Trino Smart Routing: ML Design Document

1. Executive Summary & Decisions

1.1. PROBLEM & OBJECTIVES

The current non-curated Trino environment uses a symmetric, round-robin routing strategy across its 405 nodes. Query log analysis shows a significant imbalance: **~2% of "heavy" queries consume >95% of all CPU resources, while the remaining ~98% of "light" queries are resource-frugal.** This setup leads to resource contention, where small, interactive queries can be starved by large, batch-style queries, degrading performance and inflating costs due to over-provisioning for the peak minority workload.

This document proposes a **"Smart Routing"** solution. We will introduce a lightweight, ML-powered routing service that classifies incoming queries as *light* or *heavy* in real-time. This allows us to route queries to two distinct, right-sized Trino clusters: a small, high-concurrency cluster for light queries and a large, high-power cluster for heavy ones.

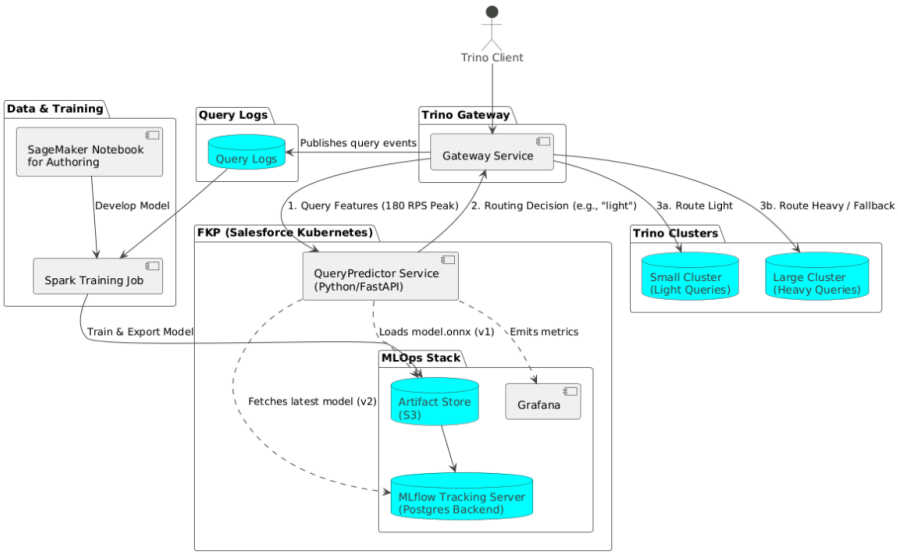
OBJECTIVES:

- **Improve Performance:** Reduce P90/P99 query latency for 98% of queries.
- **Reduce Cost:** Achieve >25% reduction in Trino compute costs (CTS) by right-sizing clusters.
- **Increase Stability:** Isolate workloads to prevent "noisy neighbor" problems.

Non-Goals:

- This is not a query planner or optimizer. It operates purely at the routing layer.
- It will not rewrite user SQL.
- It will not manage user-level permissions or resource groups beyond routing.

1.2. ARCHITECTURE OVERVIEW



1.3. EXECUTIVE DECISION TABLE

	A	B	C	D	E
1	**Area**	**Decision**	**Alternatives Considered**	**Why**	**Downsides / Mitigations**
2	**Model Format**	**XGBoost serialized to ONNX (Open Neural Network Exchange) .**	Native XGBoost binary, LightGBM, shallow Neural Net, **Spark MLlib, MLeap**.	**Portability & Performance.** ONNX provides a universal, high-performance runtime, decoupling the serving environment from the training framework (Python/XGBoost). It supports batching and SIMD optimizations.	Minor conversion overhead. **Mitigation:** Automate ONNX conversion in the training pipeline.
3	**Serving**	**Python QueryPredictor service on FKP** with blue-green deployments via Argo CD.	KServe on FKP.	**Simplicity & Control.** A simple FastAPI? service is sufficient for this task, has a low resource footprint, and gives us full control over logic, logging, and metrics. KServe is overkill for v1.	**Mitigation:** Create a reusable service template.
4	**Shadow Strategy**	**Async Mirroring on Trino Gateway**	Envoy `mirror()`	**Highest Fidelity.** Preserves all original request headers, session properties, and user context. Avoids complexities of recreating state. It's the most direct way to compare production vs. shadow.	Requires custom Java development in the Gateway. **Mitigation:** Develop the plugin behind a feature flag with extensive unit/integration tests. Isolate the shadow call in a separate thread pool to prevent impacting production latency.
5	**Fallback Logic**	**On timeout or error, default route to the Large cluster.**	Default to Small cluster, round-robin, return error to user.	**Prioritizes Safety & Availability.** Sending a potentially heavy query to the Large cluster is the safest failure mode. It avoids overwhelming the Small cluster and impacting the 98% of light queries.	May slightly over-utilize the Large cluster during router outages. **Mitigation:** Implement aggressive alerting on fallback rate to detect and resolve router issues quickly.
6	**MLOps Strategy**	**Phased Approach:** **v1:** DIY model management using an **existing S3 bucket** for artifacts. **v2:** Integrate with self-hosted MLflow on FKP migrating artifacts to MinIO.	SageMaker (Endpoints/Feature Store disabled), KServe, BentoML	**Maximizes velocity.** Using S3 avoids the setup/approval overhead for MinIO, de-risking the initial timeline. We can launch the core routing functionality and add robust tracking with MLflow later.	**v1 has manual model governance.** Less auditability and higher risk of human error. **Mitigation:** Use strict, date-based versioning for model files in S3 and document all model promotions manually in quip/git/confluence.

2. Context, Requirements & SLAs

2.1. FUNCTIONAL REQUIREMENTS

- **FR1: Classify:** The QueryPredictor must classify incoming Trino queries as `light` or `heavy`.
- **FR2: Route:** The Trino Gateway (TGW) must route queries to the corresponding Small or Large cluster based on the classification.
- **FR3: Fallback:** TGW must implement a safe fallback mechanism in case of model inference failure, timeout, or system unavailability.
- **FR4: Versioning:** The TGW must support versioned models to enable safe rollouts (e.g., via headers `X-Model-Version: v1.2.0`).
- **FR5: Idempotency:** The QueryPredictor decision for a given query, config (user, sessionProperties) and model version should be idempotent.

2.2. NON-FUNCTIONAL REQUIREMENTS (NFRS)

- **NFR1: Latency:** QueryPredictor inference latency (p99) must be < 500ms to stay within the overall query submission budget. This includes network hops, featurization, and model prediction.
- **NFR2: Throughput:** Must handle **180 RPS** with 50% headroom, targeting **~270 RPS**.
- **NFR3: Availability:** The QueryPredictor service must have an SLO of **99.95%**. The error budget is ~22 minutes of downtime per month.
- **NFR4: Security:** All communication must be over TLS. The service must run with the least privilege. No PII should be logged without redaction. Secrets managed via Vault.

2.3. SUCCESS METRICS

	A	B	C	D
1	**Metric**	**Current Baseline**	**Target**	**Description**
2	**CTS Reduction**	405 nodes @ ~55% util.	**>25% reduction**	Cost savings from right-sizing clusters (e.g., to 300 Large + 21 Small nodes).
3	**P90 ExecutionTime (Light)**	4.7s	**≥ 30% improvement**	Faster execution for the 98% of queries on a dedicated, less-congested cluster.
4	**P99 ExecutionTime (Light)**	1hr	**≥ 20% improvement**	Reduced tail latency for interactive queries.
5	**Heavy-to-Light FNR**	N/A	**< 1%**	False Negative Rate: The percentage of heavy queries misclassified as light. Critical to protect the Small cluster.
6	**Router Error Rate**	N/A	**< 0.05%**	Percentage of requests to the router that fail or time out.

3. Workload Analysis & Capacity Assumptions

3.1. WORKLOAD DISTRIBUTIONS

- **CPU-Time:** 98% of queries use < 5 minutes of CPU time, but consume < 5% of total CPU resources.
- **Memory:** 90% of queries use < 1GB peak memory; 98% use < 10GB. The top 2% have massive memory requirements.
- **Concurrency:** Peak gateway traffic to the non-curated cluster is **180 RPS**.

3.2. DEFINING LIGHT VS. HEAVY

- **Definition:** A query is classified as **"light"** if its `totalCpuTime` is **< 300 seconds (5 minutes)**. All other queries are **"heavy"**.
- **Assumption:** CPU-time is the most reliable proxy for resource impact. Memory is a secondary signal. We will validate this during offline analysis.
- **Class Balance:** The expected split is **98% light / 2% heavy**. This is a highly imbalanced classification problem.

3.3. ROUTER SERVICE CAPACITY

- **Target QPS:** 180 RPS * 1.5 (headroom) = **270 QPS**.
- **Latency Budget:** 500ms p99.
- **Concurrency:** Using Little's Law: `Concurrency = Arrival Rate * Latency`.
  - `270 queries/sec * 0.500 sec = 135`
  - We need to handle at least **135 concurrent requests** within the service at any given time.
- **Assumption:** This level of concurrency is significant. We will scale pods based on CPU utilization, targeting 70%.
  - **Min Pods:** 3 (for high availability across AZs)
  - **Max Pods:** 10+ (to handle unexpected spikes)

4. Architecture & Serving

4.1. End-to-End Flow

The routing process is a request-response interaction between the Trino Gateway and the QueryPredictor service.

1. **Request:** The Trino Gateway receives a query. Instead of routing directly, it calls the **QueryPredictor Service** sending the query text and session properties.
2. **Featurize & Infer:** The QueryPredictor service receives the request, generates a feature vector from the text and uses

the loaded ONNX model to classify the query.

3. **Decision:** The QueryPredictor returns a simple JSON response (e.g., `{"decision": "light"}`) back to the Trino Gateway. It does **not** route the query itself.
4. **Route:** The Trino Gateway receives the decision and directs the original query to the appropriate Trino cluster (Small or Large).
5. **Fallback:** If the call to the QueryPredictor fails or times out (>500ms), the Gateway's internal logic defaults to routing the query to the **Large cluster** as a safety measure.

4.2. API CONTRACT

REQUEST: `POST /V1/ROUTE`

```
{
  "queryId": "20250811_123456_00012_abcde",
  "user": "service-account-etl",
  "source": "tableau-dashboard-x",
  "sqlText": "SELECT count(*) FROM hive.logs.my_table WHERE day = '2025-08-10'",
  "sessionProperties": {
    "catalog": "hive",
    "schema": "logs"
  }
}
```

Response (Success): `200 OK`

```
{
  "queryId": "20250811_123456_00012_abcde",
  "decision": "light",
  "targetCluster": "trino-small-cluster.internal",
  "modelVersion": "v1.2.1-a4bcf1",
  "modelScore": 0.97
}
```

Response (Error): `500 Internal Server Error`

```
{
  "error": "Inference engine failed",
  "queryId": "20250811_123456_00012_abcde"
}
```

4.3. CIRCUIT BREAKERS & FALLBACKS

- **Gateway side Timeout:** The Gateway HTTP client will have a strict **500ms timeout** on requests to QueryPredictor. On timeout, it will immediately route to the Large cluster and increment a `qp.timeout` metric.
- **Default Route:** The primary safety mechanism is the **default-to-large** fallback, ensuring no query is ever dropped due to a router inference failure.

4.4. FAILURES, MISCLASSIFICATIONS & CACHING

- **Small** cluster will have resource group limits (on cputime and memory). If a query fails with an `INSUFFICIENT_RESOURCES` error on the small cluster, **TGW retries the request on large cluster immediately.**

- We don't want to send the error back to the user mainly because the user has not control on which cluster to query and an `INSUFFICIENT_RESOURCES` on a small cluster is due to cost optimization on Trino's end. The same error on a large cluster puts the ball in user's court as far as optimization is concerned. On the small cluster, the user doesn't have any recourse except to retry (and still fail due to QP misclassification as a slow query).
- Retries are tricky particularly if its a write/update/create request due to .... To minimize such retries, Trino will setup a distributed cache (Redis) and save queries (with templates for timestamps in WHERE clauses) that failed with `INSUFFICIENT_RESOURCES` in it. If a new query has a cache hit, TGW will route it to the large cluster by default without calling the QueryPredictor.

5. Data, Features, Labels & Modeling

5.1. FEATURES

A combination of static and historical features will be used, derived from the query text and its context. Feature generation must be extremely fast and operate within a strict latency budget. If parsing a query with `sqlglot` fails or exceeds a **50ms timeout**, the system will fall back to using only token-based features.

- **Text Statistics & Heuristics (Low-level):**
  - **Counts:** Raw SQL length, token count.
  - **Keyword Counts:** `JOIN`, `GROUP BY`, `ORDER BY`, `WINDOW`, `DISTINCT`, `UNION`, `CROSS JOIN`, `CTE`, `CTAS`, `INSERT`, `UPDATE`, `DELETE`, `CREATE`, `ANALYZE`.
  - **Markers:** Presence of wildcard (`*`), shuffle-risk markers (e.g., multi-way joins combined with `ORDER BY` or `DISTINCT`).
- **AST-Derived Features (via `sqlglot`):**
  - **Structural Complexity:** Abstract Syntax Tree (AST) depth, number of CTEs/subqueries.
  - **Join Analysis:** Total number of joins, density of the join graph.
  - **Operator Usage:** Window function count, use of `DISTINCT`, `ORDER BY`, `LIMIT`, `HAVING`.
  - **Predicate Complexity:** Number of `AND`/`OR` clauses, use of inequalities, size of `IN`-lists, use of `LIKE`/`REGEXP`.
  - **Set Operations:** Presence of `UNION`, `INTERSECT`, etc.
- **Contextual & Historical Features:**
  - **Context:** `catalog`, `schema`, `user`, `source` (client info), `tableName`.
  - **Temporal:** `hour_of_day`, `day_of_week`.
  - **Historical (v2):** `avg_runtime_for_user`, `avg_runtime_for_template`, rolling heavy-rate for a given query template.
- **Final Representation:**
  - Features will be represented as a combination of a **dense vector** of the statistical/contextual features and a **sparse vector** from **TF-IDF n-grams** (1- and 2-grams) on the SQL tokens, with a vocabulary size of 5k-20k.

5.2. LABELS

Labels will be generated from historical Trino query logs.

- **Label Source:** Completed query logs containing `queryId`, `totalCpuTime`, `peakUserMemoryBytes`, `errorCode`, etc.
- **Label Definition:** A query is labeled `is_heavy = 1` if any of the following are true, otherwise `is_heavy = 0`:
  - `totalCpuTime >= 300` (seconds)
  - `peakUserMemoryBytes > 10737418240` (10 GiB)

- `errorCode = 'INSUFFICIENT_RESOURCES'`

- **Design Rationale for the "Heavy" Label:**

  This multi-faceted definition was chosen over a simple CPU threshold to create a more robust and production-aware label.

  - **Beyond CPU:** A query can be resource-light on CPU but consume vast amounts of memory (a "memory bomb"). Including `peakUserMemoryBytes` prevents these queries from being misclassified as "light" and destabilizing the small cluster.
  - **Self-Correction:** Labeling queries that failed with `INSUFFICIENT_RESOURCES` as heavy creates a self-correcting feedback loop. On retry, the model is more likely to route these queries to the Large cluster where they have a better chance of success.
  - **Holistic Impact:** Together, these conditions define "heavy" based on a query's true resource impact (CPU, memory, or failure risk), providing a much richer training signal for the model.

- **Data Cleaning:**
  - Filter out failed queries not relevant to resource exhaustion.
  - Deduplicate retried queries, using the final attempt's stats.
  - Control for cluster state: Exclude queries run during known incidents or high-load periods to avoid noisy labels. [*TBD. Will have to collate with PD incidents data*]
  - **Handling Write Queries:** Initially, all DML/DDL queries (e.g., `INSERT`, `CREATE`, `UPDATE`) will be heuristically routed to the **Large cluster** for safety and labeled as "heavy" by default. This is a critical safety measure to prevent data corruption, as write operations are not idempotent and cannot be safely duplicated during the shadow and canary rollout phases.

5.3. MODELING

- **Baseline:** A simple heuristic model (e.g., `IF sql_length > 2000 OR num_joins > 3 THEN heavy ELSE light`). This will be our performance floor.
- **Primary Model:** Gradient Boosted Decision Tree (GBDT). We will evaluate **XGBoost** and **LightGBM**. XGBoost is the initial choice due to its robustness. The model will be trained to optimize for **Recall** on the `heavy` class to minimize the False Negative Rate.
- **Serialization and Serving Format:** The choice of model format is critical for a low-latency, high-throughput serving environment.
  - **Decision: ONNX (Open Neural Network Exchange).** The trained XGBoost model will be converted to the ONNX format for serving.
    - **Portability:** ONNX is a universal standard that completely decouples the training environment (Python, Spark, XGBoost) from the serving environment (lightweight Python/FastAPI service). We can change training frameworks in the future without rewriting the serving code.
    - **Performance:** The ONNX Runtime is a high-performance C++ engine optimized for fast inference. It leverages hardware-level CPU features like **SIMD (Single Instruction, Multiple Data)** to perform mathematical operations in parallel, significantly reducing prediction latency.
  - **Alternatives Considered:**
    - **Spark MLlib Native Format:**
      - **Pros:** Natively integrated with the Spark training environment.
      - **Cons:** Unsuitable for serving. Loading a Spark model requires a running Spark session, which is a heavyweight dependency with high memory overhead and slow startup time, making it incompatible with our low-latency service requirements.
    - **MLeap:**

- **Pros:** A strong contender specifically designed to export Spark models for low-latency serving without a Spark context.
- **Cons:** MLeap is tightly coupled to the Spark/JVM ecosystem. ONNX was chosen for its broader, more universal industry support, offering greater flexibility and compatibility with a wider range of tools, hardware accelerators, and future serving architectures.

5.4. TRAINING-SERVING SKEW & DRIFT MONITORING

Training-serving skew is a critical production risk where inconsistencies between the training and serving environments cause model performance to degrade. This can happen in two ways:

- **Feature Skew:** The logic used to generate features in the real-time service differs slightly from the logic used in the offline training pipeline.
- **Drift:** The statistical properties of the live data change over time, making the original training data a poor representation of the current reality.
  - **Data Drift:** The inputs change (e.g., a new application starts sending structurally different queries).
  - **Concept Drift:** The meaning of the data changes (e.g., a query that was once "light" becomes "heavy" because its underlying table grew massively).

Mitigations: Our strategy to combat this is multi-layered:

- **Shared Featurization Library:** A single, versioned Python library will be used for both the offline Spark training job and the online FastAPI service to guarantee identical feature generation logic.
- **Schema Contracts:** We will use Pydantic to enforce a strict schema for the feature data, ensuring that the type and structure of data entering the model at serving time exactly matches what it was trained on. [**Medium term**]
- **Automated Drift Monitoring:** We will use **EvidentlyAI** (available at https://nexus-proxy.repo.local.sfdc.net/nexus/repository/pypi-all/simple/) to run a nightly job that compares the statistical distributions of the last 24 hours of production data against a reference (training) dataset. If significant drift is detected, it will trigger an alert, signaling that the model needs to be investigated or retrained. MLflow itself is used as a system of record, not a diagnostic tool, making a specialized library like Evidently a better fit for this active monitoring task. [**Long Term. To begin with we will use trigger criteria listed in 7.1**]

6. Serving on FKP & MLOps Plan

6.1. QUERYPREDICTOR SERVICE ON FKP

- **Framework:** Utilize current QP service that is already onboarded to FKP
- **Containerization:** Docker image with a minimal base, pinning all Python dependencies.
- **Deployment:** Blue-green deployments managed by **Argo CD**. The Kubernetes Service object will be switched to point to the new deployment's pods only after health checks pass.
- **Resource Sizing:**
  - We will fine tune the sizing during shadow dial up.

6.2. MLOPS STACK & PHASED ROLLOUT

Our choice of MLOps tooling is heavily influenced by the constraints of using existing platforms and minimizing costs. After evaluating the options, we have decided on a phased approach to de-risk the project timeline from internal approval processes.

TOOLING ANALYSIS:

	Tool	Pros	Cons	Decision for this Project
1	**Self-hosted MLflow**	• **Full Control & Low Cost:** No vendor lock-in or licensing fees. • **Industry Standard:** Portable, well-understood tool for tracking and registry. • **Compliant:** Avoids onboarding new AWS services and can be multi substrate if needed.	• **Operational Overhead:** Requires initial setup and ongoing maintenance (backups, upgrades).	**Chosen as the long-term solution (Phase 2).** Its robust tracking and governance capabilities are the right end-state. The operational cost is acceptable and can be mitigated with automation (Helm/ArgoCD).
2	**SageMaker**	• **Managed Environment:** Provides a managed Jupyter notebook for model authoring.	• **Incomplete Solution for Us:** Key features (Endpoints, Pipelines, Registry) are disabled by project constraints. **Limited to Authoring:** Cannot be used for the core MLOps tasks of tracking, registry, or serving.	**Use for authoring only.** We will leverage the managed notebook environment for development but not for the MLOps platform itself.
3	**BentoML**	• **Excellent for Serving:** Simplifies packaging models into efficient, containerized API endpoints.	• **Serving-focused:** Its experiment tracking and registry features are less mature than MLflow's. • **Misaligned Priority:** Our primary need is a strong tracking/registry foundation, not serving optimization.	**Not chosen for v1.** While a great tool, it doesn't solve our most immediate MLOps problem. It could be considered in the future for serving more complex models.

Phased MLOps Plan:
Phase 1: Initial Deployment (MLflow-less)

- **Artifact Store:** To maximize velocity, we will use an **existing, approved S3 bucket** for artifact storage.
- **Tracking & Registry:** Experiment tracking will be manual. The validated ONNX model file will be stored directly in a versioned path in S3 (e.g., `s3://our-bucket/smart-router/models/v1.2.0/model.onnx`).
- **Model Deployment:** The QueryPredictor service will be configured via a Kubernetes ConfigMap to load a specific model version by its S3 path. Promoting a model involves updating this ConfigMap and triggering a rolling deployment via Argo CD.

Phase 2: MLflow Integration (Post-Launch)

- **Upgrade Path:** Once the self-hosted MLflow instance is approved and deployed, the training pipeline and serving service will be upgraded.
- **Artifact Store:** We will migrate artifacts to a self-hosted server, which will be managed as part of the MLflow deployment on FKP, keeping the entire MLOps stack within the same platform.
- **Tracking & Registry:** All training runs will be logged to the **MLflow Tracking Server**. The model promotion lifecycle (Staging → Production) will be formally managed via the **MLflow Model Registry**.
- **Service Update:** The QueryPredictor service will be updated to fetch the latest model marked as `Production` from the registry at startup, automating the deployment process.

7. Lifecycle, Rollout, Safety & Cost

7.1. DATA & MODEL LIFECYCLE

PHASE 1 (MANUAL LIFECYCLE):

1. **Data Ingestion:** Trino query logs are available for training.
2. **Retraining Trigger:** Manual trigger based on performance degradation (Heavy Recall < 98%) or known data changes (such as 455 rollout or cluster resizing or instance type change such as graviton).
3. **Promotion:** The newly trained model (challenger) is evaluated against the current model (champion) on a hold-out test set. If it performs better, the new ONNX file is uploaded to S3 and its version is manually documented.
4. **Deployment:** The model path in the QueryPredictor's deployment configuration is manually updated, triggering a new

rollout.

Phase 2 (MLflow-driven Lifecycle):

1. **Retraining Trigger:** Automated triggers based on drift detection from Evidently or performance degradation alerts similar to phase 1.
2. **Promotion:** The challenger model is promoted from `Staging` to `Production` within the MLflow Model Registry UI.
3. **Deployment:** The router service automatically pulls the latest `Production` model on its next restart or via a refresh endpoint.

7.2. SAFE ROLLOUT PLAN

This is the most critical phase. We will proceed with extreme caution.

	Phase	Stage	Traffic Impact	What to Measure	Exit Criteria
1	1	**Offline Replay**	0%	Replay 2 weeks of query logs. Calculate misclassification cost (wasted CPU + SLA impact).	**Heavy Recall ≥ 98%**, FNR ≤ 1%.
2	2	**Shadow Predict**	100% (predict only)	QP receives live traffic but TGW doesn't route. Log predicted class vs. actual class (from logs).	Router p99 latency **< 500ms**. Model accuracy holds. No QP crashes for 48h. **Featurization consistency validated:** <1% deviation between offline and online feature distributions.
3	3	**Provision Small Cluster**	--	Provision a small cluster for eventual routing.	Small cluster is live and can take traffic from TGW.
4	4	**Canary Route**	2% -> 10% -> 50% → 100%	Route a percentage of live traffic (reads + writes) to small cluster . Monitor router metrics, cluster health, and user-facing SLAs.	**No P99 latency regression > 5%**. Error rates stable. FNR < 1%. Hold at each stage for 24h. **Automatic rollback** on critical alerts.
5	5	**Right-size & Cleanup**	100%	After 100% of traffic is on Smart Routing for 1 week, begin decommissioning nodes from the old symmetric cluster.	All business and system metrics remain green. **Cost savings > 25%** are realized.

Risk: Cost Constraints on Provisioning a New Small Cluster

Phase 3 (provisioning small cluster) needs to be ready before canary routing. Here are two options to provision small cluster from cost constraints perspective.

Option 1: Downsize Existing Cluster & Re-provision (Cost Neutral)
Provision a new 24 node small cluster by down sizing existing non-curated cluster. Take 8 nodes (6%) off each of the three backends with 135 nodes and create a small cluster.

- **Pros:**
  - **No Additional Budget:** Avoids the need for new funding by using already allocated capacity.
- **Cons:**
  - **High Operational Complexity:** Forces the team to manage two changing variables at once: the canary rollout and the shrinking capacity of the production cluster. Requires tuning Trino resource group limits on the current production cluster as it shrinks and also the small cluster during the dial up.
  - **Distraction:** Diverts focus from the primary goal of validating+tuning the ML routing logic and the small cluster setup in Phase 4 by merging Phases 3, 4 and 5 into one complicated step.

Option 2: Provision New Cluster with Additional Budget (Recommended)
Provisioning a new 24-node small cluster for the 4-6 week rollout period would incur a temporary additional cost of ~$33K.

- **Pros:**
  - **Simplicity & Isolation:** The current cluster remains stable and untouched. The team can focus exclusively on

validating the new small cluster and routing logic, making troubleshooting far easier. This is the safest and cleanest engineering approach. Down sizing the large cluster (Phase 5) can be done after small cluster is fully dialed up and is stable.

- **Cons:**
  - **Requires Temporary Budget**: The primary drawback is the need for **an additional ~$33K budget** (24 nodes * $229 weekly cost per node * 6 weeks in the worse case) for the duration of the rollout.

7.3. COST MODEL

- **Current Cost:** 405 nodes * `C` (cost per c6a.32xl node) = **405C**
- **Proposed Cost:**
  - Large Cluster: 270 nodes * `C` = 270C
  - Small Cluster: 24 nodes * `C` = 24C
    - small queries currently use a peak of 2500 cores out of 45K available cores. With 105 out of 128 vCPUs available for Trino. 2500 cores approximate to 23 (or 8 each over 3 backends)
  - Router + MLOps on FKP: ~5C (estimate for pods/storage)
  - **Total:** 270C + 25C + 5C = **300C**
- **Estimated Savings:** `(1 - 300 / 405) * 100` ≈ **25.9%**.
  - **Assumption:** This meets the >25% goal, with potential for further optimization by tuning the Large cluster size down.
- A c6i.32xlarge instance costs $229.46 weekly (both on demand and reserved according to Amogh Garg from capacity team).

8. Observability, Security & Compliance

8.1. METRICS & DASHBOARDS

A dedicated Grafana dashboard will be created to monitor the entire system:

- **Router Service:** QPS, p50/p95/p99 latency, error rate, fallback rate, CPU/memory usage per pod.
- **Model Performance:** Predictions per second, confidence score distribution, **live FNR/FPR** (calculated by joining router logs with query completion logs).
- **Business KPIs:** P90/P99 latency for Light vs. Heavy queries (compared to control), CTS savings.

8.2. SECURITY & COMPLIANCE

- **Network:** Kubernetes `NetworkPolicy` will restrict ingress to the router service to only the Trino Gateway pods. Egress will be restricted to the Trino clusters.
- **Secrets:** All secrets (DB passwords for MLflow, etc.) will be stored in **Vault** and injected into pods.
- **Data Handling:**
  - The QueryPredictor will only log `queryId` and metadata, not the full `sqlText`, to minimize PII exposure.
  - Data in the MLflow artifact store will be encrypted at rest.
  - Data retention policies will be applied to logs and artifacts.

9. Risks, Open Questions & Project Plan

9.1. RISKS & MITIGATIONS

	Risk	Likelihood	Impact	Mitigation
1	**Read/Write confusion in Shadow Mode**	Low	Critical	**Mitigation:** Develop a robust SQL parser (e.g., using an existing Java library) to classify queries as `SELECT` vs. `INSERT`/`CREATE`/etc. Only mirror `SELECT` queries during the Shadow Route phase. All DML/DDL defaults to the Large cluster.
2	**Heavy query misclassified as Light (FNR)**	Medium	High	**Primary Mitigation:** Tune model for high recall. **Secondary:** Set strict resource group limits on the Small cluster to kill runaway queries. **Tertiary:** Implement a retry mechanism where a query killed on the Small cluster is automatically retried on the Large cluster.
3	**Lack of Gateway Fanout for Shadowing**	High	Medium	**Mitigation:** Implement async mirroring with extensive testing and safety features (feature flags, thread pool isolation).
4	**Feature/Concept Drift**	Medium	High	**Mitigation:** Implement automated drift detection with **EvidentlyAI**. Set up alerts for significant drift and have a clear process for triggering model retraining.
5	**Adversarial Queries**	Low	Medium	**Mitigation:** A user could craft a query that looks simple but explodes in complexity. The resource group limits on the Small cluster are the primary defense here.

9.2. OPEN QUESTIONS

	Question	Hypothesis / Assumption
1	How will this interact with existing Resource Groups?	The router is a coarse-grained layer; RGs are a fine-grained enforcement layer. They are complementary.
2	Can we get features from the Trino planner in time?	**Assumption: No.** Calling the planner for features would likely violate the 500ms latency budget. We will rely on static SQL analysis.
3	How to handle hot-key users or tenants?	A single user submitting many heavy queries could still be an issue.
4	Is there significant seasonality in query patterns?	**Assumption: Yes.** ETL jobs run at night, analytics during the day. Time-based features should capture this.

9.3. PROJECT PLAN (HIGH-LEVEL)

- **Phase 1: Foundation & Offline Analysis (TK weeks)**
  - **Milestone:** Develop v1 featurization library and train baseline XGBoost model.
  - **Milestone:** Store validated model artifact in S3.
  - **Exit Criteria:** Offline replay shows model meets >98% recall target.
- **Phase 2: Serving & Shadowing (TK weeks)**
  - **Milestone:** Build QP service (loading model from S3) and Gateway async mirroring (for shadow predictions).
  - **Milestone:** Deploy QP service and run in Shadow Predict mode.
  - **Milestone:** Provision Small cluster and run in Shadow Route mode.
  - **Exit Criteria:** System is stable in shadow mode for 1 week with no SLA regressions.
- **Phase 3: Canary Rollout & Go-Live (TK weeks)**
  - **Milestone:** Execute phased canary rollout to 100%.
  - **Milestone:** Right-size the final clusters.
  - **Exit Criteria:** Smart Routing is live for 100% of traffic for 1 week with improved SLAs and cost savings.
- **Phase 4: MLflow Integration & Handoff (TK weeks)**
  - **Milestone:** Set up self-hosted MLflow on FKP.

    - **Milestone:** Integrate training pipeline and serving service with MLflow for tracking and registry.
    - **Milestone:** Hand off operations to the BDI team with runbooks.

What's Next Checklist:

- [ ] Finalize resource sizing for the Small/Large clusters based on offline analysis.
- [ ] Begin development of the Gateway async mirroring prototype.
- [ ] Submit MLflow and other libraries for internal open-source review.
- [ ] Schedule a review of this design with the Trino and Security teams.

Appendix

QUESTIONS:

- Tune resource group thresholds (during? and) after canary routing (as the number of instances in the large cluster is downscaled).
    - RGs tied to worker counts.
    - **Question: Different clusters with different limits. Explainability to users.**
        -
    - Question: Downsizing large cluster would mean moving from 3 AZs to 2 AZs to maintain the node count in a AZ.
        - **Define fallback to large cluster in resource groups
    - Question: Retain 32xl for small cluster too.
    - While making changes to TGW, honor the session properties.
    - What is the work involved if we change the threshold for small vs large?

```
@startuml
!theme vibrant

actor "Trino Client" as Client

package "Trino Gateway" {
  component "Gateway Service" as Gateway
}

package "FKP (Salesforce Kubernetes)" {
  component "QueryPredictor Service\n(Python/FastAPI)" as Predictor

  package "MLOps Stack" {
    database "MLflow Tracking Server\n(Postgres Backend)" as MLflow
    database "Artifact Store\n(S3)" as ArtifactStore
    component "Grafana" as Monitoring
  }
}

package "Trino Clusters" {
    database "Small Cluster\n(Light Queries)" as SmallCluster
```

```
    database "Large Cluster\n(Heavy Queries)" as LargeCluster
}

package "Query Logs" {
    database "Query Logs" as Logs
}

package "Data & Training" {
    component "Spark Training Job" as SparkJob
    component "SageMaker Notebook\nfor Authoring" as SageMaker
}


' --- Connections ---
Client --> Gateway

' Online Flow - Use directional arrows to avoid overlap
Gateway -down-> Predictor : 1. Query Features (180 RPS Peak)
Predictor -up-> Gateway : 2. Routing Decision (e.g., "light")

Gateway --> SmallCluster : 3a. Route Light
Gateway --> LargeCluster : 3b. Route Heavy / Fallback

' MLOps & Monitoring
Predictor ..> ArtifactStore : Loads model.onnx (v1)
Predictor ..> MLflow : Fetches latest model (v2)
Predictor ..> Monitoring : Emits metrics

' Offline Training Flow
Gateway -> Logs : Publishes query events
Logs --> SparkJob
SageMaker --> SparkJob : Develop Model
SparkJob --> ArtifactStore : Train & Export Model
ArtifactStore --> MLflow

@enduml
```