Block

Block is a instantiation of the component registered in the component registry, block is a collection of multiple services and one or more replicas of the instances of servers that are responsible for serving the component which is usually a AI inference of any generl computational workload. Block is scheduled on a cluster and a block cannot spawn across multiple clusters.

Instance:

1. The container image built using AIOS instance SDK will be instantiated as an instance of the block.
2. The instance is always scheduled on the node of a cluster that satisfies the resource requirements, multiple instances can be scheduled on the same node.
3. One or more GPUs can be mounted to an instance based on the requirements, multiple instances can be allocated the same GPU if resource allocation policy permits.
4. System defined metrics and custom metrics (build using AIOS metrics module) are emitted by the instance, these metrics are collected and used by various services for decision making.
5. Instances can be up-scaled and down-scaled as per the demand, which is taken care by the auto-scaling policy.
6. Load across the instances are distributed based on the rules written in load balancer policy which runs in executor.

Services of blocks:

1. **Executor**: Executor acts as a gateway for the block, tasks submitted to the block are load balanced using a load balancer policy, also implements executor metrics.
2. **Mapper service**: Mapper service monitors for instance additions and removals periodically, when instances are added/removed, the new list of instances will be populated and sent to all the block's services, so they can maintain the updated state of instances.
3. **Auto-scaler**: Auto-scaler periodically monitors the metrics of all the instances, executes auto-scaler policy to determine if the instances needs to be scaled/down-scaled based on the workload demand.
4. **Health checker**: This service periodically calls the health endpoint of all the instances of the block, runs health checker policy to determine if all the instances are healthy and takes the necessary action (defined in the policy) if anomalies are detected.

5. **Metrics system**: Metrics system is spread across the executor and instances, metrics system consumes the metrics and actively writes it to the local cluster metrics DB and the global metrics DBs.

Side-cars:

Side-cars are utility pods deployed as a part of the block, these pods listen for inputs from the instances and process them, side-cars can be used to perform actions like - saving the task outputs to a remote DB, checking for useful information from the outputs and alerting the associated external users/services etc.

Block specification:

A **Block Specification** defines a deployable unit that is derived from a registered component. Blocks allow for per-deployment customization such as runtime parameters, initialization data, policies, and scaling configurations. Each block references a component by its URI and may override or extend component-level defaults.

The `block_create_IR` function processes this specification, enriching it with metadata and default values from the component if they are not explicitly defined in the block spec.

Top-Level Structure

The spec must be submitted under the following structure:

```
{
  "body": {
    "spec": {
      "values": {
        ...
      }
    }
  }
}
```

Fields

Field	Type	Required	Description
`blockComponentURI`	`string`	Yes	URI of the registered component to use as the base for this block.
`blockId`	`string`	No	Unique ID of the block. If not provided, it is auto-generated.
`minInstances`	`number`	Yes	Minimum number of runtime instances to deploy.
`maxInstances`	`number`	Yes	Maximum number of runtime instances to deploy.
`blockInitData`	`object`	No	Initialization data specific to the block. If not provided, inherited from the component.
`initSettings`	`object`	No	Runtime settings for the block. If not specified, inherited from the component.
`parameters`	`object`	No	Runtime or inference parameters. If not provided, inherited from the component.
`policyRulesSpec`	`array`	No	Array of custom policy rule definitions to override or extend component-level policies. See below for structure.

Great catch — you're absolutely right. The documentation should reflect the **conditional override behavior** of policies based on the code logic.

Here's the corrected and refined version of the relevant section:

Policy Rule Specification (`policyRulesSpec`)

Policies in a block specification are optional and only override the component-level policies if explicitly provided.

- The block **inherits all policies from the component by default**.
- If a policy with the same `name` exists in both the component and the block's `policyRulesSpec`, the block-level policy **overrides** the one from the component.
- If a new policy is defined in `policyRulesSpec` that doesn't exist at the component level, it is **added** to the block's policy set.
- If `policyRulesSpec` is **omitted**, the block uses the component's policies as-is with no changes.

Each entry in `policyRulesSpec` must follow this structure:

```
{
  "values": {
    "name": "policyName",
    "policyRuleURI": "policy.rule.uri",
    "parameters": { ... },
    "settings": { ... }
  }
}
```

Field	Type	Description
`name`	`string`	Unique name identifying the policy. Used as a key for merging or overriding.
`policyRuleURI`	`string`	URI of the policy rule to apply.
`parameters`	`object`	Policy input parameters.
`settings`	`object`	Runtime or configuration settings for the policy.

Inherited Fields from Component

If not explicitly specified in the block spec, the following fields are copied from the associated component:

Field	Description
`blockInitData`	Defaults to `componentInitData`.
`initSettings`	Defaults to `componentInitSettings`.
`parameters`	Defaults to `componentParameters`.
`blockMetadata`	Populated from `componentMetadata`.
`inputProtocol`	Populated from `componentInputProtocol`.
`outputProtocol`	Populated from `componentOutputProtocol`.
`tags`	Copied from `component.tags`.
`policies`	Starts with `component.policies`, then overridden or extended via `policyRulesSpec`.

Block policies:

Here is the list of policies that should be provided in the spec when a block should be created:

Policy Name	Type	Description
`clusterAllocator`	Object	Used for selecting the initial list of candidate clusters and performing final cluster selection based on ranking logic.
`resourceAllocator`	Object	Responsible for allocating resources to block instances within the selected cluster and for scaling and re-assignment of instance in block life-cycle.
`loadBalancer`	Object	Determines the target instance for a given task based on real-time metrics of active instances.
`stabilityChecker`	Object	Acts as a health check policy to assess the health status of block instances using monitoring data.

Policy Name	Type	Description
`autoscaler`	Object	Makes scaling decisions by analyzing metrics collected from instances and nodes.

Block allocation Flow

The cluster allocator policy is used for the initial selection of the cluster on which the target block must be allocated. Before we dive into cluster selection, it's important to understand the end-to-end allocation process. Here are the steps:

1. The block specification is submitted to the parser for allocation.
2. Once the specification passes validation, the JSON data extracted from the spec is passed to the cluster controller gateway.
3. The cluster controller gateway parses the data and retrieves the cluster allocator policy specification.
4. The filter rule specified in the cluster allocator policy is executed first, returning a list of clusters.
5. This list of clusters serves as the initial set of candidates and is passed to the cluster allocator policy with `"action": "selection"`.
6. The cluster allocator policy may apply additional (optional) filtering and then selects a final list of candidate clusters.
7. A resource allocation request is made in `dry_run` mode for each of these clusters. During this phase, each cluster loads the block's `resourceAllocation` policy in `dry_run` mode. The output of the `dry_run` execution is a feasibility score between 0 and 1. For example, a value of 0.9 means the cluster can run the block with 90% feasibility. It is recommended to run feasibility analysis for `maxInstances`.
8. The feasibility scores from all clusters are collected and passed to the cluster allocator policy with `"action": "post_dry_run"`. In this mode, the policy selects one cluster from the list based on the reported feasibility scores.
9. The final cluster is selected in step 8, and the allocation request is sent to the cluster controller of the selected cluster, where the block gets allocated.

Summary:

Here is the list of actions that needs to be selected for each policy:

Cluster Allocator Policy – Supported Actions

Action	Description
`selection`	Performs ranking or additional filtering on the candidate clusters returned from the `filter` step.
`post_dry_run`	Receives feasibility scores from candidate clusters (based on dry run resource allocation) and selects the final cluster for allocation.

Resource Allocator Policy – Supported Actions

Action	Description
`dry_run`	Simulates resource allocation for the block and returns a feasibility score (ranging from 0.0 to 1.0). This score is used by the Cluster Controller Gateway to rank clusters for final selection.
`allocation`	Performs actual resource allocation for a newly created block. Similar to a dry run, but the policy must return the target node ID and one or more GPU IDs to be assigned to the instance.
`scale`	Allocates resources for a new instance of an existing block. The policy must return the node ID and one or more GPU IDs for the additional instance.
`reassignment`	Allocates resources for an existing instance that is being reassigned. The policy may take into account the current allocation details when determining the new placement.

Writing cluster allocator policy:

Here is the structure of cluster allocator policy:

```
class ClusterSelectorPolicyRule:

    def __init__(self, rule_id, settings, parameters):
        # Initialize the policy with rule ID, configuration settings, and external parameter
        self.rule_id = rule_id
```

```
        self.settings = settings
        self.parameters = parameters

    def eval(self, parameters, input_data, context):
        """
        Evaluates the cluster allocator policy based on the action phase.

        Supported actions:
        - "selection": Select clusters after filtering (could be all or a subset).
        - "post_dry_run": Select a single cluster based on feasibility scores.
        """
        try:

            # Action: selection
            if input_data['action'] == "selection":
                # list of cluster from the initial filter is obtained here
                filter_result = input_data['filter_result']

                # do some custom filtering
                result = {}

                result['clusters'] = filter_result
                return result

            # Action: post_dry_run:
            if input_data['action'] == "post_dry_run":
                selected_clusters = input_data['clusters']
                """
                [
                    {
                       "score_data": {

                                # feasibility score - must be between 0-1, 1 means max feas
                                    "score": 0.9,
                                    "node_info": {
                                        # node_id where the instance will be scheduled
                                        "node_id": <node_id>,
                                        # gpu_ids that will be assigned to the instance
                                        "gpus": [....]
                                    }
                            }
                    }
                ]
                """

                # select one of the clusters based on it's score (index of the selected clus
```

```
                return selected_clusters[idx]

        except Exception as e:
            # Bubble up exception for the caller to handle
            raise e
```

Example:

```
class ClusterSelectorPolicyRule:

    def __init__(self, rule_id, settings, parameters):
        # Initialize the policy with rule ID, configuration settings, and external parameter
        self.rule_id = rule_id
        self.settings = settings
        self.parameters = parameters

    def eval(self, parameters, input_data, context):
        """
        Evaluates the cluster allocator policy based on the action phase.

        Supported actions:
        - "selection": Select clusters after filtering (could be all or a subset).
        - "post_dry_run": Select a single cluster based on feasibility scores.
        """
        try:

            # Action: selection
            if input_data['action'] == "selection":

                print(parameters, input_data, context)

                # Retrieve filtered cluster candidates from the filter step.
                filter_result = input_data['filter_result']
                if len(filter_result) == 0:
                    raise Exception("no results found in the filter")

                result = {}

                # Optionally apply custom filtering or ranking logic here
                # Currently, returns all filtered clusters as-is
                result['clusters'] = filter_result

                return result

            # Action: post_dry_run
            if input_data['action'] == "post_dry_run":
```

```
            selected_clusters = input_data['clusters']

            # Select the cluster with the highest feasibility score from dry_run
            highest_index = 0
            current_highest = -1
            idx = 0

            for entry in selected_clusters:
                score_data = entry["score_data"]
                if score_data['score'] > current_highest:
                    current_highest = score_data['score']
                    highest_index = idx
                idx += 1

            # Return the cluster with the highest score
            return selected_clusters[highest_index]

    except Exception as e:
        # Bubble up exception for the caller to handle
        raise e
```

Writing block resource allocation policy:

The policy for resource allocation needs to be written in such a way that accommodates `dry_run`, `allocation`, `scale` and `reassignment` requests.

General input data format Input is passed to the resource allocator policy in the following structure:

```
{
    "action": "dry_run" // can be - 'dry_run', 'allocation', 'scale', 'reassignment',
    "payload": {} // action specific input payload (check the examples below)
}
```

Payload formats

1. For `dry_run`:

```
{
    // data of the block that is being created
    "block": <complete-block-entry>,
    // data of the current cluster
    "cluster": <complete-cluster-entry>,
    // metrics of the current cluster at that point in time
    "cluster_metrics": <metrics of the cluster>,
    // list of nodes that are healthy in the current cluster at that point in time
    "healthy_nodes": <list of node IDs that are healthy>
}
```

Return:

```
{
    "selection_score_data": {
        // feasibility score - must be between 0-1, 1 means max feasibility, 0 means - this
        "score": 0.9,
            "node_info": {
                // node_id where the instance will be scheduled
                "node_id": <node_id>,
                // gpu_ids that will be assigned to the instance
                "gpus": [....]
            }
    }
}
```

---

2. For `allocation`:

```
{
    // data of the block that is being created
    "block": <complete-block-entry>,
    // data of the current cluster
    "cluster": <complete-cluster-entry>,
    // metrics of the current cluster at that point in time
    "cluster_metrics": <metrics of the cluster>,
    // list of nodes that are healthy in the current cluster at that point in time
    "healthy_nodes": <list of node IDs that are healthy>
}
```

Return:

```
{
    // node_id where the instance will be scheduled
    "node_id": <node_id>,
    // gpu_ids that will be assigned to the instance
    "gpus": [....]
}
```

---

3. For `scale`:

```
{
    // data of the block that is being created
    "block": <complete-block-entry>,
    // data of the current cluster
    "cluster": <complete-cluster-entry>,
    // metrics of the current cluster at that point in time
    "cluster_metrics": <metrics of the cluster>,
```

```
    // metrics of the current block for which the new instance is being created
    "block_metrics": <metrics of the block>,
    // list of nodes that are healthy in the current cluster at that point in time
    "healthy_nodes": <list of node IDs that are healthy>
}
```

Return:

```
{
    // node_id where the instance will be scheduled
    "node_id": <node_id>,
    // gpu_ids that will be assigned to the instance
    "gpus": [....]
}
```

---

4. For `reassignment`:

```
{
    // data of the block that is being created
    "block": <complete-block-entry>,
    // data of the current cluster
    "cluster": <complete-cluster-entry>,
    // metrics of the current cluster at that point in time
    "cluster_metrics": <metrics of the cluster>,
    // metrics of the current block for which the new instance is being created
    "block_metrics": <metrics of the block>,
    // list of nodes that are healthy in the current cluster at that point in time
    "healthy_nodes": <list of node IDs that are healthy>,
    // id of the instance being re-assigned
    "instance_id": <instance-id>,
    // name of the pod being re-assigned, i.e name of the instance pod
    "pod_name": <name of the pod being re-assigned>,

}
```

Return:

```
{
    // node_id where the instance will be scheduled
    "node_id": <node_id>,
    // gpu_ids that will be assigned to the instance
    "gpus": [....]
}
```

**Example resource allocation policy structure:

```python
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.settings = settings
        self.parameters = parameters


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

        '''
        Evaluates the policy rule.

        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 acces
        '''

        '''
           input_data={
            "action": "dry_run"/ "allocation" /  "scale" / "reassignment",
            "payload": <as defined in the previous section based on the action>
           }
        '''

        # handle the logic here

        if input_data['action'] in ["allocation", "scale", "assignment"]:
            return {
                # node_id where the instance will be scheduled
                "node_id": <node_id>,
                # gpu_ids that will be assigned to the instance
                "gpus": [....]
            }
        elif input_data['action'] == "dry_run":
            {
                "selection_score_data": {
                    # feasibility score - must be between 0-1, 1 means max feasibility, 0 m
```

```
                    "score": 0.9,
                        "node_info": {
                            # node_id where the instance will be scheduled
                            "node_id": <node_id>,
                            # gpu_ids that will be assigned to the instance
                            "gpus": [....]
                        }
                }
        }
```