NYCPS TMS: Prescriptive GIS Data Management & Integration Strategy

I. Introduction: The Geospatial Foundation

This document mandates the comprehensive, hyper-detailed strategy for managing all Geographic Information System (GIS) data underpinning the NYCPS Transportation Management System (TMS). Accurate, up-to-date, and performant geospatial data – primarily the NYC LION street centerline file and associated address information, potentially supplemented by other sources or ESRI platform data – is the absolute foundation for core TMS functions, including geocoding student addresses, generating optimized routes, calculating accurate ETAs, displaying maps, and executing geofencing rules. Given this criticality, managing the GIS data lifecycle requires a dedicated, rigorous, and specialized approach integrated within the overall Data Governance and DevSecOps frameworks.

This strategy details the non-negotiable processes for acquiring, validating, processing, storing, optimizing, updating, and integrating GIS data, including handling official updates (LION) and incorporating user-sourced corrections (from OPT staff) in a controlled, auditable manner. It emphasizes performance optimization for routing-critical spatial queries and defines the technical architecture choices within AWS GovCloud.

Inaccurate, outdated, or poorly performing GIS data will directly lead to incorrect routes, unreliable ETAs, user frustration, operational inefficiencies, and potentially impact student safety. Rigorous management is essential.

II. GIS Data Governance & Roles

Specific expertise and clear ownership are required for effective GIS data management.

Implementation How-To:

1. Establish specific roles within the project structure:
   - **GIS Lead / Specialist (Dedicated Role(s)):** Possesses deep expertise in GIS concepts, spatial databases (PostGIS), ETL processes for geospatial data, spatial analysis, and potentially ESRI technology (if used). **Responsibilities:** Owns this GIS strategy execution, designs/manages**

GIS data processing pipelines, defines spatial data models/schemas, oversees LION update process, manages user correction workflow, optimizes spatial query performance, primary technical POC for all things GIS.

- Data Steward (Address/Location Data):** Designated OPT SME responsible for defining business rules related to addresses, stop locations, borough/district boundaries, and validating user-submitted map corrections for operational accuracy. Works closely with GIS Lead.
- Data Engineer(s):** Implement and maintain automated ETL pipelines for LION updates and potentially other GIS data sources, working under guidance from GIS Lead. Implement data quality checks.
- **DBA (PostgreSQL Focus):** Responsible for tuning the RDS PostgreSQL/PostGIS instance, managing backups/recovery, monitoring performance, implementing

indexing strategies defined by GIS Lead.

- **Backend Developers (Routing Engine):** Work closely with GIS Lead to understand spatial data schema and develop optimized spatial queries for routing algorithms.
- **Technical Review Board (TRB):** Reviews significant changes to GIS architecture, data models, or processing pipelines.

2. Integrate GIS data governance within the overall TMS Data Governance Policy, ensuring alignment on classification, security, access control, and quality standards.
3. Establish a regular (e.g., monthly) **GIS Review Meeting** involving GIS Lead, Data Steward, relevant DE/DBA/Backend leads to discuss LION update status, pending corrections, performance metrics, and upcoming requirements.

Responsibility: Project Leadership (Establishing Roles), GIS Lead (Strategy Execution), Data Governance Lead (Policy Alignment).

III. Base Map Data Management: NYC LION Lifecycle

This section details the end-to-end process for acquiring, processing, storing, and updating the authoritative NYC LION dataset, which forms the core street network for routing and geocoding.

A. LION Data Acquisition & Update Monitoring

Implementation How-To:

1. **Identify Authoritative Source:** Confirm the official NYC Department of City Planning (DCP) Open Data portal as the single source for LION releases (typically Shapefile or File Geodatabase format). Document exact URL/API endpoint.
2. **Update Frequency:** Determine the official LION release cadence (e.g., quarterly, semi-annually).
3. **Automated Monitoring:** Implement a scheduled **AWS Lambda function** triggered by **EventBridge Scheduler** (e.g., daily or weekly) that:
   - Checks the DCP Open Data portal (via API or web scraping if necessary) for a new LION release version/date

compared to the currently ingested version stored in a tracking database/parameter store.

- If a new version is detected, trigger an **SNS notification** to the GIS Lead and Data Engineering team AND/OR automatically initiate the LION Ingestion Pipeline (Step B).

4. **Manual Check Backup:** GIS Lead performs manual check for new releases periodically (e.g., monthly) as a backup to automation.

Tools: AWS Lambda, EventBridge Scheduler, SNS, Python (`requests`, `beautifulsoup` if needed), Parameter Store/DynamoDB (for version tracking).

Responsibility: GIS Lead (Source Identification/Manual Check), Data Engineer (Automation Script).

Automating the check for new LION releases ensures timely updates.

B. LION Data Ingestion & Processing Pipeline (ETL)

Implementation How-To:

Implement an automated, versioned ETL pipeline, preferably using AWS Glue or orchestrated scripts (Python/Lambda/Step Functions).

1. **Trigger:** Initiated automatically by the monitoring function (Step A) or manually by GIS Lead via pipeline trigger upon new release notification.

2. **Download:** Securely download the new LION release files (Shapefile/FileGDB zip) from the DCP portal to a designated **S3 "Raw/Landing" Zone** bucket (versioning enabled).

3. **Unzip/Extract:** Unpack the downloaded archive into constituent files (.shp, .dbf, .shx, .prj, etc. or GDB contents) in a staging area within S3.

4. **Validation (Initial):**
    - Verify presence and basic integrity of all expected files.
    - Check projection/coordinate system information (.prj file or GDB metadata). **Mandatory:** Ensure it matches the project standard (e.g., EPSG:2263 - NAD83 / New York Long Island (ftUS)). Log warning/error if mismatch.
    - Perform basic record count checks against previous version or release notes if available.

5. **Transformation & Loading (PostGIS):**

- Use **AWS Glue (PySpark with GeoSpark/custom libraries)** OR a robust **Python script (using libraries like `geopandas`, `psycopg2`, `sqlalchemy-geo`)** running on Lambda/Fargate/EC2 to:
    - Read the LION Shapefile/FileGDB features from S3.
    - Perform necessary transformations: Rename columns to match target schema, handle null values appropriately, potentially filter out unnecessary attributes based on TMS requirements.
    - **Mandatory:** Explicitly re-project geometries to the standard project SRID (EPSG:2263) during loading if the source

differs, using functions like `ST_Transform`.

- Load the transformed data into designated staging tables within the **RDS PostgreSQL/PostGIS database**. Use bulk loading mechanisms (`COPY` command, `geopandas.to_postgis`) for efficiency.

6. **Post-Load Validation (Spatial & Attribute):**
    - Run SQL queries within PostGIS against staging tables to perform deeper validation:
        - Check for invalid geometries (`ST_IsValid`).
        - Verify attribute data integrity (expected value ranges, code lookups).

- Compare record counts/key metrics against previous version loaded into production tables (identify significant changes).
- Run spatial consistency checks (e.g., check for significant gaps/overlaps in street segments if topology is critical).

- Log validation results. **Quality Gate:** Pipeline halts and alerts GIS Lead/DE if critical validation checks fail.

7. **Production Table Update Strategy:**
   - **Method:** Use a "Blue/Green" table swap approach for minimal downtime during update.
     a. Load validated data into new set of tables (e.g., `lion_streets_vN+1`, `lion_address_points_vN+1`).

b. **Create necessary spatial indexes on new tables.**

c. **Run final validation queries comparing key metrics between old (`_vN`) and new (`_vN+1`) tables.**

d. **In a short, planned maintenance window (coordinated with Routing Engine team): Update database views or application configurations to point to the new `_vN+1` tables.**

e. **Keep old `_vN` tables for a short rollback period (e.g., 1-2 days), then drop them.**

- ****Audit:** Log the entire ETL process execution (start, end, steps, validation** results, success/failure). Log the production table update event.

8. **Archive:** Move processed LION source files from Raw/Landing zone to an "Archive" prefix within S3 (with Glacier transition policies).

Tools: AWS S3, AWS Lambda, EventBridge Scheduler, SNS, AWS Glue (ETL, Data Quality), Python (`geopandas`, `psycopg2`), RDS PostgreSQL/PostGIS, SQL, Terraform (for Glue jobs/Lambda), Step Functions (Orchestration).

Responsibility: Data Engineers (Pipeline Dev/Ops), GIS Lead (Validation Rules, Transformation Logic, Prod Update Oversight), DBA (DB Performance during load).

> ***Automating the LION ETL pipeline is critical for timely and reliable base map updates. Rigorous validation at each step prevents propagation of bad data.***

IV. ESRI ArcGIS Integration Strategy (Conditional)

We will prioritize using cloud-native solutions (AWS Location Service, PostGIS) for GIS capabilities. Integration with self-managed ESRI ArcGIS Enterprise will only be pursued if a mandatory, routing-critical function

(identified during detailed design/POC) cannot be adequately fulfilled by native options within AWS GovCloud.

Self-managing ArcGIS Enterprise in AWS GovCloud adds significant complexity, cost (licensing, infrastructure, specialized skills), and operational overhead compared to using managed services.

Implementation How-To (If ESRI is Deemed Absolutely Necessary):

1. ****Formal Justification & Approval:** Document the specific, critical functional gap(s) in AWS Location Service/PostGIS and the justification for needing ArcGIS Enterprise. Obtain formal approval from the TRB and Steering Committee, acknowledging the cost/complexity implications.**
2. ****Architecture & Deployment (IaC):****
    - **Design a secure, highly available ArcGIS Enterprise deployment (Server, Portal, Data Store - potentially using enterprise GDB on RDS PostgreSQL/PostGIS) on EC2 instances within private VPC subnets in AWS GovCloud, following ESRI best practices for AWS.**
    - **Provision the entire ESRI stack using **Terraform**.**

- **Implement robust monitoring (CloudWatch, ArcGIS Monitor) and backup strategies for the ESRI components.**
- **Manage ESRI software licensing according to vendor terms.**

3. **Data Synchronization Strategy:**

- **Define the mechanism for keeping the ESRI Enterprise Geodatabase synchronized with the authoritative LION data (now residing in PostGIS) and potentially user corrections. Options:**
    - ****ETL (Preferred if feasible):** Develop dedicated ETL jobs (Glue, FME Server on EC2, Python `arcpy` scripts) to regularly export data from PostGIS and load/update the Enterprise GDB.**
    - ****GDB Replication:** If using RDS PostgreSQL for the enterprise GDB,**

investigate ESRI geodatabase replication features, but be aware of complexity and potential limitations.

- Schedule and monitor synchronization jobs rigorously. Implement validation checks post-sync.

4. **Integration with TMS:**
    - Routing Engine or other TMS microservices interact with ArcGIS Server via its published REST APIs (Map Services, Geocode Services, Network Analyst Services) over secure HTTPS connections within the VPC.
    - Manage authentication to ArcGIS services securely (e.g., using tokens).

Tools (If Used): ESRI ArcGIS Enterprise (Server, Portal, Data Store), ArcGIS Pro (for admin/publishing), ArcGIS Monitor, AWS EC2, RDS PostgreSQL/PostGIS, FME Server, Python (`arcpy`), Terraform.

Responsibility (If Used): GIS Lead (ESRI Expertise), Cloud Architect (AWS Infra), DevOps (IaC/Deployment), DBA (GDB on RDS), Data Engineers (Sync ETL), Security (Securing ESRI Stack).

V. Spatial Data Storage Architecture (AWS GovCloud)

Our primary strategy leverages the power and maturity of PostGIS on managed RDS within AWS GovCloud for storing and querying authoritative geospatial data.

Implementation How-To:

1. **Primary Datastore:** **AWS RDS for PostgreSQL** with the **PostGIS extension enabled**.
    - Provisioned via Terraform module (\`modules/rds_postgres/\`).
    - Deployed in a **Multi-AZ** configuration for high availability.
    - Located in **private subnets**.
    - Security Groups configured for least privilege access (only from specific App Tier SGs on port 5432).
    - Encryption at rest enabled using KMS (dedicated CMK).

- Automated backups enabled with cross-region replication to DR region.
- Performance Insights enabled for query tuning.
- Appropriately sized instance class (e.g., `db.r6g` Graviton or `db.m6i` Intel) based on performance testing, with sufficient RAM (`work_mem` crucial for spatial queries).

2. **Data Schema:**
    - Use `geometry` data types for storing spatial features (street segments, address points, geofences, user corrections).
    - **Mandate use of a consistent Spatial Reference Identifier (SRID)** for *all* geometry data stored in the database, matching the official NYC projection (e.g., **EPSG:2263** - NAD83 / New York Long Island (ftUS)). Enforce via database constraints if possible.
    - Store LION street segments, address ranges, aliases, and other relevant attributes in dedicated, versioned

tables (e.g., `lion_streets_vXX`, `lion_address_points_vXX`).

- Store user-submitted map corrections in a separate table (e.g., `map_corrections`) linking to the affected LION feature ID, storing the corrected attribute value, status (pending, approved, rejected, implemented, retired), effective dates, submitter ID, approver ID, and timestamps (audit trail).
- Create database views (e.g., `current_streets_view`) that intelligently combine the latest approved LION version with active, approved corrections for use by the routing engine.

3. **Spatial Indexing (Mandatory):** Create **GiST (Generalized Search Tree) indexes** on *all* `geometry` columns that will be used in spatial query predicates (`WHERE` clauses using `ST_Intersects`, `ST_DWithin`, `ST_Contains`, etc.). Regularly analyze and potentially `REINDEX` spatial indexes if fragmentation occurs.
4. **Attribute Indexing:** Create standard B-tree indexes on non-spatial attributes frequently used for

filtering (e.g., street names, address range numbers, borough codes).

Tools: AWS RDS PostgreSQL, PostGIS Extension, SQL, Terraform.

Responsibility: DBA, GIS Lead, Data Architect, DevOps (IaC).

VI. Spatial Query Performance Optimization Strategy

Ensuring millisecond-level performance for critical spatial queries underpinning real-time routing and geocoding is paramount.

Implementation How-To (Continuous Process):

1. **Query Design Best Practices:**
    - **Mandatory Index Usage:** Ensure all spatial queries leverage appropriate GiST indexes. Use `EXPLAIN ANALYZE` extensively during development and testing to verify index usage and identify bottlenecks (sequential scans on geometry columns are unacceptable for performance-critical queries).

- **Bounding Box First:** Use efficient bounding box intersection operators (`&&`) as a first filter in spatial queries where appropriate, before applying more computationally expensive functions like `ST_Distance` or `ST_DWithin` on the reduced dataset.
- **Function Selection:** Use the most efficient PostGIS function for the required task (e.g., `ST_DWithin` is often faster than `ST_Distance < radius` when using an index).
- **Limit Results:** Use `LIMIT` clauses where only the nearest feature(s) are needed.
- **Pre-Calculation:** Identify expensive calculations performed repeatedly in queries and explore pre-calculating results and storing them in indexed columns or materialized views if the underlying data changes infrequently.
- **Projection Consistency:** Ensure all geometries involved in a spatial operation use the *same* SRID

(EPSG:2263) to avoid implicit, costly `ST_Transform` operations within the query execution plan.

2. **Database Tuning (PostgreSQL/PostGIS):**
    - Tune key PostgreSQL parameters via RDS Parameter Groups based on workload and instance size: `shared_buffers` (e.g., 25% of instance RAM), `work_mem` (significantly increase for complex spatial joins/sorts, monitor usage), `maintenance_work_mem` (for index creation/vacuum).
    - Run `VACUUM ANALYZE` regularly (or configure autovacuum aggressively) on spatial tables to update statistics and prevent bloat, which is critical for query planner efficiency.
    - Monitor RDS Performance Insights to identify wait events and resource bottlenecks related to spatial queries.

3. **Application-Level Caching:**
    - Identify spatial query results that are relatively static or change infrequently (e.g., geocoding results for fixed

school addresses, lookup of street segments by name).

- Implement caching for these results using **AWS ElastiCache (Redis or Memcached)** deployed via Terraform.
- Application code checks the cache first before querying PostGIS. Implement appropriate cache invalidation strategies when underlying data (LION updates, corrections) changes.

4. **Performance Testing:** Include specific test scenarios simulating high-concurrency spatial queries (geocoding bursts, simultaneous route calculations) during performance testing (as per Test Strategy) to validate latency and throughput against NFRs.

Tools: PostGIS SQL, `EXPLAIN ANALYZE`, AWS RDS Performance Insights, AWS ElastiCache, Performance Testing Tools (k6, JMeter).

Responsibility: Backend Developers (Query Writing), GIS Lead (Query Optimization Guidance), DBA (DB Tuning), SRE/Ops (Caching Infra), Performance Engineer (Testing).

VII. User-Sourced Map Correction Workflow (Controlled & Auditable)

Leveraging the operational expertise of OPT users (Routers, Admins) to identify map inaccuracies is valuable, but requires a highly controlled process to maintain data integrity.

Implementation How-To:

1. ****Submission Interface:****
    - **Provide a dedicated form/interface (potentially integrated into the OPT Admin Console map view using tools like Leaflet Draw, or a standalone Jira Service Management portal/form) for authorized OPT users to submit map correction requests.**
    - ****Mandatory Fields:** Location (user clicks on map or enters address/intersection), Type of Correction (Wrong Street Name, Incorrect One-Way, Missing Segment, Bad Address Range, Turn Restriction, Other), Description of Issue, Suggested Correction,**

Justification/Source (if known), Submitter ID (auto-captured), Submission Timestamp. Allow attachment of screenshots/supporting docs.

2. **Tracking:** Each submission creates a "Map Correction Request" ticket in Jira/ADO, assigned to the GIS Lead/Data Steward queue. Status: `Submitted`.
3. **Triage & Validation:** GIS Lead/Data Steward reviews incoming requests daily/weekly.
   - Check for duplicates.
   - Assess clarity and completeness. Request more info from submitter via Jira if needed.
   - Perform initial validation: Does the request seem plausible? Cross-reference with recent satellite imagery, NYC DOT street closure feeds, other official sources. Field verification by OPT staff may be requested for ambiguous cases.
   - Update Jira status: `Under Review`.
4. **Decision & Approval:** Based on validation, GIS Lead/Data Steward makes a decision:

- **Approve:** Correction is valid and needed.
- **Reject:** Correction is invalid, already fixed in pending LION update, or cannot be verified. Provide clear rationale in Jira ticket.
- **Defer:** Requires further investigation or depends on other factors.

Document decision and rationale in Jira ticket. Update status: `Approved`, `Rejected`, `Deferred`.

5. **Implementation (Into Corrections Layer):**
    - If **Approved**, GIS Lead/Data Steward implements the correction *not* by modifying the base LION tables, but by adding/updating a record in the dedicated `map_corrections` table in PostGIS.
    - The record includes: Link to original LION feature ID(s), the attribute being corrected (e.g., `street_name`, `oneway_direction`, `geometry`), the new corrected value, status (`Implemented`), effective start date (can be immediate or future), optional

expiration date, link to the Jira approval ticket, approver ID, implementation timestamp.

- For geometry changes (new segments, splits), store the new geometry in the corrections table. For attribute changes, store the new attribute value.
- Update Jira status: `Implemented`.

6. **Integration with Routing Engine:** The routing engine's spatial queries *must* be designed to:
    - First query the `map_corrections` table for any active, approved corrections relevant to the area/features being considered.
    - If a relevant correction exists, use the corrected value/geometry from the `map_corrections` table.
    - Otherwise, use the default value/geometry from the base `lion_streets_vXX` table.
    - This ensures corrections are applied dynamically without altering the authoritative base map data directly.

7. **Feedback Loop:** Configure Jira automation or manual process to notify the original submitter when their request status changes (Approved, Rejected, Implemented).

8. **Periodic Reconciliation:** During each LION update cycle (Step III.B), GIS Lead reviews all `Implemented` corrections in the `map_corrections` table. If a correction is now reflected in the new LION base data, mark the correction status as `Retired/Superseded`.

Tools: Jira/ADO (Tracking/Workflow), Confluence (Process Documentation), PostGIS, SQL, potentially OPT Admin Console UI elements (Leaflet Draw/similar for submission).

Responsibility: GIS Lead (Process Owner, Implementation), Data Steward (Validation/Approval), OPT Users (Submission), Backend Developers (Integrating correction logic into routing queries).

Maintaining corrections in a separate, audited layer preserves the integrity of the base LION data while allowing timely operational adjustments. Requires careful query design in the routing engine.

VIII. Integration with Routing Engine & Other Systems

Ensuring the routing engine and other consumers have efficient, reliable access to accurate GIS data.

> *Implementation How-To:*
>
> 1. ****Routing Engine Access:****
>   - **The routing engine (running on EC2/ECS/Fargate) connects directly to the RDS PostgreSQL/PostGIS database instance via secure connections within the VPC (using Secrets Manager for credentials).**
>   - **Queries *must* be optimized for performance, leverage spatial indexes, and query the combined view of LION + active corrections.**
>   - **Consider using RDS Read Replicas if routing query load becomes a bottleneck for the primary write instance (requires application logic to direct reads appropriately).**
> 2. ****Geocoding Service:** Address geocoding (converting student/stop addresses to coordinates) can be handled via:**

- **PostGIS Geocoder:** Utilize the built-in PostGIS geocoder functions (requires TIGER data loading/setup).
- **AWS Location Service:** If available/approved in GovCloud, use its geocoding API.
- **Self-Managed ESRI Geocode Service:** If ArcGIS Enterprise is deployed.
- Implement as a dedicated internal microservice accessed via API by other TMS components (Student Management, Routing Engine). Cache results aggressively (ElastiCache) as addresses change relatively infrequently.

3. **Map Visualization:** User interface modules (Admin Consoles, Parent/Student Apps) will typically use:
    - **Base Map Tiles:** Consume standard base map tile services (e.g., from AWS Location Service Maps, potentially ESRI base maps if licensed, or approved open source providers compatible with GovCloud usage).

- **Overlay Data (Routes, Stops, Bus Locations):** Retrieve dynamic data (real-time bus locations, calculated route lines, stop points) via dedicated TMS backend APIs, displayed as overlays on the base map using frontend libraries (Leaflet, Mapbox GL JS, platform-native map SDKs).

Responsibility: Backend Developers (Routing Engine Queries), GIS Lead (Geocoding Strategy/Service), Frontend/Mobile Developers (Map Integration), Cloud Architect (Service Selection).

IX. Ongoing Maintenance & Monitoring

Continuously monitor the health, performance, and accuracy of the GIS data platform.

Implementation How-To:

1. **LION Update Automation Monitoring:** Monitor the automated LION update check (Lambda) and ETL pipeline (Glue/Step Functions) execution via CloudWatch Alarms. Alert GIS Lead/DE on failures.

2. **Database Monitoring:** Monitor core RDS PostgreSQL metrics via CloudWatch (CPU, Memory, IOPS, Disk Space, Replica Lag). Set alarms on critical thresholds. Use RDS Performance Insights to proactively identify slow or resource-intensive spatial queries.

3. **Spatial Index Maintenance:** Schedule periodic `REINDEX` operations or `VACUUM ANALYZE` on key spatial tables if performance degradation related to index bloat/fragmentation is observed.

4. **Data Quality Monitoring:** Monitor results from automated DQ checks (Glue DQ reports, Lambda validation results sent to CloudWatch). Alert Data Stewards on significant quality issues.

5. **Geocoding Service Monitoring:** Monitor latency, error rates, and cache hit ratio (if applicable) of the geocoding service. Track geocoding success rate as a key quality indicator.

6. **Backup/Recovery Verification:** Ensure GIS database backups are included in standard RDS backup/restore testing procedures.

Responsibility: SRE/Ops Team, DBA, GIS Lead, Data Engineers.

X. Conclusion: Ensuring Geospatial Accuracy & Performance

This Prescriptive GIS Data Management and Integration Strategy provides the detailed framework necessary to manage the critical geospatial foundation of the NYCPS TMS. By implementing rigorous, automated processes for LION data lifecycle management, leveraging the power of PostGIS on AWS RDS GovCloud, optimizing spatial query performance relentlessly, establishing a controlled workflow for user-sourced corrections, and continuously monitoring data quality and system health, we ensure the highest levels of accuracy, reliability, and performance for routing, geocoding, and map visualization.

The emphasis on clear roles (especially the dedicated GIS Lead), standardized procedures, automated validation, performance tuning, and integration with overall data governance provides the necessary control and assurance for this fundamental system component. Adherence to this strategy is critical for delivering accurate ETAs, efficient routes, and a trustworthy user experience, ultimately supporting the core mission of safe and reliable student transportation.