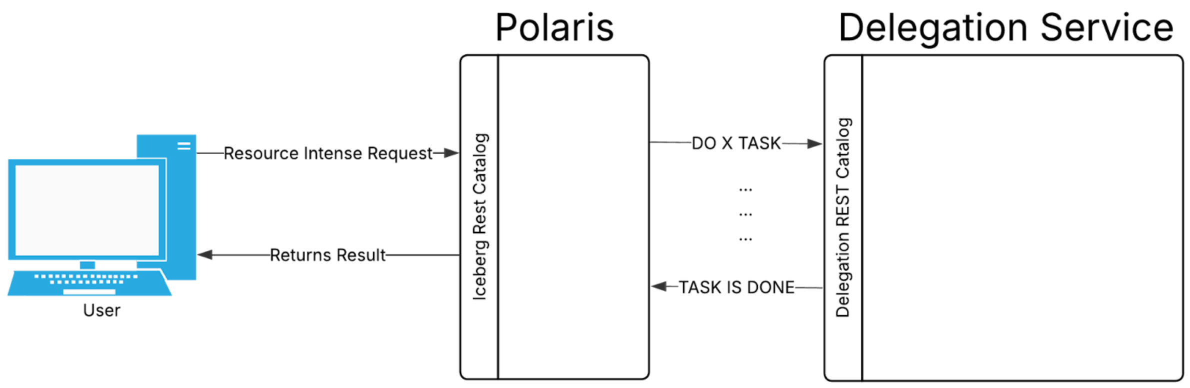
Polaris Delegation Service

1. Introduction & Motivation

The Polaris Catalog is optimized for low-latency metadata operations, but certain resource-intensive tasks, such as purging data files for dropped tables, can impact its core performance. The workload for these long-running, I/O-heavy tasks also scales differently from the primary metadata workload. The core motivation is to decouple these two concerns, ensuring the Polaris Catalog remains highly responsive for metadata requests while allowing heavy background tasks to be managed and scaled independently.

2. Proposal



We propose the introduction of the **Polaris Delegation Service**, an optional and independent component responsible for executing long-running background operations offloaded from the main catalog. The initial implementation will focus on handling the data file deletion process for DROP TABLE WITH PURGE commands, which will be delegated synchronously from the Polaris Catalog. This service will be built on a reliable task framework, establishing a pattern that is extensible for future delegated and scheduled operations such as data compaction and snapshot garbage collection.

3. Goals

- Decouple long-running tasks from the main Polaris catalog service to maintain its low-latency performance for metadata operations.
- Allow independent scaling of the task execution workload (Delegation Service) and the metadata workload (Polaris Catalog).
- Provide a resilient mechanism for executing delegated tasks, with recovery capabilities.
- Establish a secure communication and trust model between the Polaris Catalog and the Delegation Service.
- Initially support DROP WITH PURGE operations, with a design extendible to other delegatable tasks (e.g. format conversion, compaction, snapshot expiry actions beyond metadata changes).

4. Non-Goals

- The Delegation Service will not handle core metadata CRUD operations. These remain the sole responsibility of the Polaris REST Catalog.
- The Delegation Service will not resolve or execute user queries.

5. Polaris Changes

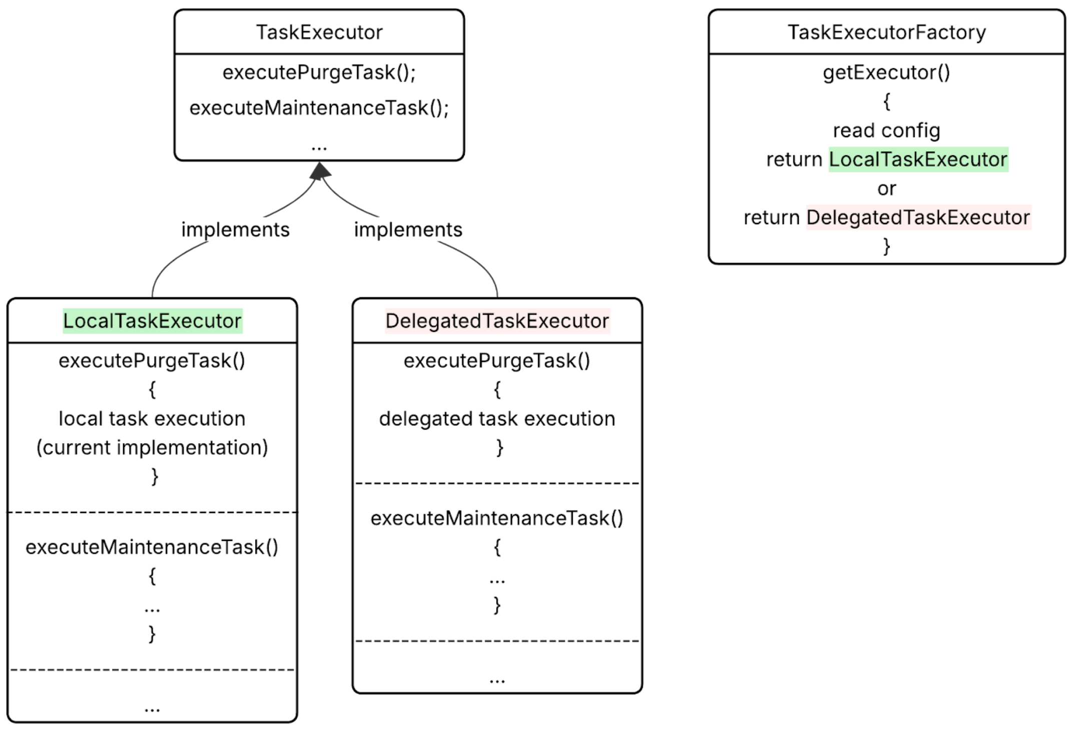
Config changes:

- New configs in `applications.properties`

```
None
polaris.delegation-service.task-type=delegated
polaris.delegation-service.endpoint=http://delegation-service:818
2
```

Code Refactors:

We will use a Factory and Strategy design pattern to provide the correct task execution method depending on the config setup above.



- `TaskExecutor`: interface that has a method for each type of task (e.g. `executePurgeTask`).
    a. `LocalTaskExecutor` and `DelegatedTaskExecutor` implements `TaskExecutor` with their own methods of task execution.
- `TaskExecutorFactory` reads the configuration from `application.properties` and provides the correct executor instance.
- `TaskExecutorFactory` is injected into the metastore manager

Following the proposed architecture, a submitted DROP TABLE PURGE task will follow the path outlined below:

1: Request reception (API call)

2: Asks a metastore manager to drop the table (`dropEntityIfExists()`)

3: Metastore manager asks injected `TaskExecutorFactory` to provide the correct task executor per the setup configuration

4. Actual execution of the task

5. Result is returned to the user

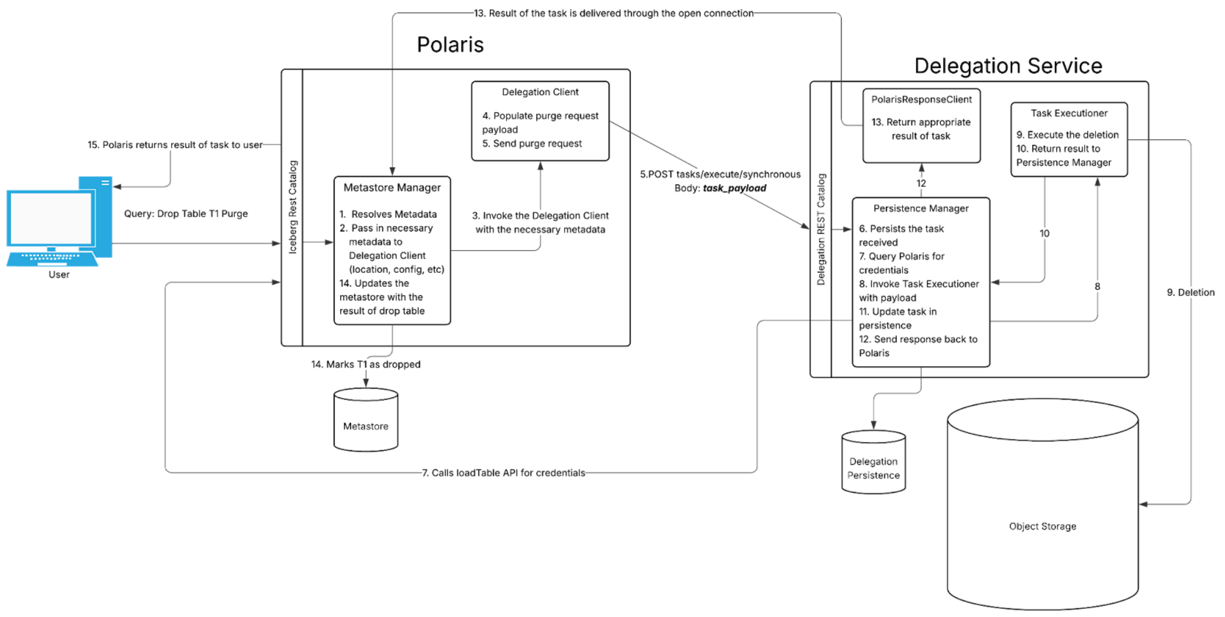
6. Delegation Service REST API spec:

Task Execution Endpoint (Synchronous)

This endpoint is the main entry point for delegating a task where the Polaris Catalog requires immediate & blocking execution and a final result.
We are choosing a one-to-one Polaris Realm to Delegation Service model and using synchronous task execution for the MVP.
To read more about this decision, see: 9. Design Discussion.

Endpoint:

- POST /tasks/execute/synchronous


13. Result of the task is delivered through the open connection
Polaris
Delegation Service
Delegation Client
4. Populate purge request payload
5. Send purge request
15. Polaris returns result of task to user
Query: Drop Table T1 Purge
User
Iceberg Rest Catalog
Metastore Manager
1. Resolves Metadata
2. Pass in necessary metadata to Delegation Client (location, config, etc)
14. Updates the metastore with the result of drop table
3. Invoke the Delegation Client with the necessary metadata
5.POST tasks/execute/synchronous
Body: task_payload
Delegation REST Catalog
PolarisResponseClient
13. Return appropriate result of task
Task Executioner
9. Execute the deletion
10. Return result to Persistence Manager
12
Persistence Manager
6. Persists the task received
7. Query Polaris for credentials
8. Invoke Task Executioner with payload
11. Update task in persistence
12. Send response back to Polaris
10
8
9. Deletion
14. Marks T1 as dropped
Metastore
7. Calls loadTable API for credentials
Delegation Persistence
Object Storage

7. Persistence

Delegation Service is optional, therefore we will be leaving the Polaris persistence unchanged and instead implement a separate, unique task schema within the Delegation Service as the source of truth for delegated tasks.

Delegation Service Task Schema:

Field Name	**Data Type**	**Description**
task_id	UUID (PK)	Unique identifier for the task within the Delegation Service.
version	Integer (PK)	Version number for optimistic concurrency control on this task record within the Delegation Service's persistence.
parent_id	UUID	Point to where this sub-task came from.
task_type	Enum	Defines the nature of the work (e.g., DATA_FILE_PURGE, TABLE_COMPACTION).
task_payload	String (JSON)	The actual data required to execute the task (see below)
status	Enum	Current state of the task within the Delegation Service (see below).
received_timestamp	Timestamp	Timestamp when the Delegation Service ingested/received this task.
last_status_change_ts	Timestamp	Timestamp of the last status update to this task record in the Delegation Service's persistence.
attempt_count	Integer	Number of times the Delegation Service has attempted to execute this specific task.
created_ts	Timestamp	Timestamp when the task was first persisted.
lease_acquired_ts	Timestamp	Timestamp when the current attempt was assigned.
result_summary	String	(Optional) A summary of the last encountered success, error or terminal failure reason (runtime, etc).
result_extended	String	(Optional) More detailed log or stack trace from the last failure, or a history of critical errors.

Status Enum:

Status ENUM Value	**Definition**	**Can Transition To**
SUBMITTED	The task has been successfully submitted to and persisted by the Delegation Service but is not yet picked up for active processing by a worker. It's awaiting resource availability or scheduling.	ACQUIRING_RESOURCES, RUNNING
ACQUIRING_RESOURCES	A worker has picked up the task, but it's in a preparatory phase, primarily waiting for external dependencies like credentials from the Polaris Catalog before the main work (e.g. data deletion) can begin.	RUNNING, RETRY_SCHEDULED (if resource acquisition fails transiently), FAILURE
RUNNING	The task is actively being executed by a worker. The worker maintains a lease on this task.	SUCCESS, RETRY_SCHEDULED, FAILURE

RETRY_SCHEDULED	The task encountered a transient error during ACQUIRING_RESOURCES or RUNNING and has been scheduled for a future retry attempt.	ACQUIRING_RESOURCES or RUNNING (when the retry attempt begins).
SUCCESS	The task has finished all its intended work successfully. Any results or summaries are recorded.	Terminal (or SUBMITTED if scheduled task).
FAILURE	The task has failed permanently after exhausting all configured retries or encountering a non-retryable error. No further automatic attempts will be made. Error information is recorded.	Terminal

Global task_payload:

Field Path	**Data Type**	**Description**
operation_type	String	Specifies the action to be performed by the Delegation Service.
created_ts	Timestamp	Timestamp when the task was persisted in Delegation Service..
realm_id	String	The identifier of the Polaris realm that this task originated from

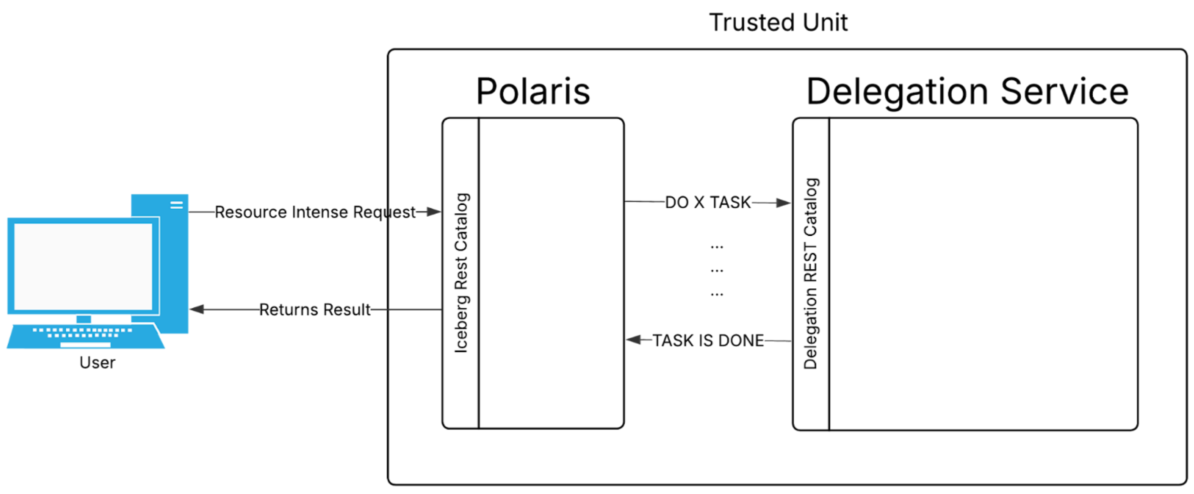
PURGE_TABLE specific task_payload:

table_identity	**Object**	**Description**
table_identity.catalog_name	String	The name of the Polaris Catalog the table belonged to.
table_identity.namespace_levels	Array of Strings	An ordered list of namespace levels leading to the table.
table_identity.table_name	String	The name of the table.
properties	Object (Map)	A catch-all for key-value pairs providing specific instructions for the purge operation itself. This is kept separate from config to distinguish operational parameters from I/O/credential configuration.

Full Example of PURGE_TABLE task_payload:

```JSON
{
  "common_payload": {
    "operation_type": "TABLE_PURGE",
    "request_timestamp_utc": "2025-06-11T16:21:00Z",
    "realm_identifier": "prod_us_west",
    "polaris_correlation_id": "drop-t1-request-uuid-1234"
  },
  "operation_parameters": {
    "table_identity": {
      "table_entity_id": "polaris-entity-id-for-T1",
      "catalog_name": "prod_catalog",
      "namespace_levels": ["NS1", "NESTED1"],
      "table_name": "T1"
    },
    "properties": {
      "skipTrash": "true",
      "deleteMarkerFiles": "false"
    }
  }
}
```

8. Security Model and Trust Architecture



The Delegation Service is designed as a private, internal backend component, inaccessible directly by end-users. Polaris should act as the sole, trusted gateway for all delegated tasks, ensuring every operation is subject to its centralized governance and security controls. To establish a secure and exclusive one-to-one relationship between a Polaris realm and its dedicated Delegation Service instance, the foundational security model will be the following:

8.1 Network-Level Isolation (Firewalls / Security Groups)

This is the first and most basic layer of defense, restricting which machines can even attempt to communicate with the Delegation Service through network configurations.

- Use network firewall rules or cloud security groups (e.g. AWS Security Groups or GCP Firewall Rules) to control traffic to the Delegation Service's exposed port.
- The rule would state: "Only allow incoming traffic on the Delegation Service port from the specific IP address of the Polaris instance." All other traffic from the internet or even from other machines within the private network would be blocked.

This guarantees that only a Polaris instance can reach the Delegation Service's port, but it doesn't verify the identity of the application making the call.

8.2 Transport-Level Authentication (Mutual TLS)

This is the layer for guaranteeing the identity of both services, creating the security binary discussed above.

- A private, trusted Certificate Authority (CA) is created for the Polaris-Delegation Service security binary.
- Both the Polaris instance and Delegation Service instance are issued a unique client TLS certificate signed by this CA.
- Both services are configured to require mutual authentication. When Polaris calls the Delegation Service, it presents its certificate. The Delegation Service validates it against the CA and vice-versa.

This approach provides strong, cryptographic proof that the service on the other end of the connection is the legitimate, expected instance of the paired counterpart. Even if an unauthorized service managed to get through the firewall, it could not establish a connection because it wouldn't have the required TLS certificate. This enforces the exclusive one-to-one trust, effectively preventing network-level impersonation and establishing a secure, encrypted communication channel for all traffic between them.

9. Design Discussion

9.1 Delegation Mode

Approach	**How it Works**	**Pros**	**Cons**
Synchronous (Preferred)	Polaris calls the Delegation Service and holds the connection open, waiting for a final success or failure response before it completes its own operation and responds to the user.	• Unambiguous State • Immediate Error Reporting • Simple Catalog Logic	• API calls are blocked for the entire duration of the purge • Temporary Tokens can expire and lead to incomplete purge
Fire-and-Forget (current task execution)	Polaris sends a task to the Delegation Service and immediately drops the table and returns success to the user.	• Fast User Response • Simple Catalog Logic	• No Guarantee of Execution • No Error Reporting • Orphaned Data
Asynchronous w/ Polaris Polling	Polaris submits the task to the Delegation Service, which immediately accepts it and returns a task ID. The status of the long-running task can be checked later by polling an API endpoint with the task ID.	• Fully Decoupled • High Reliability (task is persisted in Polaris) • Observable	• Complex to implement, requiring spec changes within Polaris
Asynchronous w/ DS pushing	Polaris submits a task to the Delegation Service, which immediately accepts it and returns an acknowledgement. When the task eventually completes, the Delegation Service sends the final status back to a pre-agreed callback URL.	• Fully Decoupled • Efficient: No polling overhead • High Reliability	• Complex to implement, requiring Polaris spec changes as well as exposing a secure, reliable, and publicly addressable endpoint for callbacks

Due to the limited time allocated for this project, we will try to avoid any Polaris spec changes to limit the number of blockers. Leading us to choose the synchronous delegation model. However, the fire-and-forget approach can also be easily included within MVP to match the current task execution behavior.

The future plan/final goal of the Delegation Service is to be fully asynchronous and independent.

9.2 Credential Vending

Credential Vending Approach	**How It Works**	**Pros**	**Cons**
Delegation Service Has Its Own Principal	The Delegation Service is a registered PrincipalEntity in Polaris. Polaris sends table identity with only the context of the table to purge. The Delegation Service then authenticates as itself, calls a dedicated	• Centralized Governance: All access policies and credential logic remain within Polaris.	• A new secured internal API endpoint must be created in Polaris. • Token Lifetime Risk: The vended token's lifetime must

	Polaris API to request credentials for that specific context, and Polaris vends them after authorizing the request.		be long enough to cover the entire purge operation.
Delegation Service Has Its Own Principal & Uses loadTable API (Preferred)	The Delegation Service is a registered PrincipalEntity. It receives the table identity and makes a standard loadTable call to the public Polaris REST API. The LoadTableResult response from Polaris contains the necessary vended credentials for the DS to use.	• Reuses Existing API: No new internal API needs to be created on the Polaris Catalog. • Centralized Governance	• Token Lifetime Risk
Catalog Pre-Vends Credentials	Polaris calls its internal PolarisCredentialVendor then includes the temporary, scoped credentials in the ***task_payload*** sent to the Delegation Service.	• Simpler Delegation Service logic: No need for a call back to fetch credentials. • Fewer network trips	• Token Lifetime Risk • Credentials will get persisted according to our task persistence schema
Delegation Service Has Its Own Object Storage Credentials	The Delegation Service is configured with its own long-lived cloud credentials. It completely bypasses Polaris for storage access and uses these static credentials to delete data directly from the storage system.	• Simple with no callbacks to Polaris for credential vending. • Fully Decoupled for storage access, the DS doesn't depend on the Polaris credential vending mechanism being available.	• Insecure as the Delegation Service becomes a major security liability with persisting powerful, long-lived credentials. • All storage access control policies defined in Polaris are ignored.

We will choose the loadTable approach as it avoids a Polaris spec change and persisting credentials.

9.3 Polaris to Delegation Service Relationship

Aspect	**Shared Delegation Service (Many-to-One)**	**Dedicated Delegation Service (One-to-One) (Preferred)**
Security & Isolation	**Complex.** The Delegation Service must be a "super-principal" with permissions to operate across all realms it serves.	**Simple & Strong.** The Delegation Service is a standard principal within its single, dedicated realm. Its permissions are naturally scoped by Polaris's existing RBAC model. No risk of cross-realm credential requests or data access.
Authorization Logic	**Complex.** Every callback from the DS to Polaris (e.g. for loadTable or credential vending) requires the DS to dynamically select the correct principal identity for the specific realm of the task it is processing.	**Simple.** The DS has only one identity—its own principal within its paired realm. All callbacks to Polaris are made with this single, consistent identity, and authorization is handled by standard Polaris mechanisms.
Configuration	**Complex.** The DS must be configured with realm-specific principals for every Polaris realm it serves.	**Simple.** Each Polaris realm configuration points to its dedicated DS endpoint. Each DS is configured with the client_id/secret for its single principal and its allowed realm_identifier.
Resource Utilization / Cost	**More Efficient.** A single, shared DS instance can handle tasks from multiple realms, potentially reducing idle resource costs compared to running many dedicated, under-utilized instances.	**More Predictable.** Resources are scaled based on the workload of a single realm. While this might lead to higher costs if many realms have low traffic, it prevents a "noisy neighbor" problem where a busy realm monopolizes the shared DS resources, starving other realms.
Blast Radius	**Larger Blast Radius.** If the shared Delegation Service instance fails or has a performance degradation, it impacts task execution for all Polaris realms	**Smaller Blast Radius.** If a dedicated Delegation Service for a specific realm fails, only that single realm's background task processing is affected, isolating the failure.

	that rely on it.	

Although there are many benefits for the user when exploring the many-to-one relationship, for sake of simplicity and security, we will choose the one-to-one relationship between a Polaris realm and Delegation Service.

9.4 Order Of Deletion: Metadata & Object Storage

A PURGE operation involves two distinct systems: the Polaris metastore and the underlying object storage. As a distributed transaction across these two systems is not atomic, the order of operations must be carefully considered. This section analyzes the trade-offs between the two possible sequences to determine the most consistent and reliable approach.

Ultimately, since a true atomic operation is not possible, we must pick our battles and choose the failure mode that is safer and more recoverable.

Approach	**How it works**	**Mid-purge Failure Scenario**
1. Purge First Drop Metadata Second **(Preferred)**	Polaris calls the Delegation Service and waits for a success confirmation. Only after receiving this success signal, Polaris drops the table within its persistence.	• Metadata drop transaction is aborted. The table remains visible and fully defined in the catalog when it is a broken table in object storage. • DS can ask for credentials again and retry the purge.
2. Drop Metadata First Purge Second	Polaris immediately drops the table from its persistence. After this commit, it calls the Delegation Service to perform the data purge.	• The table is not visible once the task is received by Polaris. • There will be orphaned files. • DS will not be able to ask for credentials to retry the purge.

10. Future Work

10.1 Asynchronous Task Submission & Status Endpoints

To allow for true decoupling of long-running tasks, the API could be extended with the following asynchronous endpoints.

Task Submission

Endpoint:

- POST /tasks

Description:

- Submits a task for asynchronous background processing. The Delegation Service immediately accepts the task, persists it with a QUEUED status, and returns a response.

Request Body:

- The ***task_payload***

Success Response (202 Accepted):

- Returned when the task has been accepted successfully.
- Body:

```json
{
      "delegation_task_id": "ds-async-task-abc-123",
      "status": "QUEUED",
      "status_query_url":
"/delegation-service/api/v1/tasks/ds-async-task-abc-123"
}
```

Task Status Inquiry

Endpoint:

- GET /tasks/{taskId}

Description:

- Retrieves the full current state of a previously submitted asynchronous task from the Delegation Service's persistence layer.

Success Response (200 OK):

- The response body would be the full task record from the DS persistence, including status, attempt count, timestamps, error info, etc.

Error Response (404 Not Found):

- Returned if a task with the specified taskId does not exist.

10.2 Asynchronous Task Framework

This proposal and the [Polaris Asynchronous & Reliable Tasks]() proposal both address the need for handling background work, at different architectural layers and with different scopes.

Whereas the Delegation Service is a simple worker, the Asynchronous Task Framework is for scheduling, executing, and tracking a wide variety of tasks within Polaris itself. Key features being the following:

- Unique “task behaviors”
- Unified submit() API
- Handling retries and restarts of lost tasks
- Using a shared persisted state for coordination

The primary difference is one of scope and abstraction. The Asynchronous Task Framework is a broad, internal platform change for managing any task, while the Delegation Service is a specific, external worker designed to execute a particular class of task. Therefore, the two proposals are highly synergistic. As the Delegation Service does not alter the task schema of Polaris, it can be a synergistic consumer of tasks managed by the more advanced Asynchronous Task Framework.

For example, here's how they would work hand-in-hand for a fully asynchronous DROP TABLE ... PURGE operation:

1. **Task Submission** (via Asynchronous Task Framework):
    - A user's DROP TABLE ... PURGE request would trigger a call to the Tasks.submit() function
    - A TaskBehavior specifically for "delegated purge" would be submitted. This behavior would define the task as retryable and would generate a unique, persisted task record in the Polaris metastore.
2. **Scheduling & Handoff:**
    - The Asynchronous Task Framework's scheduler would see this new CREATED task.
    - The TaskFunction associated with the "delegated purge" behavior would call the Delegation Service's API, passing the necessary task_payload.
    - The framework would manage the reliability of this handoff, retrying the API call to the Delegation Service if it fails, leveraging the persisted task state.
3. **Execution:**
    - The Delegation Service receives the task, executes the data purge, and completes its work, returning the result back to the Async Task Framework to be updated within the Polaris persistence.

In this model, the Asynchronous Task Framework acts as the robust, reliable "frontend" and scheduler, while the Delegation Service acts as the specialized "backend" worker. This allows for the leverage of the advanced features of the task framework without having to build that complex logic into the Delegation Service itself.

11. Open Questions & Notes

- Tombstoning in Polaris (unqueryable but supporting credential vending)
-

APPENDIX:

Delegation Service OpenAPI REST spec:

```
None
---
openapi: 3.1.1
info:
  title: Apache Polaris Delegation Service API
  license:
    name: Apache 2.0
    url: https://www.apache.org/licenses/LICENSE-2.0.html
  version: 0.1.0
  description: >
    Defines the REST API for a Delegation Service designed to handle
long-running or
    I/O-intensive tasks offloaded from a Polaris Catalog instance. This
specification
    covers the synchronous execution of tasks, where the client (Polaris
Catalog)
    waits for the task to complete.
```

```
servers:
  - url: "https://{host}:{port}/delegation-service/api"
    description: Generic base server URL for the Delegation Service.
    variables:
      host:
        description: The host address for the Delegation Service.
        default: localhost
      port:
        description: The port for the Delegation Service.
        default: "8182"

security:
  - OAuth2: [execute_tasks]
  - BearerAuth: []

paths:
  /v1/tasks/execute/synchronous:
    post:
      tags:
        - Task Execution API
      summary: Submits a task for immediate, synchronous execution
      description: >
        Submits a task for immediate execution. The Delegation Service will
perform the
        entire task and only return a response once the task has reached a
terminal state
        (COMPLETED_SUCCESS or FAILED_TERMINAL). The HTTP connection from the
client
        (Polaris Catalog) is held open during this time.
      operationId: executeSynchronousTask
      requestBody:
        description: The full task payload defining the operation to execute.
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/TaskPayload'
      responses:
        '200':
          description: The synchronous task was executed successfully.
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/TaskSuccessResponse'
        '400':
          $ref: '#/components/responses/BadRequest'
        '401':
          $ref: '#/components/responses/Unauthorized'
        '403':
          $ref: '#/components/responses/Forbidden'
        '409':
          $ref: '#/components/responses/Conflict'
        '500':
          $ref: '#/components/responses/InternalServerError'
        '502':
          $ref: '#/components/responses/BadGateway'

components:
  securitySchemes:
    OAuth2:
      type: oauth2
      description: >
        The Delegation Service is secured via OAuth2. The client (Polaris
Catalog)
        must first obtain an access token from an authorization server using
its
        client_id and secret, then use that token for requests.
      flows:
        clientCredentials:
          tokenUrl: http://polaris-catalog-auth-server/v1/oauth/tokens # This
URL points to the authorization server
          scopes:
            execute_tasks: Allows submitting tasks to the Delegation Service.
    BearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT

  schemas:
    # DTO for the main request body
    TaskPayload:
      type: object
      description: The complete payload for a delegated task.
      required:
        - common_payload
        - operation_parameters
      properties:
        common_payload:
          $ref: '#/components/schemas/CommonPayload'
        operation_parameters:
          $ref: '#/components/schemas/PurgeTableOperationParameters'

    # Common/Global part of the payload
    CommonPayload:
      type: object
      description: Contains metadata common to all delegated tasks.
      required:
        - operation_type
        - realm_identifier
      properties:
        operation_type:
          type: string
          description: Defines the operation to be performed and the schema of
`operation_parameters`.
          enum:
            - SYNCHRONOUS_TABLE_PURGE
        request_timestamp_utc:
          type: string
          format: date-time
          description: Timestamp (UTC) when the Polaris Catalog initiated the
delegation request.
        realm_identifier:
          type: string
          description: The Polaris realm this task belongs to. Essential for
all callbacks.
        polaris_correlation_id:
          type: string
          description: A unique ID for tracing this request end-to-end.

    # Operation-specific parameters for a PURGE task, modeled after Iceberg's
LoadTableResult
    PurgeTableOperationParameters:
      type: object
      description: Parameters specific to a SYNCHRONOUS_TABLE_PURGE operation,
modeled after the Iceberg LoadTableResult schema.
      required:
        - table_identity
        - metadata_location
        - metadata
        - config
      properties:
        table_identity:
          $ref: '#/components/schemas/TableIdentity'
        metadata_location:
          type: string
          format: uri
          description: The URI to the table's latest Iceberg metadata file.
        metadata:
          $ref: '#/components/schemas/TableMetadata'
        config:
          type: object
          description: >
            A map containing the temporary, scoped credentials vended by the
Polaris Catalog.
            Keys are standard Iceberg credential properties (e.g.,
's3.access-key-id').
          additionalProperties:
            type: string
        properties:
          $ref: '#/components/schemas/PropertiesMap'

    TableIdentity:
      type: object
      description: Uniquely identifies the table to be purged.
      required:
        - table_entity_id
        - catalog_name
        - namespace_levels
        - table_name
      properties:
        table_entity_id:
          type: string
          description: The Polaris Entity ID of the dropped table.
        catalog_name:
          type: string
        namespace_levels:
          type: array
          items:
            type: string
        table_name:
          type: string

    TableMetadata:
      type: object
      description: A representation of the Iceberg TableMetadata JSON object,
containing info for the purge.
      required:
        - location
        - table-uuid
        - format-version
      properties:
        location:
          type: string
          format: uri
          description: The root location of the table. This path is the target
for recursive deletion during a purge.
        table-uuid:
          type: string
          format: uuid
        format-version:
          type: integer
        # Other relevant fields from the full Iceberg spec like schemas,
snapshots, etc.,
        # can be included here if the DS needs them for more advanced logic.

    PropertiesMap:
      type: object
      description: A generic key-value map for operation-specific parameters
(corresponds to cleanupProperties).
      additionalProperties:
        type: string

    # --- Response Schemas ---
    TaskSuccessResponse:
      type: object
      description: The response body for a successfully executed synchronous
task.
      required:
        - delegation_task_id
        - status
      properties:
        delegation_task_id:
          type: string
          format: uuid
          description: The unique ID assigned to the task by the Delegation
Service for its internal tracking and auditing.
        status:
          type: string
          enum: [COMPLETED_SUCCESS]
        message:
          type: string
          description: A human-readable message confirming success.
        execution_result:
          type: object
          description: (Optional) A summary of the work performed.
          properties:
            files_deleted:
              type: integer
              format: int64
            bytes_deleted:
              type: integer
              format: int64

    ErrorResponse:
      type: object
      description: The standard error response body.
      required:
        - error_code
        - message
      properties:
        delegation_task_id:
          type: string
          format: uuid
          description: The task ID, if one was assigned before the error
occurred.
        status:
          type: string
          enum: [FAILED_TERMINAL]
        error_code:
          type: string
          description: A machine-readable error code specific to the Delegation
Service.
          example: "CREDENTIAL_VENDING_FAILED"
        message:
          type: string
          description: A human-readable description of the error.
        details:
          type: string
          description: (Optional) Further details or a stack trace snippet for
debugging.

  responses:
    BadRequest:
      description: The request payload is malformed or invalid.
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/ErrorResponse'
    Unauthorized:
      description: The request lacks valid authentication credentials.
      content:
        application/json:
```

```yaml
          schema:
            $ref: '#/components/schemas/ErrorResponse'
    Forbidden:
      description: The authenticated client is not authorized to perform this
action.
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/ErrorResponse'
    Conflict:
      description: The request could not be completed due to a conflict with
the current state of the resource (e.g., idempotency key violation).
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/ErrorResponse'
    InternalServerError:
      description: An unexpected error occurred on the server while processing
the task.
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/ErrorResponse'
    BadGateway:
      description: The service failed to get a valid response from a downstream
dependency (e.g., the storage system).
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/ErrorResponse'
```

Example of version updates within Delegation Service persistence:

Description of Action Triggering New Version	task_id	version	status
Task Ingestion: The Delegation Service receives the task_payload from the Polaris Catalog and creates the first record for the task.	delete_table1	1	QUEUED
Processing Start: A worker picks up the task, increments the version, and marks the status as PROCESSING.	delete_table1	2	RUNNING
Transient Failure: The worker fails to connect to storage. It creates a new record with the RETRY_SCHEDULED status and records the error.	delete_table1	3	RETRY_SCHEDULED
Retry Attempt: After a backoff period, a worker picks up the task again. It creates a new record, re-enters the PROCESSING state.	delete_table1	4	RUNNING
Successful Completion: The second attempt succeeds. A final record is created with the COMPLETED_SUCCESS status.	delete_table1	5	COMPLETED_SUCCESS

LoadTableResult:

Field Name	Data Type	Description
metadata-location	String (URI)	Full URI pointing to the specific table metadata file (e.g., vX.metadata.json) in the storage system.
metadata	Object	Complete Iceberg TableMetadata object, serialized as JSON. Contains everything about the table's schema, partition specs, sort orders, snapshots, and location.
config	Object (Map)	A map of key-value string pairs for table-specific configuration that can override the client's or catalog's default settings. This is a critical field for passing back credentials and I/O settings. This can include: a bearer token for authenticating subsequent requests for this table, cloud-specific credentials: Keys like s3.access-key-id, s3.secret-access-key, s3.session-token. And FileIO settings: Overrides for the FileIO implementation class.
storage-credentials	Array of Objects	(Optional) An alternative, more structured way to provide credentials for different storage location prefixes.

Delegation Service will take LoadTableResult:

metadata_location	String (URI)	From LoadTableResult: The URI to the table's latest Iceberg

		metadata file. Used by the Delegation Service to traverse the table's history.
metadata	Object	From LoadTableResult: The full Iceberg TableMetadata JSON object for the dropped table. The metadata.location field within this object provides the table_base_location needed for a full recursive purge.
config	Object (Map)	From LoadTableResult: A map containing the vended, temporary credentials and any other relevant FileIO configuration needed to access storage.

Failure States (Synchronous):

Failure Point	**Resulting System State**	**Analysis**
1. Polaris -> Delegation Service Call Fails	**Scenario**: The Polaris Catalog tries to send the purge task to the Delegation Service, but the call fails (e.g. network error, DS is down). **State**: • Polaris Metastore: The table metadata remains untouched and the table is still visible. • Object Storage: Unchanged.	**Good**. This is a clean, recoverable failure. The user receives an error for delegating the DROP command and the system state has not been altered. The user can safely retry the entire operation later.
2. DS -> Polaris Credential Vending Fails	**Scenario**: The Delegation Service receives the task but fails to get credentials when it calls back to the Polaris loadTable API. **State**: • Polaris Metastore: Unchanged. • Object Storage: Unchanged.	**Good**. This is also a clean failure. The DS would return an error (e.g. a 502 Bad Gateway) to Polaris. Polaris then aborts the drop transaction, leaving everything unchanged. The user gets a clear error and can investigate the permission or connectivity issue before retrying.
3. DS Fails Midway Through Object Storage Deletion	**Scenario**: The DS gets credentials, starts deleting data files, but crashes or hits a non-retryable error after some files are deleted. This results in the connection timing out. **State (Depending on Approach in 9.4)**: 1. Polaris sees a table but it is broken in object storage 2. Polaris does not see a table and it is incompletely deleted in object storage	**Design Decision**; refer to 9.4
4. DS Fails After Deletion, Before Responding	**Scenario**: The DS successfully deletes all data from object storage but crashes before it can send the 200 OK success response back to Polaris. **State**: • Polaris Metastore: Table metadata remains visible.	**Acceptable**. The connection will time out and the table will still exist in Polaris. However, queries against the table will fail. The user needs to re-run the DROP command (TBD feature in Polaris).

	• Object Storage: The data is deleted.	

Task Retry:

1. Configuration of Retry Policies

Retries are configured in Delegation Service itself with task-specific overrides allowing for operational flexibility.

Example `delegation-service/application.properties`:

```
None
# --- Task Resiliency and Lease Management ---
delegation.task.lease.lost-timeout-duration=1h

# --- Default Retry Policy ---
delegation.retry.default.max-attempts=5
delegation.retry.default.initial-backoff-duration=30s
delegation.retry.default.backoff-multiplier=2.0
delegation.retry.default.max-backoff-duration=1h

# --- Task-Type Specific Overrides ---
delegation.retry.TABLE_PURGE.max-attempts=10
delegation.retry.TABLE_PURGE.initial-backoff-duration=1m
```

2. Retry Logic

2.1 How a Task is Leased:

The logic for leasing a task is handled by a single, atomic database query.
This query finds available tasks by checking three conditions:

- **New Tasks**: Looks for tasks in the SUBMITTED state.
- **Scheduled Retries**: Looks for tasks in the RETRY_SCHEDULED state whose next_scheduled_run_ts is in the past.
- **Lost or Timed-out Tasks**: Looks for tasks that have been in the RUNNING or ACQUIRING_RESOURCES state for longer than the configured timeout duration, indicating the original worker failed.

A worker that successfully executes this query and updates the task's status to RUNNING wins the lease and begins execution.

Example Execution Loop Pseudocode:

```
Python
POLLING_INTERVAL_SECONDS = 5

def worker_process_loop():
  """
  The main loop for a worker thread. It continuously tries to
lease and
  execute tasks from the shared database.
  """
  while True:
    task = lease_next_available_task()

    if task:
      execute_task(task)
    else:
      sleep(POLLING_INTERVAL_SECONDS)

def lease_next_available_task():
  """
  Atomically finds, claims, and returns an available task
  from the database. Returns None if no tasks are available.
  """
  task_record = query.sql_and_lock_row("""
    WITH available_tasks AS (
      -- Step 1: Find tasks that have been submitted but not yet
started.
      SELECT *, last_status_change_ts AS scheduled_time, 'new' AS
source
      FROM tasks WHERE status = 'SUBMITTED'
      UNION ALL

      -- Step 2: Find tasks that have failed and are already
scheduled for a retry.
```

```
      SELECT *, last_status_change_ts AS scheduled_time, 'retry' AS
source
      FROM tasks WHERE status = 'RETRY_SCHEDULED'
      UNION ALL

      -- Step 3: Find lost or failed tasks that were running but
have timed out.
      SELECT *, last_status_change_ts AS scheduled_time, 'running'
AS source
      FROM tasks WHERE status = RUNNING AND last_status_change_ts +
'${timeout_duration}' < NOW()

      SELECT *, last_status_change_ts AS scheduled_time,
'acquiring' AS source
      FROM tasks
      WHERE status = ACQUIRING_LEASE AND last_status_change_ts +
'${timeout_duration}' < NOW()
    )
    SELECT * FROM available_tasks
    ORDER BY scheduled_time ASC
    LIMIT 1
    FOR UPDATE SKIP LOCKED;
  """)

  if not task_record:
    return None

  # If we successfully leased a 'lost' task, log it for
observability.
  if task_record.source in ('running', 'acquiring'):
    LOG.info("Re-leasing possibly stuck task {task_record.id}")

  # Atomically update the task's status to 'PROCESSING' to
complete the lease.
  task = update_task_status(task_record.id, 'PROCESSING')
```

```
    return task
```

2.2 How a Task Enters the Retry State:

If a worker is executing a task and encounters a transient, recoverable error:

- The worker increments the attempt_count for the task in the database.
- It checks this count against the configured max-attempts.
- If more retries are allowed, it calculates the next run time using the backoff policy, sets the task status to RETRY_SCHEDULED, and updates the task record. The task will then be picked up again by the leasing query after its backoff period has passed.
- If the attempt_count reaches the max-attempts, the task is moved to a terminal FAILED_TERMINAL state.

If the Delegation Service crashes mid task execution:

- Failed tasks will be in RUNNING or ACQUIRING_RESOURCES state in the persistence
- The task executor will check when these tasks have been leased and if it should be considered lost/failed and thus retried.

Deployment Model:

1. Initial One-To-One Deployment

Both Polaris and the Delegation Service will be published as Docker images, remaining consistent with the existing Polaris docker-compose examples and providing a portable, consistent runtime environment.

Local Development: For local testing and development, the Polaris-DS pair can be launched using a docker-compose.yml file. This file will define both the polaris and delegation-service as services, connecting them on a shared Docker network.

Production Deployment: In a production environment, the pair would be deployed within a container orchestrator like Kubernetes. Both the Polaris and Delegation Service containers would typically be deployed into the same Kubernetes Namespace to provide network isolation and simplified discovery.

2. Future Evolution: Shared Service Model (Many-to-One)

With a future goal of supporting a single Delegation Service that serves multiple Polaris instances, the production deployment model would evolve from a tightly-coupled unit to a more centralized service architecture.

The Delegation Service would be deployed as its own independent, centralized service in a separate Kubernetes Namespace. It would no longer be deployed as a sidecar of Polaris or within the same unit as any single Polaris instance.