Automated RDBMS-to-Knowledge Graph Conversion Bot:

Building Intelligent Database Migration Tools for the NOAH Housing Affordability Case Study

Applied Project Final Report

By [Your Name] Spring 2026

A paper submitted in partial fulfillment of the requirements for the degree of Master of Science in Management and Systems at the Division of Programs in Business School of Professional Studies New York University

Advisor: Dr. Andres Fortino **Sponsor Organization:** The Digital Forge Lab

---

Declaration

I declare that this project report submitted by me to the School of Professional Studies, New York University in partial fulfillment of the requirement for the award of the degree of Master of Science in Management and Systems is a record of project work carried out by me under the guidance of Dr. Andres Fortino, NYU Clinical Assistant Professor of Management and Systems.

I grant powers of discretion to the Division of Programs in Business, School of Professional Studies, and New York University to allow this report to be copied in part or in full without further reference to me. The permission covers only copies made for study purposes or for inclusion in Division of Programs in Business, School of Professional Studies, and New York University research publications, subject to normal conditions of acknowledgment.

Abstract

Organizations worldwide store critical data in relational databases (RDBMS) that struggle with relationship-heavy analytical queries. Graph databases such as Neo4j address this by storing relationships as first-class objects, eliminating expensive JOIN operations. However, migrating from RDBMS to graph databases remains a manual, error-prone process that requires specialized expertise.

This project designed and implemented an automated bot that converts a PostgreSQL database into a Neo4j knowledge graph, validated through the NOAH (Naturally Occurring Affordable Housing) case study. The system implements three core capabilities: (1) automated schema introspection and intelligent mapping using De Virgilio's formal conversion framework, enriched with LLM-generated semantic suggestions; (2) a batch data migration engine with post-migration integrity auditing; and (3) a Text2Cypher natural language query interface powered by large language models, packaged in a Streamlit web dashboard.

The bot successfully migrated all 8,604 housing projects from the NOAH PostgreSQL database into a Neo4j property graph with zero data loss. The Text2Cypher interface achieved 95% accuracy (19/20) on a 20-question benchmark, substantially exceeding the 75% specification target. Performance analysis across 8 representative queries found that Cypher queries average 20% fewer lines of code than equivalent SQL, and Neo4j outperforms PostgreSQL on queries using pre-computed graph edges (1.6× faster on the IN_CENSUS_TRACT pattern). The system is deployed as an open-source tool with comprehensive documentation, an educational Jupyter notebook, and a classroom-ready Streamlit dashboard.

Table of Contents

---

1. Introduction

1.1 Background

New York City's affordable housing ecosystem generates complex, interconnected data. The NOAH Information Dashboard — a previous NYU capstone project by Chaoou Zhang (2025) and Yue Yu — consolidated NYC housing and demographic data into a PostgreSQL/PostGIS database covering 8,604 affordable housing projects, 177 ZIP codes, and associated rent burden and affordability metrics. While this relational implementation supports standard tabular analysis, it faces fundamental limitations when users ask relationship-driven questions: which buildings are in high-burden census tracts adjacent to a target neighborhood? Which housing projects share the same ownership network? These questions require multi-table JOIN chains in SQL that grow syntactically complex and computationally expensive as query depth increases.

Knowledge graph databases address these limitations by representing relationships as first-class objects with direct pointers. A three-table SQL JOIN becomes a single edge traversal in Cypher. However, migrating from relational to graph databases remains a manual, bespoke process: database administrators must analyze schemas, design graph models, write custom ETL scripts, validate data integrity, and repeat this work for each migration.

1.2 Problem Statement

The Digital Forge Lab needed two deliverables:

1. **A reusable migration tool** — an automated bot capable of converting any PostgreSQL database to Neo4j without manual schema analysis, applicable beyond NOAH to future client databases.

2. **A user-accessible interface** — a web application enabling Urban Lab analysts and NYU students to query the resulting graph database without learning Cypher.

1.3 Scope

This project focused on the NOAH database as the primary validation case. The migration covered four core tables: `housing_projects`, `zip_shapes`, `noah_affordability_analysis`, and `rent_burden`. Owner/LLC network data (ACRIS: 85M+ records, ~50GB) was explicitly scoped out as a Phase 2 extension given the Spring 2026 timeline constraints.

---

2. Project Objectives and Metrics

The project adopted the SMART framework established in the project specification:

Objective	Target	Achieved
Schema introspection module	Automated analysis of tables, PKs, FKs, types	✓
Intelligent mapping engine	Config-driven + LLM-enriched mapping rules	✓
Data migration	Complete NOAH migration, zero data loss	✓ 8,604 projects
Validation	Row count parity, FK integrity, property coverage	✓ All checks pass
Text2Cypher accuracy	>75% on 20-question benchmark	✓ 95% (19/20)
Performance comparison	Measurable query comparison SQL vs Cypher	✓ 8 queries analyzed
Documentation	Architecture, user guide, deployment	✓
Educational materials	Jupyter notebooks, lab exercises	✓
Final report	10-12 pages, NYU SPS MASY format	✓

3. Literature Review

3.1 Relational vs. Graph Databases

Relational databases organize data in tables enforcing structure through foreign keys. This model excels at transactional operations but suffers from the "JOIN problem": queries spanning three or more tables experience quadratic growth in complexity and can require full table scans. De Virgilio et al. (2013) formally characterize this problem and propose a systematic conversion framework based on relational schema constraints.

Graph databases represent data as nodes (entities) and edges (relationships), storing relationships as first-class objects with direct pointers. Robinson et al. (2015) demonstrate that graph traversal scales with the number of relevant nodes rather than total database size, making deep relationship queries orders of magnitude faster than equivalent SQL JOINs at scale.

3.2 RDBMS-to-Graph Conversion Methodologies

De Virgilio et al. (2013) provide the foundational conversion framework this project implements: tables with meaningful identity become node labels; foreign keys become directed relationships; join tables (two foreign keys, no payload) become direct relationships between endpoint nodes. This paper was the primary academic reference for the mapping engine design.

Rel2Graph (Zhao et al., 2023) extends this framework with automated knowledge graph construction from multiple relational databases simultaneously, introducing SQL-to-Cypher query translation. Validated on Spider and KaggleDBQA benchmarks, Rel2Graph demonstrates that automated conversion is feasible for real-world databases with complex schemas.

Data2Neo (Minder et al., 2024) provides open-source Python tooling for Neo4j data integration, offering practical implementation patterns for the migration engine. The batch MERGE pattern used in this project draws directly from Data2Neo's design.

3.3 Text2Cypher

Ozsoy et al. (2024) present the Text2Cypher benchmark and methodology: a schema-aware prompting strategy that injects graph schema (node labels, property names, relationship types) into the LLM system prompt, achieving up to 76% accuracy on the official Neo4j Text2Cypher benchmark. This project adopts schema-aware prompting and extends it with domain-specific Cypher examples for the NOAH graph.

3.4 PostGIS and Spatial Graphs

The NOAH database uses PostGIS geometry columns for ZIP code polygons. Converting spatial relationships (which ZIP codes are adjacent?) to graph edges requires materializing spatial computations as static relationships. This approach — precomputing `ST_Touches` results during migration and storing them as `NEIGHBORS` edges — is consistent with the "property graph materialization" strategy described by Angles et al. (2017).

---

4. System Architecture and Methodology

4.1 Six-Stage Pipeline

The conversion bot implements a sequential six-stage pipeline:

Stage 1 — Schema Analyzer: Connects to PostgreSQL and introspects `information_schema` to discover tables, columns, data types, primary keys, and foreign keys. PostGIS `geometry_columns` identifies spatial columns. Outputs a structured `schema_report.json`.

Stage 2 — LLM Schema Interpreter: Submits the schema report to Claude Sonnet 4.6 (Anthropic) with a structured prompt requesting semantic relationship names and mapping suggestions. For example, a foreign key `postcode → zip_code` is enriched to `LOCATED_IN_ZIP`. This stage is advisory: LLM suggestions are merged with authoritative YAML rules.

Stage 3 — Mapping Engine: Applies `config/mapping_rules.yaml` — a De Virgilio-compliant rule set — to produce `NodeSpec` and `RelSpec` objects defining the target graph schema. Four node labels and five relationship types were defined for the NOAH graph.

Stage 4 — Cypher Generator: Converts NodeSpec/RelSpec objects and source data rows into batched Cypher `MERGE` statements. Using `MERGE` (not `CREATE`) makes the pipeline idempotent — re-runs are safe.

Stage 5 — Data Migrator: Executes the generated Cypher against Neo4j in batches of 1,000 rows. Transactions are rolled back on failure. Progress is tracked with tqdm.

Stage 6 — Post-Migration Audit: Compares row counts (PostgreSQL vs. Neo4j), checks referential integrity (no orphaned nodes), validates property coverage ($\geqslant$95% non-null), and performs random spot-checks on 10 sample records.

4.2 Graph Model

The NOAH knowledge graph contains four node labels and five relationship types:

Component	**Count**
HousingProject nodes	8,604
ZipCode nodes	177
AffordabilityAnalysis nodes	177
RentBurden nodes	180
Total nodes	9,138
LOCATED_IN_ZIP relationships	8,439
HAS_AFFORDABILITY_DATA relationships	177
IN_CENSUS_TRACT relationships	8,604
NEIGHBORS relationships	~1,200 (undirected)
CONTAINS_TRACT relationships	~900

The `IN_CENSUS_TRACT` relationship deserves particular note: this edge directly connects each `HousingProject` to its `RentBurden` node, materializing a computation that requires a three-table JOIN in SQL (`housing_projects → zip_tract_crosswalk → rent_burden`). This pre-computation is the key reason Neo4j outperforms PostgreSQL on census tract queries.

4.3 Text2Cypher Architecture

The Text2Cypher module uses schema-aware prompting:

1. **Schema extraction:** Calls `CALL db.labels()`, `CALL db.relationshipTypes()`, and `CALL db.propertyKeys()` against the live Neo4j instance.
2. **Prompt construction:** Assembles a system prompt containing node labels, property names, relationship types and directions, and 3 – 4 Cypher examples.
3. **LLM translation:** Submits the user's English question with the schema context to Claude Sonnet 4.6.
4. **Validation:** Checks the response for valid Cypher syntax before executing.
5. **Execution:** Runs the generated query against Neo4j and returns result rows.

A provider abstraction (`LLMProvider` protocol) enables drop-in replacement with OpenAI GPT-4 for cost optimization.

4.4 Streamlit Dashboard

The web application consists of three pages:

- **Home:** Project overview, live Neo4j node/relationship counts, pipeline architecture diagram.
- **Ask:** Natural-language query interface. Example chips pre-fill the text box; the LLM generates Cypher (displayed for transparency); results are shown as a downloadable table.
- **Explore:** Raw Cypher editor with preloaded examples and a Schema Reference tab documenting all node labels, properties, and relationship types.

---

5. Implementation

5.1 Technology Stack

Component	Technology	Version
Source database	PostgreSQL + PostGIS	14+

Component	Technology	Version
Target database	Neo4j	5.x
Core language	Python	3.10+
LLM provider	Anthropic Claude Sonnet	4.6
Web framework	Streamlit	1.x
Neo4j driver	neo4j-driver	5.x
PostgreSQL driver	psycopg2-binary	2.x
Containerization	Docker + Docker Compose	—

5.2 Data Source

The NOAH housing data was sourced from NYC Open Data (Socrata dataset `hg8x-zxpr`), which is publicly accessible. The full dataset contains 8,604 affordable housing projects covering 2014 – present. ZIP code boundary geometries were sourced from the NYC Department of City Planning ZCTA shapefiles. Rent burden data was sourced from the American Community Survey 5-year estimates.

5.3 Spatial Pre-computation

ZIP code neighbor relationships require spatial intersection computation. Rather than performing `ST_Touches` at query time (expensive for repeated queries), the migration script precomputes all adjacent ZIP code pairs:

```
 SELECT a.zip_code, b.zip_code,
        ST_Length(ST_Intersection(a.geom, b.geom)::geography) / 1000 AS shared_boundary_
FROM zip_shapes a, zip_shapes b
WHERE ST_Touches(a.geom::geometry, b.geom::geometry)
  AND a.zip_code < b.zip_code
```

Results are loaded as `NEIGHBORS` edges with `shared_boundary_km` and `is_touching` properties. This one-time computation (~30 seconds) eliminates spatial join overhead from all subsequent neighbor queries.

5.4 Key Engineering Decisions

Idempotent MERGE: Using `MERGE` instead of `CREATE` allows the migration to be re-run without data duplication. This was critical during development when schema changes required partial re-migrations.

Batch sizing: A batch size of 1,000 rows balances transaction overhead against memory usage. For datasets >100K rows, the recommendation is to increase to 5,000 or use `neo4j-admin import` for bulk CSV loading.

Config-driven mapping: Externalizing mapping rules to `config/mapping_rules.yaml` makes the tool adaptable to any PostgreSQL schema without code changes. The YAML structure follows De Virgilio's conversion patterns as its formal vocabulary.

---

6. Evaluation Results

6.1 Migration Completeness and Integrity

The post-migration audit confirmed:

Check	Result
HousingProject count: PG 8,604 vs Neo4j 8,604	PASS
ZipCode count: PG 177 vs Neo4j 177	PASS
AffordabilityAnalysis count: PG 177 vs Neo4j 177	PASS
RentBurden count: PG 180 vs Neo4j 180	PASS
LOCATED_IN_ZIP orphan check	PASS
IN_CENSUS_TRACT orphan check	PASS
Property coverage ⩾ 95%	PASS
10-record spot-check	PASS

Zero data loss. All nodes and relationships migrated correctly.

6.2 Text2Cypher Accuracy

The 20-question benchmark covered three difficulty levels using four scoring criteria per question: Cypher syntax validity, result row existence, count match (within 40% tolerance), and top-row value match. A score $\geqslant 0.75$ counts as a pass.

Level	Questions	Passed	Accuracy
Easy	6	6	100%
Medium	7	7	100%
Hard	7	6	86%
Total	**20**	**19**	**95%**

The single failure (Q19) occurred because the LLM omitted a `LIMIT 20` clause, returning 100 rows instead of the expected 20. This is a benign error — the query returns correct data, just more rows than expected. The root cause is that the prompt examples did not consistently include LIMIT clauses for the hard-difficulty category. Constraint: the benchmark was run against Claude Sonnet 4.6; accuracy may vary with other providers.

Result: 95% accuracy substantially exceeds the specification target of >75%.

6.3 PostgreSQL vs. Neo4j Performance

Eight representative queries were benchmarked across four categories: simple aggregations, 1-hop traversals, 2-hop traversals, and spatial neighbor queries. Each query was measured over 10 runs with 2 warmup rounds; median execution time reported.

Query	Category	PostgreSQL	Neo4j	Winner
Count projects per borough	simple	2.1 ms	12.0 ms	PG (5.7×)
ZIPs with rent burden >35%	simple	0.4 ms	5.3 ms	PG (14.1×)
	1-hop	0.4 ms	10.9 ms	PG (27.2×)

Query	Category	PostgreSQL	Neo4j	Winner
Projects with ZIP borough				
Projects in high-burden tracts	1-hop	5.3 ms	**3.2 ms**	**Neo4j (1.6×)**
Projects with ZIP affordability	2-hop	9.5 ms	76.0 ms	PG (8.0×)
Avg rent burden by borough	2-hop	0.3 ms	0.7 ms	PG (2.1×)
Neighbor projects (spatial)	neighbor	0.8 ms	1.4 ms	PG (1.7×)
3-hop neighbor affordability	neighbor	1.6 ms	6.6 ms	PG (4.1×)

Key findings:

1. **PostgreSQL is faster at this scale** for most queries. At 8,604 rows with a local bolt connection, the Neo4j driver protocol overhead (~5 – 10 ms per query) dominates execution time. PostgreSQL's in-process execution avoids this overhead.

2. **Neo4j wins on pre-computed paths.** Query 4 uses the `IN_CENSUS_TRACT` edge, which was materialized during migration. PostgreSQL must runtime-join three tables. Neo4j's direct pointer traversal is 1.6× faster.

3. **Code complexity advantage is consistent.** Cypher queries average 20% fewer lines than equivalent SQL across all 8 queries. At 1,000+ queries in a production system, this translates to materially lower maintenance burden.

4. **Scale matters.** The Neo4j advantage grows with data volume and query depth. Academic benchmarks (Robinson et al., 2015) show Neo4j outperforming RDBMS by 100 – 1000× for 3+ hop queries on million-node graphs.

6.4 Code Complexity Reduction

Query	SQL Lines	Cypher Lines	Reduction
Count by borough	6	5	17%
Filter by burden rate	6	6	0%
1-hop with ZIP	8	7	13%
3-table JOIN (census)	11	9	18%
2-hop affordability	11	10	9%
2-hop aggregation	9	7	22%
Spatial neighbor	12	7	42%
3-hop spatial	15	9	40%
Average	9.75	7.5	~20%

The largest reductions occur in neighbor queries (40 – 42%), where SQL must perform spatial `ST_Touches` joins while Cypher simply follows pre-computed `NEIGHBORS` edges.

---

7. Issues Encountered

7.1 PostGIS Geometry Serialization

Problem: Neo4j does not natively support WKB/WKT geometry objects. The migration initially failed when attempting to store PostGIS geometry blobs as node properties.

Resolution: The `SpatialHandler` class was added to extract scalar properties from geometry columns — specifically `center_lat` , `center_lon` , and `area_km2` — using PostGIS functions (`ST_Y(ST_Centroid(geom))` , etc.). Full geometry polygons are not stored in Neo4j; they are retained in PostgreSQL for GIS operations.

7.2 Census Tract Matching

Problem: Housing projects reference census tracts as numeric strings (e.g., `"10100"`), while the `rent_burden` table uses Census GEOID format (e.g., `"36005010100"` = state FIPS + county FIPS + tract). Direct FK matching failed.

Resolution: The `IN_CENSUS_TRACT` relationship uses a computed join key: `borough_to_fips[borough] + zero_padded(census_tract) = geo_id`. This borough-to-county FIPS mapping was hardcoded for the five NYC boroughs.

7.3 Streamlit Session State Bugs

Three UI bugs were discovered and fixed during development:

1. **Search button no reaction:** The text area `value=` parameter reset the content on every rerun (when `_pending` was already popped), causing `question.strip() == ""`. Fixed by giving the text area `key="ask_question"` so Streamlit persists its value in session state.
2. **Load → no effect in Explore:** Writing to `_cypher` session key while the text area used `key="cypher_editor"` — Streamlit's widget state overrides `value=`. Fixed by writing directly to `st.session_state["cypher_editor"]`.
3. **Cypher display broken:** LLM-generated Cypher injected directly into HTML via `st.markdown()` — `<` and `>` characters were interpreted as HTML tags. Fixed with `st.code(language="cypher")`.

7.4 Sidebar Contrast

Problem: Streamlit hardcodes `rgb(49,51,63)` on the inner `<span>` of each sidebar navigation link via inline styles, overriding `a` element CSS rules.

Resolution: CSS targeting `[data-testid="stSidebarNavLink"] span` with `!important` overrides the inline style successfully. Active page highlighted in `#F4A261` (orange) to match the project theme.

---

8. Lessons Learned

8.1 Precomputation is a First-Class Design Decision

The single instance where Neo4j outperformed PostgreSQL (Q4, 1.6×) directly resulted from migrating a computed relationship (`IN_CENSUS_TRACT`) rather than a raw foreign key. This validates a key graph database design principle: **model the queries you want to answer, not just the data you have**. Future migrations should systematically identify multi-table joins in production SQL and pre-compute them as direct graph edges.

8.2 Scale Determines the Performance Story

At 8,604 rows, PostgreSQL's local execution advantage dominates. The honest finding is that Neo4j provides its performance advantage at scale (millions of nodes, deep traversal queries). For an honest evaluation, the project framing should emphasize code simplicity and scalability potential rather than claiming immediate runtime advantages for small datasets.

8.3 LLM-Augmented Schema Interpretation Has Real Value

The LLM schema interpreter added genuine value beyond what static rules could provide: it suggested the semantically meaningful name `LOCATED_IN_ZIP` rather than the literal `FK_postcode`, and identified that `zip_tract_crosswalk` should become a computed edge rather than an intermediate node. However, LLM suggestions required human review — the interpreter occasionally proposed nonsensical relationships. The hybrid approach (LLM suggestions + authoritative YAML) proved more robust than either alone.

8.4 Streamlit State Management Requires Careful Design

Streamlit's execution model (full script re-run on every interaction) creates subtle bugs when `value=` and `key=` parameters interact. The key lesson: always use `key=` for persistent widget state and write to `st.session_state[key]` directly to programmatically update widget content. The `value=` parameter only sets the initial value on first render.

8.5 Documentation as a Product Deliverable

Early documentation drafts focused on code comments and API references. User testing (simulated by running the app with a fresh terminal and no memory of the implementation) revealed that the most critical documentation was the user guide — specifically the Cypher tips table explaining `rent_burden_rate > 0.35` vs `> 35` and the direction of `NEIGHBORS` edges. Documentation quality directly determines whether the KG is adopted or ignored.

---

9. Conclusion and Future Work

9.1 Conclusion

This project successfully designed and implemented an automated PostgreSQL-to-Neo4j conversion bot, validated through complete migration of the NOAH housing affordability database. All primary success criteria were met or exceeded:

- **Zero data loss:** 8,604 housing projects, 177 ZIP codes, and all relationships migrated correctly with full audit validation.
- **Text2Cypher accuracy:** 95% (19/20) on the 20-question benchmark, substantially exceeding the 75% specification target.
- **Performance analysis:** Documented across 8 queries with honest discussion of when PostgreSQL vs. Neo4j is faster and why.
- **Code simplicity:** 20% average line reduction in Cypher vs. SQL.
- **Comprehensive deliverables:** Working prototype, Streamlit dashboard, educational notebook, architecture documentation, user guide, and this report.

The project demonstrates that automated RDBMS-to-graph conversion is technically feasible using a combination of formal conversion frameworks (De Virgilio), LLM-assisted semantic enrichment, and config-driven ETL pipelines. The Text2Cypher interface — achieving near-human accuracy on representative queries — validates the promise of natural language interfaces for democratizing graph database access.

9.2 Future Work

Phase 2: Owner/LLC Network Data The ACRIS dataset (NYC property records: 85M rows across 3 tables, ~50GB) would add an owner-to-building relationship layer to the graph, enabling the multi-hop ownership chain queries described in the project briefing. This requires bulk import tooling (`neo4j-admin import`) rather than Python-mediated batch MERGE.

Parameterized Question Library The briefing identified a "Question Library" as the highest-value UI feature for non-technical users: 5 – 10 templates with dropdowns for geography, indicator, and relationship depth. This would provide repeatable, validated queries that Urban Lab analysts can run without LLM involvement.

Variable-Depth Graph Traversal Cypher's `[:RELATIONSHIP*1..3]` syntax enables variable-depth traversal (1 to N hops) with a single query. Exposing this through the UI as a "relationship depth" slider would unlock a class of network discovery queries not currently supported.

Production Hardening The current deployment is designed for local/classroom use. A production deployment would require: authentication/authorization, rate limiting on the Text2Cypher endpoint, caching common queries, and integration with Neo4j Aura (managed cloud Neo4j).

Broader Database Support The mapping engine's YAML-driven configuration is database-agnostic. Extending the schema analyzer to support MySQL and SQL Server would generalize the tool beyond PostgreSQL, addressing the Digital Forge Lab's stated goal of a broadly applicable migration bot.

---

10. References

Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J., & Vrgoc, D. (2017). Foundations of modern query languages for graph databases. ACM Computing Surveys, 50(5), 1 – 40.

De Virgilio, R., Maccioni, A., & Torlone, R. (2013). Converting relational to graph databases. In Proceedings of the First International Workshop on Graph Data Management Experiences and Systems (GRADES) (pp. 1 – 6). ACM.

Minder, P., Kindler, L., & Laparra, E. (2024). Data2Neo: A Tool for Complex Neo4j Data Integration. arXiv:2406.04995.

Neo4j, Inc. (2023). The Definitive Guide to Graph Databases for the RDBMS Developer. Neo4j Technical Documentation.

Ozsoy, O., Aktas, S., Ulker, Y., & Temizel, A. (2024). Text2Cypher: Bridging Natural Language and Graph Databases. arXiv:2412.10064.

Robinson, I., Webber, J., & Eifrem, E. (2015). Graph Databases: New Opportunities for Connected Data (2nd ed.). O'Reilly Media.

Zhao, F., Xu, W., & Bagherzadeh, N. (2023). Rel2Graph: Automated Mapping From Relational Databases to a Unified Property Knowledge Graph. arXiv:2310.01080.

Zhang, C. (2025). NOAH Information Dashboard: A Proof-of-Concept Housing Affordability Analytics Tool for Urban Labs [Capstone Report]. NYU School of Professional Studies.

Yu, Y. (2025). NOAH PostgreSQL/PostGIS Implementation [Capstone Project]. NYU School of Professional Studies. https://github.com/Becky0713/NOAH

NYC Open Data. (2024). Affordable Housing Production by Building [Dataset]. NYC Department of Housing Preservation and Development. https://data.cityofnewyork.us/Housing-Development/Affordable-Housing-Production-by-Building/hg8x-zxpr

---

Word count: approximately 3,200 words (body sections 1 – 9, excluding tables) Document version: 1.0 · Submitted Spring 2026