AIOS Instance SDK

The AIOS Instance SDK is a Python library designed as a foundational framework for implementing custom computational logic that operates as servable instances within a modular block architecture. It is general-purpose and supports a wide range of computational workloads, including AI model inference, web application backends, and more. AIOS instances can also establish outbound network connections to interact with external services.

Core Features of the AIOS Instance SDK

1. Execution logic can be modularized into distinct phases within a single module, such as preprocessing, inference, and postprocessing.
2. Custom management operations can be defined by implementing a management command handler callback, which is invoked by the SDK in response to specific management events.
3. Native batching support is provided via an integrated batcher module, enabling efficient batched execution of compatible workloads (e.g., batched object detection).
4. A built-in multiplexing (muxing) module enables the instance to synchronize and process multiple concurrent inputs, facilitating the construction of virtual Directed Acyclic Graphs (vDAGs).
5. AIOS instances are GPU-compatible, allowing seamless integration of GPU-accelerated computations.
6. Utility modules such as in-memory and Redis-backed state dictionaries are available, supporting stateful computational workflows.
7. Custom telemetry and performance metrics can be instrumented using the AIOS metrics module. These metrics can be leveraged by load balancers and auto-scaling controllers to drive intelligent traffic distribution and resource scaling decisions.
8. The AIOS LLM SDK extension augments the base SDK with specialized utilities for serving large language models (refer to the LLM serving documentation for further details).
9. Custom side-cars can be deployed along with the block, these side-cars are deployed as pods externally to extend the functionalities like external auditing, drawing, output saving etc.

How to build on top of the SDK?

1. First, install the library:

```
# from the project root:
cd services/block/instance
```

```
# install:
pip3 install -e .
```

2. Importing the library:**

```
from aios_instance import *
```

3. Create a custom class that agrees with the protocol:

Here is the structure of the class with explanation of the parameters:

```
class YourCustomBlock:
    def __init__(self, context):
        """
        Constructor for the custom block class.

        Parameters:
        - context (Context): An object containing:
            - metrics (AIOSMetrics): For registering and updating Prometheus metrics.
            - events (BlockEvents): For handling custom external events (check Cluster conti
            - block_data (dict): The full block configuration retrieved from the blocks data
            - block_init_data (dict): Initial configuration for the block (usually static).
            - block_init_settings (dict): Runtime settings for the block, possibly modified
            - block_init_parameters (dict): Modifiable runtime parameters (can be updated v
            - allocation_info (dict): {"gpus": [], "node_id": "", "cluster_id": ""}
            - sessions (SessionsManager): Object to maintain session-level data across requ

        This method is typically used to initialize models, load resources, or set up interr
        """
        self.context = context

        # Example: Initialize a threshold from block parameters
        self.threshold = context.block_init_parameters.get("threshold", 0.5)

    def on_preprocess(self, packet):
    """
    If muxing is enabled, `packet.data` will be a dict with key "inputs" that is a list of t

    You need to handle both:
    - packet.data = {"inputs": [...]}
    - packet.data = {...} (when no muxer is used)
    """
        try:
            data = packet.data
            if isinstance(data, str):
                data = json.loads(data)
```

```
            if "inputs" in data:
                # Handle merged packet (N > 1)
                results = []
                for input_data in data["inputs"]:
                    result = PreProcessResult(packet=packet, extra_data={"input": input_data
                    results.append(result)
                return True, results
            else:
                # Normal single input
                return True, [PreProcessResult(packet=packet, extra_data={"input": data})]
        except Exception as e:
            return False, str(e)

    def on_data(self, preprocessed_entry):
        """
        Performs the main data processing or model inference logic.

        Parameters:
        - preprocessed_entry (PreProcessResult): Contains:
            - packet (AIOSPacket): The input packet, possibly enriched in `on_preprocess`.
            - extra_data (dict): Additional metadata returned from `on_preprocess`.

        Returns:
        - (True, OnDataResult): On success.
            - OnDataResult contains:
                - output (dict): A JSON-serializable Python dictionary with the inference o
        - (False, str): On failure, returns a string with the error message.

        Notes:
        - The returned dictionary will be JSON-encoded and placed into the `data` field of
        - The `output_ptr` field in the packet determines where this result is forwarded.
        """
        try:
            # Example: Access original input
            input_data = json.loads(preprocessed_entry.packet.data)

            # Example output
            output = {
                "result": "inference complete",
                "input_summary": input_data,
                "processed_at": time.time(),
                "metadata": preprocessed_entry.extra_data
            }

            return True, OnDataResult(output=output)
```

```
        except Exception as e:
            return False, str(e)

    def on_update(self, updated_parameters):
        """
        Updates internal state of the block with new parameters.

        Parameters:
        - updated_parameters (dict): A dictionary of key-value pairs to update block behavi
            These may come from:
            - Redis parameter update queue
            - HTTP POST to `/setParameters`

        Returns:
        - (True, dict): On success, returns the updated parameters.
        - (False, str): On failure, returns an error message explaining the failure.

        Example use case:
        - Updating model thresholds, control flags, or runtime settings.
        """
        try:
            # Update internal parameter
            if "threshold" in updated_parameters:
                self.threshold = updated_parameters["threshold"]

            return True, updated_parameters
        except Exception as e:
            return False, str(e)

    def health(self):
        """
        Responds to `/health` HTTP endpoint to indicate readiness/liveness.

        Parameters:
        - None

        Returns:
        - Any JSON-serializable object indicating health status (e.g., dict or str).

        This method should not raise exceptions unless the block is truly unhealthy.
        """
        return {
            "status": "healthy",
            "timestamp": time.time()
        }
```

```python
def management(self, action, data):
    """
    Handles special management commands sent via the `/mgmt` endpoint.

    Parameters:
    - action (str): The management command (e.g., 'reset', 'pause', 'reload').
    - data (dict): Optional parameters associated with the command.

    Returns:
    - Any JSON-serializable object: Response payload indicating the result of the manag

    Example use cases:
    - Resetting internal state or counters.
    - Reloading models or resources.
    """
    try:
        if action == "reset":
            self.context.sessions.clear()
            return {"message": "Session state reset"}
        elif action == "status":
            return {"message": "All systems operational"}
        else:
            return {"message": f"Unknown action '{action}'"}
    except Exception as e:
        return {"error": str(e)}

def get_muxer(self):
    """
    Returns an optional Muxer instance to manipulate or filter packets before preproces

    Parameters:
    - None

    Returns:
    - Muxer: An instance of a Muxer subclass that implements `process_packet(raw_bytes)
    - None: If no muxing is required.

    Example use cases:
    - Filter out irrelevant packets.
    - Modify packet routing dynamically.
    - Split a large message into multiple logical packets.
    """
    return None  # Or return a custom Muxer instance

def get_muxer(self):
    """
```

```
        Returns:
        - An instance of `Muxer` to group packets with the same session ID and sequence numb

        Behavior:
        - If `merge_count` is 1, packets are passed through unchanged.
        - If `merge_count` > 1, packets are buffered and merged before being passed to `on_p

        Notes:
        - Ensure that your `on_preprocess` function handles packets with `data = {"inputs":
        """
        return Muxer(N=self.merge_count)
```

Register the class and instantiate:

Once the class is designed as per the specified structure, it needs to be registered and the instance must be started as specified below:

```
from aios_instance import Block

if __name__ == "__main__":
    block = Block(YourCustomBlock)
    block.run()
```

Building the container image:

1. Build the AIOS instance base image:

```
docker build . -f Dockerfile_base -t aios_instance:v1
```

2. Later this use base image to build custom images:

```
FROM aios_instance:v1

COPY . /app

WORKDIR /app

# install prerequisites and other dependencies

# install requirements
pip3 install -r requirements.txt

ENTRYPOINT ["python3", "main.py"]
```

Publishing and registering the component in component registry:

TODO

Got it — no emojis, just clean and professional documentation. Let's dive in.

Custom Block Class Developer Guide

This guide outlines how to implement a custom `block_class` that integrates with the AIOS runtime. The class will handle job packets (`AIOSPacket`), process them through your logic (e.g., preprocessing, inference), and return results downstream via Redis or gRPC, depending on the system configuration.

`AIOSPacket`: Job Payload Format

Each job that arrives at your block is serialized as an `AIOSPacket`. Here's the Protobuf definition with detailed field usage:

```
message AIOSPacket {
    string session_id = 1;      // Unique identifier for the session, enables stateful logic
    uint64 seq_no = 2;          // Monotonically increasing sequence number for this session
    string data = 4;            // JSON-encoded input payload; format depends on block logic
    double ts = 5;              // Optional Unix timestamp (used for latency metrics or ordering)
    string output_ptr = 6;      // JSON-encoded pointer to downstream blocks; defines the output
    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
}
```

Muxing and Batching: How Jobs Arrive

AIOS supports advanced job grouping via `Muxer`, `Batcher`, and `TimeBasedBatcher`.

Strategy	Description	Resulting `packet.data` format
`Muxer(N)`	Merges N packets with the same `(session_id, seq_no)` into one	`{"inputs": [<data1>, <data2>, ...]}`
`Batcher(N)`	Aggregates packets into a batch of N sequentially	`[AIOSPacket, ...]` (not merged)
`TimeBasedBatcher(N, T)`	Triggers a batch when N packets arrive or T seconds elapse	`[AIOSPacket, ...]` (not merged)

Only the `Muxer` modifies the structure of the packet's `.data`. Other batchers work at the scheduling level.

Required Methods in `block_class`

`__init__(self, context)`

This method is called once when the block is started. The `context` object includes:

- `context.block_data`: Block metadata from the Blocks DB.
- `context.block_init_data`: Initial static configuration.
- `context.block_init_parameters`: Runtime parameters that can be updated.
- `context.metrics`: Metric registry for gauges and counters.
- `context.sessions`: Session state manager.
- `context.events`: Object of the class `BlockEvents` used for pushing events to the external block events handler.

Use this method to load models, initialize internal caches, or register metrics.

`on_preprocess(self, packet)`

Called for every incoming job (or merged job, if using a `Muxer`).

Parameters:

- `packet (AIOSPacket)`: The job input. If `Muxer` is active, the `.data` field will be a dictionary like `{"inputs": [...]}`. Otherwise, it's a JSON string.

Returns:

- `(True, List[PreProcessResult])`: Each `PreProcessResult` wraps the original `packet` and any additional metadata in `extra_data`.
- `(False, "error message")`: If preprocessing fails.

Typical flow:

```
def on_preprocess(self, packet):
    try:
        data = packet.data
        if isinstance(data, str):
            data = json.loads(data)

        if "inputs" in data:
            # Muxed case: return multiple PreProcessResult instances
            return True, [
                PreProcessResult(packet=packet, extra_data={"input": item})
                for item in data["inputs"]
            ]
        else:
            # Normal single-packet case
            return True, [PreProcessResult(packet=packet, extra_data={"input": data})]
    except Exception as e:
        return False, str(e)
```

`on_data(self, preprocessed_entry)`

Called for each `PreProcessResult` from `on_preprocess`.

Parameters:

- `preprocessed_entry (PreProcessResult)`:
  - `packet`: The original `AIOSPacket`.
  - `extra_data`: A dictionary with custom metadata created in `on_preprocess`.

Returns:

- `(True, OnDataResult)`: The result is a JSON-serializable dictionary placed into the `data` field of the packet.
- `(False, "error message")`: To log and skip the packet if processing fails.

Typical flow:

```
def on_data(self, preprocessed_entry):
    try:
        result = {
            "processed_at": time.time(),
```

```
            "input": preprocessed_entry.extra_data["input"],
            "status": "ok"
        }
        return True, OnDataResult(output=result)
    except Exception as e:
        return False, str(e)
```

`on_update(self, updated_parameters)`

Handles updates to the block's parameters at runtime (via Redis or HTTP).

Parameters:

- `updated_parameters (dict)`: New key-value pairs to apply.

Returns:

- `(True, dict)`: On success.
- `(False, "error message")`: If the update is invalid or fails.

Example:

```
def on_update(self, updated_parameters):
    try:
        if "threshold" in updated_parameters:
            self.threshold = updated_parameters["threshold"]
        return True, updated_parameters
    except Exception as e:
        return False, str(e)
```

`health(self)`

Responds to health checks on the `/health` endpoint.

Returns: Any JSON-serializable object indicating block health.

```
def health(self):
    return {"status": "healthy"}
```

`management(self, action, data)`

Handles custom management operations (via `/mgmt` endpoint).

Parameters:

- `action (str)`: Management command (e.g., `"reset"`, `"status"`).
- `data (dict)`: Additional data for the command.

Returns: JSON-serializable result.

```python
def management(self, action, data):
    if action == "reset":
        self.context.sessions.clear()
        return {"message": "state reset"}
    return {"message": f"Unhandled action: {action}"}
```

`get_muxer(self)`

Returns an instance of `Muxer` to control packet merging.

Returns: `Muxer` instance or `None`.

```python
def get_muxer(self):
    return Muxer(N=4)  # Wait for 4 packets with same session_id + seq_no before merging
```

When `Muxer(N)` is used, the incoming `AIOSPacket.data` becomes:

```
{
  "inputs": [
    {"key": "value"},
    {"key": "value"},
    ...
  ]
}
```

Your `on_preprocess` logic must handle this structure.

Summary Table

Method	Purpose	Input	Output
`__init__`	Initialize state, models, config	`context`	None
`on_preprocess`	Prepare inputs	`AIOSPacket`	`(bool, List[PreProcessResult])`
`on_data`	Core inference or logic	`PreProcessResult`	`(bool, OnDataResult)`
`on_update`	Runtime parameter update	`dict`	`(bool, dict or str)`
`health`	Healthcheck for liveness probe	None	JSON-serializable health status

Method	Purpose	Input	Output
`management`	Runtime management commands	`action: str, data: dict`	JSON-serializable result
`get_muxer`	Optional packet merging	None	`Muxer` instance or `None`

Designing Custom Muxers and Batchers in AIOS

In the AIOS block system, job packets (`AIOSPacket`) can be **aggregated or merged** before being sent to the `on_preprocess()` stage. This is handled using optional components:

- **Muxer**: Groups multiple logically related packets (e.g. by session) and merges them into one.
- **Batcher**: Collects a fixed number of packets and emits them as a list.

These utilities help optimize inference throughput, enable multi-input processing, and reduce I/O overhead.

AIOSPacket: The Base Unit of Processing

Every job is encapsulated in an `AIOSPacket`. Here's the structure (from the protobuf definition):

```
message AIOSPacket {
    string session_id = 1;      // Used to group packets by logical session
    uint64 seq_no = 2;          // Sequence number within a session
    string data = 4;            // JSON-encoded data payload (can be merged into a list via
    double ts = 5;              // Optional timestamp (used for latency analysis)
    string output_ptr = 6;      // Optional routing structure in JSON
    repeated FileInfo files = 7;// Optional list of attached files
}

message FileInfo {
    string metadata = 1;        // JSON describing the file
    bytes file_data = 2;        // Raw bytes of the file
}
```

Muxer: Merging Logically Related Packets

A **Muxer** groups together `N` packets with the same `(session_id, seq_no)` and produces a single merged packet. It's typically used for combining inputs for multi-view models, ensembles, or grouped inference logic.

Class Overview

```python
class Muxer:
    def __init__(self, N: int):
        self.N = N
        self.store = defaultdict(list)
        self.counts = defaultdict(int)

    def process_packet(self, packet):
        if self.N == 1:
            return packet  # No merging needed

        key = (packet.session_id, packet.seq_no)
        self.store[key].append(packet)
        self.counts[key] += 1

        if self.counts[key] == self.N:
            merged_packet = self._merge_packets(self.store[key])
            del self.store[key]
            del self.counts[key]
            return merged_packet

        return None

    def _merge_packets(self, packets):
        merged_data = {"inputs": [json.loads(p.data) for p in packets]}
        merged_files = [file for p in packets for file in p.files]

        base_packet = copy.deepcopy(packets[0])
        base_packet.data = merged_data
        base_packet.files = merged_files

        return base_packet
```

Behavior

- **Input**: Individual `AIOSPacket` objects.
- **Output**: One merged `AIOSPacket` with `.data = {"inputs": [...]}` and `.files` merged.
- **Trigger**: Emits a merged packet when `N` matching packets are received.

Integration

In your `block_class`, provide it via:

```
def get_muxer(self):
    return Muxer(N=3)
```

In `on_preprocess`, check for `{"inputs": [...]}`:

```
def on_preprocess(self, packet):
    try:
        data = packet.data
        if isinstance(data, str):
            data = json.loads(data)

        if "inputs" in data:
            return True, [PreProcessResult(packet=packet, extra_data={"input": d}) for d in
        else:
            return True, [PreProcessResult(packet=packet, extra_data={"input": data})]
    except Exception as e:
        return False, str(e)
```

Batcher: Fixed-Count Packet Grouping

A **Batcher** collects `N` packets and emits them as a batch. It does not merge or modify them. This is useful for model inference methods that accept multiple inputs at once (e.g., transformers, image batches).

Class Definition

```
class Batcher:
    def __init__(self, N: int):
        self.N = N
        self.batch = []

    def add_to_batch(self, packet):
        self.batch.append(packet)

        if len(self.batch) >= self.N:
            batch_copy = self.batch
            self.batch = []
            return batch_copy

        return None
```

Behavior

- **Input**: Individual `AIOSPacket` objects.
- **Output**: List of `N` packets.
- **Trigger**: Emits a batch when exactly `N` packets are collected.

Usage Pattern

This pattern works best when you implement **manual batching logic** in your block loop. For example:

```
def listen_for_jobs(self):
    while True:
        packet = self.fetch_from_redis()
        batch = self.batcher.add_to_batch(packet)

        if batch:
            for p in batch:
                # process each packet in a loop
```

Unlike the `Muxer`, the `Batcher` is not automatically plugged into the AIOS pipeline — you manage it explicitly.

Choosing Between Muxer and Batcher

Feature	Muxer	Batcher
Output	1 merged `AIOSPacket`	List of `AIOSPacket`
Trigger	N packets with same session & seq_no	Any N packets
Modifies packet	Yes (`data → {"inputs": [...]}`)	No
Internal state	Maintains group-by buffers	Simple FIFO buffer
Use cases	Session-wise aggregation	Mini-batching, multi-sample input

---

Recommendations

- Use a **Muxer** if your block expects to merge multiple views/inputs into a single job (e.g., a multi-camera model).
- Use a **Batcher** if your block supports vectorized inference and can handle a list of jobs at once.
- Always handle edge cases: incomplete merges, invalid JSON, oversized batches, and session expiry.

Metrics in AIOS Blocks

The `AIOSMetrics` class provides Prometheus-compatible monitoring for AIOS blocks. Each block instance can define, update, and expose performance and operational metrics — both via HTTP (for Prometheus scraping) and Redis (for centralized collection). These metrics help you measure things like throughput, latency, queue size, and model performance.

Metric Types

Metric Type	Description
Counter	Tracks a cumulative total that only increases (e.g., number of processed images).
Gauge	Represents a value that may increase or decrease (e.g., queue size, FPS).
Histogram	Records samples and bucketizes them (e.g., latency distribution).

Registering Metrics

Each metric must be explicitly registered before being used.

`register_counter(name, documentation, labelnames=None)`

Registers a new counter.

```
metrics.register_counter(
    "images_processed_total",
    "Total number of images processed by the block"
)
```

`register_gauge(name, documentation, labelnames=None)`

Registers a new gauge.

```
metrics.register_gauge(
    "input_queue_length",
    "Number of pending jobs in the Redis queue"
)
```

register_histogram(name, documentation, labelnames=None, buckets=None)

Registers a histogram. If not specified, default latency buckets are used.

```
metrics.register_histogram(
    "inference_latency_seconds",
    "Latency of inference step in seconds",
    buckets=[0.05, 0.1, 0.2, 0.5, 1, 2]
)
```

Updating Metrics

Once registered, metrics can be updated during processing.

increment_counter(name)

Increments a counter by one.

```
metrics.increment_counter("images_processed_total")
```

set_gauge(name, value)

Sets a gauge to a specific value.

```
metrics.set_gauge("input_queue_length", queue_length)
```

observe_histogram(name, value)

Adds a value sample to a histogram.

```
metrics.observe_histogram("inference_latency_seconds", latency)
```

Application Example: Object Detection Block

Consider an object detection block that receives images and returns bounding boxes. Here's how metrics can be defined and used:

Registration in `__init__`

```
metrics.register_counter("images_processed_total", "Total number of images processed")
metrics.register_counter("detections_emitted_total", "Total number of detected objects")
metrics.register_gauge("input_queue_length", "Jobs in Redis input queue")
metrics.register_histogram("inference_latency_seconds", "Time taken for model inference")
metrics.register_histogram("objects_per_image", "Number of detections per image", buckets=[0
```

Usage in processing loop

```
# Count the image
metrics.increment_counter("images_processed_total")

# Measure inference time
start = time.time()
results = model.detect_objects(image)
latency = time.time() - start
metrics.observe_histogram("inference_latency_seconds", latency)

# Record detection statistics
metrics.increment_counter("detections_emitted_total")
metrics.observe_histogram("objects_per_image", len(results))

# Queue monitoring
queue_length = redis_client.llen("block_inputs")
metrics.set_gauge("input_queue_length", queue_length)
```

BlockEvents

The `BlockEvents` class provides a Redis-backed interface for emitting structured runtime events from an AIOS block. It is accessible via `context.events`, these events are consumed by the custom block events handler in Cluster controller gateway.

Each event is pushed to a Redis queue (default: `"EVENTS_QUEUE"`) as a JSON payload containing: - `event_name`: A string representing the type of event. - `block_id`: Automatically filled from the `BLOCK_ID` environment variable. - `event_data`: Any additional metadata or contextual information you want to include.

Example Usage

Pushing an Inference Start Event

```
self.context.events.push_event(
    event_name="inference_started",
    event_data={
        "session_id": packet.session_id,
        "seq_no": packet.seq_no,
        "ts": packet.ts
    }
)
```

BlockSideCars: Sidecar Communication in AIOS

The `BlockSideCars` class enables a block to interact with auxiliary sidecar services via Redis queues. These sidecars may include preprocessing workers, feature generators, model servers, or any block-specific helper components running as separate services.

Sidecars are addressed by name and pushed data is serialized using Protobuf. You can configure multiple sidecars per block and send messages to one or all of them using the provided interface.

The object is available inside every block as `context.side_cars`.

Method: `push_to_sidecar(name, input_proto)`

Pushes a serialized Protobuf message to a specific named sidecar's Redis queue (`INPUTS`).

Parameters

Name	Type	Description
`name`	`str`	The name of the sidecar (must match the keys in the sidecars config).
`input_proto`	`google.protobuf.Message`	Job packet

Behavior:

- Resolves the Redis URL for the sidecar using naming conventions:
  - Internal: `<name>-<block_id>-svc.sidecars.svc.cluster.local`
  - External: Uses `external_redis_url` from the config if `external: true`.
- Serializes the input using `.SerializeToString()`.
- Pushes the message to the Redis list `"INPUTS"`.
- On failure, retries connection once before giving up.

Example:

```
# Push a packet to the "feature_generator" sidecar
self.context.side_cars.push_to_sidecar(
    name="feature_generator",
```

```
    input_proto=packet  # Must be a protobuf message (e.g. AIOSPacket)
)
```

Method: `push_to_all_sidecars(input_proto)`

Broadcasts a single Protobuf message to all configured sidecars.

Parameters

Name	Type	Description
`input_proto`	`google.protobuf.Message`	The job packet

Behavior:

- Iterates over all sidecar names in the configuration.
- Calls `push_to_sidecar()` for each one.
- Logs errors but continues to attempt sending to all sidecars even if one fails.

Example:

```
# Push the packet to all registered sidecars
self.context.side_cars.push_to_all_sidecars(input_proto=packet)
```

Sample implementation:

This example shows: - Initialization - Preprocessing - Processing - Handling runtime parameter updates - Health checks - Management actions

---

```
class SimpleBlock:
    def __init__(self, context):
        """
        Initializes the block with initial parameters.
        """
        self.context = context
        self.threshold = context.block_init_parameters.get("threshold", 0.5)

    def on_preprocess(self, packet):
        """
        Parses the JSON data from the packet.
        """
        try:
```

```
            data = json.loads(packet.data)
            return True, [PreProcessResult(packet=packet, extra_data={"input": data})]
        except Exception as e:
            return False, str(e)

    def on_data(self, preprocessed_entry):
        """
        Processes the input and returns a result based on a simple threshold.
        """
        try:
            input_data = preprocessed_entry.extra_data["input"]
            value = input_data.get("value", 0)

            result = {
                "value": value,
                "passed": value > self.threshold
            }

            return True, OnDataResult(output=result)
        except Exception as e:
            return False, str(e)

    def on_update(self, updated_parameters):
        """
        Updates the threshold value dynamically.
        """
        try:
            if "threshold" in updated_parameters:
                self.threshold = updated_parameters["threshold"]
            return True, updated_parameters
        except Exception as e:
            return False, str(e)

    def health(self):
        """
        Returns basic health status.
        """
        return {"status": "healthy"}

    def management(self, action, data):
        """
        Handles simple management commands.
        """
        if action == "get_threshold":
            return {"threshold": self.threshold}
        elif action == "reset":
```

```
            self.threshold = 0.5
            return {"message": "Threshold reset to default"}
        else:
            return {"message": f"Unknown action: {action}"}

    def get_muxer(self):
        """
        No muxing needed for this example.
        """
        return None
```

Extending the example with Muxer and Batcher:

```
from collections import deque
from .tools import Muxer, Batcher

class BlockWithMuxerAndBatcher:
    def __init__(self, context):
        """
        Initializes the block with parameters and sets up both a Muxer and a Batcher.

        - context: The standard AIOS block context containing config, sessions, and metrics
        - threshold: A configurable float parameter used for logic checks.
        - batcher: Collects N outputs from on_data before emitting a final result.
        """
        self.context = context
        self.threshold = context.block_init_parameters.get("threshold", 0.5)

        # Create a batcher to group 4 processed packets before final output
        self.batcher = Batcher(N=4)

    def on_preprocess(self, packet):
        """
        Parses the packet and handles muxed or single-packet inputs.

        - packet: An AIOSPacket; may contain either a single JSON string or a merged dict w

        Returns:
        - (True, List[PreProcessResult]) if parsing is successful.
        - (False, error message) if parsing fails.
        """
        try:
            data = packet.data
            if isinstance(data, str):
                data = json.loads(data)
```

```
            # If data has been muxed, it will have a list under the 'inputs' key
            if "inputs" in data:
                return True, [
                    PreProcessResult(packet=packet, extra_data={"input": item})
                    for item in data["inputs"]
                ]
            else:
                # Normal case: just one input
                return True, [PreProcessResult(packet=packet, extra_data={"input": data})]
        except Exception as e:
            return False, str(e)

    def on_data(self, preprocessed_entry):
        """
        Processes one PreProcessResult and appends the output to the Batcher.

        - preprocessed_entry: Contains original packet and extracted input.
        - If a batch is ready (size N), it emits all results as a single dictionary.
        - If not, returns None to indicate hold-back.

        Returns:
        - (True, OnDataResult) when a batch is emitted.
        - (True, None) when batch is still collecting.
        - (False, error message) on failure.
        """
        try:
            input_data = preprocessed_entry.extra_data["input"]
            value = input_data.get("value", 0)

            # Apply threshold logic
            result = {"value": value, "passed": value > self.threshold}

            # Update the original packet's data field
            packet = preprocessed_entry.packet
            packet.data = json.dumps(result)

            # Add to batcher and check if batch is full
            batch = self.batcher.add_to_batch(packet)

            if batch:
                # Emit the complete batch
                return True, OnDataResult(output={"batch": [json.loads(p.data) for p in batc

            # No output yet; waiting for batch to fill
            return True, None
```

```
        except Exception as e:
            return False, str(e)

    def on_update(self, updated_parameters):
        """
        Dynamically updates block configuration (e.g., threshold).

        - updated_parameters: Dictionary of parameters to apply.

        Returns:
        - (True, dict) if update was applied.
        - (False, error message) on failure.
        """
        try:
            if "threshold" in updated_parameters:
                self.threshold = updated_parameters["threshold"]
            return True, updated_parameters
        except Exception as e:
            return False, str(e)

    def health(self):
        """
        Responds to /health endpoint with status information.

        Returns:
        - Dictionary representing the current health status of the block.
        """
        return {"status": "healthy"}

    def management(self, action, data):
        """
        Handles runtime management commands sent to /mgmt endpoint.

        - action: A string representing the management command.
        - data: Optional dictionary of command-specific parameters.

        Returns:
        - Dictionary describing the result of the command.
        """
        if action == "get_threshold":
            return {"threshold": self.threshold}
        elif action == "reset":
            self.threshold = 0.5
            return {"message": "Threshold reset to default"}
        else:
            return {"message": f"Unknown action: {action}"}
```

```
def get_muxer(self):
    """
    Returns a Muxer instance to enable merging of N packets with same session ID and se

    - This Muxer collects 3 packets before forwarding them as one combined packet.

    Returns:
    - Muxer instance
    """
    return Muxer(N=3)
```

Testing framework:

The `BlockTester` class is a lightweight testing utility designed to simulate the full lifecycle of a block — from input packet creation to preprocessing and data processing — **without needing Redis, metrics, batching infrastructure, or Flask servers**.

It is useful for writing unit tests, debugging block logic, or validating preprocessing and inference behavior locally.

1. Create a BlockTester Instance

To quickly test a block with default settings:

```
from your_blocks import YourBlockClass
tester = BlockTester(YourBlockClass)
```

This will: - Create a mock context (`TestContext`) - Instantiate your block with it - Prepare the tester for running

---

2. Run Inference with a Simple Input

```
result = tester.run({"value": 0.9})
print(result)
```

- The input dictionary will be converted into an `AIOSPacket`.
- `on_preprocess` and `on_data` are executed in sequence.
- Output from `on_data` is collected and returned as a list of dictionaries.

---

3. Attach Files to the Input Packet

```
test_files = [
    ({"filename": "image.jpg", "type": "jpeg"}, b"\xff\xd8\xff...")
]

result = tester.run(
    data={"value": 1.0},
    files=test_files
)
```

Each file is defined by a tuple: - The first element is a `metadata` dictionary (will be JSON-encoded). - The second is the raw file content as `bytes`.

All files are attached to the `files` field in the packet, just like in production.

Using a Custom Context

For advanced testing, you can initialize the tester with a pre-built context:

```
from your_blocks import YourBlockClass
from your_runtime import AIOSMetrics, SessionsManager, BlockEvents

context = TestContext()
context.block_init_parameters = {"threshold": 0.75}
context.metrics = AIOSMetrics()
context.sessions = SessionsManager()
context.events = BlockEvents()

tester = BlockTester.init_with_context(YourBlockClass, context)

result = tester.run({"value": 0.6})
```

This allows you to: - Override block parameters - Attach fake metrics or session state - Inject mock event publishers

---

Output Format

The `run()` method returns a list of processed results from `on_data`. Each result is a JSON-parsed Python dictionary:

```
[
  {"value": 0.9, "passed": True}
]
```

If your block batches results (e.g. with `Batcher`), the result may look like:

```
[
  {
    "batch": [
      {"value": 0.6, "passed": True},
      {"value": 0.4, "passed": False},
      ...
    ]
  }
]
```

BlockTester helps you develop and debug AIOS blocks with confidence — without needing the full runtime infrastructure.