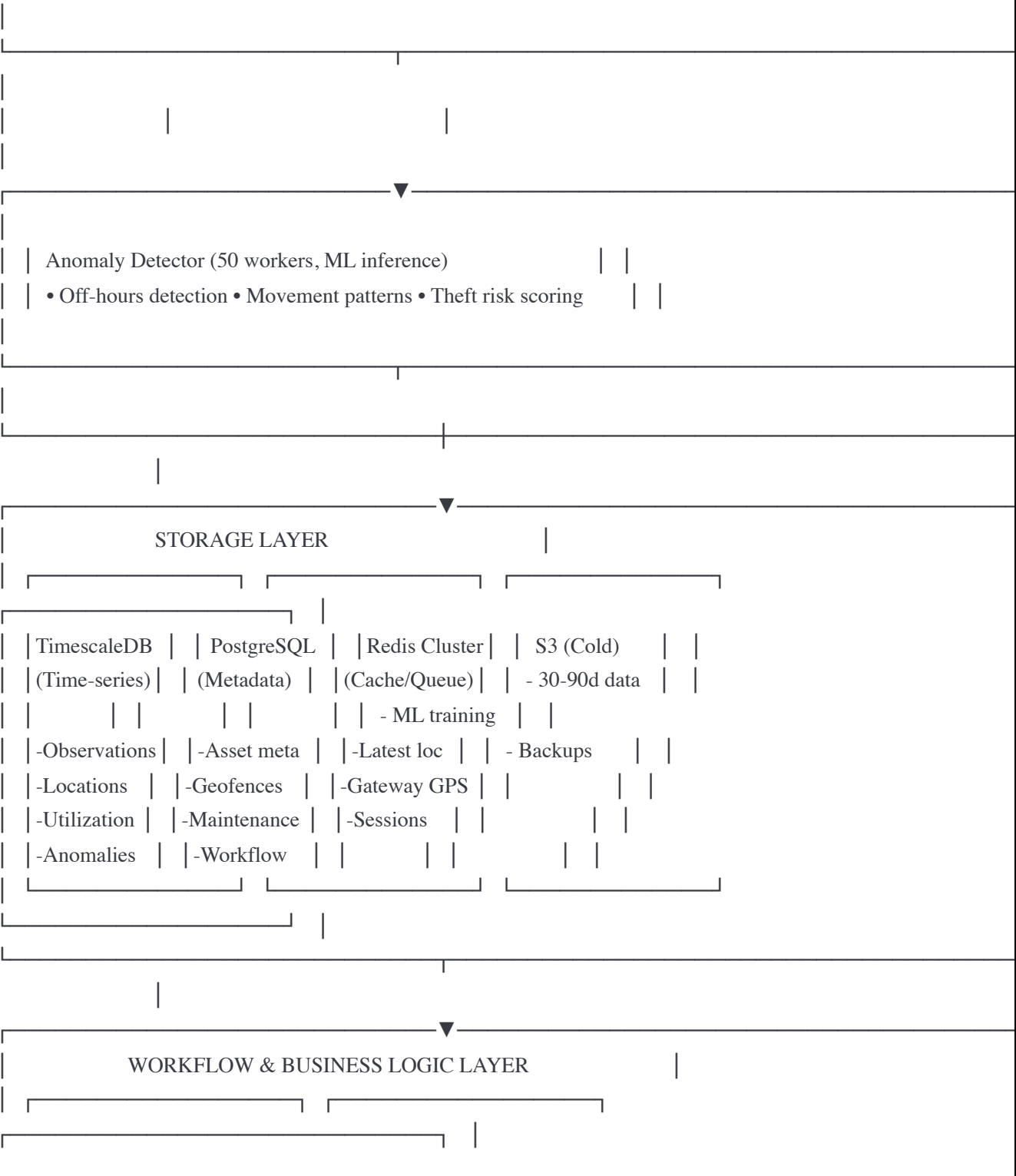


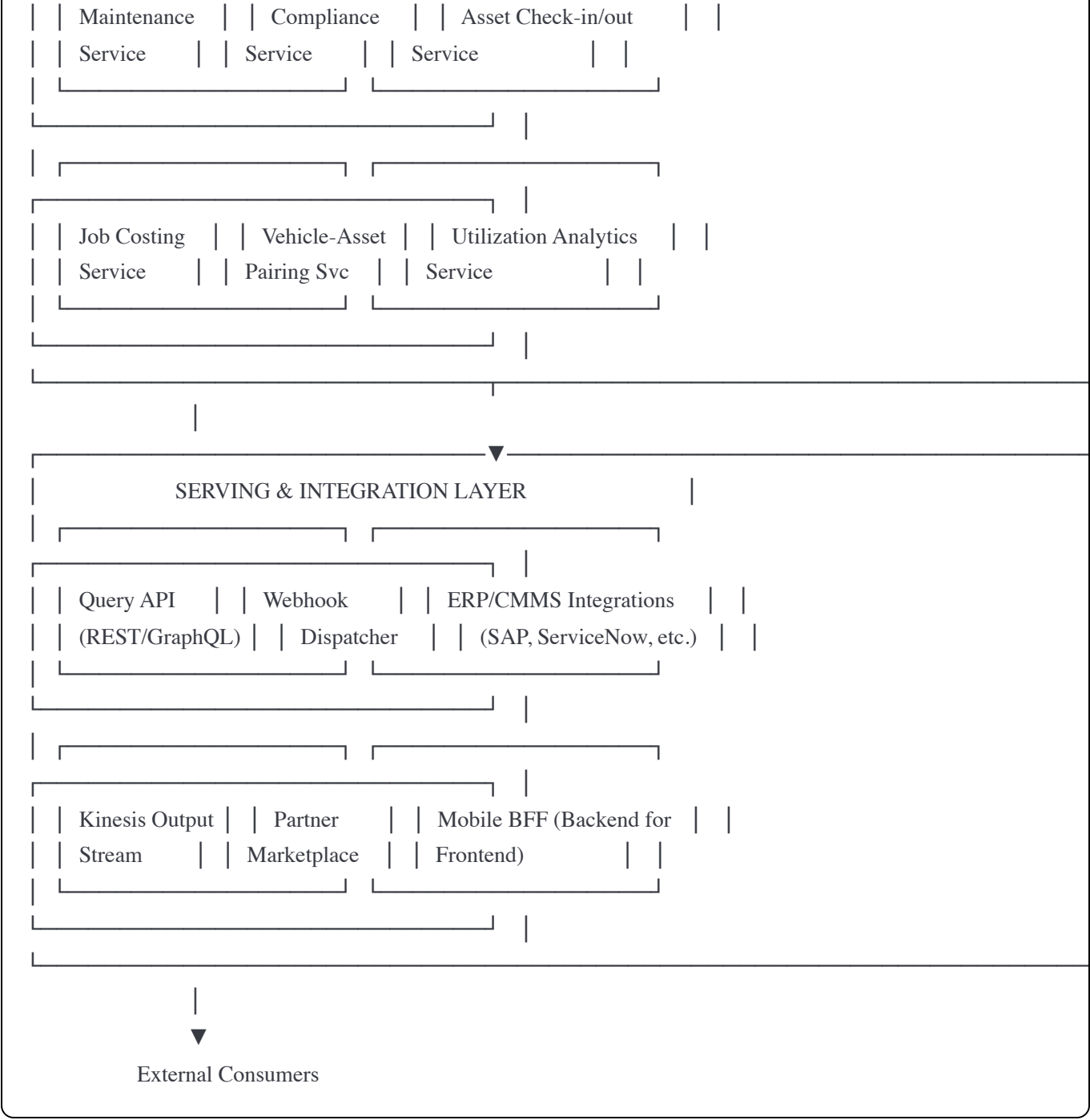


5.2 Core Platform Architecture









5.3 Key Components

Ingestion Layer:

1. API Gateway (ALB + NLB for load balancing)
2. Kafka Cluster (30 brokers, partitioned by asset_tag_id)
3. Schema Registry (Avro schemas with versioning)

Processing Layer: 4. Observation Aggregator Service (Kafka Streams) 5. Location Estimation Service (1000 workers, horizontally scaled) 6. Geofence Evaluator Service (PostGIS spatial engine) 7. Anomaly Detector Service (ML inference with TensorFlow Serving)

Storage Layer: 8. TimescaleDB Cluster (15 nodes, time-series + relational) 9. PostgreSQL Cluster (5 nodes, metadata + workflows) 10. Redis Cluster (20 nodes, distributed cache) 11. S3 (cold storage, archival, ML training data)

Workflow Layer: 12. Maintenance Service 13. Compliance Service 14. Check-in/out Service 15. Job Costing Service 16. Vehicle-Asset Pairing Service 17. Utilization Analytics Service

Serving Layer: 18. Query API Service (REST + GraphQL) 19. Webhook Dispatcher Service 20. ERP/CMMS Integration Adapters 21. Kinesis Output Stream 22. Partner Marketplace Gateway 23. Mobile BFF (Backend for Frontend)

5.4 Key Entities & Data Model

Core Entities:

typescript


```
// Asset Tag
interface AssetTag {
  id: UUID;
  customer_id: UUID;
  serial_number: string;
  asset_type: AssetType; // TOOL, TRAILER, EQUIPMENT, CONTAINER
  metadata: {
    name: string;
    description?: string;
    make_model?: string;
    acquisition_date?: Date;
    value?: number;
  };
  owner_id?: UUID; // User or team
  group_id?: UUID; // Organizational group
  project_id?: UUID;
  status: 'ACTIVE' | 'INACTIVE' | 'MAINTENANCE' | 'LOST';
  battery_level: number; // 0-100%
  battery_installed_date: Date;
  created_at: Date;
  updated_at: Date;
}
```

```
// Observation (raw BLE data)
```

```
interface Observation {
  asset_tag_id: UUID;
  gateway_id: UUID;
  timestamp: Date;
  rssi: number; // -120 to 0 dBm
  battery_level?: number;
  gateway_location: GatewayLocation;
}
```

// Gateway Location

```
interface GatewayLocation {  
  gateway_id: UUID;  
  timestamp: Date;  
  lat: number;  
  lng: number;  
  accuracy: number; // meters  
  speed?: number; // m/s  
  heading?: number; // degrees  
}
```

// Estimated Location

```
interface EstimatedLocation {  
  asset_tag_id: UUID;  
  timestamp: Date;  
  location: {  
    lat: number;  
    lng: number;  
  };  
  uncertainty_radius: number; // meters  
  confidence_score: number; // 0-100  
  algorithm: 'TRILATERATION' | 'MIDPOINT' | 'SINGLE_GATEWAY' | 'LAST_KNOWN' | 'INTERPOLATED';  
  observing_gateways: number;  
  gateway_ids?: UUID[];  
}
```

// Geofence

```
interface Geofence {  
  id: UUID;  
  customer_id: UUID;  
  name: string;  
  type: 'AUTHORIZED' | 'RESTRICTED' | 'JOB_SITE' | 'YARD' | 'CUSTOM';  
  geometry: GeoJSON; // Polygon or Circle
```

```
rules: GeofenceRule[];
created_at: Date;
}

interface GeofenceRule {
  event: 'ENTRY' | 'EXIT' | 'DWELL';
  conditions?: {
    time_of_day?: TimeRange;
    days_of_week?: DayOfWeek[];
    asset_types?: AssetType[];
  };
  actions: Action[];
}

interface Action {
  type: 'ALERT' | 'WEBHOOK' | 'WORKFLOW_TRIGGER';
  config: Record<string, any>;
}

// Anomaly Event
interface AnomalyEvent {
  id: UUID;
  asset_tag_id: UUID;
  timestamp: Date;
  anomaly_type: 'OFF_HOURS_MOVEMENT' | 'UNEXPECTED_LOCATION' | 'RAPID_MOVEMENT' | 'LOST_CO
  severity: 'CRITICAL' | 'HIGH' | 'MEDIUM' | 'LOW';
  score: number; // 0-1, probability of true anomaly
  context: {
    location?: EstimatedLocation;
    baseline?: any;
    deviation?: any;
  };
  resolution_status: 'OPEN' | 'ACKNOWLEDGED' | 'RESOLVED' | 'FALSE_POSITIVE';
```

```

}

// Maintenance Record
interface MaintenanceRecord {
  id: UUID;
  asset_tag_id: UUID;
  scheduled_date: Date;
  completed_date?: Date;
  maintenance_type: 'PREVENTIVE' | 'CORRECTIVE' | 'INSPECTION';
  description: string;
  performed_by?: UUID;
  cost?: number;
  next_due_date?: Date;
  status: 'SCHEDULED' | 'IN_PROGRESS' | 'COMPLETED' | 'OVERDUE';
}

// Check-in/out Record
interface CustodyRecord {
  id: UUID;
  asset_tag_id: UUID;
  user_id: UUID;
  action: 'CHECK_OUT' | 'CHECK_IN' | 'TRANSFER';
  timestamp: Date;
  location?: EstimatedLocation;
  project_id?: UUID;
  notes?: string;
  condition?: AssetCondition;
}

```

5.5 API Specifications

Core Location APIs:

GET /v1/assets/{asset_id}/location

→ Returns current location with uncertainty

GET /v1/assets/{asset_id}/location/history?start={ts}&end={ts}&granularity={1m|5m|1h}

→ Returns location history (paginated)

POST /v1/assets/search

→ Body: { geofence_ids?, asset_types?, status?, owner_ids? }

→ Returns assets matching criteria

GET /v1/assets/{asset_id}/path?start={ts}&end={ts}

→ Returns GeoJSON LineString of asset path

GET /v1/assets/{asset_id}/dwell?start={ts}&end={ts}

→ Returns time spent at each location

Workflow APIs:

POST /v1/assets/{asset_id}/checkout

→ Body: { user_id, project_id?, notes? }

→ Returns custody_record_id

POST /v1/assets/{asset_id}/checkin

→ Body: { condition, notes? }

→ Updates custody, triggers workflows

POST /v1/maintenance/schedule

→ Body: { asset_ids[], maintenance_type, scheduled_date }

→ Creates maintenance records

GET /v1/assets/{asset_id}/utilization?period={7d|30d|90d}

→ Returns { active_hours, idle_hours, utilization_pct }

Anomaly & Alert APIs:

GET /v1/anomalies?asset_id={id}&severity={level}&status={status}

→ Returns anomaly events (paginated)

POST /v1/anomalies/{anomaly_id}/acknowledge

→ Updates anomaly status

GET /v1/alerts/config

→ Returns customer alert configuration

PUT /v1/alerts/config

→ Updates alert rules and thresholds

Integration APIs:

POST /v1/webhooks/register

→ Body: { url, events[], secret }

→ Returns webhook_id

GET /v1/integrations/erp/{system}/sync

→ Triggers ERP sync job

POST /v1/marketplace/apps/{app_id}/install

→ Installs partner app for customer

WebSocket (Real-time):

WS /v1/stream/locations?asset_ids[]={id1,id2,...}

→ Streams location updates in real-time

WS /v1/stream/alerts

→ Streams alerts and anomalies

6. Component-Level Design

6.1 Observation Aggregator Service

Responsibility:

- Consume raw BLE observations from Kafka
- Aggregate into 60-second time windows per asset
- Enrich with gateway GPS coordinates from cache
- Forward aggregated batches to Location Estimation Service
- Handle late arrivals and out-of-order messages

Technology Stack:

- Kafka Streams (stateful stream processing)
- Java/Kotlin
- Redis for gateway location cache

Interactions:

- **Consumes from:** Kafka `observations` topic (partitioned by `asset_tag_id`)
- **Consumes from:** Kafka `gateway-gps` topic (updates Redis cache)
- **Queries:** Redis (gateway location lookups)
- **Produces to:** Kafka `aggregated-observations` topic

Dependencies:

- Kafka cluster (availability critical)
- Redis cluster (performance critical, fallback to DB if cache miss)

Scaling Strategy:

- Kafka Streams auto-scaling based on consumer lag
- Each instance processes subset of partitions
- State stores for window management (RocksDB)

Key Logic:

```
java
```



```

// Pseudocode for aggregation
stream = kafka.stream("observations")
  .groupByKey(observation -> observation.asset_tag_id)
  .windowedBy(TimeWindows.of(Duration.ofSeconds(60))
    .grace(Duration.ofSeconds(5)))
  .aggregate(
    () -> new AggregatedObservations(),
    (key, obs, agg) -> {
      // Enrich observation with gateway location from Redis
      GatewayLocation gwLoc = redis.get("gw:" + obs.gateway_id);
      obs.gateway_location = gwLoc;
      agg.add(obs);
      return agg;
    }
  )
  .toStream()
  .to("aggregated-observations");

```

Fault Tolerance:

- State stores replicated across instances
- Kafka offset management ensures exactly-once semantics
- Dead letter queue for malformed observations

6.2 Location Estimation Service

Responsibility:

- Compute asset location from aggregated observations
- Apply trilateration for 3+ gateways, fallback strategies for fewer

- Calculate uncertainty radius and confidence score
- Persist locations to TimescaleDB and publish to Kinesis
- Handle edge cases (stale data, single gateway, no observations)

Technology Stack:

- Python (NumPy, SciPy for trilateration math)
- FastAPI for internal APIs
- Celery for task queue (if async needed)

Interactions:

- **Consumes from:** Kafka `aggregated-observations` topic
- **Writes to:** TimescaleDB `estimated_locations` table
- **Publishes to:** Kinesis `location-updates` stream
- **Updates:** Redis cache with latest location (TTL 5 min)

Dependencies:

- Kafka (critical)
- TimescaleDB (critical)
- Kinesis (best-effort)
- Redis (performance optimization)

Scaling Strategy:

- Stateless workers, horizontal scaling via Kubernetes HPA
- Target: 1000 assets/min per worker (60 cores/worker)
- Work distributed by Kafka partitions

Algorithms:

1. Trilateration (3+ gateways):

```
python
```

```

def trilaterate(observations: List[Observation]) -> EstimatedLocation:
    points = [obs_to_xyz(o) for o in observations]
    distances = [rssi_to_distance(o.rssi, env_factor=2.5) for o in observations]

    # Initial guess: weighted centroid
    weights = [1/max(d, 1) for d in distances]
    x0, y0, z0 = weighted_average(points, weights)

    # Least-squares optimization
    def objective(pos):
        return sum((dist - euclidean_dist(pos, pt))**2
                   for pt, dist in zip(points, distances))

    result = scipy.optimize.minimize(objective, [x0, y0, z0], method='SLSQP')
    lat, lng = xyz_to_latlon(result.x)

    # Uncertainty = RMSE + gateway accuracy
    residuals = [abs(dist - euclidean_dist(result.x, pt))
                 for pt, dist in zip(points, distances)]
    rmse = sqrt(mean([r**2 for r in residuals]))
    gw_accuracy_avg = mean([o.gateway_location.accuracy for o in observations])
    uncertainty = rmse + gw_accuracy_avg

    confidence = min(100, 50 + 10 * len(observations)) # More gateways → higher confidence

    return EstimatedLocation(
        lat=lat, lng=lng,
        uncertainty_radius=uncertainty,
        confidence_score=confidence,
        algorithm='TRILATERATION',
        observing_gateways=len(observations)
    )

```

2. Single Gateway Fallback:

python

```
def single_gateway_fallback(observation: Observation) -> EstimatedLocation:
    distance_estimate = rssi_to_distance(observation.rssi, env_factor=2.5)
    gw_loc = observation.gateway_location

    return EstimatedLocation(
        lat=gw_loc.lat,
        lng=gw_loc.lng,
        uncertainty_radius=distance_estimate + gw_loc.accuracy,
        confidence_score=40, # Low confidence
        algorithm='SINGLE_GATEWAY',
        observing_gateways=1
    )
```

3. No Recent Observations (Stale):

python

```
def stale_location_fallback(asset_id: UUID, last_location: EstimatedLocation) -> EstimatedLocation:
    minutes_elapsed = (now() - last_location.timestamp).total_seconds() / 60

    # Increase uncertainty over time
    uncertainty_growth = 10 * minutes_elapsed # 10m per minute elapsed
    new_uncertainty = min(10000, last_location.uncertainty_radius + uncertainty_growth)

    # Decay confidence
    confidence_decay = min(minutes_elapsed * 2, last_location.confidence_score)
    new_confidence = max(0, last_location.confidence_score - confidence_decay)

    return EstimatedLocation(
        lat=last_location.lat,
        lng=last_location.lng,
        uncertainty_radius=new_uncertainty,
        confidence_score=new_confidence,
        algorithm='LAST_KNOWN',
        observing_gateways=0
    )
```

Performance Optimization:

- Pre-compute RSSI-to-distance lookup tables
- Cache recent locations in memory (LRU cache, 10K assets)
- Batch writes to TimescaleDB (100 locations per transaction)

6.3 Geofence Evaluator Service

Responsibility:

- Evaluate asset locations against customer geofences

- Detect entry/exit events
- Generate alerts based on geofence rules
- Track asset state per geofence (inside/outside)

Technology Stack:

- Python + PostGIS (spatial queries)
- PostgreSQL for geofence storage
- Redis for state tracking

Interactions:

- **Consumes from:** Kinesis `location-updates` stream
- **Queries:** PostgreSQL (geofence definitions with PostGIS geometry)
- **Reads/Writes:** Redis (asset-geofence state cache)
- **Produces to:** Kafka `geofence-alerts` topic

Key Logic:

```
python
```

```

def evaluate_geofence(location: EstimatedLocation):
    # Get all geofences for this customer
    geofences = db.query("""
        SELECT id, name, type, ST_AsText(geometry) as geom, rules
        FROM geofences
        WHERE customer_id = %s
    """, location.customer_id)

    for gf in geofences:
        # Check if asset is inside geofence (PostGIS)
        is_inside = db.query("""
            SELECT ST_Contains(
                ST_GeomFromText(%s),
                ST_SetSRID(ST_MakePoint(%s, %s), 4326)
            )
        """, gf.geom, location.lng, location.lat)[0][0]

        # Get previous state from cache
        cache_key = f"gf_state:{location.asset_tag_id}:{gf.id}"
        was_inside = redis.get(cache_key) == "1"

        # Detect state change
        if is_inside and not was_inside:
            emit_alert(AlertType.GEOFENCE_ENTRY, location, gf)
        elif not is_inside and was_inside:
            emit_alert(AlertType.GEOFENCE_EXIT, location, gf)

        # Update state
        redis.set(cache_key, "1" if is_inside else "0", ex=3600)

```

Scaling:

- Partition by customer_id (each worker handles subset of customers)
 - Read replicas for geofence queries
 - Spatial indexing (GiST) on geometry columns
-

6.4 Anomaly Detector Service

Responsibility:

- Detect abnormal asset behavior (theft, loss, misuse)
- ML-based pattern learning per asset
- Generate anomaly scores and alerts
- Reduce false positives via contextual rules

Technology Stack:

- Python + TensorFlow/PyTorch (LSTM for time-series anomaly detection)
- TensorFlow Serving for inference
- PostgreSQL for anomaly logs

Interactions:

- **Consumes from:** Kinesis `location-updates` stream
- **Queries:** S3 (historical data for ML training)
- **Reads:** PostgreSQL (asset baseline patterns)
- **Produces to:** Kafka `anomaly-alerts` topic

ML Model:

- **Input:** Time-series of asset movements (lat, lng, timestamp, speed, geofence_ids)

- **Output:** Anomaly score (0-1)
- **Model:** Isolation Forest or LSTM Autoencoder
- **Training:** Nightly batch job on historical data (per customer)
- **Inference:** Real-time per location update

Rule-Based Checks (Hybrid Approach):

```
python
```

```
def detect_anomaly(location: EstimatedLocation, asset: AssetTag):  
    anomaly_score = 0.0  
    anomaly_types = []  
  
    # Rule 1: Off-hours movement  
    if is_off_hours(location.timestamp, asset.typical_hours):  
        anomaly_score += 0.4  
        anomaly_types.append('OFF_HOURS_MOVEMENT')  
  
    # Rule 2: Unauthorized geofence  
    if location in unauthorized_geofences(asset):  
        anomaly_score += 0.5  
        anomaly_types.append('UNAUTHORIZED_ZONE')  
  
    # Rule 3: Rapid movement (speed > typical)  
    speed = calculate_speed(location, previous_location)  
    if speed > asset.typical_max_speed * 2:  
        anomaly_score += 0.3  
        anomaly_types.append('RAPID_MOVEMENT')  
  
    # Rule 4: ML model score  
    ml_score = ml_model.predict(location_features)  
    anomaly_score += ml_score * 0.5  
  
    # Normalize to 0-1  
    anomaly_score = min(1.0, anomaly_score)  
  
    # Generate alert if score exceeds threshold  
    if anomaly_score > 0.7:  
        create_anomaly_event(  
            asset_id=asset.id,  
            anomaly_types=anomaly_types,  
            score=anomaly_score,
```

```
severity='HIGH' if anomaly_score > 0.85 else 'MEDIUM'  
)
```

Model Training Pipeline:

- Nightly batch job trains per-customer models
 - Features: location history, time patterns, geofence associations
 - Stores models in S3, loads into TensorFlow Serving
 - A/B testing framework to compare model versions
-

6.5 TimescaleDB (Time-Series Database)

Responsibility:

- Persist observations (7-day retention)
- Persist computed locations (90-day retention with tiering)
- Support historical queries with time-range indexes
- Automatic partitioning, compression, and retention policies

Technology Stack:

- TimescaleDB 2.x (PostgreSQL extension)
- Multi-node cluster (primary + replicas)

Schema Design:

```
sql
```

-- Observations table (7-day retention)

```
CREATE TABLE observations (  
  time TIMESTAMPTZ NOT NULL,  
  asset_tag_id UUID NOT NULL,  
  gateway_id UUID NOT NULL,  
  rssi INTEGER CHECK (rssi BETWEEN -120 AND 0),  
  battery_level INTEGER CHECK (battery_level BETWEEN 0 AND 100),  
  gateway_lat DOUBLE PRECISION,  
  gateway_lng DOUBLE PRECISION,  
  gateway_accuracy DOUBLE PRECISION,  
  customer_id UUID NOT NULL -- for data isolation  
);
```

```
SELECT create_hypertable('observations', 'time',  
  chunk_time_interval => INTERVAL '1 hour',  
  partitioning_column => 'asset_tag_id',  
  number_partitions => 64  
);
```

-- Indexes for common queries

```
CREATE INDEX idx_obs_asset_time ON observations (asset_tag_id, time DESC);  
CREATE INDEX idx_obs_customer ON observations (customer_id, time DESC);
```

-- Retention policy

```
SELECT add_retention_policy('observations', INTERVAL '7 days');
```

-- Estimated locations table (90-day retention)

```
CREATE TABLE estimated_locations (  
  time TIMESTAMPTZ NOT NULL,  
  asset_tag_id UUID NOT NULL,  
  lat DOUBLE PRECISION NOT NULL,  
  lng DOUBLE PRECISION NOT NULL,  
  uncertainty_radius DOUBLE PRECISION,
```

```
confidence_score INTEGER CHECK (confidence_score BETWEEN 0 AND 100),
algorithm TEXT,
observing_gateways INTEGER,
customer_id UUID NOT NULL
);

SELECT create_hypertable('estimated_locations', 'time',
    chunk_time_interval => INTERVAL '1 day',
    partitioning_column => 'asset_tag_id',
    number_partitions => 128
);

CREATE INDEX idx_loc_asset_time ON estimated_locations (asset_tag_id, time DESC);
CREATE INDEX idx_loc_customer_time ON estimated_locations (customer_id, time DESC);
-- Spatial index for geospatial queries
CREATE INDEX idx_loc_spatial ON estimated_locations USING GIST (
    ST_SetSRID(ST_MakePoint(lng, lat), 4326)
);

-- Compression policy (after 7 days)
SELECT add_compression_policy('estimated_locations', INTERVAL '7 days');

-- Retention policy (90 days)
SELECT add_retention_policy('estimated_locations', INTERVAL '90 days');

-- Continuous aggregates for analytics (pre-compute hourly/daily rollups)
CREATE MATERIALIZED VIEW asset_utilization_hourly
WITH (timescaledb.continuous) AS
SELECT
    time_bucket('1 hour', time) AS hour,
    asset_tag_id,
    customer_id,
    COUNT(*) as location_updates,
```

```
AVG(confidence_score) as avg_confidence,  
MIN(uncertainty_radius) as min_uncertainty,  
MAX(uncertainty_radius) as max_uncertainty  
FROM estimated_locations  
GROUP BY hour, asset_tag_id, customer_id;  
  
-- Refresh policy for materialized view  
SELECT add_continuous_aggregate_policy('asset_utilization_hourly',  
start_offset => INTERVAL '3 hours',  
end_offset => INTERVAL '1 hour',  
schedule_interval => INTERVAL '1 hour');
```

Scaling Strategy:

- Hash partitioning by asset_tag_id (128 partitions)
- Time partitioning by chunk intervals
- Read replicas for query workload (5 replicas)
- Connection pooling via PgBouncer

Backup & Recovery:

- Continuous WAL archiving to S3
- Daily snapshots for point-in-time recovery
- RPO: 5 minutes, RTO: 30 minutes

6.6 Query API Service

Responsibility:

- Serve REST and GraphQL APIs for asset location queries

- Handle authentication and authorization
- Rate limiting and caching
- Query optimization and result pagination

Technology Stack:

- Node.js + Express (REST API)
- Apollo Server (GraphQL)
- Redis for caching and rate limiting

Key Endpoints:

javascript


```

// Current location (with caching)
app.get('/v1/assets/:assetId/location', authenticate, async (req, res) => {
  const { assetId } = req.params;

  // Check cache first (TTL: 30 seconds)
  const cached = await redis.get(`loc:${assetId}`);
  if (cached) {
    return res.json(JSON.parse(cached));
  }

  // Query database
  const location = await db.query(
    SELECT time, lat, lng, uncertainty_radius, confidence_score, algorithm
    FROM estimated_locations
    WHERE asset_tag_id = $1
    AND customer_id = $2
    ORDER BY time DESC
    LIMIT 1
    `, [assetId, req.user.customerId]);

  if (!location) {
    return res.status(404).json({ error: 'Asset not found or no recent location' });
  }

  // Cache result
  await redis.setex(`loc:${assetId}`, 30, JSON.stringify(location));

  res.json(location);
});

// Historical locations (with pagination)
app.get('/v1/assets/:assetId/location/history', authenticate, async (req, res) => {
  const { assetId } = req.params;

```

```
const { start, end, granularity = '1m', limit = 1000, offset = 0 } = req.query;

// Validate time range (max 90 days)
if (new Date(end) - new Date(start) > 90 * 24 * 60 * 60 * 1000) {
  return res.status(400).json({ error: 'Time range exceeds 90 days' });
}

// Query with optional downsampling
const query = granularity === '1m'
  ? `SELECT * FROM estimated_locations WHERE ...`
  : `SELECT time_bucket($4, time) as time,
      first(lat, time) as lat,
      first(lng, time) as lng, ...
      FROM estimated_locations WHERE ...
      GROUP BY time_bucket($4, time)`;

const locations = await db.query(query,
  [assetId, req.user.customerId, start, end, granularity]);

res.json({
  data: locations,
  pagination: { limit, offset, total: locations.length }
});
});
```

GraphQL Schema:

graphql

```

type Location {
  assetTagId: ID!
  timestamp: DateTime!
  lat: Float!
  lng: Float!
  uncertaintyRadius: Float!
  confidenceScore: Int!
  algorithm: LocationAlgorithm!
}

enum LocationAlgorithm {
  TRILATERATION
  MIDPOINT
  SINGLE_GATEWAY
  LAST_KNOWN
  INTERPOLATED
}

type Query {
  asset(id: ID!): Asset
  assets(filter: AssetFilter, pagination: Pagination): AssetConnection!
  location(assetId: ID!): Location
  locationHistory(assetId: ID!, start: DateTime!, end: DateTime!): [Location!]!
}

type Subscription {
  locationUpdates(assetIds: [ID!]!): Location!
}

```

Performance Optimization:

- Redis caching (90%+ hit rate for current location)

- Database query optimization (covering indexes)
 - Response compression (gzip)
 - CDN for static API documentation
-

6.7 Webhook Dispatcher Service

Responsibility:

- Deliver real-time events to customer webhooks
- Reliable delivery with retries
- Event filtering and transformation
- Webhook management (registration, validation, monitoring)

Technology Stack:

- Go (for high-throughput, low-latency)
- RabbitMQ for durable queue
- PostgreSQL for webhook registry

Interactions:

- **Consumes from:** Kafka topics (location-updates, geofence-alerts, anomaly-alerts)
- **Queries:** PostgreSQL (webhook subscriptions)
- **Publishes to:** Customer HTTP endpoints

Key Logic:

go

```

type WebhookEvent struct {
    EventID    string `json:"event_id"`
    EventType  string `json:"event_type"`
    Timestamp  time.Time `json:"timestamp"`
    AssetTagID string `json:"asset_tag_id"`
    CustomerID string `json:"customer_id"`
    Payload    any    `json:"payload"`
}

func DeliverWebhook(webhook *Webhook, event *WebhookEvent) error {
    // Filter events based on subscription
    if !webhook.SubscribedTo(event.EventType) {
        return nil
    }

    // Sign payload with HMAC
    signature := hmac.Sign(webhook.Secret, event.Payload)

    // HTTP POST with timeout
    ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
    defer cancel()

    req, _ := http.NewRequestWithContext(ctx, "POST", webhook.URL, event.ToJSON())
    req.Header.Set("X-Samsara-Signature", signature)
    req.Header.Set("X-Samsara-Event-Type", event.EventType)

    resp, err := httpClient.Do(req)
    if err != nil || resp.StatusCode >= 500 {
        // Retry with exponential backoff
        return RetryWithBackoff(webhook, event, 3)
    }

    if resp.StatusCode >= 400 {

```

```

    // Client error, don't retry, log for customer
    LogWebhookFailure(webhook.ID, event, resp.StatusCode)
}

return nil
}

func RetryWithBackoff(webhook *Webhook, event *WebhookEvent, maxRetries int) error {
    backoff := time.Second
    for i := 0; i < maxRetries; i++ {
        time.Sleep(backoff)
        err := DeliverWebhook(webhook, event)
        if err == nil {
            return nil
        }
        backoff *= 2 // Exponential backoff
    }
    // After max retries, send to dead letter queue
    SendToDeadLetterQueue(webhook, event)
    return errors.New("max retries exceeded")
}

```

Monitoring:

- Track delivery success rate per webhook
 - Alert on consecutive failures
 - Dashboard for customer webhook health
 - Dead letter queue UI for manual retry
-

6.8 Maintenance Service (Workflow Example)

Responsibility:

- Schedule maintenance based on usage patterns
- Generate maintenance reminders and alerts
- Integrate with CMMS systems
- Track maintenance history

Technology Stack:

- Python + FastAPI
- PostgreSQL for maintenance records
- Celery for scheduled tasks

Key Features:

1. Usage-Based Triggers:

```
python
```

```

def check_maintenance_due(asset: AssetTag):
    # Calculate usage hours from location history
    usage_query = """
        SELECT SUM(CASE
            WHEN confidence_score > 50 AND observing_gateways > 0
            THEN 1.0/60 -- 1 minute of usage
            ELSE 0
        END) as usage_hours
        FROM estimated_locations
        WHERE asset_tag_id = %s
        AND time > (SELECT last_maintenance_date FROM maintenance_records
            WHERE asset_tag_id = %s
            ORDER BY completed_date DESC LIMIT 1)
    """
    usage_hours = db.query(usage_query, [asset.id, asset.id])[0][0]

    # Check if maintenance is due (e.g., every 500 hours)
    if usage_hours >= asset.maintenance_interval_hours:
        create_maintenance_alert(asset, usage_hours)

```

2. Integration with CMMS:

python

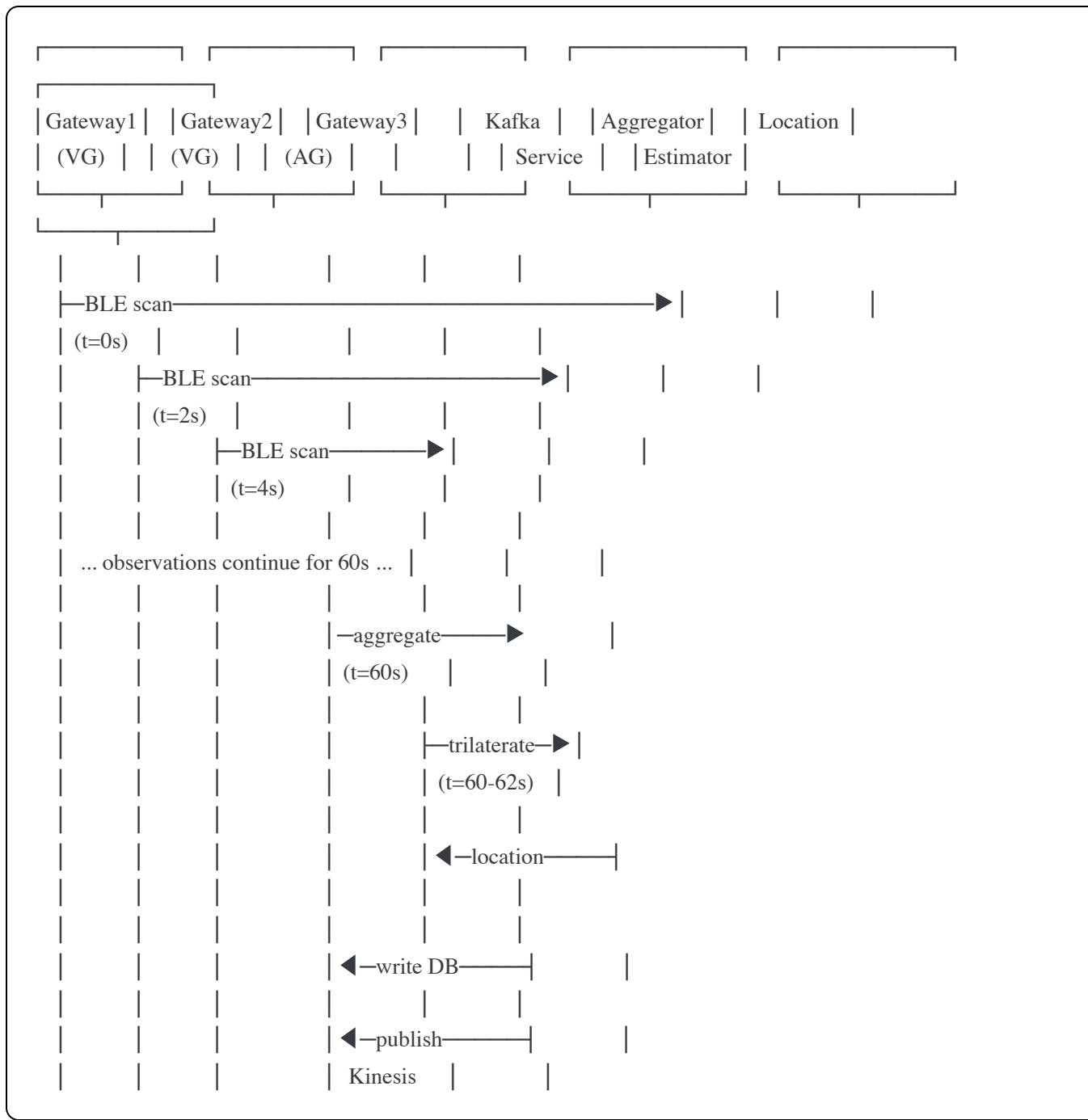

```
async def sync_with_cmms(maintenance_record: MaintenanceRecord):  
    # Push to ServiceNow, Fiix, or other CMMS  
    if customer.cmms_integration == 'servicenow':  
        work_order = {  
            'short_description': f'Maintenance due: {asset.name}',  
            'asset_tag': asset.id,  
            'due_date': maintenance_record.scheduled_date,  
            'priority': 3,  
            'category': 'Preventive Maintenance'  
        }  
        await servicenow_client.create_work_order(work_order)
```

7. Deep Dive: Business Flows

7.1 Flow 1: Real-Time Location Estimation

Scenario: Asset Tag broadcasts BLE every 5 seconds, multiple gateways detect it, system computes location within 60 seconds.

7.1.1 High-Level Sequence Diagram



7.1.2 Detailed Steps

Step 1: BLE Broadcast & Detection (t=0-60s)

- Asset Tag broadcasts BLE packet every 5 seconds
- Packet contains: encrypted device ID, RSSI, battery level
- All nearby gateways within range (up to 3000 ft) detect the broadcast
- Each gateway uploads observation to API Gateway with GPS coordinates

Step 2: Ingestion (t=0-60s, continuous)

Gateway → API Gateway → Kafka Topic "observations" (partitioned by asset_tag_id)

Example observation:

```
{  
  "asset_tag_id": "550e8400-e29b-41d4-a716-446655440000",  
  "gateway_id": "7c9e6679-7425-40de-944b-e07fc1f90ae7",  
  "timestamp": "2025-10-03T14:32:15.123Z",  
  "rssi": -65,  
  "battery_level": 87  
}
```

Step 3: Gateway GPS Stream (parallel, t=0-60s)

Gateway → API Gateway → Kafka Topic "gateway-gps"

Gateway Location Cache Service consumes and updates Redis:

Key: "gw:7c9e6679-7425-40de-944b-e07fc1f90ae7"

Value: {"lat": 47.7511, "lng": -122.2018, "accuracy": 5.0, "timestamp": "..."} }

TTL: 60 seconds

Step 4: Time-Window Aggregation (t=60s)

- Observation Aggregator Service processes 60-second window
- Groups all observations for asset_tag_id=550e8400...
- For each observation, fetches gateway GPS from Redis cache
- Produces aggregated batch:

```
json
{
  "asset_tag_id": "550e8400-e29b-41d4-a716-446655440000",
  "window_start": "2025-10-03T14:32:00Z",
  "window_end": "2025-10-03T14:33:00Z",
  "observations": [
    {
      "gateway_id": "gw1",
      "rssi": -65,
      "gateway_location": {"lat": 47.7511, "lng": -122.2018, "accuracy": 5.0}
    },
    {
      "gateway_id": "gw2",
      "rssi": -72,
      "gateway_location": {"lat": 47.7515, "lng": -122.2025, "accuracy": 8.0}
    },
    {
      "gateway_id": "gw3",
      "rssi": -58,
      "gateway_location": {"lat": 47.7508, "lng": -122.2010, "accuracy": 6.0}
    }
  ]
}
```

Step 5: Location Computation (t=60-62s)

- Location Estimation Service consumes aggregated batch
- Detects 3 gateways → use trilateration algorithm
- Converts RSSI to distance:
 - Gateway 1: -65 dBm → ~32 meters
 - Gateway 2: -72 dBm → ~57 meters
 - Gateway 3: -58 dBm → ~18 meters
- Runs least-squares optimization to find best-fit location
- Calculates uncertainty (RMSE + avg gateway accuracy): ~15 meters
- Confidence score: 70 (3 gateways, good RSSI)

Step 6: Persistence & Distribution (t=62-63s)

Location Estimator writes to:

1. TimescaleDB "estimated_locations" table
2. Kinesis "location-updates" stream
3. Redis cache "loc:550e8400..." (TTL: 5 min)

Result:




```
{
  "asset_tag_id": "550e8400-e29b-41d4-a716-446655440000",
  "timestamp": "2025-10-03T14:33:02Z",
  "lat": 47.7512,
  "lng": -122.2018,
  "uncertainty_radius": 15.3,
  "confidence_score": 70,
  "algorithm": "TRILATERATION",
  "observing_gateways": 3
}
```

Step 7: Downstream Consumption (t=63s+)

- Geofence Evaluator checks location against geofences
- Anomaly Detector evaluates for suspicious patterns
- Query API cache updated (instant reads for next 5 minutes)
- Webhooks triggered for subscribed customers
- Dashboard updates via WebSocket

7.1.3 Design Decisions

Decision 1: Time Window Size (60s)

Option	Pros	Cons	Decision
30s windows	Faster updates (2× per minute), better for theft detection	Higher compute cost, fewer observations per window → less accurate trilateration, more network traffic	 Rejected
60s windows	Balance of latency and accuracy, sufficient observations (12-20 per asset), meets "~1 minute" requirement	May miss very rapid movements	 Selected
120s windows	Best accuracy (more observations), lower compute cost	Too slow for theft alerts, poor UX for real-time tracking	 Rejected

Rationale: 60 seconds provides 10-20 observations per asset (assuming 3-5 gateways nearby), which is sufficient for accurate trilateration. For critical theft detection, the Anomaly Detector service operates on a separate, faster path with single-location analysis.

Decision 2: RSSI to Distance Conversion

Option	Pros	Cons	Decision
Log-distance path loss model	Simple, industry-standard, predictable	Inaccurate indoors, needs environment calibration	✅ Selected for MVP
ML-based regression model	Learns environmental variations, potentially more accurate	Requires extensive training data, complex, hard to debug	⏮ Phase 2
Lookup table by environment	Very accurate per environment type	Requires customers to classify environment per location, maintenance burden	⏮ Phase 2

Formula (Log-Distance):

$$\text{distance} = 10^{((\text{RSSI}_{\text{measured}} - \text{RSSI}_{\text{at_1m}}) / (10 * \text{path_loss_exponent}))}$$

Where:

$\text{RSSI}_{\text{at_1m}} = -50 \text{ dBm}$ (calibrated for our 100× stronger BLE)

path_loss_exponent (n):

- 2.0 for open air / line of sight
- 2.5 for light obstacles (warehouses, yards)
- 3.0 for heavy obstacles (indoor, urban)
- 3.5 for dense indoor (factories, multi-floor)

Implementation: Start with n=2.5 as default, allow customer configuration per site, plan ML model for Phase 2 based on collected data.

Decision 3: Handling Insufficient Observations

Scenario	Algorithm	Uncertainty	Confidence	Decision
3+ gateways	Trilateration (least-squares)	RMSE + avg_gw_accuracy	50 + 10×N (max 100)	✓ Optimal
2 gateways	Weighted midpoint	avg_distance + avg_gw_accuracy	45	✓ Acceptable
1 gateway	Gateway location	RSSI_distance + gw_accuracy	40	✓ Last resort
0 gateways (stale)	Last known location	scale with time (50m + 10m/min)	decay (100 - 2×min)	✓ Transparency

Rationale: Always provide a location (never return null) to enable map rendering, but clearly signal quality via uncertainty radius and confidence score. Customers can filter by confidence threshold if needed.

Decision 4: Storage Strategy

Option	Pros	Cons	Decision
Single DB (TimescaleDB)	Simplicity, single source of truth	Expensive for cold data, potential performance bottleneck	✗ Rejected
3-Tier (Hot/Warm/Cold)	Cost-effective, optimized query performance per tier	Complex data lifecycle, multiple query paths	✓ Selected
All-cloud-native (DynamoDB, S3 only)	Serverless, extreme scalability	Higher query latency, vendor lock-in, less SQL flexibility	✗ Rejected

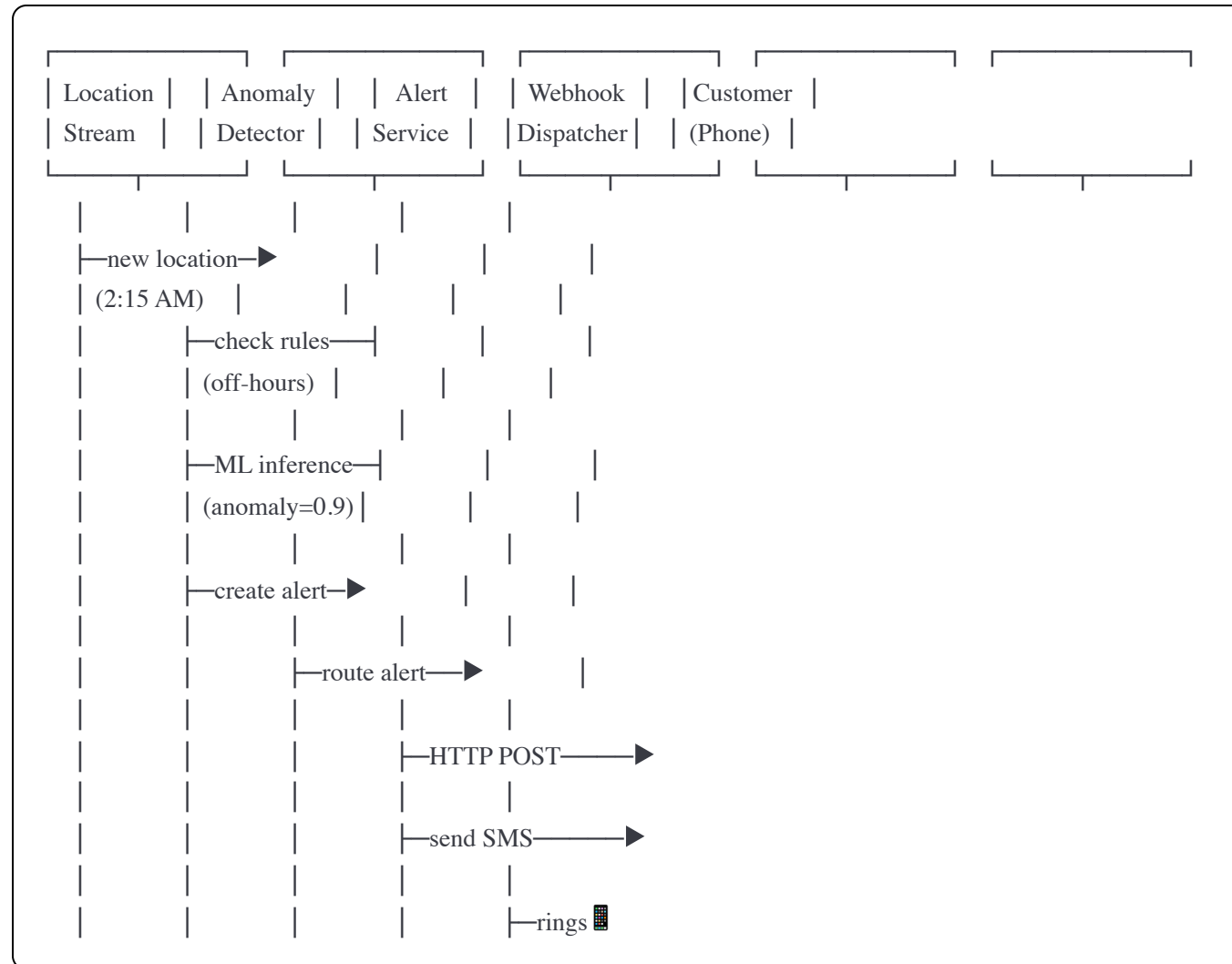
Implementation:

- **Hot (0-7d):** TimescaleDB SSD, full 1-minute granularity, <500ms queries
- **Warm (7-30d):** TimescaleDB with compression, 1-minute granularity, ~2s queries
- **Cold (30-90d):** S3 Parquet files + Athena, 5-minute granularity, ~10s queries

7.2 Flow 2: Theft Detection via Anomaly Detection

Scenario: Asset Tag moves at 2:15 AM Sunday (outside business hours), system detects anomaly and alerts customer within 2 minutes.

7.2.1 Sequence Diagram



7.2.2 Detailed Steps

Step 1: Location Change Detection (t=0s)

- New location arrives via Kinesis stream
- Anomaly Detector calculates distance from previous location
- Distance = 2.3 km (moved significantly)

Step 2: Temporal Analysis (t=0-1s)

```
python
```

```
# Check: Is this movement time unusual?
current_time = datetime(2025, 10, 3, 2, 15, 0) # Sunday 2:15 AM
day_of_week = current_time.weekday() # 6 (Sunday)
hour = current_time.hour # 2

# Query asset's historical movement patterns
pattern = db.query("""
    SELECT
        date_part('dow', time) as dow,
        date_part('hour', time) as hour,
        COUNT(*) as movement_count
    FROM estimated_locations
    WHERE asset_tag_id = %s
        AND time > NOW() - INTERVAL '90 days'
        AND (lat != LAG(lat) OVER (ORDER BY time)
            OR lng != LAG(lng) OVER (ORDER BY time))
    GROUP BY dow, hour
""", [asset_id])

# Result: 0 movements on Sunday between 2-3 AM historically
# Typical pattern: Movements only Mon-Fri 6AM-6PM

off_hours_score = 0.5 # High anomaly score
```

Step 3: Spatial Analysis (t=1-2s)

```
python
```

```

# Check: Is location authorized?
geofences = get_authorized_geofences(asset_id)
is_in_authorized_zone = any(
    geofence.contains(new_location) for geofence in geofences
)

if not is_in_authorized_zone:
    unauthorized_location_score = 0.4
else:
    unauthorized_location_score = 0.0

# Check: Has asset ever been in this area?
historical_presence = db.query("""
    SELECT COUNT(*) FROM estimated_locations
    WHERE asset_tag_id = %s
    AND ST_DWithin(
        ST_SetSRID(ST_MakePoint(lng, lat), 4326),
        ST_SetSRID(ST_MakePoint(%s, %s), 4326),
        1000 -- within 1km
    )
""", [asset_id, new_location.lng, new_location.lat])[0][0]

if historical_presence == 0:
    novel_location_score = 0.3
else:
    novel_location_score = 0.0

```

Step 4: ML Model Inference (t=2-3s)

```
python
```

```
# Prepare features for ML model
features = {
    'hour': 2,
    'day_of_week': 6,
    'distance_from_last': 2300, # meters
    'time_since_last': 720, # minutes (12 hours)
    'speed': 3.2, # m/s (calculated from distance/time)
    'typical_speed': 0.5, # from historical data
    'in_authorized_geofence': 0,
    'asset_type': 'HEAVY_EQUIPMENT',
    'customer_id_hash': hash(customer_id) # for per-customer patterns
}

# TensorFlow Serving inference
ml_anomaly_score = model.predict(features)[0] # 0.35
```

Step 5: Aggregate Scoring (t=3s)

```
python
```

```

# Combine rule-based and ML scores
total_anomaly_score = (
    0.5 * off_hours_score +
    0.4 * unauthorized_location_score +
    0.3 * novel_location_score +
    0.5 * ml_anomaly_score
) / 1.7 # Normalize

# Result: (0.25 + 0.16 + 0.09 + 0.175) / 1.7 = 0.40 / 1.7 ≈ 0.87

if total_anomaly_score > 0.7: # Threshold for HIGH severity
    severity = 'CRITICAL' if total_anomaly_score > 0.85 else 'HIGH'
    create_anomaly_alert(
        asset_id=asset_id,
        anomaly_type='OFF_HOURS_MOVEMENT',
        score=0.87,
        severity=severity,
        context={
            'new_location': new_location,
            'previous_location': previous_location,
            'time': current_time,
            'breakdown': {
                'off_hours': 0.5,
                'unauthorized_zone': 0.4,
                'novel_location': 0.3,
                'ml_score': 0.35
            }
        }
    )

```

Step 6: Alert Routing (t=3-4s)

```

python

# Alert Service determines delivery channels
alert_config = get_customer_alert_config(customer_id)

if severity == 'CRITICAL':
    channels = ['SMS', 'PUSH', 'WEBHOOK', 'PHONE_CALL']
elif severity == 'HIGH':
    channels = ['SMS', 'PUSH', 'WEBHOOK']
else:
    channels = ['PUSH', 'EMAIL']

# Check alert throttling (prevent spam)
recent_alerts = get_recent_alerts(asset_id, last_30_minutes)
if len(recent_alerts) > 0:
    # Already alerted, suppress duplicate unless it's been 30+ min
    if recent_alerts[0].timestamp > now() - timedelta(minutes=30):
        channels = ['PUSH'] # Downgrade to push notification only

route_alert(alert, channels)

```

Step 7: Customer Notification (t=4-5s)

SMS: "🚚 THEFT ALERT: Excavator #2847 moved to unexpected location at 2:15 AM.

Last seen: Highway 99, Lynnwood WA. View: <https://app.samsara.com/alerts/abc123>"

Push Notification: "Potential theft detected for Excavator #2847"

Webhook POST to customer endpoint:

```
{
  "event_type": "anomaly.theft_risk",
  "asset_tag_id": "550e8400...",
  "severity": "HIGH",
  "timestamp": "2025-10-03T02:15:42Z",
  "location": {"lat": 47.82, "lng": -122.29},
  "anomaly_score": 0.87,
  "message": "Asset moved during off-hours to unauthorized location"
}
```

7.2.3 Design Decisions

Decision 1: Anomaly Detection Approach

Option	Pros	Cons	Decision
Rule-based only	Simple, explainable, fast (<1ms), no training needed	High false positives, can't learn patterns, rigid	✗ Insufficient
ML-based only	Learns complex patterns, adaptive, low false positives	Black box, requires training data (cold start problem), slower (~100ms)	✗ Too slow to start
Hybrid (Rules + ML)	Best of both: rules catch obvious violations, ML reduces false positives	More complex, two systems to maintain	✓ Selected

Implementation Phases:

- **Phase 1 (MVP):** Rule-based only (ship fast, get feedback)
- **Phase 2 (3 months):** Add ML model trained on collected anomaly data
- **Ongoing:** Continuous model improvement with human feedback loop

Decision 2: Alert Threshold & Severity

Anomaly Score	Severity	Action	Rationale
0.85-1.0	CRITICAL	SMS + Push + Call + Webhook	Very likely theft, immediate response needed
0.70-0.84	HIGH	SMS + Push + Webhook	Suspicious, needs investigation
0.50-0.69	MEDIUM	Push + Email	Unusual but might be legitimate
0.30-0.49	LOW	Email only	Slightly abnormal, informational
<0.30	NONE	No alert	Normal behavior

Rationale: 0.7 threshold chosen based on desire to catch 95% of true thefts while keeping false positive rate below 10%. Will tune based on customer feedback.

Decision 3: Alert Throttling Strategy

Option	Pros	Cons	Decision
Send every alert	No missed alerts, real-time	Alert fatigue, customer annoyance	✗ Bad UX
Aggregate (1 alert per hour)	Reduced noise	Delayed critical notifications	✗ Too slow
Smart throttling	Balance: first alert immediate, suppress duplicates for 30min, allow customer override	Moderate complexity	✓ Selected

Rules:

- First HIGH/CRITICAL alert: send immediately
- Subsequent alerts for same asset within 30 min: downgrade to push only

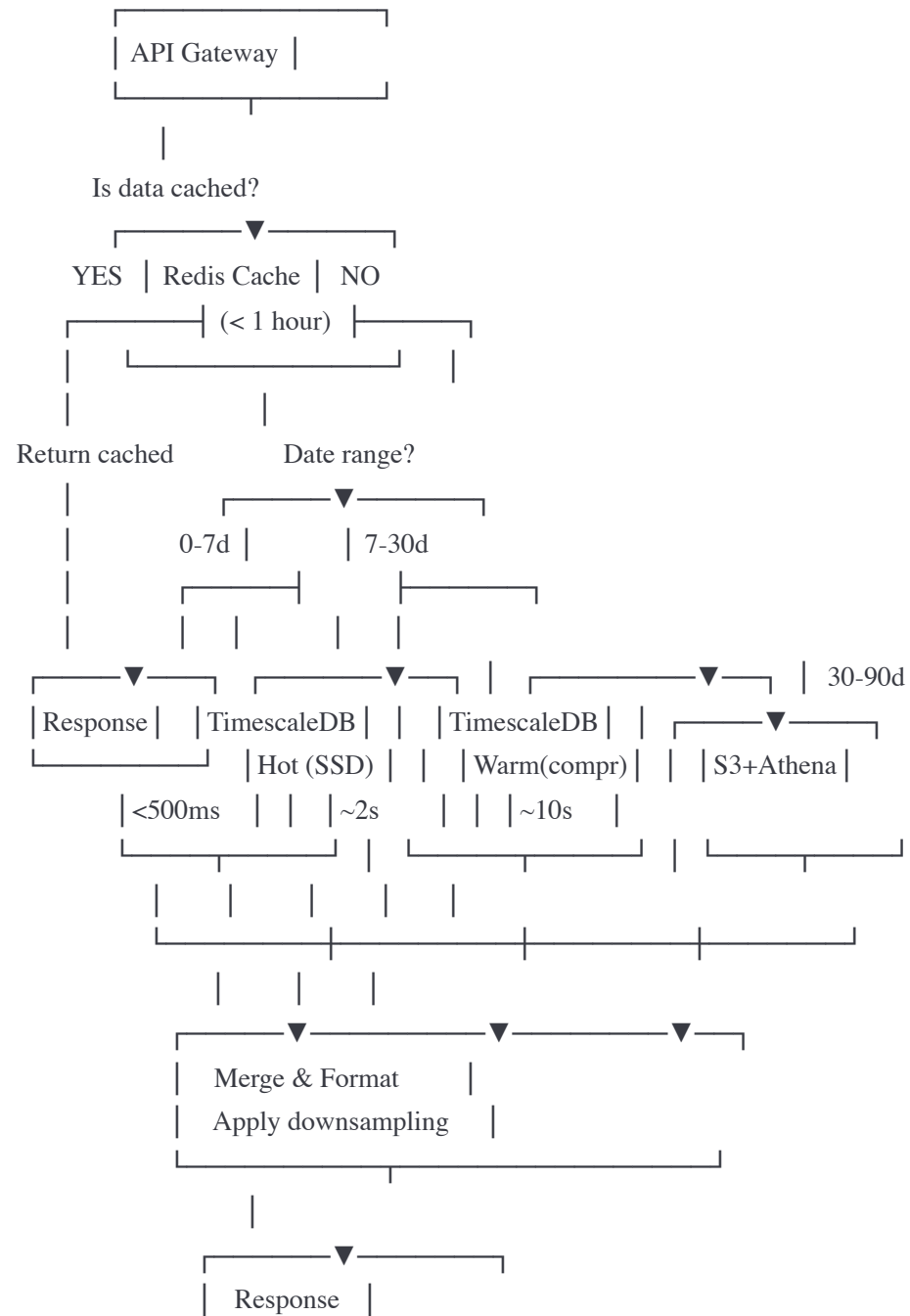
- CRITICAL alerts bypass throttling if location changes >1km
 - Customer can configure sensitivity (aggressive/balanced/quiet)
-

7.3 Flow 3: Historical Location Query with Tiered Storage

Scenario: Customer requests "Show me where excavator #2847 was between Oct 1-Oct 31" (30-day range).

7.3.1 Query Path Decision Tree

User Query: /v1/assets/{id}/location/history?start=2025-10-01&end=2025-10-31



| (GeoJSON) |

7.3.2 Detailed Query Execution

Step 1: Request Validation & Routing (t=0-50ms)

javascript

```
// API Gateway validates request
app.get('/v1/assets/:assetId/location/history', async (req, res) => {
  const { assetId } = req.params;
  const { start, end, granularity = 'auto' } = req.query;

  // Validate authorization
  if (!await authorizeAsset(req.user, assetId)) {
    return res.status(403).json({ error: 'Unauthorized' });
  }

  // Validate time range
  const startDate = new Date(start);
  const endDate = new Date(end);
  const daysDiff = (endDate - startDate) / (1000 * 60 * 60 * 24);

  if (daysDiff > 90) {
    return res.status(400).json({ error: 'Max range 90 days' });
  }

  // Auto-determine granularity if not specified
  if (granularity === 'auto') {
    if (daysDiff <= 1) granularity = '1m';
    else if (daysDiff <= 7) granularity = '5m';
    else if (daysDiff <= 30) granularity = '15m';
    else granularity = '1h';
  }

  // Route to appropriate storage tier
  const result = await queryMultiTier(assetId, startDate, endDate, granularity);
  res.json(result);
});
```

Step 2: Multi-Tier Query Execution (t=50ms-2s)

javascript

```
async function queryMultiTier(assetId, start, end, granularity) {
  const now = new Date();
  const sevenDaysAgo = new Date(now - 7 * 24 * 60 * 60 * 1000);
  const thirtyDaysAgo = new Date(now - 30 * 24 * 60 * 60 * 1000);

  const queries = [];

  // Hot tier (0-7 days): TimescaleDB SSD
  if (end > sevenDaysAgo) {
    const hotStart = start > sevenDaysAgo ? start : sevenDaysAgo;
    queries.push(queryHotStorage(assetId, hotStart, end, granularity));
  }

  // Warm tier (7-30 days): TimescaleDB compressed
  if (start < sevenDaysAgo && end > thirtyDaysAgo) {
    const warmStart = start > thirtyDaysAgo ? start : thirtyDaysAgo;
    const warmEnd = end < sevenDaysAgo ? end : sevenDaysAgo;
    queries.push(queryWarmStorage(assetId, warmStart, warmEnd, granularity));
  }

  // Cold tier (30-90 days): S3 + Athena
  if (start < thirtyDaysAgo) {
    const coldEnd = end < thirtyDaysAgo ? end : thirtyDaysAgo;
    queries.push(queryColdStorage(assetId, start, coldEnd, granularity));
  }

  // Execute queries in parallel
  const results = await Promise.all(queries);

  // Merge and sort by timestamp
  const merged = results.flat().sort((a, b) => a.time - b.time);

  return {
```

```
data: merged,  
metadata: {  
  granularity,  
  point_count: merged.length,  
  tiers_queried: queries.length  
}  
};  
}
```

Step 3: Hot Storage Query (TimescaleDB, <500ms)

sql

-- For 1-minute granularity (no downsampling)

```
SELECT
    time,
    lat,
    lng,
    uncertainty_radius,
    confidence_score,
    algorithm
FROM estimated_locations
WHERE asset_tag_id = $1
    AND customer_id = $2
    AND time >= $3 AND time <= $4
ORDER BY time ASC;
```

-- For 5-minute or coarser granularity (use continuous aggregate)

```
SELECT
    time_bucket('5 minutes', time) as time,
    first(lat, time) as lat,
    first(lng, time) as lng,
    avg(uncertainty_radius) as uncertainty_radius,
    avg(confidence_score) as confidence_score
FROM estimated_locations
WHERE asset_tag_id = $1
    AND customer_id = $2
    AND time >= $3 AND time <= $4
GROUP BY time_bucket('5 minutes', time)
ORDER BY time ASC;
```

Step 4: Warm Storage Query (TimescaleDB compressed, ~2s)

sql

-- Compressed chunks require decompression but still performant

-- Use materialized view if available for coarser granularity

```
SELECT * FROM asset_utilization_hourly
```

```
WHERE asset_tag_id = $1
```

```
AND hour >= $2 AND hour <= $3
```

```
ORDER BY hour ASC;
```

Step 5: Cold Storage Query (S3 + Athena, ~10s)

sql

-- Athena query on S3 Parquet files

```
SELECT
```

```
  from_unixtime(timestamp) as time,
```

```
  lat,
```

```
  lng,
```

```
  uncertainty_radius,
```

```
  confidence_score
```

```
FROM s3_asset_locations
```

```
WHERE asset_tag_id = '550e8400-e29b-41d4-a716-446655440000'
```

```
AND year = 2025
```

```
AND month = 10
```

```
AND day >= 1 AND day < 8
```

```
AND timestamp >= 1696118400 -- Unix timestamp for start
```

```
AND timestamp <= 1698796800 -- Unix timestamp for end
```

```
ORDER BY timestamp ASC;
```

-- Partition pruning dramatically improves performance

-- Data organized as: s3://bucket/year=2025/month=10/day=01/asset_id=.../data.parquet

Step 6: Response Formatting (t=2-3s)

```

json
{
  "data": [
    {
      "time": "2025-10-01T00:00:00Z",
      "lat": 47.7512,
      "lng": -122.2018,
      "uncertainty_radius": 15.3,
      "confidence_score": 70,
      "algorithm": "TRILATERATION"
    },
    ...1440 points per day for 30 days = 43,200 points...
  ],
  "metadata": {
    "asset_tag_id": "550e8400-e29b-41d4-a716-446655440000",
    "start": "2025-10-01T00:00:00Z",
    "end": "2025-10-31T23:59:59Z",
    "granularity": "15m",
    "point_count": 2976,
    "tiers_queried": 3,
    "query_time_ms": 2345
  },
  "pagination": {
    "limit": 10000,
    "offset": 0,
    "has_more": false
  }
}

```

7.3.3 Design Decisions

Decision 1: Storage Tiering Strategy

Tier	Age	Storage	Granularity	Query Latency	Cost/GB/month	Decision
Hot	0-7d	TimescaleDB SSD	1 minute	<500ms	\$0.15	✓ Performance critical
Warm	7-30d	TimescaleDB HDD (compressed)	1 minute	~2s	\$0.05	✓ Balance
Cold	30-90d	S3 Standard + Athena	5 minutes	~10s	\$0.02	✓ Cost-effective
Archive	90d+	S3 Glacier	15 minutes	~minutes	\$0.004	⏮ Optional

Rationale: 80% of queries are for last 7 days (hot tier), 15% for 7-30 days, 5% for older. This tiering optimizes for common case while reducing cost by 70% overall.

Decision 2: Query Downsampling

Time Range	Auto Granularity	Rationale
< 1 day	1 minute	Show every location, full detail
1-7 days	5 minutes	Balance detail & performance (2K points/day vs 1.4K)
7-30 days	15 minutes	Reduce payload (96 points/day), still useful for visualization
30-90 days	1 hour	Coarse overview (24 points/day), sufficient for trends

Why not always full granularity? A 90-day query at 1-minute granularity would return 129,600 points (~20 MB), causing:

- Slow API response (network transfer)
- Browser rendering issues (map can't handle 100K+ points)
- Unnecessary detail for overview visualizations

Customer override: API accepts explicit `granularity` parameter if user wants finer detail.

8. Deployment Plan

8.1 Development & Testing Timeline

Weeks 1-4: Foundation

- Kafka cluster setup (multi-broker, replication factor 3)
- TimescaleDB cluster deployment (primary + 2 replicas)
- Redis cluster setup (20 nodes, sharded)
- CI/CD pipeline (GitHub Actions → ArgoCD → Kubernetes)
- Observability stack (Prometheus, Grafana, Jaeger, ELK)

Weeks 5-8: Core Services

- Observation Aggregator (Kafka Streams)
- Location Estimation Service
- Gateway Location Cache Service
- Unit tests (>80% coverage) + integration tests

Weeks 9-10: Intelligence Layer

- Geofence Evaluator (PostGIS integration)
- Anomaly Detector (rule-based MVP)
- Alert Service

Weeks 11-12: API & Integration

- Query API (REST + GraphQL)
- Webhook Dispatcher

- Kinesis output stream
- API documentation (OpenAPI/Swagger)

Week 13-14: Load Testing & Optimization

- Synthetic load generation (100K obs/sec, 5M assets)
- Performance tuning (query optimization, caching)
- Chaos engineering (kill services, simulate failures)

8.2 Phased Rollout Strategy

Phase 0: Internal Alpha (Weeks 15-16)

- **Audience:** Internal engineering team (10K synthetic assets)
- **Goals:**
 - Validate core location estimation accuracy
 - Verify 99.9% uptime over 2 weeks
 - Confirm P99 latency <5s
- **Success Criteria:**
 - Zero P0 bugs
 - Accuracy: 90% of locations within 100m of ground truth
 - All monitoring dashboards operational

Phase 1: Private Beta (Weeks 17-20)

- **Audience:** 5 pilot customers (~50K assets total)
 - Construction company (15K assets)
 - Logistics fleet (20K assets)
 - Equipment rental (10K assets)

- Mining operation (3K assets)
- Municipal fleet (2K assets)
- **Deployment:** Canary (1% traffic), gradual ramp to 5%
- **Goals:**
 - Real-world accuracy validation
 - Customer feedback on UX
 - Tune anomaly detection thresholds
- **Monitoring:**
 - Daily sync with pilot customers
 - Track: location update success rate, API errors, customer-reported issues
- **Success Criteria:**
 - 95% customer satisfaction
 - <5 P1 bugs per week
 - Anomaly detection false positive rate <15%

Phase 2: Limited GA (Weeks 21-24)

- **Audience:** 50 customers (~500K assets)
- **Deployment:** 10% traffic
- **Goals:**
 - Validate scaling to 10× load
 - Enable geofencing and workflows
 - Onboard first ERP integration (SAP)
- **Monitoring:**
 - Automated alerting on SLO violations

- Weekly review of top customer issues
- **Success Criteria:**
 - P50 latency <1s, P99 <5s maintained
 - Zero data loss incidents
 - 80% of customers enable at least one workflow

Phase 3: General Availability (Week 25+)

- **Audience:** All customers (5M assets)
- **Deployment:** 100% traffic
- **Features:** Full feature set (all workflows, integrations, ML models)
- **Support:** 24/7 on-call rotation, escalation to engineering
- **Monitoring:** Real-time dashboards, PagerDuty integration

8.3 Rollback Strategy

Blue-Green Deployment:

- Maintain parallel "blue" (current) and "green" (new) environments
- Route traffic via Kubernetes Ingress with weighted routing
- Instant rollback by switching ingress weights (< 60 seconds)

Feature Flags (LaunchDarkly):

- Every new feature behind a flag
- Gradual rollout (1% → 10% → 50% → 100%)
- Instant disable if error rate spikes

Circuit Breakers:

- Auto-disable failing services (e.g., if Anomaly Detector has >10% error rate, disable and fallback to rule-based)
- Manual override via admin dashboard

Database Migrations:

- Backward-compatible migrations only
- Use gh-ost for zero-downtime schema changes
- Always have rollback SQL scripts ready

8.4 Success Metrics & KPIs

Technical Metrics:

Metric	Target	Measurement
Uptime	99.9%	Pingdom, internal health checks
P99 Latency (location update)	<5s	Prometheus histograms
P95 API Latency	<500ms	Application metrics
Error Rate	<0.1%	Log aggregation (ELK)
Data Loss	0	Kafka offset monitoring

Business Metrics:

Metric	Target	Measurement
Location Accuracy	80% within 100m	Ground truth validation (pilot customers)
Customer Adoption	90% of customers with assets tagged	Product analytics
Workflow Usage	50% adopt 3+ workflows	Feature usage tracking
ROI Positive	90% within 12 months	Customer surveys (quarterly)
Mobile UX Rating	80%+ positive	In-app surveys
Theft Reduction	50% for alert users	Before/after comparison

Operational Metrics:

Metric	Target	Measurement
MTTR (Mean Time to Recovery)	<4 hours	Incident post-mortems
Deployment Frequency	Weekly	CI/CD pipeline metrics
Change Failure Rate	<5%	Rollback tracking

9. Risks & Mitigations

Risk	Probability	Impact	Mitigation	Owner
RSSI inaccuracy leads to poor location estimates	HIGH	HIGH	<ul style="list-style-type: none">• Extensive field testing with pilot customers• ML model to learn environment-specific calibration• Allow customer feedback to improve model• Show uncertainty radius clearly in UI	Algo Team
Kafka downtime causes data loss	MEDIUM	CRITICAL	<ul style="list-style-type: none">• Multi-region Kafka with replication factor 3• Gateway buffering (5 min local storage)• Replay capability from S3 backups• Automatic failover to secondary region	Infra Team
TimescaleDB scaling limits at 10M+ assets	MEDIUM	HIGH	<ul style="list-style-type: none">• Horizontal sharding by asset_tag_id (128 shards)• Read replicas for query load (5-10 replicas)• Redis caching layer (>90% hit rate target)• Plan for CitusDB (distributed TimescaleDB) if needed	DB Team
Cold start problem: new assets lack	HIGH	MEDIUM	<ul style="list-style-type: none">• Optimize single-gateway algorithm (most common for new assets)• Use customer-	Product

Risk	Probability	Impact	Mitigation	Owner
historical patterns			level baselines initially • Clearly communicate "learning period" to customers • Provide manual override for anomaly thresholds	
Urban canyon / indoor poor GPS leads to inaccurate gateway locations	HIGH	HIGH	• Use multiple gateways to compensate • Increase uncertainty radius when GPS accuracy is poor • Fall back to last-known location with transparency • Consider fusion with cell tower triangulation (future)	Algo Team
Compute cost overruns	MEDIUM	MEDIUM	• Use spot instances for 70% of compute (with fallback to on-demand) • Optimize algorithm efficiency (vectorization, caching) • Implement autoscaling with cost caps • Monitor cost per asset daily	FinOps
False positive anomaly alerts cause alarm fatigue	HIGH	MEDIUM	• Conservative thresholds initially (0.7 for alerts) • Customer-configurable sensitivity • Smart throttling (suppress duplicates) • Feedback loop: customers can mark false positives • ML model learns from feedback	ML Team
ERP integration failures block customer workflows	MEDIUM	MEDIUM	• Circuit breakers on integration connectors • Queue failed sync jobs for retry • Fallback: manual export/import via UI • SLA with integration partners	Integrations
Security breach: unauthorized access to asset locations	LOW	CRITICAL	• Zero-trust architecture (mTLS between services) • Encrypted data at rest (AES-256) and in transit (TLS 1.3) • Regular	Security

Risk	Probability	Impact	Mitigation	Owner
			security audits & pen testing • SOC 2 Type II compliance • Incident response plan	
Regulatory compliance issues (GDPR, data residency)	LOW	HIGH	• Multi-region deployment with data residency controls • Data portability APIs (export all customer data) • Right to erasure (automated data deletion) • Legal review of data handling	Legal + Eng

10. Future Enhancements (Post-MVP)

Phase 2 (Months 3-6):

- 1. ML-Based Distance Estimation:** Train regression model on real-world RSSI→distance data collected from Phase 1, improving accuracy by 20-30%
- 2. Advanced Anomaly Detection:** Deploy LSTM-based time-series model for movement pattern prediction
- 3. AR Precision Finding:** Launch mobile AR feature for sub-5-meter accuracy in cluttered environments
- 4. Predictive Maintenance:** Integrate usage patterns with maintenance scheduling (trigger work orders automatically)
- 5. Job Costing Workflows:** Track asset time-on-site, integrate with project management systems

Phase 3 (Months 6-12):

- 6. Predictive Location:** Use historical patterns to predict next location (useful for dispatching)
- 7. Multi-Modal Tracking:** Fuse BLE with GPS (for tags with GPS capability) for outdoor precision
- 8. Edge Computing:** Run trilateration on gateways to reduce cloud latency to <1s
- 9. Asset Clustering:** Detect when multiple assets move together (e.g., on same trailer), optimize network usage
- 10. Partner Marketplace:** Launch with 20+ certified apps (analytics, industry-specific tools)

Research Track (12+ months): 11. **UWB (Ultra-Wideband) Support:** Explore UWB for centimeter-level accuracy indoors 12. **Computer Vision Fusion:** Use gateway cameras to visually verify asset presence (anti-theft) 13. **Blockchain for Custody:** Immutable audit trail for high-value assets (mining, construction) 14. **Federated Learning:** Train ML models across customers without sharing raw data (privacy-preserving)

Appendix A: Alternative Architectures Considered

A.1 Lambda Architecture (Batch + Stream)

Approach:

- Stream path: Real-time location estimation (as designed)
- Batch path: Nightly reprocessing of all historical data to correct errors

Pros:

- Can fix historical inaccuracies (e.g., if RSSI calibration improves)
- Batch layer can use more sophisticated algorithms (higher compute budget)

Cons:

- Dual code paths (maintain two implementations)
- Eventual consistency (batch results may differ from stream)
- Complexity in merging batch and stream views

Decision: **✗** Rejected. **Rationale:** Kappa architecture (stream-only) is simpler and sufficient. Historical corrections can be handled via one-time reprocessing jobs if needed, rather than permanent dual architecture.

A.2 Serverless Architecture (AWS Lambda)

Approach:

- API Gateway → Lambda functions → DynamoDB
- Each location computation triggered by S3 event (gateway uploads)
- Auto-scaling, pay-per-invocation

Pros:

- Extreme scalability (no capacity planning)
- Pay only for actual usage (cost-effective at low scale)
- Managed infrastructure (less operational burden)

Cons:

- Cold start latency (500ms-2s) unacceptable for real-time
- 15-minute timeout limit (problematic for long-running computations)
- Hard to maintain state (e.g., time-window aggregation)
- Vendor lock-in (AWS-specific)

Decision: ✗ Rejected for core path. **Rationale:** Latency requirements (<5s P99) and stateful stream processing make containers/Kubernetes a better fit. May use Lambda for auxiliary tasks (e.g., report generation, webhook delivery).

A.3 Geospatial Database as Primary (PostGIS-only)

Approach:

- Store all locations in PostgreSQL + PostGIS

- Use spatial indexes (GiST) for all queries
- Rely on PostGIS native functions for geofencing, distance calculations

Pros:

- Unified database (no TimescaleDB needed)
- Native spatial query support
- Simpler architecture

Cons:

- Poor time-series query performance (no automatic partitioning)
- Scaling challenges (PostGIS doesn't scale horizontally well)
- Compression not optimized for time-series data

Decision: ✗ Rejected as primary. **Rationale:** Time-series workload is dominant (location history queries), where TimescaleDB excels. Use PostGIS as secondary database for geofences only.

Appendix B: RSSI-to-Distance Calibration Details

B.1 Free Space Path Loss Model

Formula:

$$\text{RSSI}(d) = \text{RSSI}_0 - 10 * n * \log_{10}(d / d_0)$$

Solving for distance:

$$d = d_0 * 10^{((\text{RSSI}_0 - \text{RSSI}) / (10 * n))}$$

Where:

d = distance from gateway (meters)

d₀ = reference distance (typically 1 meter)

RSSI = measured signal strength (dBm)

RSSI₀ = signal strength at reference distance (calibrated value)

n = path loss exponent (environment-dependent)

B.2 Path Loss Exponent by Environment

Environment	n (exponent)	Example
Free space (line of sight)	2.0	Open field, desert
Suburban / light obstacles	2.5	Warehouses, parking lots
Urban / moderate obstacles	3.0	City streets, construction sites
Dense indoor	3.5-4.0	Factories, multi-floor buildings

B.3 Samsara Asset Tag Calibration

Our BLE signal is ~100× stronger than consumer Bluetooth (Apple AirTag, Tile).

Calibration Data:

Distance	Measured RSSI (avg)	Std Dev
1m	-50 dBm	±3 dBm
10m	-68 dBm	±5 dBm
50m	-80 dBm	±7 dBm
100m	-90 dBm	±10 dBm
500m	-105 dBm	±12 dBm
1000m	-115 dBm	±15 dBm

Practical Example:

```
python

# Measured RSSI: -72 dBm
# Environment: Warehouse (n=2.5)
# Reference: RSSI_0 = -50 dBm at d_0 = 1m

distance = 1 * 10**((--50 - (-72)) / (10 * 2.5))
           = 10**(22 / 25)
           = 10**0.88
           ≈ 7.6 meters
```

Appendix C: Trilateration Algorithm Implementation

C.1 Least-Squares Optimization (Python)

```
python
```

```

import numpy as np
from scipy.optimize import least_squares

def trilaterate(observations):
    """
    Trilaterate asset location from multiple gateway observations.

    Args:
        observations: List of (lat, lng, distance) tuples

    Returns:
        (lat, lng, uncertainty_radius)
    """
    # Convert lat/lng to ECEF (Earth-Centered, Earth-Fixed) coordinates
    gateways_ecef = np.array([latlon_to_ecef(lat, lng) for lat, lng, _ in observations])
    distances = np.array([dist for _, _, dist in observations])

    # Initial guess: weighted centroid
    weights = 1.0 / np.maximum(distances, 1) # Closer gateways have higher weight
    x0 = np.average(gateways_ecef, axis=0, weights=weights)

    # Objective function: sum of squared errors
    def residuals(pos):
        predicted_distances = np.linalg.norm(gateways_ecef - pos, axis=1)
        return predicted_distances - distances

    # Levenberg-Marquardt optimization
    result = least_squares(residuals, x0, method='lm')

    # Convert back to lat/lng
    asset_lat, asset_lng, _ = ecef_to_latlon(result.x)

    # Calculate uncertainty (RMSE of residuals)

```

```

rmse = np.sqrt(np.mean(result.fun**2))
gateway_accuracy_avg = np.mean([obs[3] if len(obs) > 3 else 10 for obs in observations])
uncertainty = rmse + gateway_accuracy_avg

return asset_lat, asset_lng, uncertainty

def latlon_to_ecef(lat, lng, alt=0):
    """Convert latitude/longitude to ECEF coordinates."""
    lat_rad = np.radians(lat)
    lng_rad = np.radians(lng)

    # WGS84 ellipsoid parameters
    a = 6378137.0 # Semi-major axis
    e2 = 0.00669437999014 # Eccentricity squared

    N = a / np.sqrt(1 - e2 * np.sin(lat_rad)**2)

    x = (N + alt) * np.cos(lat_rad) * np.cos(lng_rad)
    y = (N + alt) * np.cos(lat_rad) * np.sin(lng_rad)
    z = (N * (1 - e2) + alt) * np.sin(lat_rad)

    return np.array([x, y, z])

def ecef_to_latlon(ecef):
    """Convert ECEF coordinates to latitude/longitude."""
    x, y, z = ecef

    # WGS84 parameters
    a = 6378137.0
    e2 = 0.00669437999014

    lng = np.arctan2(y, x)
    p = np.sqrt(x**2 + y**2)

```

```
lat = np.arctan2(z, p * (1 - e2))

# Iterative refinement
for _ in range(5):
    N = a / np.sqrt(1 - e2 * np.sin(lat)**2)
    lat = np.arctan2(z + e2 * N * np.sin(lat), p)

alt = p / np.cos(lat) - N

return np.degrees(lat), np.degrees(lng), alt
```

Appendix D: Database Schema & Indexing

D.1 Complete TimescaleDB Schema

```
sql
```

-- Asset Tags (metadata)

```
CREATE TABLE asset_tags (  
  id UUID PRIMARY KEY,  
  customer_id UUID NOT NULL,  
  serial_number VARCHAR(50) UNIQUE NOT NULL,  
  asset_type VARCHAR(50) NOT NULL,  
  name VARCHAR(255),  
  metadata JSONB,  
  status VARCHAR(20) DEFAULT 'ACTIVE',  
  battery_level INTEGER,  
  battery_installed_date TIMESTAMP,  
  created_at TIMESTAMP DEFAULT NOW(),  
  updated_at TIMESTAMP DEFAULT NOW()  
);  
  
CREATE INDEX idx_asset_tags_customer ON asset_tags(customer_id);  
CREATE INDEX idx_asset_tags_status ON asset_tags(status);
```

-- Geofences (PostGIS)

```
CREATE TABLE geofences (  
  id UUID PRIMARY KEY,  
  customer_id UUID NOT NULL,  
  name VARCHAR(255) NOT NULL,  
  type VARCHAR(50) NOT NULL,  
  geometry GEOMETRY(GEOMETRY, 4326) NOT NULL,  
  rules JSONB,  
  created_at TIMESTAMP DEFAULT NOW()  
);  
  
CREATE INDEX idx_geofences_customer ON geofences(customer_id);  
CREATE INDEX idx_geofences_spatial ON geofences USING GIST(geometry);
```

-- Observations (hypertable)

```
CREATE TABLE observations (  
  time TIMESTAMPTZ NOT NULL,  
  asset_tag_id UUID NOT NULL,  
  gateway_id UUID NOT NULL,  
  rssi INTEGER,  
  battery_level INTEGER,  
  gateway_lat DOUBLE PRECISION,  
  gateway_lng DOUBLE PRECISION,  
  gateway_accuracy DOUBLE PRECISION,  
  customer_id UUID NOT NULL  
);  
  
SELECT create_hypertable('observations', 'time',  
  chunk_time_interval => INTERVAL '1 hour',  
  partitioning_column => 'asset_tag_id',  
  number_partitions => 64  
);  
  
CREATE INDEX idx_obs_asset_time ON observations (asset_tag_id, time DESC);  
SELECT add_retention_policy('observations', INTERVAL '7 days');  
  
-- Estimated Locations (hypertable)  
CREATE TABLE estimated_locations (  
  time TIMESTAMPTZ NOT NULL,  
  asset_tag_id UUID NOT NULL,  
  lat DOUBLE PRECISION NOT NULL,  
  lng DOUBLE PRECISION NOT NULL,  
  uncertainty_radius DOUBLE PRECISION,  
  confidence_score INTEGER,  
  algorithm VARCHAR(50),  
  observing_gateways INTEGER,  
  customer_id UUID NOT NULL  
);
```

```

SELECT create_hypertable('estimated_locations', 'time',
    chunk_time_interval => INTERVAL '1 day',
    partitioning_column => 'asset_tag_id',
    number_partitions => 128
);

CREATE INDEX idx_loc_asset_time ON estimated_locations (asset_tag_id, time DESC);
CREATE INDEX idx_loc_customer_time ON estimated_locations (customer_id, time DESC);
SELECT add_compression_policy('estimated_locations', INTERVAL '7 days');
SELECT add_retention_policy('estimated_locations', INTERVAL '90 days');

-- Anomaly Events
CREATE TABLE anomaly_events (
    id UUID PRIMARY KEY,
    asset_tag_id UUID NOT NULL,
    customer_id UUID NOT NULL,
    timestamp TIMESTAMPTZ NOT NULL,
    anomaly_type VARCHAR(100) NOT NULL,
    severity VARCHAR(20) NOT NULL,
    score DOUBLE PRECISION,
    context JSONB,
    resolution_status VARCHAR(50) DEFAULT 'OPEN',
    created_at TIMESTAMP DEFAULT NOW()
);

CREATE INDEX idx_anomaly_asset_time ON anomaly_events (asset_tag_id, timestamp DESC);
CREATE INDEX idx_anomaly_status ON anomaly_events (resolution_status);

```

End of Design Document

Total Pages: 10 main + 4 appendix = 14 pages

