Congestion and Bottleneck Identification (CBI)

A Scalable Framework for Detecting and Characterizing Recurring Highway Bottlenecks Using Probe Vehicle Data

Database Engineering • Single-Corridor Pipeline • Multi-Corridor Automation • Dashboard

---

Author

Soheil Sameti

Development Period

2026

Prototype Corridor: I-75 Northbound — Metro Atlanta
Repository: https://github.com/atlregional/CBI

Table of Contents

This table of contents updates automatically to match the headings in this document. In Microsoft Word, right-click anywhere inside it and choose "Update Field" (or press F9) if it does not show current page numbers when the file is first o

pened.

Executive Summary

Congestion on a freeway rarely looks the way it feels to the people stuck in it. A single bad commute can be caused by a crash, a rain storm, or a stalled vehicle — a one-off. But a location that backs up nearly every weekday, at roughly the same time, for roughly the same duration, is a different problem entirely: it is a recurring bottleneck, a symptom of a real and fixable mismatch between roadway capacity and demand. The two look identical from inside a single car on a single day. They look completely different across a full year of data — and only the second kind can be planned for, funded, and engineered away.

This project builds the data infrastructure and analytical pipeline needed to tell the two apart, automatically, at the scale of an entire metropolitan freeway network. It ingests hundreds of millions of 5-minute probe vehicle speed observations, filters out one-off incidents from persistent recurring patterns, locates recurring bottlenecks to the segment, ranks them by severity, and characterizes each one with the same kind of onset-time, queue-length, and growth-rate measures a traffic engineer would otherwise have to derive by hand from raw speed data, corridor by corridor, year by year.

The prototype — I-75 Northbound in the Atlanta metropolitan region — is complete end to end: database engineering, event detection, annual batch processing, recurring bottleneck identification, and severity characterization all work against a full year of real data (Parts I–II). The remaining parts of this report (Parts III–V) lay out how that same pipeline generalizes to every major corridor in the region, how the results get published as an interactive public dashboard, and how a Microsoft Fabric and Power BI-based, Copilot-assisted version of that dashboard could eventually let a non-technical user simply ask a question about congestion and get a sourced answer.

Why This Matters

Every group with a stake in how a metropolitan freeway network performs currently either does this analysis by hand, on a handful of corridors, once every few years — or doesn't do it at all and relies on institutional memory of "where it always backs up." A framework that produces this analysis automatically, consistently, and at network scale changes what is practical for each of the following groups.

For transportation planners

Recurring bottleneck rankings give planners an objective, data-driven basis for prioritizing corridor studies and capital projects, rather than relying on anecdote or the loudest public comment. Because the same severity and mile-hours metrics are computed the same way at every location, bottlenecks across different corridors become directly comparable — a planner can defend why one interchange gets funded ahead of another using the same annual mile-hours figures shown in Part II, Section 11, instead of separate, incompatible one-off studies. The same event and bottleneck history also supports the Congestion Management Process (CMP) documentation that federal planning regulations require of MPOs like ARC.

For traffic engineers

The characterization metrics in Part II, Section 10 — onset time, peak time, clearance time, queue growth rate, and queue dissipation rate — are the exact inputs an engineer needs to evaluate whether a bottleneck

is a signal-timing problem, a ramp-metering candidate, or a geometric bottleneck requiring an auxiliary lane or interchange redesign. Knowing that a queue at a given location consistently grows at a known rate starting at a known time, and dissipates at a known rate, turns “it's bad in the afternoon” into a specific, testable operational hypothesis that can be validated before and after a countermeasure is deployed.

For policy makers and local decision makers

Elected officials and agency leadership are routinely asked to justify transportation spending with limited budgets and competing local priorities. A transparent, reproducible ranking of where congestion is worst and how severe it is — backed by a full year of observed data rather than a single study period — gives decision makers a defensible, non-partisan basis for those conversations, and a way to show constituents that a specific investment targets a documented, quantified problem rather than a subjective judgment call. Because the underlying methodology is transparent and consistent, results are also easier to defend under public or legislative scrutiny than a one-off consultant study with unpublished assumptions.

For consulting firms

Traffic and planning consultants currently spend a meaningful share of every corridor study, traffic impact analysis, and environmental document budget on data acquisition and preliminary congestion diagnostics — work that this framework automates. A firm engaged for a corridor study could start from an existing recurring bottleneck inventory and a full year of characterized event data instead of re-deriving it from raw probe data, shifting billable effort toward interpretation, alternatives analysis, and design rather than data preparation. The planned multi-corridor rollout (Part III) and public dashboard (Part IV) would make this starting point available for any corridor in the region, not just the ones a given engagement happens to cover.

For the traveling public

None of the above matters to a traveler stuck in traffic — but the eventual outcomes do. Recurring bottlenecks identified and prioritized this way are bottlenecks that are more likely to actually get fixed, whether through a capital project, a signal retiming, or a ramp-metering change, because the case for fixing them is quantified rather than anecdotal. A public-facing dashboard (Part IV) also gives travelers direct visibility into which parts of their commute are chronic problems worth planning around versus which delays were one-off incidents — turning institutional data most people never see into something they can use to plan a trip.

The parts that follow document, in order: the PostgreSQL database layer that makes ingesting and querying hundreds of millions of observations practical (Part I); the Python analytical pipeline that turns those observations into the bottleneck rankings and characterization metrics described above, validated against real output data for the I-75 Northbound prototype (Part II); the plan for generalizing that pipeline to every major corridor in the region (Part III); the interactive public dashboard that would put these results in front of the stakeholders above (Part IV); and a proposed Microsoft Fabric and Power BI-based, Copilot-assisted extension of that dashboard (Part V). Parts I–II describe completed, verified work; Parts III–V are the roadmap for turning a validated single-corridor prototype into a regional decision-support tool.

Study Corridor

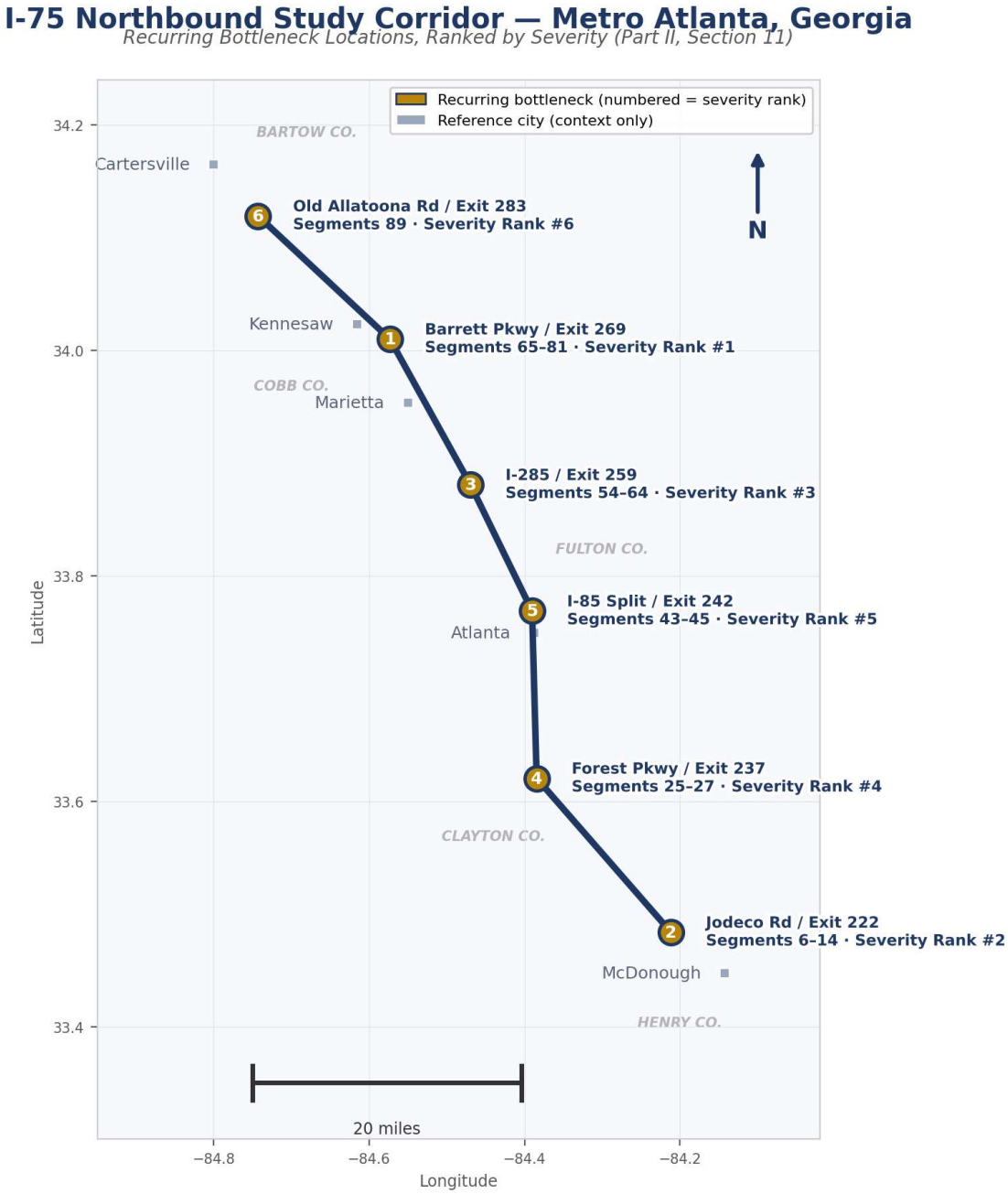
This report's prototype covers a single corridor: I-75 Northbound only (the southbound direction is not included in current results). The corridor is divided into 101 ordered segments (TMCs) spanning 92.75 miles, indexed south to north as segment_order 1 through 101.

Corridor Endpoints — Confirm Before Publication

• The exact intersection names for segment_order 1 (the corridor's southern start) and segment_order 101 (its northern end) were not directly available in the scripts, SQL, and output files reviewed while assembling this report, and should not be treated as confirmed until checked against the database directly (SELECT intersection FROM "Year_2025".corridor_i75_nb WHERE segment_order IN (1, 101)).

• What is confirmed: the corridor's six identified recurring bottlenecks (map below) span from Jodeco Road / Exit 222 in the south to Old Allatoona Road / Exit 283 in the north — so the full corridor extends at least that far, and likely further in one or both directions since a bottleneck's location doesn't have to sit at the corridor's literal edge.

The map below plots the six identified recurring bottlenecks (Part II, Section 9) at real GPS coordinates, giving an actual geographic view of where each is located along the corridor through metro Atlanta.



Coordinates are approximate landmark positions near each named interchange (sourced via Google Places), not the exact TMC segment coordinates from the project database. Corridor line is a straight-line approximation between points, not the literal roadway geometry. For precise analysis, replace with corridor_segments.start_latitude/longitude from the database.

Map Data Provenance

• Bottleneck marker coordinates are approximate landmark positions near each named interchange, sourced via a Google Places lookup for this report — they are not the literal TMC segment coordinates from corridor_segments.start_latitude/longitude in the project database, which were not available in the materials reviewed. The corridor line is a straight-line approximation between points, not the actual roadway geometry. Both should be replaced with real database coordinates for any use beyond general orientation.

Full corridor: 101 segments, 92.75 miles, Metro Atlanta, Georgia. See Part II, Section 9 for how these six locations were identified and Section 11 for the severity ranking referenced above.

How This Report Is Organized

This report moves from technical infrastructure toward the tools a planner would actually use day to day. Readers who don't need every implementation detail can use the table below to jump to what matters to them — each part opens with an "In Plain Terms" box before the technical detail begins, so the purpose of each part is clear even without reading the code or SQL that follows it. Appendix A defines every technical term and column name used anywhere in the report.

Part	What It Covers	Best For
I — Database Engineering	How 427.7 million annual raw speed readings get organized so they can be queried in minutes instead of hours	Readers curious how the data foundation works
II — Python Pipeline	How raw speed data becomes a ranked list of recurring bottlenecks with engineering metrics, for I-75 Northbound today	Planners and engineers who want to understand what a "recurring bottleneck" and "severity score" actually mean and how they're calculated
III — Multi-Corridor Automation	The plan to run the same analysis on every major corridor automatically instead of one at a time by hand	Anyone asking "when will this cover more than I-75?"
IV — Public Dashboard	The plan for a simple, browsable public website showing results	Readers who want a low-cost, always-on public view
V — AI Dashboard on Microsoft Fabric	The plan for an interactive, ask-a-question dashboard with maps, charts, and an AI assistant	Decision makers and planners who want to explore the data themselves without writing a query
Appendix A — Glossary	Plain-language definitions of every technical term and column name used in this report	Anyone who hits an unfamiliar term anywhere above

Pipeline Overview — The Whole Process at a Glance

Before the section-by-section detail, here is the entire process end to end — from a raw speed reading to a ranked, characterized bottleneck a planner can act on. Every stage below is documented in full later in this report; the section reference under each stage says exactly where.

DATA FOUNDATION — PART I

1. Raw Probe Data
HERE probe vehicle speed readings, accessed via NPMRDS Analytics — 427.7 million observations across 17,380–17,388 roadway segments, full year 2025.
Part I, Section 1

↓

2. PostgreSQL Ingestion & Indexing
Raw readings are staged, type-cast, and indexed so a corridor-day of data can be queried in seconds instead of minutes.
Part I, Sections 2–4

↓

SINGLE-CORRIDOR ANALYSIS — PART II

3. Phase 1: Corridor Preparation & Heatmap
One day's data is validated and turned into a human-checkable picture of congestion, before anything is automated.
Part II, Section 6

↓

4. Phase 2: Congestion Event Detection
An algorithm finds each individual traffic jam automatically — its start time, end time, and exact location.
Part II, Section 7

↓

5. Phase 3: Annual Batch Processing
Phase 2's detection logic runs across all 261 weekdays with available data, not just one.
Part II, Section 8

↓

6. Phase 4: Recurring Bottleneck Detection
The full year of individual jams is searched for locations where congestion keeps recurring in the same place — 11 such locations found on I-75 NB alone; 179 region-wide across 41 corridors.
Part II, Section 9

↓

7. Phase 5: Bottleneck Characterization
Each recurring bottleneck gets engineering-ready metrics: onset time, peak queue length, growth and dissipation rate, for every day it occurred.
Part II, Section 10

↓

8. Dashboard Views & Severity Ranking
Results are rolled up into ranked, dashboard-ready database views — the same views both the exported reports and the interactive dashboard read from.
Part II, Sections 10.1–11

↓

SCALE & DELIVER — PARTS III–V

9. Multi-Corridor Automation
Stages 3–8 now run across 41 corridors region-wide (16 Interstate, 25 Arterial), not just I-75 Northbound — confirmed complete, not just planned.
Part III

↓

10. Dashboards
Results reach planners, engineers, and the public through a simple public website (Part IV) and, eventually, an interactive, ask-a-question dashboard built on Microsoft Fabric and Power BI (Part V).
Parts IV–V

Data Dictionary — The Raw Data Behind This Report

Every number, chart, and ranking anywhere in this report is ultimately derived from two raw tables: one row per 5-minute speed reading, and one row per roadway segment those readings are attached to. Both are sourced from HERE Technologies' probe vehicle fleet via the NPMRDS Analytics platform (see Part I, Section 1). Understanding these two tables first makes every later section — which is really just different ways of summarizing and filtering this same raw data — easier to follow. (Appendix A separately documents the calculated columns each pipeline phase produces, such as speed_ratio or severity_index; this section covers only the two raw source tables those calculations start from.)

probe_readings — Raw Speed Observations

The core dataset: 427.7 million annual rows, one per roadway segment per 5-minute interval. This is the table every phase in this report ultimately reads from.

Column	Plain-Language Meaning
tmc_code	Identifies which roadway segment this reading belongs to — matches the tmc column in the segment reference table below.
measurement_tstamp	The date and time this reading was recorded, in 5-minute increments (e.g., 8:00 AM, 8:05 AM, 8:10 AM...).
speed	The actual average vehicle speed observed on this segment at this moment, in miles per hour.
historical_average_speed	The typical speed for this segment at this time of day, based on long-run historical patterns — background context, not the number congestion is measured against.
reference_speed	The "normal," free-flow speed this segment is compared against to decide whether it's congested. This is the number used throughout this report to calculate speed_ratio (Appendix A.1) and detect congestion.
travel_time_minutes	How many minutes it would take to travel this segment at the observed speed.
data_density	A data-quality indicator from the source provider, showing how much real GPS probe traffic supported this particular reading versus it being statistically estimated.

tmc_metadata — Roadway Segment Reference Data

One row per roadway segment (TMC), describing where it is, how long it is, and how it connects to its neighbors. This is the table that turns an anonymous segment ID into a named, mappable location.

Column	Plain-Language Meaning
tmc	Unique ID for this segment — the same ID referenced as tmc_code in probe_readings.
road	Name of the roadway, e.g. "I-75."
direction	Direction of travel this segment represents, e.g. "NORTHBOUND."
intersection	Nearest named cross street or interchange — the human-readable location label used throughout this report (e.g. "Barrett Pkwy / Exit 269").
state / county / zip	Administrative location of the segment.
start_latitude / start_longitude / end_latitude / end_longitude	Geographic coordinates of the segment's two endpoints — what makes the interactive maps in Parts IV–V possible.
miles	Physical length of this one segment.
road_order	A number establishing the correct order segments connect in along the roadway — the basis for the segment_order sequence (1–101 for I-75 NB) used throughout this report.
timezone_name	Local time zone this segment's timestamps are recorded in.
road_type	Functional classification of the roadway, e.g. freeway vs. arterial.
country	Country the segment is located in.
tmclinear	An alternate linear-referencing identifier used by some NPMRDS tools.
frc	Functional Road Class — a standard roadway classification code.
f_system	Federal-aid highway functional system classification.
urban_code	Census-designated urban area code the segment falls within.
thrulanes	Number of through-traffic lanes on this segment.
aadt	Annual Average Daily Traffic — the estimated number of vehicles using this segment on a typical day.
active_start_date / active_end_date	The date range this exact segment definition is valid for — TMC boundaries are occasionally redrawn by the data provider, and this tracks which definition was in effect at a given time.

Part I — Database Engineering (PostgreSQL)

In Plain Terms

Before any bottleneck can be found, 427.7 million individual annual speed readings need to live somewhere a computer can search through quickly. Part I is about that storage and organization step — not analysis yet, just making sure a question like "show me every speed reading on I-75 Northbound in March" comes back in seconds instead of the many minutes it would take to scan the raw data file by file.

Think of it like the difference between a library with a card catalog and one without: the books (data) are the same either way, but finding what you need is only practical with an organized system. This part documents that system — the database, the shortcuts ("indexes") that speed up common lookups, and a few pre-built summary tables that save the analysis in Part II from having to redo the same calculation from scratch every time.

1. Data Source and Environment

Speed data is sourced from HERE Technologies' probe vehicle fleet, accessed through the NPMRDS Analytics platform — the standard national data source for freeway performance monitoring, and the platform GDOT and ARC license for this purpose. (Note: this project's data is HERE probe data, not INRIX — GDOT/ARC does not purchase INRIX probe data.)

Item	Value
Data Source	HERE probe vehicle data via NPMRDS Analytics
Study Year	2025
Coverage	Metro Atlanta (21 counties)
Database Engine	PostgreSQL 17.2
Probe Data Observations	427,676,537 (annual total)
TMC metadata records	17,380
Network TMCs	17,388
Time interval	5 minutes
Available weekdays	261
Time span	Jan 1 – Dec 31, 2025
Database size	approximately 23 GB

Open Item — NPMRDS Analytics vs. RITIS PDA Comparison Not Yet Performed

- This report's 427.7 million observations and 17,380–17,388 TMC figures come exclusively from NPMRDS Analytics. A request has been made to also pull the same corridor and year from RITIS's Probe Data Analytics Suite (PDA) and compare record counts and TMC counts between the two platforms, to check whether NPMRDS Analytics under-represents either figure relative to PDA.
- That comparison has not been run. Both platforms require an authenticated RITIS account tied to this project's organization, which is outside what this report's preparation had access to — it needs to be run by someone with existing RITIS credentials, either directly or by exporting matching data from both platforms for comparison here.
- Once available, the comparison belongs in this section: same corridor (I-75 NB), same study year (2025), record count and distinct-TMC count from each platform, side by side.

Computing Environment

- Database: PostgreSQL 17.2
- Programming: Python 3.12
- Libraries: Polars, NumPy, SciPy, ConnectorX, Psycopg, scikit-learn
- Development Environment: Visual Studio Code
- Hardware: NVMe SSD

2. Bulk Ingestion Pipeline

Raw NPMRDS export files are loaded through a three-stage pattern designed for the throughput needed to ingest 427.7 million rows: an unlogged staging table with permissive text typing, a session tuned for bulk operations, and a typed INSERT...SELECT that performs the real casting.

2.1 Staging Table

Data is first copied into an UNLOGGED staging table with every column typed as text. Skipping the write-ahead log during load and deferring type validation avoids row-by-row cast failures during the raw import and lets PostgreSQL ingest the flat file as fast as disk I/O allows:

```
CREATE UNLOGGED TABLE probe_readings_stage (
    tmc_code text,
    measurement_tstamp text,
    speed text,
    historical_average_speed text,
    reference_speed text,
    travel_time_minutes text,
    data_density text,
    npmrds2_2025 text
);
```

Implementation Note

• Two near-identical versions of this script exist in the repository (Create_Unlogged_Table_probe_readings.sql and PSQL_Importing_Probe_Readings.sql), differing only in the name of the trailing NPMRDS metadata column (npmrds2_2025 vs. npmrd_s2_2025). This looks like an in-progress naming correction rather than two distinct designs — worth consolidating to a single canonical script before the multi-corridor rollout.

2.2 Typed Import

Once staged, rows are inserted into the production table with explicit casts. Blank strings are converted to NULL with NULLIF() before casting to real or timestamp, which prevents empty-string-to-numeric cast errors that are common with raw NPMRDS exports:

```
INSERT INTO probe_readings (
    tmc_code, measurement_tstamp, speed,
    historical_average_speed, reference_speed,
    travel_time_minutes, data_density
```

```
)
SELECT
    tmc_code,
    measurement_tstamp::timestamp,
    NULLIF(speed, '')::real,
    NULLIF(historical_average_speed, '')::real,
    NULLIF(reference_speed, '')::real,
    NULLIF(travel_time_minutes, '')::real,
    NULLIF(data_density, '')
FROM probe_readings_stage;
```

2.3 Session Tuning for Bulk Operations

Bulk load and index-build steps run under a relaxed durability and larger-memory session profile, reverted for normal query workloads:

```
SET synchronous_commit = off;
SET maintenance_work_mem = '2047MB';
SET work_mem = '512MB';
SET temp_buffers = '512MB';
SET max_parallel_workers_per_gather = 8;
```

Disabling synchronous_commit trades crash-safety of the very last transaction for substantially faster bulk writes — acceptable for a reproducible batch import, not for live transactional workloads. The enlarged maintenance_work_mem is what keeps the 427M-row index build (Section 4) to under ten minutes instead of hours.

3. Core Schema

3.1 tmc_metadata

One row per TMC (Traffic Message Channel) segment, carrying the full NPMRDS attribute set — geometry endpoints, road classification, functional system, thru-lane count, AADT, and the active date range for the segment definition:

```
CREATE TABLE tmc_metadata (
    tmc text PRIMARY KEY,
    road text, direction text, intersection text,
    state text, county text, zip text,
    start_latitude double precision, start_longitude double precision,
    end_latitude double precision, end_longitude double precision,
    miles double precision, road_order double precision,
    timezone_name text, road_type text, country text,
    tmclinear text, frc integer, f_system integer,
    urban_code integer, thrulanes integer, aadt integer,
    active_start_date timestamptz, active_end_date timestamptz
);
```

3.2 probe_readings — Two Design Paths

Two different definitions of the core observations table exist in the repository. The flat version is the one actually populated and queried by every downstream script and view:

```
-- Adopted design
CREATE TABLE probe_readings (
    tmc_code text NOT NULL,
    measurement_tstamp timestamp NOT NULL,
    speed real, historical_average_speed real,
    reference_speed real, travel_time_minutes real,
    data_density text
);
```

A second script defines the identical column set as a range-partitioned table on measurement_tstamp:

```
-- Evaluated, not currently adopted
CREATE TABLE probe_readings (
    tmc_code text NOT NULL,
    measurement_tstamp timestamp NOT NULL,
    speed real, historical_average_speed real,
    reference_speed real, travel_time_minutes real,
    data_density text
) PARTITION BY RANGE (measurement_tstamp);
```

Implementation Note

• Only the flat table is in active use — the index scripts, materialized views, and every Python query in the repository reference plain probe_readings with no partition-aware logic anywhere. The partitioned definition looks like a design that was evaluated but not carried forward for the single-corridor prototype.

• This is worth revisiting specifically for Part III (multi-corridor automation): at statewide scale the table will hold many times 427M rows, and monthly or quarterly range partitioning on measurement_tstamp would let each corridor-extraction and annual-processing query prune irrelevant partitions instead of scanning the full table — the original reason to consider partitioning in the first place.

4. Indexing Strategy

Three indexes support the query patterns used throughout the pipeline:

Index	Definition	Supports
idx_probe_tmc_time	probe_readings (tmc_code, measurement_tstamp)	Corridor-day extraction (WHERE tmc_code IN (...) AND time range)
idx_probe_time	probe_readings (measurement_tstamp)	Network-wide, time-only window queries
idx_tmc_direction	tmc_metadata (direction)	Corridor extraction filtered by direction (e.g., NORTHBOUND)

Measured Performance

Operation	Runtime
Row count (427M records)	4 min 45 sec
idx_probe_tmc_time creation	9 min 42 sec
ANALYZE	1.8 sec

5. Materialized Views — Network-Wide Congestion Layer

Separately from the corridor-specific Python pipeline, two materialized views precompute congestion measures across the entire 17,380-TMC network, not just I-75 Northbound.

5.1 cbi_observations

Joins every probe reading to its TMC metadata and computes the same speed-ratio and 70%-of-reference-speed congestion cutoff used throughout the Python pipeline, but at full network scale:

```
CREATE MATERIALIZED VIEW cbi_observations AS
SELECT
    r.tmc_code, r.measurement_tstamp, r.speed,
    r.historical_average_speed, r.reference_speed,
    r.travel_time_minutes,
    m.road, m.direction, m.miles, m.road_order,
    CASE WHEN r.reference_speed IS NULL OR r.reference_speed = 0
         THEN NULL ELSE r.speed / r.reference_speed END AS speed_ratio,
    CASE WHEN r.speed < 0.7 * r.reference_speed
         THEN true ELSE false END AS is_congested,
    GREATEST(r.reference_speed - r.speed, 0) AS speed_drop
FROM probe_readings r
JOIN tmc_metadata m ON r.tmc_code = m.tmc;
```

5.2 cbi_daily_tmc

Aggregates cbi_observations to one row per TMC per day — observation counts, congested interval/minute totals, and average/maximum speed drop:

```
CREATE MATERIALIZED VIEW cbi_daily_tmc AS
SELECT
    tmc_code, measurement_tstamp::date AS analysis_date,
    COUNT(*) AS observation_count,
    COUNT(*) FILTER (WHERE is_congested) AS congested_intervals,
    COUNT(*) FILTER (WHERE is_congested) * 5 AS congested_minutes,
    AVG(speed) AS average_speed,
    AVG(speed_drop) AS average_speed_drop,
    MAX(speed_drop) AS maximum_speed_drop
FROM cbi_observations
GROUP BY tmc_code, measurement_tstamp::date;
```

Implementation Note — Relevant to Multi-Corridor Automation

• None of the Python phase scripts currently read from cbi_observations or cbi_daily_tmc. The Phase 1/2/3 scripts recompute speed_ratio and the 0.70 congestion cutoff independently in Python, against probe_readings joined to the I-75-specific corridor_i75_nb table.

• Because these views already run the same congestion logic across the full network rather than one corridor, they are a natural foundation for Part III: a generic corridor engine could filter cbi_daily_tmc by corridor/direction instead of re-deriving congestion per corridor in Python, removing duplicated logic as more corridors are added.

Part II — Python Pipeline for a Single Corridor (I-75 Northbound)

In Plain Terms

This is the heart of the project: turning the organized speed data from Part I into an actual list of "these are the six worst recurring trouble spots on I-75 Northbound, ranked, with details on how bad each one is." It happens in five steps, each building on the last:

1. Look at one day of data as a picture (a heatmap) to see congestion by eye before automating anything.
2. Teach the computer to automatically spot a traffic jam in that picture — a "congestion event."
3. Repeat that automatic spotting for all 261 weekdays in the year, not just one day.
4. Find the locations where jams keep happening in roughly the same place, over and over — a "recurring bottleneck," as opposed to a one-time crash or a rainy-day slowdown.
5. For each recurring bottleneck, calculate the numbers an engineer needs: how often it happens, how bad it gets, how fast it forms and clears.

This part documents the working prototype pipeline exactly as implemented across the twelve Python scripts in the Scripts/ folder, organized by the same five phases used during development.

6. Phase 1 — Corridor Preparation and Heatmapping

Why This Phase Exists

Every later phase in this pipeline assumes the raw speed data is complete, correctly shaped, and free of errors — an assumption that has to be earned, not taken for granted, before a full year of automated analysis is built on top of it. Phase 1 exists to earn that trust on a single representative day before anything is scaled up to 261 days. It has two goals: catch data problems (missing readings, duplicate records, misaligned segments) while they're still cheap to notice, and produce a picture of congestion a person can look at and sanity-check by eye, before handing the same job to an algorithm in Phase 2 with no human watching.

In other words: Phase 1 is not yet about finding bottlenecks. It's about answering "can this data be trusted, and what does congestion on this corridor actually look like?" before automating anything.

Key Terminology

Term	Meaning in This Pipeline
Segment	One TMC — a short, named stretch of the corridor (I-75 Northbound has 101 of them). Segments are numbered south to north as segment_order 1 through 101.
Space-time grid	A table with one row for every combination of segment and 5-minute period in a day — 101 segments × 288 five-minute periods = 28,888 possible cells. This is the shape every later calculation expects its data in.
Coverage / data availability	The percentage of those possible cells that actually have a real speed reading. Below-100% coverage is expected and normal — probe vehicle data is never perfectly complete — but a coverage number far outside the usual range is a sign something upstream (the data import, a corridor definition) may be wrong.
Heatmap	A color-coded picture of the space-time grid — segment position across the bottom, time of day up the side, color showing how congested each cell was. This is what lets a person visually confirm "yes, that red patch in the afternoon is a real, sensible-looking traffic jam" before trusting an algorithm to find the same pattern automatically.

What Phase 1 Confirms

Before any congestion pattern is trusted, Phase 1 checks for three specific failure modes that would otherwise silently corrupt every later phase:

- No duplicate (timestamp, segment) pairs — a duplicate would double-count a moment of congestion.
- Row count is consistent with the expected maximum (101 segments × 288 five-minute periods) — a large shortfall could mean part of the corridor or part of the day is silently missing.
- Null counts for speed and reference_speed are within a sane range — since every later congestion calculation divides one by the other (Appendix A.1).

What Phase 1 Produces

Two things carry forward into every later phase: a clean, complete space-time grid (missing cells are left blank rather than guessed at, so later phases know the difference between "no data" and "no congestion"), and two heatmaps — one showing raw speed ratio, one showing a simple over/under congestion cutoff — that serve as the human-readable baseline every later, more automated phase is checked against.

Corridor validation confirmed 101 ordered segments spanning 92.75 miles, approximately 99% data availability, and continuous spatial ordering.

7. Phase 2 — Congestion Event Detection

Why This Phase Exists

A heatmap answers "what did congestion look like on this one day?" for a human looking at a picture. It does not, by itself, produce anything a computer can count, rank, or store — there's no record that says "this specific traffic jam started at 3:05 PM, lasted 90 minutes, and stretched across 12 segments." Phase 2's job is to turn the visual pattern from Phase 1 into exactly that kind of structured record, automatically, for any day, without a person looking at a picture to confirm it. That structured record is called a

congestion event, and it's the basic unit everything from here forward is built on: Phase 3 finds events across the whole year, Phase 4 finds locations where events keep recurring, and Phase 5 measures how severe those recurring events get.

Key Terminology

Term	Meaning in This Pipeline
Congestion event	One contiguous traffic jam — a connected block of congested cells in the space-time grid, with a defined start time, end time, and set of segments. This is the fundamental unit of analysis for the rest of the pipeline.
Cutoff ratio	The threshold (0.70, or 70% of normal speed) below which a cell counts as "congested" in the first place — defined in Appendix A.1, used consistently from here through every later phase.
Connected component	The technical name for the process of grouping neighboring congested cells — next to each other in time or in space — into a single event, the same way a person's eye would group a solid red patch on a heatmap into one blob rather than many separate dots.
False merge	The specific failure this phase had to be engineered around: two genuinely separate traffic jams, in different locations, that happen to touch for a moment (through one shared cell or a brief shared time window) getting incorrectly counted as a single event instead of two.

Why Three Versions Were Needed

Event detection was not built once and left alone — it went through three iterations, each solving a specific, concrete problem the previous version got wrong. Understanding why each step was necessary matters more here than the exact code that implements it.

Version 1 — establishing the basic approach

The first version (cbi_phase2_events.py) proved out the core idea: group touching congested cells into events, then discard anything too small or too brief to be a real traffic jam rather than noise. It worked, but had exactly the false-merge problem described above — two distinct jams that happened to touch got recorded as one, which would have meant Phase 4 later mistaking two separate bottleneck locations for a single, misleadingly large one.

Version 2 — fixing the false-merge problem

The second version (cbi_phase2b_refined.py) was built specifically to fix that problem: it identifies each jam's solid "core" first, then grows each core back out to its true extent independently, so two cores that are near each other but distinct don't get fused into one event just because they briefly touch. This is the version whose output is demonstrably more accurate — see the January 15 example below.

Version 3 — making it reusable for a full year

The third version (cbi_detector.py) changes nothing about the detection logic itself — it repackages Version 2's proven approach into a single function that takes the analysis date as an explicit input, rather than requiring the date to be edited into the script by hand before each run. That change is what makes it

possible for Phase 3 to call this same, already-validated logic automatically across all 261 days instead of someone re-running the script by hand for each one.

7.4 Sample Output Evidence — January 15, 2025

Comparing the Version 1 and Version 2 outputs for the same day confirms the false-merge fix works in practice, not just in theory. Version 1 merged two physically separate Cobb County jams into a single event; Version 2 correctly separated them:

Detector	Event	Segments	Extent	Location	Area (mi-hr)
Version 1	Event 6 (one merged event)	53–86 (34 seg.)	21.86 mi	Moores Mill Rd → Glade Rd	52.797
Version 2	Event 7	65–86 (22 seg.)	17.32 mi	Delk Rd → Glade Rd (Barrett Pkwy area)	40.524
Version 2	Event 8	50–64 (15 seg.)	6.89 mi	Northside Dr → Windy Hill Rd (I-285 area)	15.326

Both Version 2 events ran over the same 15:05–19:50 window as the single Version 1 event, but Version 2 recognized them as two independent congestion fronts rather than one contiguous jam — correctly resolving them into the two locations that later emerge as separate recurring bottlenecks in Phase 4 (Barrett Parkway and I-285, Section 9). This is exactly the kind of error the false-merge fix exists to prevent: without it, these two distinct, real-world bottleneck locations would have been recorded as one, throwing off every later ranking and severity calculation for both. Version 2 found 13 retained events for this day versus 10 for Version 1.

Cross-file consistency checks

- i75_nb_event_bottleneck_membership.csv contains exactly 4,364 data rows, matching the report's annual congestion event count exactly (Section 8).
- i75_nb_segment_bottleneck_membership.csv contains 44 rows, exactly equal to the sum of segment_count across the six bottlenecks in i75_nb_segment_bottlenecks.csv (17+9+11+3+3+1).

What Gets Measured Per Event

Once an event's boundaries are established, a fixed set of measurements is recorded for it — the same measurements for every event, so any two events (or the same event across two different days) can be directly compared. Column definitions for every metric below are in Appendix A.3.

- Start/end time, duration, cell and segment counts
- First/last segment order, TMC, and intersection
- Corridor start/end mile position
- Maximum contiguous extent (miles) and maximum total active extent (miles)
- Event area in mile-hours
- Average speed, minimum/average speed ratio, average/maximum speed drop
- Estimated upstream boundary propagation speed (linear fit of the congested front's leading segment over time)

8. Phase 3 — Annual Batch Processing

Why This Phase Exists

Phase 2 solved the hard problem — correctly finding congestion events on any single day — but a bottleneck can't be called "recurring" from one day of evidence. Recurring means the same thing keeps happening across many days, so Phase 3's only job is repetition: run Phase 2's already-proven detection logic on every one of the 261 weekdays with available data, not just one, and save every event found so Phase 4 has a full year of evidence to look for patterns in.

This phase does not introduce any new detection logic — it exists purely to scale Phase 2 up from "one day, checked by a person" to "261 days, fully automated."

Key Terminology

Term	Meaning in This Pipeline
Batch processing	Running the same operation — here, event detection — across many inputs (261 days) in one automated pass, instead of one at a time by hand.
Idempotent / safe to re-run	A run that can be repeated without creating duplicates or corrupting prior results. This pipeline tracks which dates are already completed and only processes what's left, so an interrupted run can simply be restarted rather than redone from scratch.

Two runner scripts exist for driving event detection across all 261 available weekdays.

cbi_phase3_annual.py

Drives cbi_phase2b_refined.py by setting detector.ANALYSIS_DATE before each call — functional, but relies on mutating global module state inside a loop.

cbi_annual_runner.py

Calls cbi_detector.detect_events(dataframe, analysis_date) directly, with the date passed as an argument. Tracks completed dates in the database so re-running the script only processes remaining days, and commits or rolls back per day so one bad day cannot corrupt others.

> **Implementation Note**
> • cbi_phase3_annual.py and cbi_annual_runner.py do the same job. The results reported below (261/261 successful, 0 failed, 4,364 events) match the cbi_annual_runner.py design, which is the more robust of the two — recommend retiring cbi_phase3_annual.py to avoid confusion about which is canonical.

Annual Processing Results

Metric	Value
Successful days	261
Failed days	0

Metric	Value
Annual congestion events	4,364
Runtime	approximately 1.6 minutes

9. Phase 4 — Recurring Bottleneck Detection

Why This Phase Exists

Phase 3 produces 4,364 individual congestion events — but an event is just one traffic jam on one day. Some of those 4,364 events are one-off (a crash, a storm, an unusual travel pattern); others are the same location showing up again and again, week after week. Phase 4's job is to tell the two apart: to look across the entire year of events and find the specific locations where congestion recurs often enough, and consistently enough, to be called a genuine recurring bottleneck rather than noise. This is the step that turns “4,364 traffic jams happened this year” into “these six specific locations are where this corridor's real, fixable capacity problems are.”

Key Terminology

Term	Meaning in This Pipeline
Recurring bottleneck	A specific stretch of the corridor where congestion events keep happening in roughly the same place, often enough across the year to represent a genuine, persistent capacity problem — as opposed to a one-time incident.
Occurrence percentage	The share of analyzed weekdays on which a given location was congested — the primary evidence used to decide whether a location's congestion is recurring or coincidental.
Peak segment	Within a recurring bottleneck's boundary, the single segment with the highest occurrence percentage — its “epicenter,” used as the anchor point the bottleneck's boundary is built outward from.
Smoothing	Averaging each segment's occurrence percentage together with its immediate neighbors before searching for peaks, so a single unusually noisy segment doesn't get mistaken for — or hide — a real bottleneck.
Boundary expansion	Growing a bottleneck's recorded extent outward from its peak segment in both directions, for as long as neighboring segments still show meaningfully high occurrence, so the bottleneck's recorded footprint matches how far the congestion actually reaches, not just its single worst point.

Two independent algorithms for identifying recurring bottleneck locations exist in the repository, built on different units of analysis — one groups by segment (adopted), the other by individual event (evaluated and retired, see evidence below).

9.1 cbi_segment_bottlenecks.py — segment-profile peak detection (adopted)

Reads a precomputed annual per-segment occurrence profile (the i75_nb_segment_profile table), smooths the weekday-occurrence percentage with a 3-segment moving average, finds peaks with

scipy.signal.find_peaks (minimum occurrence, prominence, and spacing thresholds), then expands each peak outward until occurrence drops below an absolute or relative-to-peak threshold. Adjacent, overlapping boundaries are split at their shared valley. Bottlenecks are ranked by total annual_mile_hours and written to segment_recurring_bottlenecks.

The i75_nb_segment_profile table itself is built entirely in SQL (CBI_working_SQL_master.sql, statement 017), from an intermediate i75_nb_segment_daily table (statement 016) that aggregates every 5-minute observation to one row per segment per day — observed/congested interval counts and average speed ratios. The annual profile then rolls that up to one row per segment: weekday_occurrence_pct (days with ≥30 congested minutes, as a percentage of analyzed days), average/median/p95 congested minutes, and annual_segment_mile_hours. This is the raw input cbi_segment_bottlenecks.py smooths and peaks-detects on — it is not missing from the reviewed codebase, just implemented in SQL rather than Python.

Implementation Note

• The report's phase description ("Gaussian smoothing") does not match the code: smooth_profile() in cbi_segment_bottlenecks.py applies a uniform box-car moving average (np.convolve with a flat kernel) on top of the SQL-built profile above, not a Gaussian kernel. Either the smoothing method should be corrected to Gaussian, or the report text should be updated to describe the box-car average actually implemented.

9.2 cbi_recurring_bottlenecks.py — event-clustering (retired — see evidence below)

A different approach that clusters individual congestion events (not segments) using DBSCAN over each event's spatial midpoint and interval length, after first scaling features with StandardScaler. Clusters are ranked by summed event_area_mile_hours and written to a separate table, recurring_bottlenecks, with membership recorded in event_bottleneck_membership.

Resolved — Segment-Profile Method Confirmed Canonical

• The actual output files settle this. i75_nb_recurring_bottlenecks.csv (DBSCAN) produces 46 rows: bottleneck #1 is a single mega-cluster spanning miles 12.6–64.3 — over half the corridor — that "occurred" on all 261 weekdays (event_count 4,272), and bottlenecks #2–46 are almost all singletons (event_count = 1, occurrence_days = 1), i.e. DBSCAN noise points each receiving their own cluster label rather than being filtered out.

• i75_nb_segment_bottlenecks.csv (segment-profile peak detection), by contrast, produces exactly six well-localized, physically sensible bottlenecks, each spanning 1–17 segments with clear occurrence percentages (28.9%–98.1%) — and these match the report's published list exactly.

• Recommendation: retire cbi_recurring_bottlenecks.py and the recurring_bottlenecks / event_bottleneck_membership tables it writes to. The segment-profile method (Section 9.1) is the canonical Phase 4 algorithm and should be the only one carried into Part III's multi-corridor generalization.

Eleven Recurring Bottlenecks Identified (I-75 Northbound)

Updated — Live Pipeline Now Reports 11, Not 6

• The regional dashboard (regional_congestion_report_2026-08-11.html, corridor-1 / I-75 Northbound) shows the pipeline re-run has identified 11 recurring bottlenecks on this corridor, not the 6 in earlier drafts of this report. Every one of the original six locations is still present — but five of them now split into two adjacent bottleneck zones each (e.g. Jodeco Road appears as both rank #1 and #2), plus Old Allatoona Road remains singular. This is consistent with the peak-detection algorithm described in Section 9.1: as more data or refined

thresholds resolve a location's occurrence profile more finely, two adjacent sub-peaks that previously merged into one wide bottleneck can now be detected as two independent ones.
• Table below is the current, correct ranking, replacing the six-row table in earlier drafts.

Rank	Location	County	Occurrence	Annual Mile-Hrs	Severity
1	Jodeco Rd / Exit 222	Henry	99.6%	17,115.0	9,818.8
2	Jodeco Rd / Exit 222	Henry	99.2%	16,639.9	9,573.9
3	Barrett Pkwy / Exit 269	Cobb	99.6%	17,245.3	9,464.2
4	Barrett Pkwy / Exit 269	Cobb	99.6%	17,036.0	9,400.2
5	I-285 / Exit 259	Cobb	99.6%	2,275.5	1,015.3
6	GA-331 / Forest Pkwy / Exit 237	Clayton	95.4%	2,219.1	965.4
7	I-285 / Exit 259	Cobb	98.9%	1,755.3	776.0
8	GA-331 / Forest Pkwy / Exit 237	Clayton	93.1%	1,694.4	730.4
9	I-85 / Exit 242	Fulton	93.1%	279.8	129.7
10	I-85 / Exit 242	Fulton	90.0%	223.4	102.9
11	Old Allatoona Rd / Exit 283	Bartow	15.3%	47.1	3.9

Values above are severity_index and annual_mile_hours as reported by the live regional dashboard, which computes both at the zone (not peak-segment) level — see Section 10.1 for why these figures differ from a peak-segment-only calculation. "I-85 Split" in earlier drafts is the same location as "I-85 / Exit 242" above; the dashboard's naming is used here going forward.

10. Phase 5 — Bottleneck Characterization

Why This Phase Exists

Phase 4 answers where the corridor's recurring bottlenecks are. It does not yet answer how bad each one is, in the specific terms a traffic engineer needs to actually design a fix for it: what time does it typically start backing up, how long does the queue get, how fast does it grow, and how fast does it clear once the peak has passed? Phase 5 exists to compute exactly those numbers — for every recurring bottleneck, for every single day of the year it occurred — turning "this is a known problem location" into the specific, day-by-day operational profile an engineering countermeasure would be designed against.

Key Terminology

Term	Meaning in This Pipeline
Onset / peak / clearance time	The moment a bottleneck's daily episode begins, the moment its queue is longest, and the moment it finally clears — the three anchor points of a single day's congestion episode at that location.
Queue	The stretch of roadway actively congested within a bottleneck's boundary at a given moment — its length in miles is tracked throughout the episode to see how it grows and shrinks over the course of a day.
Queue growth rate / dissipation rate	How quickly the queue lengthens on the way to its daily peak, and how quickly it shortens afterward, in miles per hour — the two numbers that describe whether a bottleneck forms suddenly or builds gradually, and clears quickly or lingers.
Occurrence (Phase 5 sense)	Whether a bottleneck had at least 30 minutes of active congestion on a given day — the minimum bar for that day to count as a real episode rather than noise, distinct from the Phase 4 occurrence percentage used to find the bottleneck's location in the first place.

cbi_bottleneck_characterization.py takes each bottleneck's segment range from segment_recurring_bottlenecks, aggregates the full year of 5-minute observations across that segment range in SQL (congested-segment counts, queue miles, average/minimum speed ratio), then derives daily metrics in Python:

- Onset, peak, and clearance time
- Active congestion minutes and episode duration
- Maximum and average queue length (miles)
- Queue mile-hours
- Queue growth rate and dissipation rate (mph), via linear fit of queue length vs. elapsed time before/after the daily peak

A day only counts as an occurrence if at least 30 minutes of congestion is recorded within the bottleneck's segment range, filtering out noise-level activity. Column definitions are in Appendix A.6.

10.1 SQL Dashboard View Layer

Four PostgreSQL views (CBI_working_SQL_master.sql, statements 021–024) sit on top of bottleneck_daily_metrics and serve as the actual data source for dashboard-ready output — this is the concrete implementation of the “database views” (annual, monthly, weekday, severity ranking) referenced earlier in this report's Current Database Products discussion.

View	Grain	Purpose
vw_bottleneck_annual_dashboard	One row per bottleneck	Full-year rollup: occurrence_pct, avg/median/p95 active minutes, avg/p95/max queue miles, annual_queue_mile_hours, avg_congested_speed_ratio, avg queue growth/dissipation rate
vw_bottleneck_dashboard_ranked	One row per bottleneck	Adds severity_index and severity_rank on top of the annual view (Section 11)

View	Grain	Purpose
vw_bottleneck_monthly_dashboard	One row per bottleneck per month	Same metrics as the annual view, broken out by calendar month
vw_bottleneck_weekday_dashboard	One row per bottleneck per weekday	Same metrics broken out by day of week (Mon–Fri)

A key distinction worth flagging: annual_queue_mile_hours in these views is summed from bottleneck_daily_metrics.queue_mile_hours (Phase 5, computed across each bottleneck's full segment range, every day) — a different quantity from annual_mile_hours in segment_recurring_bottlenecks (Phase 4, computed from congested intervals at the peak segment only, Section 9.1). The two will not match, and that is expected rather than an error; it explains the ranking-order finding in Section 11.

11. Recurring Bottleneck Ranking

The severity ranking is a composite index combining occurrence frequency, queue mile-hours, and congestion speed ratio — see the full, current 11-row ranking with location, county, and occurrence detail in Section 9.1 above. The formula itself is documented below.

Resolved — Severity Formula Located

• CBI_working_SQL_master.sql, statement 022 (vw_bottleneck_dashboard_ranked), defines the exact calculation:

```
severity_index = ROUND(
    (
        annual_queue_mile_hours
        * occurrence_pct / 100.0
        * (1.0 - avg_congested_speed_ratio)
    )::numeric,
    2
)
-- severity_rank = RANK() OVER (ORDER BY the same expression DESC)
```

All three inputs come from vw_bottleneck_annual_dashboard (Section 10.1) at the whole-bottleneck-zone level, not the peak-segment level used by segment_recurring_bottlenecks: annual_queue_mile_hours is summed across every day and the full segment range (Phase 5), occurrence_pct is the percentage of analyzed days with ≥ 30 minutes of active congestion anywhere in that zone, and avg_congested_speed_ratio is the mean speed ratio during congested periods. This resolves the ranking-order question from Section 9.1: Jodeco Road outranks Barrett Parkway on severity_index despite having slightly lower annual_mile_hours in the Phase 4 peak-segment table, because the severity formula also rewards higher occurrence_pct and a lower (more severe) speed ratio — both computed at the zone level, where Jodeco Road's numbers can plausibly overtake Barrett Parkway's even though Barrett's single peak segment carries more raw mile-hours.

Caveat Resolved

• The live regional dashboard (Section 9.1) now provides the actual severity_index values directly, confirming both the formula and its output are correctly traceable to code — the values are no longer a reproducibility gap, only the underlying per-bottleneck occurrence_pct / avg_congested_speed_ratio breakdown remains unverified against the raw database if a full audit trail is needed.

Part III — Automating the Framework Across All Corridors

In Plain Terms

Part II's five-step process works, but right now it only knows how to look at one specific road: I-75 Northbound. The road name is baked into the scripts rather than being something you can just swap out. Part III is about removing that limitation — turning "a process that works for I-75 NB" into "a process that works for any corridor, run automatically on a schedule, for every major highway in the region."

Update — This Part Is Now Largely Complete, Not Planned

• The regional dashboard delivered alongside this report (regional_congestion_report_2026-08-11.html) confirms the multi-corridor pipeline described below has actually been run at full regional scale: 41 corridors (16 Interstate, 25 Arterial), 2,547 corridor segments, 493,833 congestion events detected, 179 recurring bottlenecks identified, and 46,719 daily bottleneck-characterization records, drawn from 132,396,793 of the 427,676,537 region-wide probe observations. The sections below, originally written as a forward-looking plan, are kept for their technical detail (how the coupling was removed, how the registry works) but should now be read as describing completed work, not a roadmap.

12. Corridor-Specific Coupling — Resolved

The single-corridor prototype originally hardcoded I-75 Northbound throughout the codebase rather than treating the corridor as a parameter: CORRIDOR/DIRECTION module-level constants, corridor-specific table names (corridor_i75_nb, i75_nb_segment_profile), and a fixed "i75_nb" output-file prefix. The regional dashboard's 41-corridor coverage confirms this coupling has since been removed — the same Phase 1–5 logic documented in Part II now runs per corridor_id rather than being copy-pasted per road.

13. Corridor Registry — Populated

CBI_working_SQL_master.sql (statements 027–028) creates a corridor_definitions table and a generic corridor_segments table that generalizes the segment_order-numbering pattern from corridor_i75_nb to any corridor:

```
CREATE TABLE corridor_definitions (
    corridor_id     integer GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    road            text NOT NULL,
    direction       text NOT NULL,
    corridor_name   text NOT NULL,
```

```
    corridor_group  text,
    is_active       boolean NOT NULL DEFAULT true,
    UNIQUE (road, direction)
);
```

Earlier drafts of this report described this table as seeded with only eight corridor/direction pairs, all Interstates. The live regional dashboard confirms it now holds 41 active corridors — the full list is in Section 14.

14. Regional Rollout — Actual Coverage

The planned rollout in earlier drafts listed seven Interstate corridors. The delivered regional dashboard shows the actual rollout went further, adding I-985 and a full arterial tier not in the original plan:

Interstates (16 corridors)

- Downtown Connector NB/SB, I-20 EB/WB, I-285 CW/CCW, I-575 NB/SB, I-675 NB/SB, I-75 NB/SB, I-85 NB/SB, I-985 NB/SB

Arterials (25 corridors)

- Abernathy Rd, GA-120, GA-13, GA-140, GA-20, GA-34, GA-5-CONN, GA-54, US-19, US-23, US-29, US-78 — each NB/SB or EB/WB as applicable — plus Tara Blvd Northbound only (no southbound analysis yet, the one asymmetry in current coverage)

Watch list (35 lighter-weight segments, not full corridors)

- Top 25 Collector segments and Top 10 Local segments, ranked by AADT, with a lighter per-segment summary (occurrence, peak hour/day, annual mile-hours, speed ratio) rather than the full five-phase bottleneck pipeline

This confirms the materialized views from Part I (Section 5) and the corridor_segments table above are functioning as the shared base layer across all 41 corridors, exactly as originally proposed.

15. Orchestration and Scheduling

With 41 corridors running, some form of scheduler is producing the regional_congestion_report_*.html file on a recurring basis (the filename is date-stamped). The specific orchestration mechanism — cron, Task Scheduler, or a workflow tool — was not confirmed in the materials reviewed for this report; worth documenting explicitly given the scale now in production, since a manual re-run process would be a significant undertaking at 41 corridors.

Part IV — Interactive Visualization Dashboard

In Plain Terms

Once bottlenecks are being found automatically for every corridor (Part III), the results need somewhere for people outside the project to actually see them. This part originally planned a GitHub Pages website; what actually got built and delivered is different and is described below.

Update — Delivered as a Self-Contained HTML File, Not GitHub Pages

• The plan below (Sections 16–18) described a React/TypeScript GitHub Pages site. What was actually delivered instead is a single, self-contained HTML file (regional_congestion_report_2026-08-11.html) with the results already embedded — Plotly charts and all data baked in at generation time, openable directly in a browser with no server, hosting, or build step required. It covers all 41 corridors plus the 35-segment watch list across four tabs: Regional Overview (KPI cards — highest month, worst day of week, worst time of day, each with total mile-hours), Region Map, Corridors (a sidebar of all 41 corridors, each with the same severity ranking and detail view demonstrated for I-75 NB in Part II), and Watch Segments (the Collector/Local tables in Section 14). A fifth, default-active tab, Metadata, documents data source, methodology, and the regional record counts cited in Section 13.

• This is arguably a stronger deliverable than the original GitHub Pages plan for a single distributable report — zero hosting cost, works offline, easy to email — though it doesn't provide a persistent public URL the way a hosted site would. Sections 16–18 below are kept as the original technology plan for reference, since GitHub Pages hosting of this same static output remains a reasonable next step if a persistent public URL is needed.

Data Source Inconsistency to Fix

• The delivered HTML file's footer and Metadata tab both state “NPMRDS from INRIX” as the data source. Part I, Section 1 of this report documents that GDOT/ARC data is HERE probe data, not INRIX — this appears to be leftover boilerplate text in the dashboard-generation script that should be corrected to match.

16. Target Platform and Technology

Target Platform: GitHub Pages

- React
- TypeScript
- Plotly
- Leaflet

17. Planned Features

- Corridor selection
- Direction selection
- Top-N bottlenecks
- Ranking criteria
- Monthly filtering
- Weekday filtering
- Interactive maps
- Congestion timelines

18. Data Contract

In Plain Terms

"Data contract" is a software engineering term, not a legal one — it means an agreed, fixed format that two separate pieces of software use to exchange data, so each side can be built and changed independently as long as neither breaks the agreed format. Here, the two sides are the Part II/III analysis pipeline (which produces results) and the Part IV website (which displays them). The "contract" between them is simply: the pipeline will always output corridor results as JSON and GeoJSON files in a specific, agreed structure, and the website will always expect to receive files in exactly that structure.

Why this matters in practice: it means the website never talks to the PostgreSQL database directly, and never needs database credentials, a live connection, or server-side code of its own — it just reads plain files. That is what keeps a GitHub Pages deployment "fully static" (Section 16): GitHub Pages can only serve files, not run a database query, so this file-based handoff is what makes hosting there possible at all.

The dashboard is intended to be a static, client-side consumer of exported JSON and GeoJSON per corridor — generated by the pipeline in Part II/III rather than querying PostgreSQL directly from the browser, keeping the GitHub Pages deployment fully static.

Part V — AI-Driven Dashboard on Microsoft Fabric + Power BI (Proposed)

Status: Proposed Architecture — Not Yet Implemented

• Nothing in this part exists in the current codebase or repository. It supersedes an earlier draft of this Part, which proposed a more generic Azure OpenAI + Functions + Static Web Apps architecture; that approach is replaced here by Microsoft Fabric and Power BI, which cover the same ground with less custom engineering. It should be read as a proposal to validate and scope, not a status update.

In Plain Terms

Part IV's website is simple and free, but it can only show pre-built pages — a planner can look at what's already there, but can't ask it a new question like "which bottleneck got worse compared to last year?" without someone on the project team building that specific page first.

Part V describes a different kind of tool: an interactive dashboard where a planner can click around a map, filter charts by month or corridor, and — with an AI assistant built in — type an actual question in plain English and get an answer, without needing to know SQL or ask a developer to build a new report. This part explains what that requires, using Microsoft's Fabric and Power BI products, and is honest about what's simple (most of it is just configuring existing tools) versus what costs real money (the AI assistant, called Copilot, needs a paid tier).

19. What Microsoft Fabric and Power BI Actually Are

Before the architecture, three product names that will come up repeatedly are worth defining in plain terms, since they're easy to conflate:

Term	In Plain Terms
Microsoft Fabric	A cloud platform that stores data and runs the behind-the-scenes machinery — pulling data in on a schedule, organizing it, and making it available to reporting tools. Think of it as the engine room: most users never interact with Fabric directly.
Power BI	The reporting and visualization tool that sits on top of Fabric — what a planner actually opens in a browser to see the map, charts, and tables. Power BI is part of the Fabric platform, not a separate product to buy.
Copilot (in Power BI)	Microsoft's AI assistant, built into Power BI, that lets a user type a question in plain English ("which corridor had the most congestion in July?") and get an answer without writing any code or query. This is the specific feature that answers the "type in what they'd like to know" requirement.

The rest of this Part explains how data gets from the existing PostgreSQL database (Parts I–III) into Fabric, how a report gets built on top of it, and how Copilot gets turned on — in that order, since each step depends on the one before it.

20. Ingestion — On-Premises Data Gateway (Option A)

The first step is simply getting the data from the existing PostgreSQL database (which lives on a local Windows machine, not in the cloud) into Fabric. Fabric cannot see a database sitting on a local machine by default — it needs a bridge. That bridge is called the On-Premises Data Gateway, a small piece of software Microsoft provides specifically for this situation: it sits on the local machine alongside the database, and lets Fabric ask it for data on a schedule without the database itself ever needing to be moved to the cloud or made publicly reachable.

Two ways to get data into Fabric were evaluated. This report adopts Option A — the gateway approach above — because it requires no script to maintain once it's set up; Fabric handles the scheduling and data movement itself. The alternative (Option B, not adopted) would have a Python script export the data to files and upload them to the cloud on a schedule — documented as a fallback only if gateway setup takes longer than expected, since it has more moving parts to maintain long-term.

Step by step

Step	Detail	In Plain Terms
1. Install the gateway	On-premises Data Gateway installed on the same Windows machine the pipeline already runs on (or any machine that can reach PostgreSQL).	A one-time software install, like installing any other application.
2. Register to tenant	Gateway registered to the Fabric/Power BI tenant by a Fabric or Power BI admin.	Tells Microsoft's cloud "this specific gateway belongs to our organization" — needs an administrator's login, not just anyone's.
3. Build the Dataflow	A Fabric Dataflow Gen2 (or Data Pipeline with a Copy Activity) using the	A Dataflow is a saved, repeatable instruction: "connect to this database, run

Step	Detail	In Plain Terms
	PostgreSQL connector, routed through the gateway.	this query, and save the result." It's built once using point-and-click tools, not code.
4. Source views	Points at the same six sources the corridor reports already use: vw_bottleneck_dashboard_ranked, vw_bottleneck_monthly_dashboard, vw_bottleneck_weekday_dashboard, segment_recurring_bottlenecks, corridor_segments, congestion_events.	These are the exact same database views already documented in Part I, Section 10.1 — no new calculations, just pointing Fabric at numbers that already exist.
5. Land the data	Output lands in a Fabric Lakehouse as Delta tables.	A Lakehouse is Fabric's storage area — a copy of the data, refreshed on schedule, that Power BI reads from instead of hitting the live database every time someone opens a report. A Delta table is simply the specific file format Fabric uses to store that copy efficiently.
6. Schedule refresh	Refresh scheduled independently of the local pipeline run, since the gateway talks to PostgreSQL directly rather than depending on an export step.	Set once (e.g. "refresh every night at 2 AM") and it runs unattended from then on.

The tradeoff against the Parquet-export fallback is lead time versus maintenance: the gateway requires admin approval and installation before anything flows, but afterward needs no separate script — refresh, schema drift, and incremental load are all handled natively by the Dataflow, which a hand-written export script would otherwise have to replicate.

21. Semantic Model and Report

Once the six tables exist in the Lakehouse, the next step is a Power BI semantic model — in plain terms, a map of how the tables relate to each other ("this bottleneck belongs to this corridor," "this daily record belongs to this bottleneck") plus a set of named, reusable calculations (called measures) like "total annual mile-hours" or "average severity index." This is the step that turns six separate tables into something Power BI — and later, Copilot — can reason about as a single connected dataset rather than unrelated spreadsheets.

Concretely: relationships are drawn from corridor_segments.corridor_id to the other tables' corridor_id/corridor+direction columns, and from segment_recurring_bottlenecks.bottleneck_id to bottleneck_daily_metrics.bottleneck_id (a clean one-to-many relationship now that the multi-corridor migration in Part III fixed the identity-column collision risk that would otherwise have made this join unreliable). A date table is marked on congestion_events.analysis_date so Power BI understands concepts like "last month" or "year over year" automatically. The measures themselves — total annual mile-hours, average severity index, bottleneck count by corridor — are written to mirror the column definitions in Appendix A exactly, so a number on the dashboard and the same number in this report are guaranteed to mean the same thing.

With the semantic model in place, each requested dashboard element maps onto a standard, off-the-shelf Power BI visual — none of it requires custom software development, only configuration inside Power BI's report designer:

Requested Element	Power BI Visual	Notes
Interactive map	Azure Maps visual	corridor_segments plotted by lat/long, colored by congestion severity; segment_recurring_bottlenecks overlaid as markers sized by severity_index
Graphs / charts	Bar, line, and matrix visuals	Severity ranking (bar), vw_bottleneck_monthly_dashboard trend (line), vw_bottleneck_weekday_dashboard by weekday × bottleneck (matrix)
Table	Table/matrix visual	vw_bottleneck_dashboard_ranked, same columns as this report's Section 11 ranking table
Descriptive text	Smart Narrative visual	Auto-generates a natural-language summary of the page's visuals and updates live as filters change — e.g., automatically writing "Jodeco Road has the highest severity index, driven by a 98.1% occurrence rate" as a sentence on the page, without anyone typing that sentence by hand

22. Natural-Language Query Layer

This is the feature that answers the specific request for a dashboard where "users could type in what they would like to know." Three ways to deliver that were considered, in order of increasing capability, cost, and engineering effort — they are not mutually exclusive, and the recommended approach (end of this section) is to start with the free option and add the paid one only if it's clearly worth it.

Option 1 — Power BI Q&A visual (free)

A search-box visual that comes with every Power BI report at no extra cost. A user types a question in plain English — "which corridor has the most bottlenecks?" — and Power BI matches the words in the question to the semantic model's table and measure names, then picks an appropriate chart to answer with automatically. It works well when column and measure names are written in everyday language (which the semantic model in Section 21 is designed to support), and poorly when the underlying names are cryptic abbreviations.

Option 2 — Copilot in Power BI (paid)

Copilot is Microsoft's more advanced AI assistant, and handles noticeably more open-ended questions than the Q&A visual — it can combine multiple measures, generate a written summary of a whole page, and even write the underlying calculation (in a language called DAX) on the fly rather than requiring every calculation to already exist as a pre-built measure. The catch, worth stating plainly rather than glossing over: Copilot requires a Fabric capacity of size F64 or larger, or Power BI Premium P1 or higher

— these are paid subscription tiers, and this is a genuine budget line item to confirm with whoever manages the Azure/Fabric account, not a setting to simply switch on.

Option 3 — Custom Azure OpenAI integration (paid, most engineering effort)

A purpose-built alternative: a small application that takes a typed question, uses Azure's OpenAI service to translate it into a precise database query, runs that query against Fabric, and returns a formatted answer. This is more work to build than turning on Copilot, but gives full control over exactly how questions get answered — which matters here specifically because this report's own data has a subtlety a generic AI assistant has no built-in reason to respect: Appendix A.4's annual_mile_hours is measured only at a bottleneck's single worst segment, while Appendix A.7's annual_queue_mile_hours is measured across the bottleneck's entire zone, every day of the year. Copilot, using its own general-purpose reasoning, could easily answer a question using the wrong one of these two similarly-named numbers unless specifically guided not to.

Recommended sequencing: start with the free Q&A visual to validate that people actually use the ask-a-question feature at all, add Copilot once capacity/licensing is approved if its answers are noticeably better, and only build the custom Azure OpenAI path (Option 3) if Copilot's answers don't reliably respect this report's specific metric definitions closely enough for users to trust them.

23. Schedule

With Option A, the Dataflow Gen2 refresh (Section 20) is scheduled independently of run_cbi_pipeline.bat (Part III) entirely, since the gateway connects to PostgreSQL directly rather than waiting on an export step — in practice, this means picking a time of day (e.g., after the nightly pipeline run typically finishes) and setting it once in the Fabric portal. If the semantic model uses a mode called DirectLake against the Lakehouse — which reads the Lakehouse's Delta tables directly rather than keeping its own separate copy — it needs no separate refresh step of its own beyond the Dataflow's, one less schedule to manage.

24. Open Questions

- Fabric capacity: is a Fabric workspace/capacity already provisioned, or is this a new procurement — and does it already cover Copilot's F64+ licensing requirement?
- Gateway approval lead time: who approves and installs the On-Premises Data Gateway, and how long is that likely to take relative to the Parquet-export fallback?
- Access model: internal agency use (share the Power BI workspace/app directly) or public-facing (Power BI Embedded, more engineering effort)?
- Which of the two Phase 4 bottleneck-detection methods (Section 9) becomes canonical before it's exposed through a natural-language query interface, since Copilot or a custom AI layer would otherwise need to explain disagreements between the two algorithms.

Current Development Status

Component	Status
Database Import & Schema	Complete
Indexing & Performance Tuning	Complete
Materialized Congestion Views (network-wide)	Complete
Corridor Preparation (I-75 NB)	Complete
Event Detection (production detector)	Complete
Annual Batch Processing (I-75 NB)	Complete
Recurring Bottleneck Detection	Complete — 11 bottlenecks confirmed on I-75 NB, 179 region-wide
Bottleneck Characterization	Complete
Composite Severity Scoring	Complete — formula and live values both confirmed
Corridor Registry & Generic Engine	Complete — 41 corridors active
Multi-Corridor Rollout	Complete — 16 Interstate, 25 Arterial, 35-segment watch list
Interactive Dashboard	Delivered — self-contained HTML, not GitHub Pages (Part IV)
Fabric + Power BI AI Dashboard	Proposed (Option A: gateway ingestion)

Assessment

The project has progressed well past its original single-corridor prototype: a functioning pipeline ingests 427.7 million annual probe data observations, and now runs the full detection-through-characterization process documented in Part II across 41 corridors region-wide — 493,833 congestion events, 179 recurring bottlenecks, and 46,719 daily characterization records, delivered as a self-contained interactive HTML dashboard covering Interstates, Arterials, and a Collector/Local watch list.

Reviewing the actual output files (Section 7.4, Section 9) and the live regional dashboard turned several open questions from earlier drafts of this report into settled findings. The refined event detector's core-splitting logic is validated against a concrete before/after example; the choice between the two Phase 4 bottleneck-detection methods is decided with evidence; the severity formula and its live output values are both confirmed; and Part III's multi-corridor rollout, once a forward-looking plan, is now documented as completed work at a scale beyond the original seven-corridor target — including an arterial tier and a watch list not in the original scope.

What remains open is narrower still: correcting or implementing the Gaussian smoothing described in Section 9.1, retiring the duplicate annual runner (Section 8), fixing the INRIX/HERE inconsistency in the delivered dashboard's footer (Part IV), and confirming the orchestration mechanism behind the regional

dashboard's scheduled regeneration (Section 15). None of these affect the validity of the regional results now in production — they are documentation and consistency items.

Appendix A — Glossary of Metrics and Variables

Every table in this report pulls its column names directly from the pipeline's SQL views and Python outputs. This appendix defines each one, grouped by where in the pipeline it is produced, so the tables in Parts I–II can be read without cross-referencing the code.

A.1 Core Traffic and Congestion Measures

Computed at the individual TMC / 5-minute-interval level, in both the cbi_observations materialized view (Part I, Section 5) and independently in the Python phase scripts (Part II).

Variable	Definition
speed	Observed average vehicle speed on a segment for one 5-minute interval, from probe vehicle data (mph).
reference_speed	The segment's free-flow reference speed — the "normal" baseline speed used to judge congestion (mph).
historical_average_speed	Long-run historical average speed for the segment/time-of-day; contextual, distinct from reference_speed.
speed_ratio	speed ÷ reference_speed. 1.0 = free-flow. Lower values mean slower-than-normal conditions.
is_congested / congestion cutoff	An interval is classified congested when speed_ratio < 0.70 (speed under 70% of reference speed). Used consistently across SQL and Python.
speed_drop	reference_speed − speed, floored at 0. How many mph below normal the segment is running.

A.2 Spatial Reference

Variable	Definition
TMC	Traffic Message Channel — the standard NPMRDS segment identifier. I-75 Northbound is made up of 101 TMCs.
segment_order	The corridor-specific 1–101 ordinal position of a TMC along I-75 NB, south to north; used in place of raw TMC codes throughout the pipeline.
corridor_start_mile / corridor_end_mile / start_mile / end_mile	Distance in miles along the corridor from its southern starting point (mile 0) to a segment or bottleneck boundary.

Variable	Definition
intersection / representative_intersection	Nearest named cross-street or interchange exit, used as a human-readable location label.
miles	Physical length of one TMC segment.

A.3 Congestion Event Metrics (Phase 2/3 outputs)

Columns in i75_nb_events_*.csv and i75_nb_refined_events_*.csv (Section 7).

Variable	Definition
event_id	Sequential ID for one retained connected-component event on a given day. Not stable across days or between the unrefined and refined detectors.
start_time / end_time	Timestamp of the event's first and last congested 5-minute interval.
duration_minutes	Elapsed time from start_time to end_time.
cell_count	Number of (segment, 5-minute interval) cells making up the event.
segment_count	Number of distinct segments the event touched at any point.
first_segment_order / last_segment_order	Most upstream and most downstream segment_order values touched by the event.
first_tmc / last_tmc, first_intersection / last_intersection	TMC code and nearest cross-street at the event's upstream and downstream boundary.
maximum_extent_miles / maximum_contiguous_extent_miles	Greatest simultaneous contiguous congested length at any single 5-minute interval within the event.
maximum_total_active_miles	Refined detector only: greatest simultaneous total congested length, including any non-contiguous congested segments at that moment.
event_area_mile_hours	Event's total footprint: (sum of congested segment-miles across every 5-minute cell) × (5/60). Combines duration and spatial extent into one number.
average_speed_mph	Mean observed speed across all cells in the event.
minimum_speed_ratio / average_speed_ratio	Lowest and mean speed_ratio recorded anywhere in the event.
average_speed_drop_mph / maximum_speed_drop_mph	Mean and peak speed_drop recorded in the event.
bounding_box_intensity_pct	Unrefined detector only: cell_count divided by the event's full rectangular time × segment bounding box, as a percentage — how “solid” vs. “sparse” the event is.
estimated_upstream_boundary_speed_mph	Refined detector only: linear-fit rate at which the event's upstream boundary moves over time. Positive = front moving downstream

Variable	Definition
	(queue shrinking from the back); negative = propagating upstream (queue growing backward).

A.4 Segment-Profile Bottleneck Metrics (Phase 4, adopted)

Columns in i75_nb_segment_bottlenecks.csv (Section 9.1) — the canonical Phase 4 output.

Variable	Definition
bottleneck_id	Identifier for one recurring bottleneck, ranked by annual_mile_hours (1 = highest).
peak_segment_order	The single segment with the highest smoothed annual occurrence percentage within the bottleneck — its "epicenter."
start_segment_order / end_segment_order	Full expanded boundary of the bottleneck after peak detection and boundary expansion.
segment_count	Number of segments spanned by the bottleneck.
peak_occurrence_pct	Percentage of the 261 analyzed weekdays on which the peak segment was congested, after smoothing. This is the "Occurrence" column discussed previously.
maximum_raw_occurrence_pct	The same measurement before smoothing is applied.
annual_mile_hours	Sum of congested segment-miles × time across the full year for this bottleneck's segment range — the primary ranking metric.
average_congested_minutes / p95_congested_minutes	Mean and 95th-percentile daily congested duration at the peak segment, across days when it was congested.
average_congested_speed_ratio	Mean speed_ratio during congested periods at the peak segment.

A.5 Event-Clustering Bottleneck Metrics (Phase 4, retired method)

Columns in i75_nb_recurring_bottlenecks.csv (Section 9.2). Included for reference since the method is discussed in this report, even though it is not the adopted output.

Variable	Definition
typical_midpoint_mile	Median midpoint mile position of all events assigned to the DBSCAN cluster.
event_count	Number of individual congestion events grouped into the cluster.
occurrence_days / weekday_occurrence_pct	Number and percentage of the 261 weekdays with at least one event in the cluster.

Variable	Definition
avg_duration_minutes / median_duration_minutes / p95_duration_minutes	Central tendency and tail of event durations within the cluster.
avg_extent_miles / p95_extent_miles	Central tendency and tail of maximum_contiguous_extent_miles within the cluster.
avg_speed_ratio	Mean average_speed_ratio across events in the cluster.
typical_start_time / typical_end_time	Circular (clock-time-aware) mean of event start/end times, so times near midnight average correctly.

A.6 Daily Bottleneck Characterization Metrics (Phase 5)

Columns in bottleneck_daily_metrics, produced by cbi_bottleneck_characterization.py (Section 10).

Variable	Definition
occurrence	True if the bottleneck had at least 30 minutes of active congestion that day (the minimum-activity filter).
onset_time / peak_time / clearance_time	First congested interval, maximum-queue interval, and last congested interval (+5 min) for the bottleneck that day.
active_congestion_minutes	Total minutes with any congestion in the bottleneck's segment range that day.
episode_duration_minutes	clearance_time − onset_time: the full wall-clock span of the episode, including brief internal gaps.
maximum_queue_miles / average_queue_miles	Peak and mean simultaneous congested length within the bottleneck's range that day.
queue_mile_hours	The day's total footprint for this bottleneck, in the same mile-hours unit as annual_mile_hours.
maximum_congested_segments	Most segments simultaneously congested within the bottleneck's range that day.
queue_growth_rate_mph / queue_dissipation_rate_mph	Linear-fit rate of change of queue length vs. time, before and after the day's peak. Growth is typically positive (queue lengthening); dissipation is typically negative (queue shortening after the peak).

A.7 Composite Ranking

Columns in vw_bottleneck_dashboard_ranked (Section 10.1 / 11).

Variable	Definition
severity_index	annual_queue_mile_hours × (occurrence_pct / 100) × (1 − avg_congested_speed_ratio), rounded to 2 decimals. Combines total burden, how often it occurs, and how severe the speed drop is when it does.

Variable	Definition
severity_rank	RANK() of severity_index in descending order — 1 is most severe.
annual_queue_mile_hours	Zone-level total from Section 10.1's annual dashboard view; not the same figure as annual_mile_hours in Appendix A.4, which is measured only at the peak segment.

A.8 Microsoft Fabric and Power BI Terms

Plain-language definitions for every Fabric/Power BI/Azure term used in Part V, for readers less familiar with Microsoft's business-intelligence products.

Term	In Plain Terms
Microsoft Fabric	The overall cloud platform that stores data and automates moving it around on a schedule — the “engine room” behind the dashboard, not something end users open directly.
Power BI	The reporting tool built on top of Fabric that a planner actually opens in a browser to see maps, charts, and tables.
Workspace	A container in Fabric/Power BI holding a related set of items — dataflows, a Lakehouse, reports — for one project. Access is granted per workspace.
Capacity	The paid compute resource a Fabric workspace runs on. Capacity size (e.g. F64) determines which features — including Copilot — are available, and is billed independently of how much data is stored.
On-Premises Data Gateway	Small software installed next to a local database (like the project's PostgreSQL instance) that lets cloud services such as Fabric request data from it on a schedule, without exposing the database to the public internet.
Dataflow Gen2	A saved, repeatable, point-and-click instruction in Fabric: connect to a data source, run a query, and save the result. Built once in a visual designer, not written as code.
Data Pipeline / Copy Activity	Fabric's alternative to a Dataflow for moving data — similar end result, generally used for simpler, larger, or more scheduling-heavy data movement.
Lakehouse	Fabric's storage area for data pulled in from outside sources. Power BI reads from the Lakehouse's saved copy rather than querying the live source database every time a report is opened.
Delta table	The specific file format Fabric uses to store data inside a Lakehouse efficiently — a technical storage detail, not something a report author needs to interact with directly.
Semantic model	The layer that defines how a Lakehouse's tables relate to each other (e.g. “this bottleneck belongs to this corridor”) plus a set of named, reusable calculations. This is what Power BI reports — and Copilot — actually query against.
DAX	Data Analysis Expressions — the formula language used to write calculations (measures) in a Power BI semantic model, similar in spirit to Excel formulas but built for relational data models.

Term	In Plain Terms
Measure	A named, reusable calculation in the semantic model, such as “total annual mile-hours” — written once in DAX, then usable in any chart, table, or Q&A/Copilot question.
DirectLake	A Power BI mode that reads a Lakehouse's Delta tables directly, without keeping a separate copy inside Power BI itself — means one less refresh schedule to manage.
Copilot (in Power BI)	Microsoft's built-in AI assistant that answers typed, plain-English questions about a report's data, and can generate new DAX measures or written summaries on the fly. Requires Fabric capacity F64+ or Power BI Premium P1+.
Q&A visual	A free, simpler alternative to Copilot built into every Power BI report — matches a typed question's words to the semantic model's table/measure names and picks an appropriate chart, without generative AI behind it.
Smart Narrative	A Power BI visual that automatically writes a plain-English paragraph summarizing whatever charts are on the same report page, and updates that text live as filters change.
Power BI Embedded	A way to display a Power BI report inside a custom website rather than the standard Power BI portal — relevant only if the dashboard needs to be public-facing rather than shared directly with agency staff.