vDAG Specification – Field Reference

This section provides an in-depth explanation of each field expected in the vDAG specification payload submitted to the Parser.

Top-Level Fields

Field	Type	Required	Description
`vdagName`	string	Yes	Unique, human-readable name assigned to the vDAG. This acts as the logical identifier used in discovery and referencing.
`vdagVersion`	object	No	Dictionary specifying the semantic version of the vDAG. It contains: • `version` — version number (e.g., `"1.0.0"`) • `release-tag` — release label (e.g., `"beta"`, `"stable"`). These are combined to construct the `vdagURI`.
`mode`	string	No	Mode of creation. Supported values: • `"create"` — default mode that persists the vDAG • `"dry-run"` — used to validate the spec without persisting.

Field	Type	Required	Description
`discoveryTags`	array of strings	No	Tags used for indexing, grouping, or categorizing the vDAG for search and discovery. Examples include domain-specific keywords like `"vision"`, `"chatbot"`, or `"gpu"`.
`controller`	object	No	Defines the runtime controller configuration for the vDAG. This governs how the vDAG receives input, initializes runtime state, and applies controller-level policies.
`nodes`	array of objects	Yes	List of nodes in the vDAG. Each node represents a computation block or another vDAG. Node specifications contain routing, assignment, and policy information.
`graph`	object	Yes	Defines the directed graph structure connecting the nodes. This includes node labels and edge definitions that control execution flow. (Structure explained separately.)

`vdagVersion`

Field	Type	Required	Description
`version`	string	No	Semantic version (e.g., `"1.0.0"`). Used in combination with the name and release-tag to version-control the vDAG.
`release-tag`	string	No	A tag identifying the release status (e.g., `"beta"`, `"stable"`, `"dev"`).

`controller`

Field	Type	Required	Description
`inputSources`	array	No	List of input sources from which the vDAG controller will pull input. These can include queue topics, stream IDs, or external endpoints.
`initParameters`	object	No	A dictionary of parameters to be made available to the controller at runtime initialization. These may influence how it reacts to events, manages sessions, or handles routing.

Field	Type	Required	Description
`initSettings`	object	No	Static configuration settings for the controller, often used to configure behavior like error handling, logging, retry policies, etc.
`policies`	array	No	List of policy rule objects that apply to the controller itself. These are distinct from node-level policies and often govern cross-cutting logic.

`nodes` (List of Node Specifications)

Each node represents a unit of computation or routing inside the vDAG.

Field	Type	Required	Description
`nodeLabel`	string	Yes	Unique label for the node. This acts as the identifier within the vDAG and must match what's referenced in the graph.

Field	Type	Required	Description
`nodeType`	string	Yes	Type of the node. Accepted values: • `"block"` — standard compute unit (default) • `"vdag"` — a reference to another registered vDAG
`vdagURI`	string	Required if `nodeType` is `"vdag"`	Fully qualified URI of the nested vDAG to invoke. Ignored for regular blocks.
`assignmentPolicyRule`	object	No	A policy rule object used to dynamically assign a block to this node. If `manualBlockId` is provided, this policy is ignored.
`preprocessingPolicyRule`	object	No	Policy rule to be executed before forwarding input to the node's instance. Used for filtering, reshaping, or enriching input.

Field	Type	Required	Description
`postprocessingPolicyRule`	object	No	Policy rule to be executed after receiving output from the instance. Useful for formatting, filtering, or result validation.
`modelParameters`	object	No	Custom model parameters that are made available to the node instance during request execution. Typically used to override defaults.
`manualBlockId`	string	No	Manually assigned block ID for this node. If specified, the `assignmentPolicyRule` is bypassed.

> All other fields defined in the NodeObject (such as `inputProtocol`, `outputProtocol`, `IOMap`) are system-generated or currently ignored at spec time and should not be included.

Absolutely. Using your real-world example — `object-detection → tracking → pose-estimation` — here's the updated **official documentation for the vDAG `graph` section**, grounded in a meaningful use case and explained clearly.

---

vDAG `graph` Field

The `graph` field defines the execution order of nodes in a vDAG using **logical node labels**. These labels correspond to nodes defined in the `nodes` array. This structure enables the specification of complex execution pipelines by expressing how data flows from one node to the next.

Conceptual Example: Object Detection → Tracking → Pose Estimation

Assume a vision pipeline where:

1. **`object_detector`** detects objects in video frames
2. **`tracker`** associates objects across frames
3. **`pose_estimator`** estimates the pose of each tracked object

Each of these steps is represented as a node in the vDAG, and they are connected in that order.

Expected Graph Structure

```json
"graph": {
  "connections": [
    {
      "nodeLabel": "tracker",
      "inputs": [
        { "nodeLabel": "object_detector" }
      ]
    },
    {
      "nodeLabel": "pose_estimator",
      "inputs": [
        { "nodeLabel": "tracker" }
      ]
    }
  ],
  "inputs": [
    {
      "nodeLabel": "object_detector"
    }
  ],
  "outputs": [
    {
      "nodeLabel": "pose_estimator"
    }
  ]
}
```

Explanation of Fields

Field	Type	Description
`connections`	array of objects	Describes edges in the DAG. Each object defines inputs to a destination node.
`nodeLabel`	string	The destination node (receiving data). In the example: `"tracker"` and `"pose_estimator"`.
`inputs`	array	List of source nodes (providing data to this node).
`inputs[].nodeLabel`	string	A source node label. It must match a node defined in the `nodes` section.

Node Flow for Example

In the example graph above:

- The **object detector** is the head node (starting point).

- It feeds its output into the **tracker**.
- The tracker then forwards its output into the **pose estimator**, which is the tail node.

This results in the following execution order:

`object_detector → tracker → pose_estimator`

Validation Guidelines

- All `nodeLabel` references in `connections` and `inputs` **must match** node labels defined in the `nodes` array.
- The graph should form a valid **Directed Acyclic Graph (DAG)** — no cycles or invalid branches.

- Nodes with no incoming connections are considered **entry points** (heads).
- Nodes with no outgoing connections are considered **exit points** (tails).

Assignment Policy

An **Assignment Policy** is responsible for selecting the most suitable block to assign to a node within a vDAG (virtual Directed Acyclic Graph). It functions as a specialized search policy that operates alongside the filtering mechanism provided by the Parser system. The assignment process begins with a filter query that retrieves one or more candidate blocks based on predefined criteria (see Filtering and Searching documentation). These blocks are then passed to the assignment policy. The policy must return a final decision in the form of a selected block ID, which will be used to assign the block to the vDAG node.

Writing assignment policy:

Here is the basic structure of an assignment policy:

```python
class AIOSv1PolicyRule:
    def __init__(self, rule_id, settings, parameters):
        self.rule_id = rule_id
        self.settings = settings
        self.parameters = parameters

        # block and cluster metrics API objects will be passed to the policy
        # these will be used to query the cluster and block metrics needed for
        # further processing:
        self.block_metrics_api = settings["block_metrics_api"]
        self.cluster_metrics_api = settings["cluster_metrics_api"]


    def eval(self, parameters, input_data, context):
        """
          inputs will be list of objects returned by the pre-filter query execution:
          - this will contain the list of blocks
        """

        # Do the search here

        # return the new results
        return []
```

Block metrics API and Cluster metrics API methods:

Here are the methods available to pull metrics from clusters and blocks in cluster_metrics_api and block_metrics_api objects passed as settings parameters to the policy:

```python
class GlobalClusterMetricsClient:
    """
    Client for accessing global cluster metrics.
    """

    """rest of the code"""

    def get_cluster(self, cluster_id):
        """
        Fetches metrics data for a specific cluster by its ID.

        Args:
            cluster_id (str): The unique identifier of the cluster.

        Returns:
            dict: Cluster metrics data.
        """
        url = f"{self.base_url}/cluster/{cluster_id}"
        response = requests.get(url)
        return self._handle_response(response)

    def query_clusters(self, query_params=None):
        """
        Queries multiple clusters using optional filters.

        Args:
            query_params (dict, optional): Dictionary of query parameters.

        Returns:
            list: A list of matching clusters.
        """
        url = f"{self.base_url}/cluster/query"
        response = requests.post(url, json=query_params)
        return self._handle_response(response)


class GlobalBlocksMetricsClient:
    """
    Client for accessing global block metrics.
    """

    """rest of the code"""
```

```python
    def get_block(self, block_id):
        """
        Fetches metrics data for a specific block by its ID.

        Args:
            block_id (str): The unique identifier of the block.

        Returns:
            dict: Block metrics data.
        """
        url = f"{self.base_url}/block/{block_id}"
        response = requests.get(url)
        return self._handle_response(response)

    def query_blocks(self, query_params=None):
        """
        Queries multiple blocks using optional filters.

        Args:
            query_params (dict, optional): Dictionary of query parameters.

        Returns:
            list: A list of matching blocks.
        """
        url = f"{self.base_url}/block/query"
        response = requests.get(url, params=query_params)
        return self._handle_response(response)
```

Pre and Post processing policies:

Pre-processing and **post-processing** policies are used to define custom logic that should run before and after the main computation for each node. When a computation task is detected for a node in a vDAG for the first time, any specified policies are automatically loaded into the AIOS SDK instance. The **pre-processing policy** is executed before the job packet is passed to the instance's core logic, while the **post-processing policy** runs after the job packet has been processed. These policies allow for modular and reusable handling of data transformations, validations, or enrichment steps around the core computation.

Writing pre and post processing policies:

The structure of pre and post processing policies are the same:

```python
class AIOSv1PolicyRule:
    def __init__(self, rule_id, settings, parameters):
        self.rule_id = rule_id
        self.settings = settings
```

```python
        self.parameters = parameters

        # vDAG information
        self.vdag = settings['vdag']

        # block information
        self.block = settings['block']

        # node label of the current node in the vDAG
        self.node_label = settings['node_label']

        # block_id of the block
        self.block_id = settings['block_id']


    def eval(self, parameters, input_data, context):
        """
          input_data["packet"] - contains the AIOSPacket proto object
        """

        # Do the search here
        packet = input_data['packet']

        # do the packet manipulation here

        return packet
```

AIOSPacket proto structure:

```protobuf
message AIOSPacket {
    string session_id = 1;      // Unique identifier for the session, enables stateful logi
    uint64 seq_no = 2;          // Monotonically increasing sequence number for this sessio
    string data = 4;            // JSON-encoded input payload; format depends on block logi
    double ts = 5;              // Optional Unix timestamp (used for latency metrics or ord
    string output_ptr = 6;      // JSON-encoded pointer to downstream blocks; defines the o
    repeated FileInfo files = 7;// Optional array of attached files
}

message FileInfo {
    string metadata = 1;        // JSON string containing metadata for the file
    bytes file_data = 2;        // Raw file content as byte array
}
```