Block

Block the core component AIOSv1 responsible for instantiating, serving, scaling and managing the AI inference or any general computational workload defined using AIOSv1 Instance SDK. A block can be scheduled on any cluster within the network of clusters that satisfies the resource requirements.

Functionalities of a block:

1. Manages one or more instances of the AI or general computational workload.
2. Spins up a load balancer (known as executor) which can be used for load balancing the tasks among the instances, the load balancing logic can be defined by the developer using load balancing policy.
3. Spins up Auto-scaler module as a side-car, this module monitors the instances and takes scaling decisions using Auto-scaling policy defined by the developer.
4. Provides APIs for executing management commands to update the configuration, query data from the auto-scaling and load balancing policies.
5. Provides APIs for fetching the metrics of instances and executor (load balancer)
6. Provides a health checker module as a side-car which is used to check the health of the instances and execute the developer defined health checker policy to check if the instances are healthy.
7. Provides a gRPC proxy server which will be used for exposing the block inference API to the outside world via the ingress.

Block creation:

Blocks are created using the block specification (defined in detail in the parser documentation).

For the creation flow, for the policies involved in block creation process - please refer to the Parser documentation, this document explains the Block components involved in the runtime like Load Balancer, Auto-scaling, health checks, metrics and management APIs.

Block inference API:

Block inference API can used to submit the inference request directly to the block, the block inference API uses gRPC protocol and defines a schema similar to the global inference server API. Here is the definition of the protobuf used for block inference:

```
syntax = "proto3";

message InferenceMessage {
   bytes rpc_data = 1;
}
```

```
message InferenceRespose {
     bool message = 1;
}

// Define the gRPC service
service InferenceProxy {
    rpc infer(InferenceMessage) returns (InferenceRespose);
}
```

For performance reasons, the proxy accepts a bytes data of the serialized protobuf structure defined below:

```
syntax = "proto3";

message FileInfo {
    string metadata = 1; // JSON serialized metadata
    bytes file_data = 2; // File data
}

message AIOSPacket {
    string session_id = 1;
    uint64 seq_no = 2;           // Sequence number
    bytes frame_ptr=3;          // frame_ptr data
    string data = 4;              // Data - json serialized
    double ts = 5;                // Timestamp
    string output_ptr = 6;       // JSON serialized string for output pointer
    repeated FileInfo files = 7; // Array of file structures
}
```

The fields are as follows:

1. **`session_id`** *(Required)*
   - A unique identifier representing an inference session.
   - Enables stateful inference by allowing the block to track the session state.
2. **`seq_no`** *(Required)*
   - Represents the sequence number of the task within the same `session_id`.
   - The combination of `(session_id, seq_no)` uniquely identifies a task.
   - This can be used to ensure inputs are processed in order.
3. **`frame_ptr`** *(Optional)*
   - Used when the task requires a very large file as input.

- Stores file references in Frame DB instead of passing the file directly.

- A JSON string containing:
  - **framedb_id**: The ID of the Frame DB.

  - **key**: A unique key identifying the file in Frame DB.

4. **data** *(Required)*
   - Contains the task's input data.

   - Typically a JSON string that acts as input for the block.
5. **ts** *(Required)*
   - A floating-point UNIX epoch timestamp indicating when the task was created.
6. **files** *(Optional)*
   - A list of small files (e.g., images, audio, PDFs) that the block can use during inference.

   - Each file should comply with the `FileInfo` struct, containing:
     - **metadata** (Optional): A JSON string with additional metadata about the file.

     - **file_data**: The binary content of the file.
7. **output_ptr** *(Optional)*
   - Used specifically for **graph inference** tasks.

   - **Keep this field empty if you are doing direct block inference**

Python example:

```
import grpc
import time
from protos import inference_pb2, inference_pb2_grpc, payload_pb2

def create_aios_packet():
    packet = payload_pb2.AIOSPacket()
    packet.session_id = "session-123"
    packet.seq_no = 1
    packet.data = '{"input": "Hello Block"}'
    packet.ts = time.time()

    # Optional: Add a small file
    file_info = payload_pb2.FileInfo()
    file_info.metadata = '{"type": "text", "description": "sample file"}'
    file_info.file_data = b"Sample file content"

    packet.files.append(file_info)
```

```
    return packet

def run():
    # Connect to the gRPC server
    channel = grpc.insecure_channel("localhost:50051")
    stub = inference_pb2_grpc.InferenceProxyStub(channel)

    # Serialize the payload
    aios_packet = create_aios_packet()
    serialized_data = aios_packet.SerializeToString()

    # Wrap it into InferenceMessage
    request = inference_pb2.InferenceMessage(rpc_data=serialized_data)

    # Send inference
    response = stub.infer(request)
    print("Inference response:", response.message)

if __name__ == "__main__":
    run()
```

Block Load Balancer:

Block load balancer distributes the requests across multiple available instances of the AI/Computational module. The load balancing logic cannot be pre-defined and fixed since different types of AI/Computational workloads follow different latency, throughput, resource utilization patterns, hence the load balancing policy can be used to implement the custom load balancing logic appropriate for the given type of AI/Computational workloads. This load balancing policy should map the given input task to one of the available instances, data points like current block metrics, cluster metrics and the block configuration information will be provided to the load balancer policy.

Here is the basic structure of load balancer policy:

```
import random

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

"""
    get_metrics is a function injected by the load balancer environment into the loc
    get_metrics() can be called anytime to obtain the current block and cluster metr
    the block is running on, this metrics can be used for load balancing decision ma

    block_data: contains the data of the block, check the documentation of DBs to us
    cluster_data: contains the data of the cluster, check the documentation of DBs t

"""
self.metrics_function = self.settings["get_metrics"]
self.block_data = self.settings["block_data"]
self.cluster_data = self.settings["cluster_data"]

"""
    session_ids_cache is not mandatory, but this technique can be used for speeding
    load balancing computations or in cases where the instances run stateful computa
    The session_id from the job can be saved along with it's target instance as valu
    session_ids_cache and return the target instance_id value if it's found, like:

    packet = input_data["packet"]

    # packet is the proto object of AIOSPacket
    session_id = packet.session_id

    if session_id in self.session_ids_cache:
        return {
            "instance_id": self.session_ids_cache[session_id]
        }

"""
self.session_ids_cache = {}

"""
    This contains the list of instances that are currently running,
    the current instances list will be passed on every input to the eval() function,
    so when new instances are added/removed, the refreshed list is always passed.
    You can optionally save it as a class member, like:

    self.current_instances = input_data["instances"]
"""
```

```
        self.current_instances = []

    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

        Returns:
            dict: A dictionary with the evaluation result and possibly modified input_data.
        """
        """
            Check if new instances have been added or deleted
            input_data["instances"] = ["instance-0", "instance-1", ...]
        """
        if len(input_data["instances"]) != len(self.current_instances):
            """
                Reset the sessions cache (For example)
            """
            self.session_ids_cache.clear()

        """
            For example: check if the session_id is already found in the sessions cache:
        """
        packet = input_data["packet"]
        # packet is the proto object of AIOSPacket
        session_id = packet.session_id
        if session_id in self.session_ids_cache:
                return {
                    "instance_id": self.session_ids_cache[session_id]
                }

        """
            For example: The current metrics can be obtained by calling the `metrics_functi
            The metrics_function returns a dict:
            {
                "block_metrics": <block metrics of the current block and it's instances>,
                "cluster_metrics": <the metrics of the cluster in which the block is running
            }

            **Note**: For the structure of block_metrics and cluster_metrics, refer to the
        """
        current_metrics = self.metrics_function()
```

```
        """
            Do the actual computation here
        """
        random_instance_id = random.choice(input_data["instances"])

        """
            save the current instance_id for the session_id in cache:
        """
        self.session_ids_cache[session_id] = random_instance_id

        return {
            "instance_id": random_instance_id
        }

    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
```

Block Auto-scaler:

Block auto-scaler is deployed as a side-car which triggers the auto-scaler policy periodically based on the interval defined during the block creation (check Block creation spec). The auto-scaler policy is provided with the functions to obtain the current block and cluster metrics. The auto-scaler policy can then use this data to decide whether new instances should be added or the current instances should be removed because they are no longer in use.

```
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

        """
            get_metrics is a function injected by the load balancer environment into the loc
            get_metrics() can be called anytime to obtain the current block and cluster metr
            the block is running on, this metrics can be used for load balancing decision ma
        """
        self.metrics_function = self.settings["get_metrics"]


    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

            block_data: contains the data of the block, check the documentation of DBs to us
            cluster_data: contains the data of the cluster, check the documentation of DBs t

        Returns:
            dict: A dictionary with the evaluation result and possibly modified input_data.
        """

        """
            input_data contains block and cluster data respectively
        """

        block_data = input_data["block_data"]
        cluster_data = input_data["cluster_data"]

        # Rule logic goes here

        """
            Return values:
```

```
            case-1: No decisions made - continue as it is:
            ---
            return {
                "skip": True
            }

            case-2: Up-scale, i.e add more instances:
            ---
            return {
                "skip": False,
                "operation": "upscale",
                "instances_count": 2  # for example - create 2 more instances
            }

            case-3: Down-scale, i.e remove instances:
            ---
            return {
                "skip": False,
                "operation": "downscale",
                "instances_list": ["instance-0", "instance-3"] # for example - remove instan
            }
        """

    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
```

Block health checker:

Block health checker is a side-car similar to auto-scaler that collects the health data of instances for fixed intervals, then calls the health checker policy to take decisions. The health checker policy can implement the suitable action required - For example - notifying the developer via Slack, calling failure policy executor server or executing a scale command via cluster controller gateway etc.

Here is the flow of health-checker: 1. Health check initiates at fixed intervals. 2. For each instance in the block, calls `http://{pod_ip}:18001/health` to obtain the health information with a timeout. 3. If the API responds with a non-200 status or the connection times-out, the instance is considered to be un-healthy. 4. Calls the policy with the health data of each instance passed as a list. 5. The health check policy can decide to take action based on this health data.

```
import requests

def call_failure_policy(api_url: str, failure_policy_id: str, inputs: dict, parameters: dict
    try:
        url = f"{api_url}/executeFailurePolicy"
        payload = {
            "failure_policy_id": failure_policy_id,
            "inputs": inputs,
            "parameters": parameters
        }
        headers = {"Content-Type": "application/json"}

        response = requests.post(url, json=payload, headers=headers)
        response.raise_for_status()
        return response.json()
    except requests.RequestException as e:
        logging.error(f"Request failed: {e}")
        return {"success": False, "message": str(e)}
    except Exception as ex:
        logging.error(f"Unexpected error: {ex}")
        return {"success": False, "message": str(ex)}

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

        # block data will be passed in settings:
        self.block_data = self.settings["block_data"]
        # cluster data will be passed in the settings:
        self.cluster_data = self.settings["cluster_data"]

        # For example: maintain a counter to count the number of times instance was unhealth

    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

        Returns:
            dict: A dictionary with the evaluation result and possibly modified input_data.
        """

        """
            'health_check_data' is a dict of instance_id: True/False
            True indicates that the instance is healthy, False indicates that the instance
            Example:
            {
                "instance-00": True,
                "instance-01": True,
                "instance-02": False
            }
        """
        health_check_data = input_data["health_check_data"]

        """
            'instances' contains the list of instance_ids.
            Example: ["instance-00", "instance-01", ""instance-02"]
        """

        for instance, is_healthy in health_check_data.items():
            if not is_healthy:
                self.counter[instance] = self.counter.get(instance, 0) + 1

                # for example, if instance was unhealthy more than 3 times:
```

```
                if self.counter[instance] > 3:
                    call_failure_policy(
                        self.settings["failure_policy_server_api_url"],
                        self.settings["failure_policy_id"],
                        inputs={
                            "block_id": self.block_data['id'],
                            "cluster_data": self.cluster_data['id'],
                            "instance_id": instance
                        },
                        parameters={}
                    )

                    self.counter[instance] = 0


        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
```

Block Services:

When a block is created, an entry is created in the ingress of the cluster at the route: `/block/<block-id>`, thus the external parties can access the block using the URL:

`http://<cluster-public-url>/block/<block-id>`

For example, if the block-id is `block-771a8` and the `cluster-public-url` is `prasanna-home-cluster.clusters.aiosv1` then:

`http://prasanna-home-cluster.clusters.aiosv1/block/block-771a8`

Block exposes multiple services with multiple ingress routes and internal ports, here are the details of the various services: Here's a table based on your provided ports and descriptions:

Purpose	Internal Port	Ingress URL Suffix	Description
Redis	6379	N/A	Redis internal port for caching or coordination
gRPC inference	50051	/rpc	Executor proxy gRPC port used for submitting AI inference requests
Executor metrics	18000	/metrics	Executor metrics port
Executor management	18001	/executor	Executor control port for load balancer management requests
Mapping	5000	N/A	For internal use
Auto-scaler management	10000	/autoscaler	Autoscaler management port
Health-checker management	19001	/health-checker	Health checker management port

---

APIs:

Executor Management:

Executor management server provides API to execute load balancer policy management commands as explained below:

Endpoint: `/mgmt`
Method: `POST`
Description:

This endpoint handles various management actions for a load balancer system. The action is specified using the `mgmt_action` field in the JSON body.

Example curl Command:

```
curl -X POST http://<cluster-public-url>/block/<block-id>/executor/mgmt \
  -H "Content-Type: application/json" \
  -d '{
    "mgmt_action": "<action>",
    "mgmt_data": {}
  }'
```

Auto-scaler management:

Endpoint: `/mgmt`
Method: `POST`
Description:

This endpoint is part of the auto-scaler management server. It accepts a management action and its associated data.

Example curl Command:

```
curl -X POST http://<cluster-public-url>/block/<block-id>/autoscaler/mgmt \
  -H "Content-Type: application/json" \
  -d '{
    "mgmt_action": "<action>",
    "mgmt_data": {}
  }'
```

Health checker management:

Endpoint: `/mgmt`
Method: `POST`
Description:

This endpoint is part of the health-checker management server. It accepts a management action and its associated data.

Example curl Command:

```
curl -X POST http://<cluster-public-url>/block/<block-id>/health-checker/mgmt \
  -H "Content-Type: application/json" \
  -d '{
    "mgmt_action": "<action>",
    "mgmt_data": {}
  }'
```

Executor metrics:

The metrics of the executor can be obtained by calling the GET API on executor metrics endpoint, here is the curl command:

```
curl -X GET http://<cluster-public-url>/block/<block-id>/metrics
```