vDAG Controller

vDAG controller service is used for performing vDAG (virtual DAG - refer to the sections below in this document to understand what is a vDAG) actions like submitting input task, performing health checks of all the blocks that are part of the vDAG, quota management (used for rate-limiting inputs to the vDAG per session), output verification etc.

Functionalities

1. Provides a gateway using which tasks can be submitted for a vDAG, these tasks are executed and returned back to the user by the vDAG controller.
2. Provides a mechanism using which vDAG provider can implement quotas and rate-limiting policy - this policy will allow the vDAG provide to limit the tasks per session ID or globally.
3. Provides a mechanism to implement the health checker policy, this policy periodically monitors the health of the blocks involved across the network.
4. Provides a mechanism to implement periodic quality checks using a quality checker policy, this policy will periodically check the final output of a vDAG against the input, this policy can submit this payload to a external QA system or for manual audit.
5. Provides APIs to execute the management commands on quota management, health checker and quality checker policies.
6. Implements metrics which can be used to monitor the performance of the vDAG - like latency, throughput etc.

Architecture

Task Submission Flow:

1. **Task submission is initiated via the gRPC API** exposed by the system, allowing external clients to submit inference tasks.
2. **Upon submission**, the `RequestsProcessor` module ingests the task and invokes the `QuotaChecker` policy engine. The policy logic, defined by the vDAG provider, allows for custom validation logic and enforces session-level or global limits using Redis-based counters or similar mechanisms.
3. **If quota validation fails**, an immediate rejection response is returned to the client. Otherwise, the task payload proceeds to the inference processing pipeline.
4. **The `PreProcessor` module** transforms the protobuf-based input into the internal format required by the Adhoc Inference Server. This transformation includes resolving the entry `block_id` and compiling the vDAG static

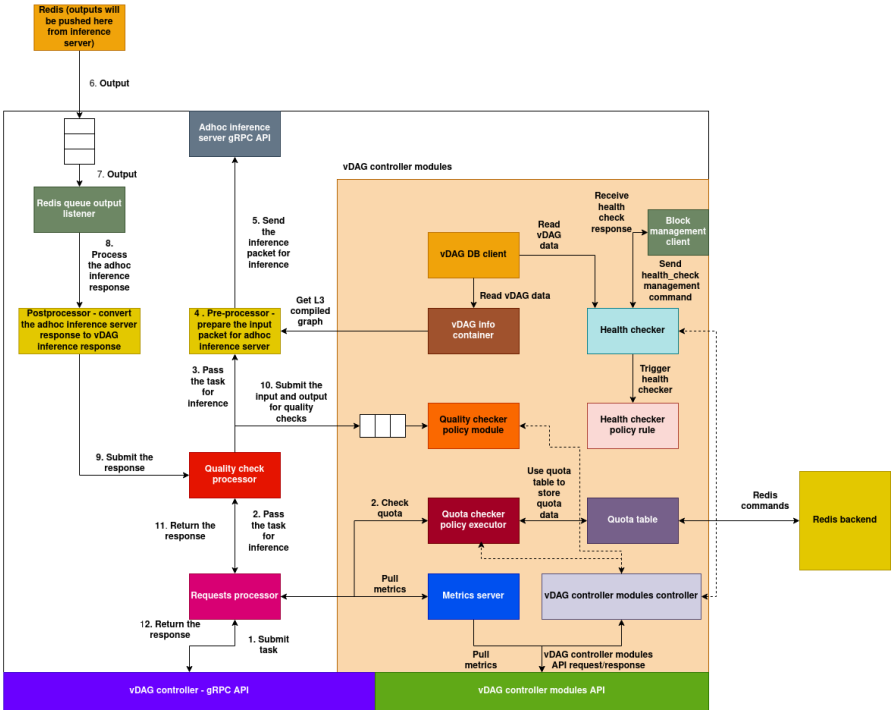

Figure 1: vDAG-controller

graph into a runtime dynamic graph. (Refer to `adhoc-inference-server` documentation for details on the dynamic graph representation.)

5. **The transformed request is then submitted** to the Adhoc Inference Server via its gRPC client. This module handles the transport layer abstraction and ensures proper formatting and connectivity with the server.
6. **A Redis-based queue listener** monitors a designated channel for inference outputs. The Adhoc Inference Server pushes the final results onto this queue upon completion.
7. **Once the output is available**, the listener retrieves the result from Redis and hands it off to the downstream modules.
8. **The output, received in the Adhoc Inference Server's native format**, is post-processed by the `PreProcessor`, which converts it back to the vDAG-compatible task output structure.
9. **The formatted output is then routed** to the `QualityCheckProcessor`.
10. **The `QualityCheckProcessor` uses both the original task input and the output** to enqueue a quality audit event into a dedicated policy queue. A background worker asynchronously picks and evaluates these samples based on the defined quality policies.
11. **The final output is passed back** to the `RequestsProcessor` module for response finalization.
12. **The processed response is returned to the originating client** over the same gRPC connection used for submission.

---

vDAG Controller Modules:

The **vDAG Controller Subsystem** provides the orchestration and auxiliary services required for task lifecycle management. It includes policy enforcement, system monitoring, configuration management, and health diagnostics.

1. Metrics Server:
Collects and reports system-level metrics such as latency, throughput, and frames-per-second (FPS). It aggregates telemetry data from the `RequestsProcessor` and periodically writes metrics to a centralized metrics database.

2. Quota Checker Module:
Executes the quota enforcement policies provided by the vDAG provider. It implements a Redis-backed `QuotaTable` for tracking and validating usage limits on a per-session or global basis.

3. Quality Checker Module:
Implements the policy executor for quality validation. A background thread

continuously polls a quality queue, executing audits against task samples using user-defined criteria.

4. Health Checker Module:
Executes periodic system and block-level health checks via a configurable health policy. It interacts with the Block Management API to fetch the current health status of individual compute blocks and passes this data through the health policy for validation.

5. vDAG DB Client:
Provides internal access to the vDAG registry. It retrieves metadata such as DAG structure, configuration parameters, and versioning details, enabling dynamic graph construction and validation by other modules.

6. Controller and API Server:
Exposes HTTP/REST APIs for policy and module management. It includes endpoints for managing metrics, health checks, quota policies, quality checks, and other auxiliary controller operations. This server acts as the centralized interface for system introspection and control.

Creating a vDAG controller:

vDAG controller can be created with the cluster controller gateway, for a vDAG controller to be created, a vDAG entry needs to exist in the registry, the vDAG creation payload should reference the vDAG URI of the vDAG along with specifying the controller ID (a unique ID for the controller) and other configuration parameters.

vDAG controller can be deployed on any cluster in the network by specifying the cluster ID.

Here is the API to create a vDAG controller:

Endpoint: `/vdag-controller/<cluster_id>`
Method: `POST`
Description:

Proxies a vDAG controller management request to the controller gateway of the specified cluster. Forwards the action and payload to the corresponding `/vdag-controller` endpoint within the target cluster.

Example curl Command:

```
curl -X POST http://<server-url>/vdag-controller/<cluster-id> \
  -H "Content-Type: application/json" \
  -d '{
    "action": "create_controller",
    "payload": {
      "vdag_controller_id": "<controller-id>",
```

```
    "vdag_uri": "<vdag-uri>",
    "config": {
      "policy_execution_mode": "local",
      "replicas": 2
    }
  }
}'
```

Parameter	Type	Description
`cluster_id`	string	The ID of the target cluster to which the request should be proxied.
`action`	string	The vDAG controller action to perform (e.g., `create_controller`
`vdag_controller_id`	string	Unique identifier for the vDAG controller being managed.
`vdag_uri`	string	URI of the vDAG to be associated with the controller.
`config`	object	Configuration for the controller. Includes fields like `policy_execution_mode`, `replicas`.
`policy_execution_mode`	string	Mode in which the controller should execute policies (e.g., `local`).
`replicas`	integer	Number of replicas to create or scale the controller to.

The vDAG controller can now be accessed at:

`http://{public-url-of-the-cluster}/<vdag_controller_id>`

For example

```
          (cluster public URL)     (vdag controller ID)
http://us-west-1.aiosv1.cluster.net/vdag-123
```

And the management server will be available at:

`http://us-west-1.aiosv1.cluster.net/vdag-123/server`

To remove the vDAG controller:

```
curl -X POST http://<server-url>/vdag-controller/<cluster_id> \
  -H "Content-Type: application/json" \
  -d '{
    "action": "remove_controller",
    "payload": {
      "vdag_controller_id": "<controller-id>"
    }
  }'
```

Parameter	Type	Description
`cluster_id`	string	The ID of the cluster to which the request will be proxied.
`action`	string	The management action to perform. Must be `"remove_controller"` in this case.
`vdag_controller_id`	string	The unique identifier of the vDAG controller to be removed.

Adhoc inference gRPC API:

Adhoc inference server provides gRPC API using which users can submit tasks to the vDAG, here is the protobuf format of the task input and output:

```
// Message to hold file metadata and content
message vDAGFileInfo {
    string metadata = 1;   // Metadata describing the file (e.g., filename, content type, et
    bytes file_data = 2;   // Actual binary data of the file
}

// Message structure used for vDAG inference requests and responses
message vDAGInferencePacket {
    string session_id = 3;       // Unique identifier for the inference session
    uint64 seq_no = 4;           // Sequence number for ordering messages in the session
    bytes frame_ptr = 5;         // Pointer to frame data stored in the frame DB.
    string data = 6;             // input json data serialized as string, contains output in
    double ts = 8;               // Timestamp of the packet (typically epoch time)
    repeated vDAGFileInfo files = 9; // List of files associated with the inference packet
}

// gRPC service for vDAG-based inference
service vDAGInferenceService {
    // Performs inference using the provided packet and returns a response packet
    rpc infer(vDAGInferencePacket) returns (vDAGInferencePacket);
}
```

Here's the table describing the fields in the `vDAGInferencePacket` message:

Field	Type	Description
`session_id`	string	Unique identifier for the inference session.
`seq_no`	uint64	Sequence number used to maintain the order of packets within a session.
`frame_ptr`	bytes (JSON string)	**Optional.** Used when the input file is large. Contains a JSON string with: • `framedb_id`: Frame DB ID • `key`: Unique file identifier.

Field	Type	Description
`data`	`string`	input json data serialized as string, contains output in case of the response packet.
`ts`	`double`	Timestamp of the packet, typically in epoch format.
`files`	`repeated vDAGFileInfo`	A list of files (binary + metadata) associated with this packet.

Python Example:

```
import grpc
import json
import time

# Import the generated gRPC classes
import vdag_pb2
import vdag_pb2_grpc

# Create a gRPC channel to the vDAG inference service
channel = grpc.insecure_channel("<vdag-controller-public-url>")
stub = vdag_pb2_grpc.vDAGInferenceServiceStub(channel)

# Example session and sequence information
session_id = "session-5678"
seq_no = 1

# Define the area of interest (bounding box coordinates)
# Format: [x_min, y_min, x_max, y_max]
area_of_interest = {
    "region_of_interest": {
        "x_min": 120,
        "y_min": 200,
        "x_max": 450,
        "y_max": 650
    }
}

# Load the image file to send as part of inference
with open("example_image.jpg", "rb") as img_file:
    image_bytes = img_file.read()

# Create vDAGFileInfo message with image
image_file = vdag_pb2.vDAGFileInfo(
    metadata=json.dumps({
        "filename": "example_image.jpg",
        "content_type": "image/jpeg"
```

```
    }),
    file_data=image_bytes
)

# Construct the vDAGInferencePacket
packet = vdag_pb2.vDAGInferencePacket(
    session_id=session_id,              # Unique session ID
    seq_no=seq_no,                      # Sequence number for packet ordering
    data=json.dumps(area_of_interest),  # Area of interest in JSON
    ts=time.time(),                     # Timestamp
    files=[image_file]                  # List of attached files
)

# Send inference request to the server
response = stub.infer(packet)

# Handle the response
print("Response received.")
print("Data:", response.data)

# Print details of any returned files
for f in response.files:
    metadata = json.loads(f.metadata)
    print(f"Returned file: {metadata.get('filename')} (type: {metadata.get('content_type')})
    # Save returned file if needed
    with open(f"output_{metadata.get('filename')}", "wb") as out_file:
        out_file.write(f.file_data)
```

---

vDAG controller policies:

Quota checker policy:

Quota checker policy provides the capability for the vDAG provider to restrict the usage of the inference API globally or per session_id.

Quota checker policy is called every time a task is submitted to check if the quota constraints are satisfied.

Quota checker module uses a redis cache backend to store the quota data, here is the definition of the class that manages the quota table:

```
class QuotaManagement:
    def __init__(self, redis_host="localhost", redis_port=6379, redis_db=0, quota_ttl=None):
        """
        Initializes the QuotaManagement instance with Redis connection details and an optio
        Establishes a connection to Redis and tests it.
```

```
        """
        pass

    def increment(self, session_id: str, amount: int = 1) -> int:
        """
        Increments the quota count for a given session ID by the specified amount.
        If TTL is set, it refreshes the expiry time.

        Args:
            session_id (str): Unique identifier for the session.
            amount (int): Amount to increment the quota by.

        Returns:
            int: The updated quota value.
        """
        pass

    def get(self, session_id: str) -> int:
        """
        Retrieves the current quota count for the specified session ID.

        Args:
            session_id (str): Unique identifier for the session.

        Returns:
            int: The current quota value, or 0 if not found.
        """
        pass

    def reset(self, session_id: str) -> None:
        """
        Resets the quota count for the specified session ID to zero.

        Args:
            session_id (str): Unique identifier for the session.
        """
        pass

    def clean(self) -> None:
        """
        Clears all quotas from the Redis database.
        """
        pass

    def remove(self, session_id: str) -> None:
        """
```

```
        Deletes the quota entry for the specified session ID from Redis.

        Args:
            session_id (str): Unique identifier for the session.
        """
        pass

    def exists(self, session_id: str) -> bool:
        """
        Checks whether a quota entry exists for the specified session ID in Redis.

        Args:
            session_id (str): Unique identifier for the session.

        Returns:
            bool: True if the session ID exists, False otherwise.
        """
        pass
```

Writing the quota checker policy:

Here is the structure of the input passed to the quota checker:

```
{
    // object of QuotaManagement class - used for accessing the quota table
    "quota_table": <object of QuotaManagement>,
    // input data in proto
    "input": <object of vDAGInferencePacket - task input>,
    // new quota - i.e current quota of that session_id + 1
    "quota": <number (int) representing the updated quota, i.e current_quota + 1> ,
    // session_id used in the task
    "session_id": <string - session_id>
}
```

The policy should return:

```
{
    "allowed": true
}
```

`allow` should be set to `false` if the request can't be allowed due to unsatisfied quota constraints.

Sample policy structure:

```
class AIOSv1PolicyRule:
```

```python
def __init__(self, rule_id, settings, parameters):

    """
        Initializes an AIOSv1PolicyRule instance.
        Args:
        rule_id (str): Unique identifier for the rule.
        settings (dict): Configuration settings for the rule.
        parameters (dict): Parameters defining the rule's behavior.

    """
    self.rule_id = rule_id
    self.settings = settings
    self.parameters = parameters

def eval(self, parameters, input_data, context):

    """
        Evaluates the policy rule.
        This method should be implemented by subclasses to define the   rule's logic.
        It takes parameters, input data, and a context object to perform  evaluation.
        Args:
            parameters (dict): The current parameters.
            input_data (any): The input data to be evaluated.
            context (dict): Context (external cache), this can be used for  storing and
    """

    """
    {
        // object of QuotaManagement class - used for accessing the quota table
        "quota_table": <object of QuotaManagement>,
        // input data in proto
        "input": <object of vDAGInferencePacket - task input>,
        // new quota - i.e current quota of that session_id + 1
        "quota": <number (int) representing the updated quota, i.e current_quota + 1> ,
        // session_id used in the task
        "session_id": <string - session_id>
    }
    """


    return {
        "allowed": True
    }

def management(self, action: str, data: dict) -> dict:
    """
```

```
    Executes a custom management command.

    This method enables external interaction with the rule instance for purposes such as
    - updating settings or parameters
    - fetching internal state
    - diagnostics or lifecycle control

    Args:
        action (str): The management action to execute.
        data (dict): Input payload or command arguments.

    Returns:
        dict: A result dictionary containing the status and any relevant details.
    """
    # Implement custom management actions here
    pass
```

Quota checker module APIs:

Endpoint: `/quota/<session_id>`
Method: `GET`
Description:
Retrieves the current quota value for the given session.

Example curl Command:

```
curl -X GET http://<server-url>/quota/session-1234
```

Endpoint: `/quota/reset/<session_id>`
Method: `POST`
Description:
Resets the quota associated with the given session ID.

Example curl Command:

```
curl -X POST http://<server-url>/quota/reset/session-1234
```

Endpoint: `/quota/exists/<session_id>`
Method: `GET`
Description:
Checks whether a quota record exists for the given session ID.

Example curl Command:

```
curl -X GET http://<server-url>/quota/exists/session-1234
```

Endpoint: `/quota/<session_id>`
Method: `DELETE`
Description:
Removes the quota associated with the specified session ID.

Example curl Command:

```
curl -X DELETE http://<server-url>/quota/session-1234
```

Endpoint: `/quota/mgmt`
Method: `POST`
Description:
Executes a management command for quota operations. The `mgmt_action` defines the operation, and `mgmt_data` provides input to that action.

Example curl Command:

```
curl -X POST http://<server-url>/quota/mgmt \
  -H "Content-Type: application/json" \
  -d '{
    "mgmt_action": "",
    "mgmt_data": {}
  }'
```

Quality checker policy:

Quality checker policy can be used to audit the performance of a vDAG by taking the request and response samples periodically, the policy can implement a logic to automatically audit the performance or can be used as a client to an external QA system where the request and response samples are saved and manually audited later.

Quality checker policy is executed in background by periodically collecting the input/output samples, thus it does not interfere with the main server thread.

Writing quality checker policy:

Following input is passed to the quality checker policy:

```
{
    // information of the vDAG (passed as is from the registry)
    "vdag_info": <vdag information>,
    "input_data": {
        // request task data
        "request": <request data - object of vDAGInferencePacket class> ,
```

```
        // response of the task
        "response": <response - object of vDAGInferencePacket class>
    }
}
```

Sample structure of the policy:

```
class AIOSv1PolicyRule:

    def __init__(self, rule_id, settings, parameters):

        """
            Initializes an AIOSv1PolicyRule instance.
            Args:
            rule_id (str): Unique identifier for the rule.
            settings (dict): Configuration settings for the rule.
            parameters (dict): Parameters defining the rule's behavior.

        """
        self.rule_id = rule_id
        self.settings = settings
        self.parameters = parameters

    def eval(self, parameters, input_data, context):

        """
            Evaluates the policy rule.
            This method should be implemented by subclasses to define the   rule's logic.
            It takes parameters, input data, and a context object to perform  evaluation.
            Args:
                parameters (dict): The current parameters.
                input_data (any): The input data to be evaluated.
                context (dict): Context (external cache), this can be used for  storing and
        """

        """
        {
            // information of the vDAG (passed as is from the registry)
            "vdag_info": <vdag information>,
            "input_data": {
                // request task data
                "request": <request data - object of vDAGInferencePacket class> ,
                // response of the task
                "response": <response - object of vDAGInferencePacket class>
            }
        }
        """
```

```
        # return value is ignored
        return {}

    def management(self, action: str, data: dict) -> dict:
        """
        Executes a custom management command.

        This method enables external interaction with the rule instance for purposes such as
        - updating settings or parameters
        - fetching internal state
        - diagnostics or lifecycle control

        Args:
            action (str): The management action to execute.
            data (dict): Input payload or command arguments.

        Returns:
            dict: A result dictionary containing the status and any relevant details.
        """
        # Implement custom management actions here
        pass
```

Quality checker management API:

Endpoint: `/quality-check/mgmt`
Method: `POST`
Description:
Executes a management command for quota operations. The `mgmt_action` defines the operation, and `mgmt_data` provides input to that action.

Example curl Command: “‘bash curl -X POST http:///quota/mgmt
-H “Content-Type: application/json”
-d ‘{ “mgmt_action”: "“,”mgmt_data": {} }’

Health checker

Health checker policy operates on the health check data of all the blocks that are part of the vDAG which is collected periodically.

This policy will be called periodically, based on the interval specified in the vDAG config.

Input data will be passed to this policy in this format:

```
{
    // information of the vDAG (passed as is from the registry)
    "vdag": <vDAG information>,
    "health_check_data": {
        // healthy block
        <block-id>: {
            "success": true,
            "data": {"instances": [list of running instances of that block]}
        },
        // unhealthy block
        <block-id>: {
            "success": false,
            "data": {
                // refer to the table below for unhealthy-reason-strings
                "mode": <unhealthy-reason-string>,
                // data (string)
                "data": <message - message string (can be ignored)>
            }
        }
    }
}
```

Unhealthy reason modes:

Mode	Description
`api_internal_error`	The block's health check API responded with a 200 status but indicated failure (e.g., `"success": False` in response). Typically indicates an internal error on the block's side.
`network_error`	The health check failed due to persistent network issues (timeouts, request exceptions, etc.) after exhausting all retries.
`general_error`	An unexpected exception occurred outside of the request/response logic (e.g., issues with URL construction, config errors, etc.).
`timeout_error`	A request to the block timed out. Used during retries, may contribute to eventual `network_error` if retries are exhausted.

Writing Health checker policy:

Here is the basic structure of health checker policy:

```
class AIOSv1PolicyRule:
```

```
def __init__(self, rule_id, settings, parameters):

    """
        Initializes an AIOSv1PolicyRule instance.
        Args:
        rule_id (str): Unique identifier for the rule.
        settings (dict): Configuration settings for the rule.
        parameters (dict): Parameters defining the rule's behavior.

    """
    self.rule_id = rule_id
    self.settings = settings
    self.parameters = parameters

def eval(self, parameters, input_data, context):

    """
        Evaluates the policy rule.
        This method should be implemented by subclasses to define the   rule's logic.
        It takes parameters, input data, and a context object to perform  evaluation.
        Args:
            parameters (dict): The current parameters.
            input_data (any): The input data to be evaluated.
            context (dict): Context (external cache), this can be used for  storing and
    """

    """
    {
        # information of the vDAG (passed as is from the registry)
        "vdag": <vDAG information>,
        "health_check_data": {
            # healthy block
            <block-id>: {
                "success": true,
                "data": {"instances": [list of running instances of that block]}
            },
            # unhealthy block
            <block-id>: {
                "success": false,
                "data": {
                    # refer to the table below for unhealthy-reason-strings
                    "mode": <unhealthy-reason-string>,
                    # data (string)
                    "data": <message - message string (can be ignored)>
                }
            }
```

```
            }
        }
        """

        # return custom data (this will not be processed by the vDAG controller)
        return {}

    def management(self, action: str, data: dict) -> dict:
        """
        Executes a custom management command.

        This method enables external interaction with the rule instance for purposes such as
        - updating settings or parameters
        - fetching internal state
        - diagnostics or lifecycle control

        Args:
            action (str): The management action to execute.
            data (dict): Input payload or command arguments.

        Returns:
            dict: A result dictionary containing the status and any relevant details.
        """
        # Implement custom management actions here
        pass
```

APIs for health checker module:

Endpoint: `/health/check`
Method: `GET`
Description:
Triggers an **ad-hoc health check** on all blocks assigned to the vDAG. The result is evaluated using a configured health check policy rule and returned.

Example curl Command:

```
curl -X GET http://<server-url>/health/check
```

Endpoint: `/health/mgmt`
Method: `POST`
Description:
Executes a **management command** on the health check policy. This can be used to modify the policy's internal configuration or behavior dynamically using the `mgmt_action` and optional `mgmt_data`.

Example curl Command:

```
curl -X POST http://<server-url>/health/mgmt \
  -H "Content-Type: application/json" \
  -d '{
    "mgmt_action": "",
    "mgmt_data": {}
  }'
```

---

vDAG controller metrics:

Following metrics are exported by vDAG controller:

vDAG metrics list

Name	Description
`inference_requests_total`	Total number of inference requests processed
`inference_fps`	Frames per second (FPS) of inference processing
`inference_latency_seconds`	Latency per inference request in seconds

These metrics can be queried from `Global vDAG metrics DB`:

Global vDAG metrics database stores vDAG metrics from all the vDAG controllers running across the clusters in the network. These metrics are reported by the vDAG controllers at fixed intervals. Global vDAG Metrics database also provides the query APIs which can be used by the systems and users for monitoring and decision making.

Global vDAG Metrics DB APIs:

Endpoint: `/vdag/<vdag_id>`
Method: `GET`
Description:

This endpoint retrieves metrics for a specific VDAG identified by its `vdag_id`. If the VDAG is not found, a corresponding message is returned. On success, the data is returned as a single object.

Example curl Command:

```
curl -X GET http://<server-url>/vdag/<vdag-id>
```

Endpoint: `/vdag/query`
Method: `POST`
Description:

This endpoint queries VDAG metrics using a MongoDB-style filter provided in the JSON request body. Supports standard MongoDB query operators like `$eq`, `$gt`, `$in`, etc.

Example curl Command:

```
curl -X POST http://<server-url>/vdag/query \
  -H "Content-Type: application/json" \
  -d '{
    "status": { "$eq": "healthy" },
    "metrics.latency": { "$lt": 200 }
  }'
```