ArmoniK SDKs

Formation Natixis

ANEO

ArmoniK Team

November 2025

Outline

Section 1

Introduction

ArmoniK Reminders

- ArmoniK implements a **Client–Worker** distributed processing model.
- It provides language-idiomatic SDKs to build scalable distributed applications.
- Responsibilities are clearly separated:
  - **Client Responsibilities**
    - Create and manage sessions.
    - Submit tasks and define execution dependencies (task graphs).
    - Manage input/output blobs and error handling.
  - **Worker Responsibilities**
    - Implement the computation logic for each task.
    - Read input blobs and produce output blobs.
    - Optionally submit new tasks (dynamic workflows).

SDKs Goals

- **Simplify** the development of Clients and Workers.
- Provide **pre-built Worker containers** following unified conventions.
- Offer a **high-level Worker API** with consistent lifecycle and error handling.
- Provide **simplified Client APIs** to create sessions, submit tasks, and manage data.
- Promote **best practices**: idempotency, error reporting, observability.
- Ensure **compatibility and long-term maintainability** across ArmoniK versions.
- Ensure **interoperability**: clients and workers can be mixed across languages.

Interoperability

- **Goal**: Enable cross-language interoperability.
  - Mix clients and workers across SDK languages.
  - Example: Java client orchestrates tasks executed by Python or C# workers.
- **Convention 1**: Payload structure
  - JSON association table mapping logical names to blob IDs.
  - Example: `{"inputs": {"A": "id1"}, "outputs": {"result": "id2"}}`
- **Convention 2**: Task options
  - Enable dynamic library loading via standardized task options.
  - Path, symbol, and blob ID for the library.

SDKs and Training

- SDKs are available in several languages:
  - C++
  - C#
  - Java
  - Python
- Each SDK provides:
  - Pre-built Worker containers.
  - A high-level Worker API to implement computation logic.
  - A simplified Client API to create sessions, submit tasks, and manage results.
- Dedicated training sessions are available for onboarding and advanced usage.

Section 2

Client SDK

Role of the Client SDK

- Provide a language-friendly interface to interact with ArmoniK's control plane.
- Create and manage **sessions**.
- Define and submit **tasks**, optionally with dependencies.
- Create and manage **input and output blobs**.
- Monitor task progress and retrieve results.
- Offer optional asynchronous event or callback mechanisms.
- Ensure consistent workflow semantics:
  - retries,
  - error propagation,
  - idempotency.

Task Submission Flow

1. The client creates a **session** to group related tasks.
2. Input data is uploaded as **blobs**.
3. The client defines one or more **task definitions**, each referencing:
   - input blobs (data dependencies),
   - expected output blobs,
   - optional task configuration.
4. Tasks are submitted to the control plane.
5. Workers execute tasks, read inputs, and produce outputs.
6. The control plane signals task completion.
7. The client retrieves output blobs or submits new tasks to continue the workflow.

Outcome: A clear, high-level flow usable by any SDK (C#, Java, C++, Python).

Section 3

Worker SDK

Role of the Worker SDK

- Provide a simple, language-friendly API to implement task computation logic.
- Expose a **task context** to read inputs, write outputs, and access metadata.
- Abstract away low-level gRPC communication and control-plane interactions.
- Ensure a consistent execution lifecycle across languages and worker modes.
- Goal: make Workers easy to write, test, deploy, and reuse.

Worker Core Responsibilities

- Receive a task request from the control plane.
- Access input data through the task context.
- Execute the user-defined computation logic.
- Produce output data and write it to output blobs.
- Maintain execution context (inputs, metadata, logs).
- Report the final task outcome (success or failure).

Dynamic and Flexible Workers

- Workers can operate in two modes:
  - **Static Mode**: computation logic is embedded into the worker container.
  - **Dynamic Mode**: computation logic is dynamically loaded at runtime.
- Dynamic loading enables:
  - Deploying multiple processing implementations without rebuilding the container.
  - Isolating user code via dedicated class loaders or library loaders.
  - Flexible per-task execution based on **WorkerLibrary** metadata.
- Supports advanced features such as:
  - Subtask submission (dynamic graphs).
  - Output delegation (workers generating tasks beyond the initial graph).

Error Handling and Reporting

- Workers must uniformly handle:
  - **Successful** task completion.
  - **Recoverable technical errors** (retries may apply).
  - **Business errors** that should be returned to the client.
  - **Fatal errors** that cause task failure.
- The Worker SDK exposes a unified way to report a task outcome to the control plane.

Key Takeaways

- The SDK unifies the task lifecycle across Client and Worker components.
- Developers gain a clear, reliable, and extensible abstraction.

Section 4

ArmoniK's Java SDK

ArmoniK's Java SDK

- Requires **Java 17+**.
- Status: Pre-1.0.0 but mature and stable API
- Designed to be **asynchronous-first** using `CompletionStage`.
- Minimal external dependencies.
- Exposes:
  - a **high-level API** (handles, definitions, listeners),
  - and the **low-level gRPC API** for advanced scenarios.
- Serialization is not handled by the SDK (except simple String)

Java Client SDK — Overview

- Main entry point: `ArmoniKClient`.
- Requires an `ArmoniKConfig` describing:
  - endpoints,
  - TLS configuration,
  - retry policy.
- Operations:
  - Create new Session
  - Retrieve existing Session
  - Cancel Session
  - Close Session
- Use try-with-resources for automatic cleanup

Client: Session Creation

- Define session with `SessionDefinition`
- Submit to get `SessionHandle`
- `SessionDefinition` components:
  - Partition IDs: where tasks can execute
  - `TaskConfiguration`: default for all tasks
  - `BlobCompletionListener`: callback when output is COMPLETED or ABORTED

Client: BlobCompletionListener

- Callback pattern for async notifications
- Interface methods:
  - `onBlobSuccess(Blob)`: called when blob is ready
  - `onBlobError(BlobError)`: called on error
- Registered in `SessionDefinition`
- Allows event-driven result handling

Client: Working with Blobs

- Blob types:
  - `InputBlobDefinition`: data sent to workers
  - `OutputBlobDefinition`: placeholder for results
  - `BlobHandle`: reference to uploaded/downloaded blobs
- Input sources:
  - In-memory byte array
  - File
  - Custom source by implementing `BlobData` interface
- `SessionHandle` can create session-scoped blobs

Client: Task Submission

- Create `TaskDefinition` with:
  - Inputs: `InputBlobDefinition` or `BlobHandle`
  - Outputs: `OutputBlobDefinition`
  - Optional: override `TaskConfiguration`
  - Optional: `WorkerLibrary` for dynamic loading
- Submit to `SessionHandle`
- Returns immediately a `TaskHandle` (non-blocking)

Client: WorkerLibrary

- Describes a dynamically loaded worker library for task execution.
- Enables dynamic loading mechanism
- Three components:
  - **path**: Path to library artifact within zip (e.g., `"MyWorker.jar"`)
  - **symbol**: Fully qualified class name (e.g., `"com.example.MyProcessor"`)
  - **libraryHandle**: BlobHandle referencing uploaded library

Client: Typical Workflow

1. Create `ArmoniKClient` with config
2. Create `SessionDefinition` with listener
3. Create `SessionHandle`
4. Define input and output blobs
5. Create and submit `TaskDefinition`
6. Wait for completion or use listener callbacks
7. Close session and client

Worker: Project Structure

- Two Maven modules:
  - `armonik-worker-domain`: SDK for implementing `TaskProcessor`, for dynamic loading
  - `armonik-worker`: Allows static `TaskProcessor` implementation
- Entry points:
  - `TaskProcessor` interface (task logic)
  - `ArmoniKWorker` class (server startup)

Worker: TaskProcessor Interface

- `@FunctionalInterface` with single method:
  - `TaskOutcome processTask(TaskContext)`
- Return values:
  - `TaskOutcome.SUCCESS`: task completed successfully
  - `TaskOutcome.error(String)`: business logic failure
- Error handling strategy:
  - Return `SUCCESS` for successful execution
  - Return `error(message)` for expected business failures
  - Let exceptions propagate for infrastructure issues

Worker: TaskContext - Input/Output Access

- Access inputs and outputs by logical name:
  - `getInput(String name)`: returns `TaskInput`
  - `getOutput(String name)`: returns `TaskOutput`
  - `hasInput(String name)`: check if input exists
  - `hasOutput(String name)`: check if output exists
- `TaskInput` provides read operations:
  - `rawData()`: read entire file as byte array (cached if $<=$ 8 MiB)
  - `stream()`: read as InputStream (for large files, bypasses cache)
  - `asString(Charset)`: read as String with encoding
  - `size()`: get file size in bytes
- `TaskOutput` provides write operations:
  - `write(byte[])`: write byte array
  - `write(InputStream)`: write from stream
  - `write(String, Charset)`: write string with encoding

Worker: Advanced TaskContext

- Blob creation:
  - `createBlob(InputBlobDefinition)`: create session-scoped blob
- Subtask submission:
  - `submitTask(TaskDefinition)`: submit tasks from worker
- Subtask inputs can be:
  - Parent task input (reuse)
  - Existing blob
  - New `InputBlobDefinition`
- Subtask outputs can be:
  - Parent task output (delegation)
  - New `OutputBlobDefinition`

Worker: Deployment Modes

Dynamic Mode

- Pre-built Docker image
- Load JAR at runtime
- Multiple task types
- No rebuild for changes
- Requires `WorkerLibrary`

Static Mode

- Custom Docker image
- Embedded processor
- Single task type
- No `WorkerLibrary` needed

Worker: Dynamic Mode Flow

1. Receive task with `WorkerLibrary` specification
2. Unzip Blob containing the library (fat JAR)
3. Load JAR with dedicated `ClassLoader`
4. Locate `TaskProcessor` by fully qualified class name
5. Instantiate processor
6. Execute `processTask(TaskContext)`
7. Return execution status
8. Unload library and cleanup

Worker: Static Mode Setup

1. Implement `TaskProcessor` interface
2. Create Main class
3. Use `ArmoniKWorker.builder().withTaskProcessor(...)`
4. Build fat JAR (Maven Shade or Gradle Shadow)
5. Create Docker image with JAR
6. Deploy to ArmoniK (edit or create a partition)

Note: Requires rebuilding Docker image for logic changes.

Section 5

TP

Cluster Setup

- We will use the provided PCs to connect to a GCP VM.
- Open visual studio code and look for the `Open remote window` option in the upper left corner, choose the `Connect to host ...` option
- Choose `ArmoniK_TP`
- Open a terminal inside vscode and check that the ArmoniK repositories have been cloned to your PC. If not

```
git clone https://github.com/aneoconsulting/ArmoniK
```

- Navigate into the ArmoniK repository and make sure to checkout the branch dedicated for the trainning, then trigger a local deployment of ArmoniK:

```
ubuntu@jdoe-vm:~$ cd ArmoniK
ubuntu@jdoe-vm:~$ git fetch && git checkout jf/formation25
ubuntu@jdoe-vm:~$ cd infrastructure/quick-deploy/localhost/
ubuntu@jdoe-vm:~$ make
```

Cluster Setup

- A succesful deployment should show an output similar to this:

```
Apply complete! Resources: 125 added, 0 changed, 0 destroyed.

Outputs:

armonik = {
  "admin_app_url" = "http://10.100.1.166:5000/admin"gitk
  "chaos_mesh_url" = null
  "control_plane_url" = "http://10.100.1.166:5001"
  "grafana_url" = "http://10.100.1.166:5000/grafana/"
  "seq_web_url" = "http://10.100.1.166:5000/seq/"
}

OUTPUT FILE: /home/ubuntu/ArmoniK/infrastructure/quick-deploy/localhost/generated/armonik-output.json
Run to point your ArmoniK CLI to this deployment:
------------------------------
export AKCONFIG=/home/ubuntu/ArmoniK/infrastructure/quick-deploy/localhost/generated/armonik-cli.yaml
```

Cluster Setup

- **Service Endpoints Overview**

  `admin_app_url` ArmoniK's web interface.
  `control_plane_url` Entry point for submitting tasks graphs.
  `grafana_url` Dashboard for real-time metrics and observability.
  `seq_web_url` Centralized log viewer for structured event traces.
  `chaos_mesh_url` (Optional) Fault injection platform — used only during dedicated Ops trainings to simulate failures and validate resilience.

- **Generated Files**

  `OUTPUT FILE` JSON file containing all deployment output variables — useful for automation or auditing.
  `AKCONFIG` CLI configuration file. Exporting it allows the `armonik` CLI to target this specific deployment without extra parameters.

ArmoniK SDK installation

- Open a second terminal inside vscode
- Clone the ArmoniK.Extensions.Java repository:

```
git clone https://github.com/aneoconsulting/ArmoniK.Extensions.Java.git
```

- Navigate into the ArmoniK.Extensions.Java repository and make sure to checkout the branch dedicated for the trainning

```
ubuntu@jdoe-vm:~$ cd ArmoniK.Extensions.Java
ubuntu@jdoe-vm:~$ git fetch && git checkout cam/formation25
```

- Install the Java SDK libraries:

```
ubuntu@jdoe-vm:~$ cd armonik-client
ubuntu@jdoe-vm:~$ ./mvnw clean install -DskipTests
ubuntu@jdoe-vm:~$ cd ../worker/
ubuntu@jdoe-vm:~$ ./mvnw clean install -DskipTests
ubuntu@jdoe-vm:~$ cd armonik-worker
ubuntu@jdoe-vm:~$ ./mvnw clean package -Pdocker -Djib.skip=true -DskipTests
ubuntu@jdoe-vm:~$ ./mvnw jib:dockerBuild -Pdocker
```

Hello World - Create Worker

- Navigate into the hello-world folder of ArmoniK.Extensions.Java repository

```
ubuntu@jdoe-vm:~$ cd ArmoniK.Extensions.Java/training/workers/hello-world/
```

- Create Docker Image for hello-world worker:

```
ubuntu@jdoe-vm:~$ ./mvnw clean package
```

Hello World - Create a partition

- Open the file: ArmoniK/infrastructure/quick-deploy/localhost/parameters.tfvars
- inside the "compute_plane" section, copy the subsection "default" and name it "helloworld"
- verify: `helloworld.socket_type = "tcp"`
- change the docker image name: `helloworld.worker.image = "hello-world-worker"`
- add the tag of the docker image: `helloworld.worker.tag = "latest"`
- update the cluster

```
cd ArmoniK/infrastructure/quick-deploy/localhost/
make
```

Calculator: Deploy the Calculator Worker

- Navigate into the `calculator` folder of ArmoniK.Extensions.Java repository

```
ubuntu@jdoe-vm:~$ cd ArmoniK.Extensions.Java/training/workers/calculator/
```

- Create Docker Image for calculator worker:

```
ubuntu@jdoe-vm:~$ ./mvnw clean package
```

- Create a partition named `calculator` by following the same steps as `helloworld` partition

Calculator: Implement the Client

- The `Calculator` Worker performs operations on an array of integers.
- The Worker expects two inputs
  1. `operation` (string): Name of the operation [`"sum"`, `"min"`, `"max"`]
  2. `values` (string): Array of integers serialized in JSON format
- The Worker returns one output
  1. `result` (string): The computed result as a String
- Implement a client application that performs the following steps
  1. Create an array of 100 random integers (values between 0 and 100)
  2. Convert the array to JSON format
  3. Submit three tasks
     - Calculate the sum of all the values
     - Find the minimum value
     - Find the maximum value
  4. Print the three results
  5. Resubmit the three tasks but use a session-scoped blob

Sum: Deploy the Sum Worker

- Navigate into the `sum` folder of ArmoniK.Extensions.Java repository

```
ubuntu@jdoe-vm:~$ cd ArmoniK.Extensions.Java/training/workers/sum/
```

- Create Docker Image for `Sum` Worker:

```
ubuntu@jdoe-vm:~$ ./mvnw clean package
```

- Create a partition named `sum` by following the same steps as `helloworld` partition

Sum: Implement the Client

- The `Sum` Worker adds two integers.
- The Worker expects two inputs:
  1. `num1` (string): First value
  2. `num2` (string): Second value
- The Worker returns one output:
  1. `result` (string): The sum as a string
- **Goal**: Calculate the sum of an array using a **task graph**
  - Use binary tree pattern: pair-wise sums at each level
  - Chain tasks: output of task A becomes input of task B
- **Exercises**:
  1. Sum 4 integers (2 levels)
  2. Sum 8 integers (3 levels)
  3. **Bonus**: Sum array of any size

Implement your own Sum Worker

- **Objective**: Implement your own `Sum` worker
- **Worker Specification**:
  - Takes two string inputs: `num1` and `num2`
  - Returns sum as string: `result`
- **Your Tasks**:
  1. Copy `training/worker-template/`
  2. Implement `SumProcessor.processTask()` method:
     - Read inputs `a` and `b`
     - Parse strings to integers
     - Compute sum
     - Write result to output `result`
     - Handle parse errors with `TaskOutcome.error()`
  3. Build Docker image: `./mvnw clean package`
  4. Create partition and deploy worker
  5. Test with your client from the previous exercise

Exercise: Calculator with Subtasking

- **Objective**: Implement `Calculator` worker that delegates to `Sum` worker
- **Worker Specification**:
  - Inputs: `operation` ("sum", "min", "max"), `values` (JSON array)
  - Output: `result` (string)
- **Implementation Strategy**:
  - Parse JSON array of integers
  - For **"sum"**: Create subtasks using `Sum` worker (build binary tree!)
  - For **"min"**/**"max"**: Compute directly

122 avenue du general Leclerc
91200 Boulogne-Billancourt
www.aneo.eu