Third-Party Instances

Third-party systems can be integrated as part of an AIOS Block. In a typical block lifecycle, a component container image—built using the AIOS Instance SDK—is onboarded into the component registry and then instantiated as a block. This block includes management services such as the executor, autoscaler, and metrics system, and it spins up the component containers as pods known as "instances." These instances can scale based on demand.

However, in the case of a third-party instance block, an external stand-alone system is deployed alongside the block's management services and instances. This third-party system, such as an inference server, can run across multiple nodes within the same cluster. It manages its own lifecycle independently. The block will redirect the tasks assigned to it to this external service, the instances of the block takes care of converting the AIOS task input to the external system specific input and vice versa in case of output.

Using this approach, an AIOS block can be linked with an external third-party system that performs the actual computation.

When Are Third-Party Instances Useful?

1. When the application logic is too complex to be efficiently implemented as an AIOS instance.
2. When your application needs to run across multiple nodes in the cluster, whereas an AIOS instance is limited to a single node.
3. When the application consists of a complex set of distributed services that manage their own scaling, lifecycle, and orchestration, which differ significantly from how AIOS operates.

Deploying a third party system along with the block:

Third party services can be linked to a block in two ways:

Here's a cleaned-up and more readable version of your section, followed by a comparison table to make it even easier to understand:

Deploying a Third-Party System Alongside a Block

Third-party services can be integrated with an AIOS block in two ways:

1. **Externally Managed**:
   In this approach, the third-party system is deployed separately by the developer—either within the same cluster or on an external environment. The internal or public URL of this system is passed as a parameter to the

block instance. The block instance then implements the logic to interact with this external system, effectively acting as a proxy.

2. **Managed by AIOS**:
   In this setup, AIOS takes responsibility for deploying and managing the third-party system. When the block is created, an **init container** (provided by the developer) runs as a pod to set up the third-party infrastructure. During block removal, the same init container is triggered to uninstall the system. This third-party system runs on the same cluster as the block and cannot be deployed externally. As in the first case, the block instance acts as a proxy to interact with the system.

Comparison Table

Feature	Externally Managed	Managed by AIOS
Deployment Responsibility	Developer	AIOS (via developer-provided init container)
Location	Anywhere (same cluster or external)	Same cluster as the block
Lifecycle Management	Handled by the developer	Fully managed by AIOS
Instance Role	Acts as a proxy to external system	Acts as a proxy to locally deployed system
Flexibility	High (can connect to any endpoint)	Limited to same cluster deployment
Setup Complexity	Developer-defined external setup	Requires a correctly configured init container

Note: In both the cases, AIOS block services doesn't manage the external system's internal load balancing and autoscaling of it's internal components.

Init container SDK:

Init container SDK can be used to build the init container which is used for deploying a third party system managed by AIOS, here is the structure of the SDK:

Certainly. Here's a detailed, formal documentation of the `InitContainerExecutor` class and the structure of the init container, including the execution flow and method responsibilities.

Init Container Execution Framework

The AIOS Init Container Execution Framework is responsible for managing the lifecycle of a third-party system when deployed as part of an AIOS block. This framework enables developers to define custom initialization, execution, and cleanup logic through a standardized interface.

Overview

The init container runs as a Kubernetes pod during block creation and deletion. It is driven by the `InitContainerExecutor` class, which orchestrates the lifecycle in three phases:

1. **Initialization (`begin`)**
2. **Main Execution (`main`)**
3. **Finalization (`finish`)**

Each phase is expected to return a success/failure status and a message.

Class: `InitContainerExecutor`

Constructor

```python
def __init__(self, container_class):
```

Initializes the executor by:

1. Fetching environment variables using `get_envs()`.
2. Fetching block and cluster metadata from the respective backends:
   - `BlocksDB().get_block_by_id(...)`
   - `ClusterClient().read_cluster(...)`
3. Establishing a Kubernetes connection (`K8sConnection`).
4. Initializing a cluster controller (`ClusterController`).
5. Instantiating the user-defined container class with the following arguments:
   - `envs`: Dictionary of environment variables.
   - `block_data`: Metadata related to the block.
   - `cluster_data`: Metadata related to the cluster.
   - `k8s`: K8sConnection instance.
   - `operating_mode`: Indicates the mode of operation (e.g., `create`, `delete`).

If any step fails, an exception is raised immediately.

Method: `execute()`

```python
def execute(self):
```

Executes the following steps in order:

1. `begin()`

- Calls `self.executor_class.begin()`
- Should perform any preparatory tasks, such as validating resources or preparing directories.
- On success, sends a status update with stage = `"begin"` and logs the message.

2. `main()`

- Calls `self.executor_class.main()`
- Should perform the core setup or teardown logic (e.g., deploying services, setting up infrastructure).
- On success, sends a status update with stage = `"main"` and logs the message.

3. `finish()`

- Calls `self.executor_class.finish()`
- Should clean up temporary resources or finalize the deployment.
- On success, sends a status update with stage = `"finish"` and logs the message.

If any step fails, a status update with `"failed"` is sent and the error is logged. The exception is then raised.

Helper Function: `execute_init_container`

```
def execute_init_container(main_class):
```

- This is the entry point for running the init container.
- It initializes the `InitContainerExecutor` with the user-defined `main_class` and calls `execute()`.
- Any exception raised results in process termination using `os._exit(0)`.

Example Implementation: `SampleContainerClass`

The class passed to `InitContainerExecutor` must implement three methods:

```
class SampleContainerClass:
    def __init__(self, envs, block_data, cluster_data, k8s, operating_mode="create"):
        ...

    def begin(self):
        return True, "Initialization successful."
    def main(self):
        # custom data can be sent as status
        return True, {}

    def finish(self):
        # custom data can be sent as status
        return True, {
            # for example: send the 'server_url' - URL of the server for the block to commu
            "server_url": "<url of the server>",
            # for example: send the 'metrics_url' - URL of the server for the metrics to be
            "metrics_url": "<url of the metrics url>"
        }
```

Method Descriptions

- `__init__`: Stores all contextual data required to manage the third-party system.
- `begin()`: Optional preparations before actual deployment/removal.
- `main()`: Core logic to set up or tear down the third-party system.
- `finish()`: Optional cleanup or confirmation tasks.

Each method must return a tuple: `(success: bool, message: str)`

Execution Flow Summary

1. `execute_init_container(SampleContainerClass)` is called.
2. `InitContainerExecutor.__init__()` gathers context and initializes the container class.
3. `execute()` runs the following in order:
    - `SampleContainerClass.begin()`
    - `SampleContainerClass.main()`
    - `SampleContainerClass.finish()`
4. After each stage, a status update is sent via the `ClusterController`.
5. On failure at any stage, a `"failed"` status is reported and execution stops.

This framework ensures that the deployment and removal of third-party systems are consistent, traceable, and integrated into the broader AIOS block lifecycle.

Here is the official-style documentation for the Kubernetes and Helm utility classes, under a new section:

Using Kubernetes Client and Helm Client in the Init Container

When developing custom init containers to deploy or remove third-party systems, it is common to interact with the Kubernetes API and Helm charts. The AIOS init container execution framework provides utility classes for both:

1. `K8sConnection`: For interacting directly with the Kubernetes API.
2. `HelmSubprocess`: For executing Helm commands as subprocesses.

These utilities simplify integration with the cluster and help developers focus on the application-specific logic.

Class: `K8sConnection`

This class encapsulates the logic required to connect to the Kubernetes API server. It supports both in-cluster and external configurations (via `kubeconfig`).

Constructor

```
def __init__(self):
    self.configuration = None
    self.api_client = None
```

Method: `connect()`

Attempts to establish a connection to the Kubernetes cluster using the following fallback mechanism:

1. **In-Cluster Configuration**: Tries to load the configuration from within a Kubernetes environment.
2. **Kubeconfig File**: If in-cluster config fails, falls back to `~/.kube/config`.

Raises a `K8sConnectionError` if both methods fail.

Method: `get_api_client()`

```
def get_api_client(self):
    return self.api_client
```

Returns an instance of `client.ApiClient`, which can be used to initialize various Kubernetes API objects (e.g., `CoreV1Api`, `AppsV1Api`, etc.).

Example Usage

```
k8s = K8sConnection()
k8s.connect()
```

```python
core_v1 = client.CoreV1Api(k8s.get_api_client())
pods = core_v1.list_namespaced_pod(namespace="default")
```

Use Cases for `K8sConnection`

- Creating, reading, or deleting Kubernetes resources (pods, services, config maps, etc.)
- Querying the cluster state
- Monitoring pods deployed as part of a third-party system
- Creating or tearing down infrastructure programmatically

Class: `HelmSubprocess`

This class wraps the `helm` CLI and provides utility methods to run Helm commands using Python subprocesses.

Constructor

```python
def __init__(self, helm_command='helm'):
    self.helm_command = helm_command
```

Optionally accepts a custom Helm binary path.

Method: `run_helm_command(args: List[str])`

```python
def run_helm_command(self, args):
```

Executes the Helm command using a list of arguments (e.g., `["install", "my-release", "my-chart"]`). Captures and logs the output.

Returns the standard output of the command if successful.

Raises `HelmSubprocessError` on failure.

Method: `run_helm_command_string(arg_string: str)`

```python
def run_helm_command_string(self, arg_string):
```

Accepts a single string and internally splits it into arguments using `shlex.split()`. Useful when constructing commands dynamically as strings.

Returns the standard output of the command.

Raises `HelmSubprocessError` on failure.

Example Usage

```python
helm = HelmSubprocess()

# Using list of arguments
helm.run_helm_command(["install", "my-release", "my-chart", "--namespace", "third-party"])

# Using string command
helm.run_helm_command_string("install my-release my-chart --namespace third-party")
```

Use Cases for `HelmSubprocess`

- Installing or uninstalling third-party Helm charts during block creation/removal
- Upgrading or rolling back Helm releases
- Querying release status or Helm-managed resources
- Integrating with any third-party system that is packaged as a Helm chart

---

Recommendations

- Always validate Helm chart input and output during development.
- Use consistent release naming conventions to ensure predictable behavior on upgrade or uninstall.
- Log command outputs and errors to aid in debugging and block status tracking.

These utilities ensure that init containers have powerful and flexible tooling to manage infrastructure cleanly, reliably, and in line with the AIOS block lifecycle.

Example init container code:

```python
import logging
from k8s_connection import K8sConnection
from helm_subprocess import HelmSubprocess
from typing import Dict

class ThirdPartyServiceInitContainer:
    def __init__(self, envs: Dict, block_data: Dict, cluster_data: Dict, k8s: K8sConnection
        self.envs = envs
        self.block_data = block_data
        self.cluster_data = cluster_data
        self.k8s = k8s
        self.operating_mode = operating_mode
        self.helm = HelmSubprocess()
        self.release_name = f"{block_data.get('block_id', 'default-block')}-triton"
        self.namespace = "third-party"
        self.chart_path = envs.get("helm_chart_path", "./charts/triton-inference-server")

    def begin(self):
        logging.info("Running begin() step of init container.")
        try:
            self.k8s.connect()
            logging.info("Kubernetes client initialized successfully.")
            return True, "Kubernetes client initialized"
        except Exception as e:
            logging.error(f"Error during begin(): {e}")
            return False, f"Kubernetes connection error: {str(e)}"

    def main(self):
        logging.info(f"Running main() step in {self.operating_mode} mode.")

        try:
            if self.operating_mode == "create":
                self.helm.run_helm_command([
                    "install", self.release_name, self.chart_path,
                    "--namespace", self.namespace,
                    "--create-namespace"
                ])
                message = f"Triton Inference Server '{self.release_name}' installed successf

            elif self.operating_mode == "delete":
                self.helm.run_helm_command([
                    "uninstall", self.release_name,
                    "--namespace", self.namespace
                ])
                message = f"Triton Inference Server '{self.release_name}' uninstalled succes

            else:
                raise ValueError(f"Unknown operating mode: {self.operating_mode}")

            logging.info(message)
            return True, message

        except Exception as e:
            logging.error(f"Error during main() in {self.operating_mode} mode: {e}")
            return False, str(e)

    def finish(self):
        logging.info("Running finish() step of init container.")
        try:
            # Construct internal cluster URLs
            service_name = f"{self.release_name}-service"
            dns_base = f"{service_name}.{self.namespace}.svc.cluster.local"

            result = {
                "inference_url": f"http://{dns_base}:8000/v2/models",
                "metrics_url": f"http://{dns_base}:8002/metrics"
            }

            logging.info(f"Finish step completed. Service DNS: {dns_base}")
            return True, result

        except Exception as e:
            logging.error(f"Error during finish(): {e}")
            return False, {"error": str(e)}
```

Querying the status on Init container:

The status of init containers deployed for blocks can be queried and tracked, the status of each init container created for the block, here is the schema used to store the status information:

```
@dataclass
class InitContainerEntry:
    # id will be same as the block id
    id: str = field(default_factory=lambda: str(uuid.uuid4()))

    # Stores metadata about the block associated with this init container run.
    # This typically includes the block ID, name, and other context from BlocksDB.
    block_data: Dict[str, Any] = field(default_factory=dict)

    # Contains detailed information about the execution status.
    # This might include logs, return messages, or any outputs from each stage.
    status_data: Dict[str, Any] = field(default_factory=dict)

    # Represents the current execution phase of the init container.
    # Valid values:
    #   "begin"  - Initial preparation logic is in progress or completed.
    #   "main"   - Main deployment logic is in progress or completed.
    #   "finish" - Final cleanup or post-deployment logic is in progress or completed.
    #   "failed" - One of the steps failed.
    status: str = ''
```

API to query status of init container:

Cluster controller gateway APIs can be used for querying the status of init containers:

Description:
Queries init container execution entries for a specific cluster. The query payload should follow MongoDB query format (e.g., using keys like `$and`, `$or`, field-based filters, etc.). `<cluster_id>` is the id of the cluster on which the init container of the block is running.

Endpoint:
`POST /initContainer/queryInitContainerData/<cluster_id>`

Curl Command:

```
curl -X POST http://<server-url>/initContainer/queryInitContainerData/<cluster_id> \
     -H "Content-Type: application/json" \
     -d '{
           "status": "failed"
         }'
```

Parameters:

- `cluster_id` (path): The unique identifier of the cluster for which the init container data is being queried.

Request Body:
A JSON object containing the MongoDB-style query. Example:

```
{
  "status": { "$in": ["failed", "begin"] },
  "block_data.id": "block-123"
}
```

Response:
Returns a list of init container entries that match the query.