

Industrial AI

AI Software Development Kit




Function Manual

<u>Introduction</u>	1
<u>Safety notes</u>	2
<u>Installing AI Software Development Kit</u>	3
<u>Using AI Software Development Kit</u>	4
<u>Guideline for writing pipeline components</u>	5

Legal information

Warning notice system

This manual contains notices you have to observe in order to ensure your personal safety, as well as to prevent damage to property. The notices referring to your personal safety are highlighted in the manual by a safety alert symbol, notices referring only to property damage have no safety alert symbol. These notices shown below are graded according to the degree of danger.

 DANGER
indicates that death or severe personal injury will result if proper precautions are not taken.
 WARNING
indicates that death or severe personal injury may result if proper precautions are not taken.
 CAUTION
indicates that minor personal injury can result if proper precautions are not taken.
NOTICE
indicates that property damage can result if proper precautions are not taken.


If more than one degree of danger is present, the warning notice representing the highest degree of danger will be used. A notice warning of injury to persons with a safety alert symbol may also include a warning relating to property damage.

Qualified Personnel

The product/system described in this documentation may be operated only by **personnel qualified** for the specific task in accordance with the relevant documentation, in particular its warning notices and safety instructions. Qualified personnel are those who, based on their training and experience, are capable of identifying risks and avoiding potential hazards when working with these products/systems.

Proper use of Siemens products

Note the following:

 WARNING
Siemens products may only be used for the applications described in the catalog and in the relevant technical documentation. If products and components from other manufacturers are used, these must be recommended or approved by Siemens. Proper transport, storage, installation, assembly, commissioning, operation and maintenance are required to ensure that the products operate safely and without any problems. The permissible ambient conditions must be complied with. The information in the relevant documentation must be observed.

Trademarks

All names identified by ® are registered trademarks of Siemens Aktiengesellschaft. The remaining trademarks in this publication may be trademarks whose use by third parties for their own purposes could violate the rights of the owner.

Disclaimer of Liability

We have reviewed the contents of this publication to ensure consistency with the hardware and software described. Since variance cannot be precluded entirely, we cannot guarantee full consistency. However, the information in this publication is reviewed regularly and any necessary corrections are included in subsequent editions.

Table of contents

1	Introduction	5
1.1	Overview of Siemens Industrial Edge.....	5
1.2	Overview of Industrial AI@Edge	7
1.3	AI Software Development Kit functionalities	8
1.4	Information about the software license	8
2	Safety notes	9
2.1	Security information	9
2.2	Note on use	9
2.3	Note regarding the general data protection regulation	10
3	Installing AI Software Development Kit	11
3.1	Install and run.....	11
4	Using AI Software Development Kit	14
4.1	Training data preparation.....	14
4.2	Training models	15
4.3	Packaging models as an inference pipeline	16
4.4	Testing the pipeline configuration package locally.....	21
4.5	Mocking the logger of the AI Inference Server	24
4.6	Deploy the packaged inference pipeline for AI@Edge.....	25
4.7	Create a delta package and deploy it to AI@Edge.....	25
5	Guideline for writing pipeline components	26
5.1	Component definition.....	26
5.2	The entrypoint	28
5.3	Input data.....	28
5.3.1	Variable types	29
5.3.2	Restrictions on type Object.....	29
5.3.3	Restrictions on type Binary	29
5.3.4	Custom data formats	30
5.4	Processing data.....	33
5.5	Python dependencies.....	33
5.6	File resources.....	35
5.7	Returning the result	36
5.7.1	Returning Binary data	37
5.8	Adding custom metrics	38

5.9	Pipeline parameters	39
5.10	Use cases.....	40
5.10.1	Processing images	40
5.10.2	Processing time series of signals.....	43
5.10.3	Processing batch data	44
5.11	Examples.....	45
5.12	Writing components for earlier versions of the AI Inference Server.....	49

Introduction

1.1 Overview of Siemens Industrial Edge

Siemens Industrial Edge is the next generation of digital automation. With Siemens Industrial Edge, you use intelligence and scalability of the cloud directly in your production - in a simple, high-performance manner and without your data leaving the manufacturing process. Siemens Industrial Edge combines local and high-performance data processing directly in the automation system itself with the advantages of the cloud: app-based data analysis, data processing and Infrastructure-as-a-Service concepts with central update functionality. In this way, you can quickly integrate apps into manufacturing and manage them with a high degree of automation.

Siemens Industrial Edge allows you to continuously make changes to your automation components and plants, analyze large volumes of data in the automation system to implement innovative functions, such as predictive maintenance, and to achieve maximum flexibility and thus productivity over the entire machine lifecycle.

Industrial Edge Hub

With the Siemens Industrial Edge Hub, you have access to an app store where you can find all Siemens apps and 3rd-party apps. From here, you can manage all licenses for your apps and devices centrally. You can install updates for security issues, device firmware, apps and Industrial Edge Management.

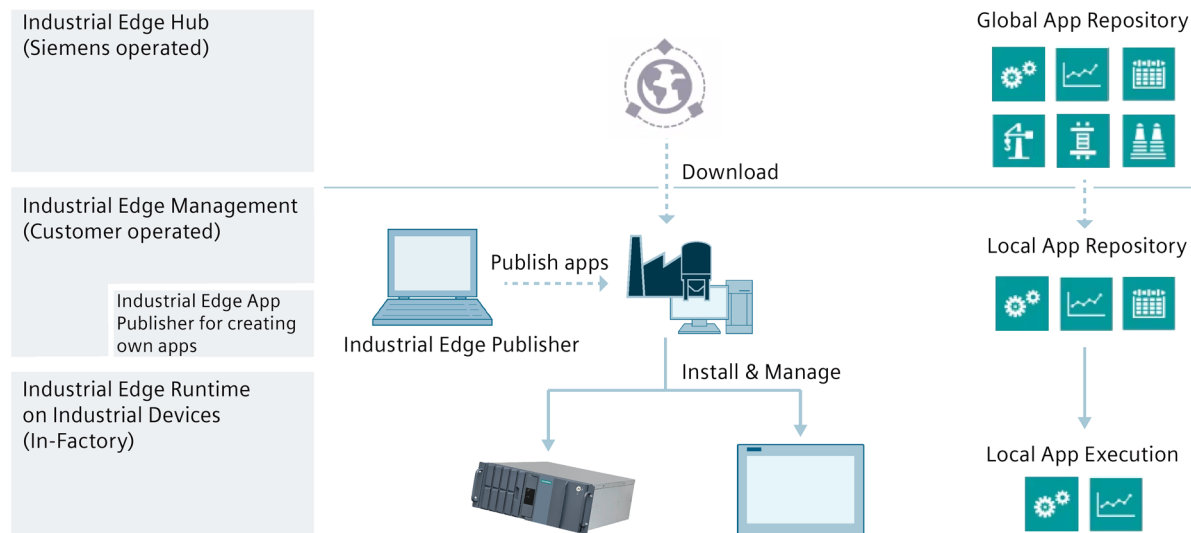
You can monitor and manage distributed Edge devices centrally in the Industrial Edge Management. In this way, new apps and software functions, for example, can be installed on all connected Edge devices company-wide. Central software management thus minimizes the workload for performing maintenance and updates on individual devices.

On the individual Industrial Edge devices, you can start and run apps and keep statistics on an Edge device, for example.

With the Industrial Edge Publisher, you can develop your own Edge apps and make them available to other users in Industrial Edge Management.

Another component of the Siemens Industrial Edge ecosystem is the Industrial Edge Runtime, that is installed on Industrial Edge Devices (IED) or Unified Comfort Panels (UCP) and on which the system, including all applications, ultimately runs.

1.1 Overview of Siemens Industrial Edge



Industrial Edge Hub

Platform for purchasing apps and software, and for monitoring management systems

- Central management location for apps, contributing to corporate standardization
- Management of all licenses in use and thus easy cost estimates
- Overview of all management system instances that are in use worldwide

Industrial Edge Management

Centralized control level for managing devices, apps, and store floor users

- Assignment of apps to the matching Edge devices (worldwide)
- Specification of user rights (e.g. app installations)
- Just a few clicks for app setup and security update cycles
- Supervision of all operations using the centralized Admin view
- Excellent usability for IT and OT users, helping to promote user adaptation and self-service

Industrial Edge Runtime on Industrial Edge Devices

Software level for app container

- Installation of scalable apps on many different Edge devices
- Supports usage in industrial environments by:
 - Ensuring security and reliability
 - Providing comprehensive user management to meet the requirements of machine manufacturers and plant operators alike
 - Adhering to Company Policy Compliance, e.g. user management integration or IT/firewall specifications
- Integrating device connectivity to cloud and automation systems

1.2 Overview of Industrial AI@Edge

Siemens Industrial Edge ecosystem is enabled with Industrial AI products. With Industrial AI, the scalable Industrial Edge ecosystem is expanded by AI capabilities that facilitate the deployment of AI models in the production environment on the shop floor.

Introduction of AI models in the shop floor

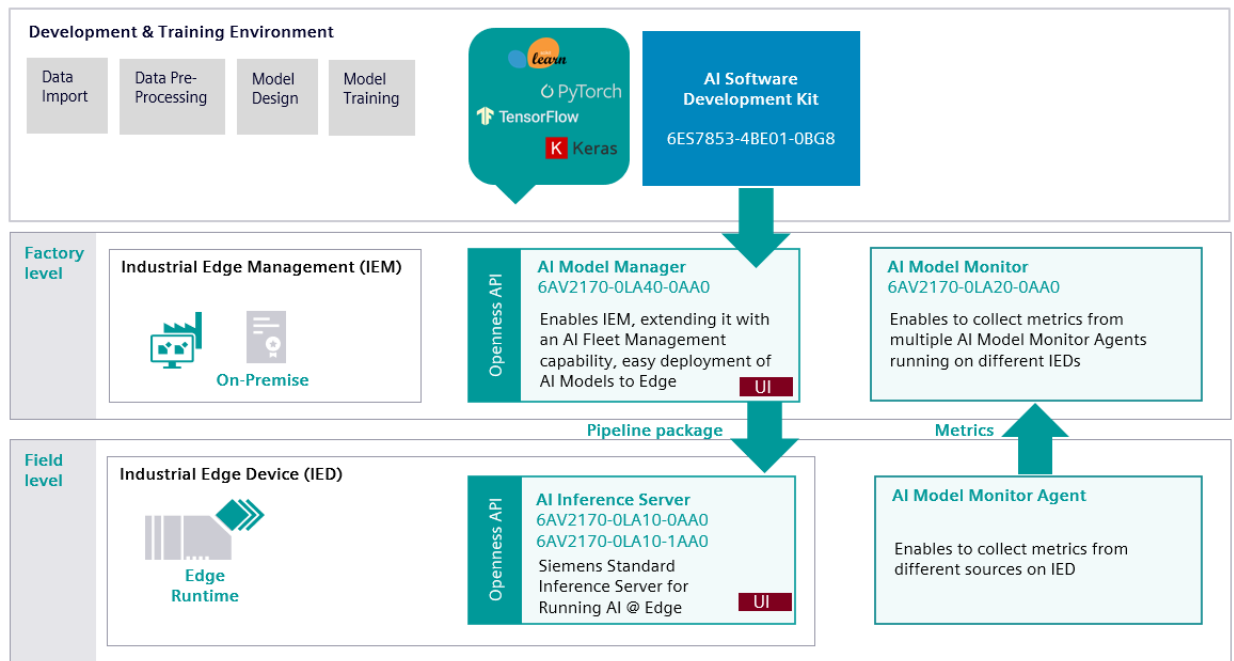
Customers can use the cloud or on-premises model training environment of their choice.

Data scientists or AI engineers can use the AI framework of their choice.

Siemens offers an easy-to-operate AI Software Development Kit (AI SDK) that has pre-configured features that generate a Siemens standard format with the developed AI pipelines. This standard format is fully compatible with AI Inference Server for Siemens Industrial Edge.

The AI Inference Server application is a ready-to-use inference runtime from Siemens that receives AI pipelines as configuration packages (content deployment). This can be done manually via the available user interface or automatically for scaling via the AI Model Manager that is the expansion of the Industrial Edge Manager for AI management.

The AI Model Monitor solution consists of two Industrial Edge applications that enable monitoring AI pipelines running on AI Inference Server distributed across multiple Industrial Edge Devices (IED) at factory level. In this infrastructure AI Model Monitor Agents are installed on IEDs separately and are connected to the central AI Model Monitor application installed at factory level. The AI Model Monitor Agents gather information about the executing IED itself and about the pipeline running on the IED.



See also

Industrial Edge Homepage (<https://new.siemens.com/global/en/products/automation/topic-areas/industrial-edge.html>)

AI@Edge Homepage (<https://new.siemens.com/global/en/products/automation/topic-areas/industrial-edge/production-machines.html>)

1.3 AI Software Development Kit functionalities

The AI Software Development Kit, or AI SDK for short, is a set of Python libraries. These libraries provide building blocks for automating the creation, packaging, and testing of inference pipelines for the AI Inference Server.

The AI SDK contains project templates that provide notebook-based workflows for training models, package them for deployment, and test those packages.

The AI SDK assumes a Machine Learning (ML) workflow that includes the following steps:

- Preparing training data
- Training of models
- Packaging of models as an inference pipeline
- Testing of packaged inference pipelines
- Generating the inference pipeline for AI@Edge

If you already have a trained model, you can skip the "Training data preparation (Page 14)" and "Training models (Page 15)" sections and start with the "Packaging models as an inference pipeline (Page 16)" section. However, to understand the concepts used, it is recommended to read through these chapters as they provide the necessary information.

The AI SDK can be used both exploratively from interactive Python notebooks and purely programmatically as part of an automated ML workflow.

1.4 Information about the software license

Software from third-party suppliers

AI SDK contains Open-Source Software and/or other software from third-party suppliers.

Copyright © SIEMENS, 2023, and licensors. All rights reserved. Parts contain Open-Source Software. More information can be found in the README_OSS.

Safety notes

2.1 Security information

Siemens provides products and solutions with industrial security functions that support the secure operation of plants, systems, machines and networks.

In order to protect plants, systems, machines and networks against cyber threats, it is necessary to implement – and continuously maintain – a holistic, state-of-the-art industrial security concept. Siemens' products and solutions constitute one element of such a concept.

Customers are responsible for preventing unauthorized access to their plants, systems, machines and networks. Such systems, machines and components should only be connected to an enterprise network or the internet if and to the extent such a connection is necessary and only when appropriate security measures (e.g. firewalls and/or network segmentation) are in place.

For additional information on industrial security measures that may be implemented, please visit (<https://new.siemens.com/global/en/products/automation/topic-areas/industrial-security.html>).

Siemens' products and solutions undergo continuous development to make them more secure. Siemens strongly recommends that product updates are applied as soon as they are available and that the latest product versions are used. Use of product versions that are no longer supported, and failure to apply the latest updates may increase customers' exposure to cyber threats.

More information about network segmentation, firewall etc. is all on these pages.

To stay informed about product updates, subscribe to the Siemens Industrial Security RSS Feed visit (<https://new.siemens.com/global/en/products/automation/topic-areas/industrial-security.html>).

2.2 Note on use

Protection of the host computer

Customers are responsible for protecting their own host computers and preventing unauthorized access to their host computers.

To protect the host computer Siemens suggests taking the following measures:

- Deploy the host computer only in isolated plant network, but not office network.
- Enable the screen saver and lock the screen when leave.
- Install suitable anti-virus software.
- Install updates and patches for the operating system and software on the host PC in time.

2.3 Note regarding the general data protection regulation

Notes on protecting administrator accounts

A user with administrator rights has extensive access and manipulation options available to the system.

Therefore, ensure there are adequate safeguards for protecting the administrator accounts to prevent unauthorized changes. To do this, use secure passwords and a standard user account for normal operation. Other measures, such as the use of security policies, should be applied as needed.

Notes on the use

- Before installing AI Software Development Kit, it is recommended to verify the SHA-256 checksum of the distribution zip package against the checksum provided on Siemens Online Industry Support.
- AI Software Development Kit can only be accessed from the host computer. Do **NOT** allow other machines in the plant network to access AI Software Development Kit.
- The current AI Software Development Kit is only applicable for non-safety critical application.
- AI Software Development Kit stores the project data without encryption on the host PC. The customer is responsible for the CIA (Confidentiality, Integrity and Availability) of the files created, stored, downloaded, or exported by AI Software Development Kit.
- AI Software development Kit might be used in conjunction with Jupyter Lab, which includes a web server that can be accessed locally or remotely. The customer is responsible for configuring Jupyter Lab with HTTPS enabled (https://jupyter-notebook.readthedocs.io/en/stable/public_server.html#using-ssl-for-encrypted-communication).
- If you use the AI Software Development Kit to create pipeline configuration packages, make sure that you only include source code and Python packages from trusted sources.
- If you use the AI Software Development Kit to run pipeline configuration packages locally, make sure that you only use pipeline configuration packages from trusted sources.

2.3 Note regarding the general data protection regulation

Siemens observes the principles of data protection, in particular the principle of data minimization (privacy by design). For this product this means:

The product does not process / store any personal data, only technical functional data (e.g. time stamp, IP addresses of the connected manufacturing devices). If the user links this data with other data (e.g. shift plans) or stores personal data on the same medium (e.g. hard disk) and thus establishes a personal reference, the user must ensure compliance with data protection regulations.

Installing AI Software Development Kit

3.1 Install and run

Depending on your background, you can choose to use AI SDK in a pure Python 3.8 environment or use the Jupyter Notebook-oriented project templates with a notebook editor of your choice.

Start with the project templates to familiarize yourself with the AI SDK. These sample solutions contain all the necessary dependencies and allow for a quick and smooth start. As you move beyond the interactive exploration phase, you might consider switching to a purely Python-based approach.

Prerequisites

Before you begin, make sure you have access to the internet. If you access the internet via a proxy, when working in a corporate network directly or via VPN, please ensure that you have configured the following tools to use the correct proxy settings:

- `pip`
- `conda` (if you also use `conda`)

Setting environment variables `http_proxy` and `https_proxy` covers both. A detailed explanation of alternative solutions is provided in:

- Using a proxy server (https://pip.pypa.io/en/stable/user_guide/#using-a-proxy-server)
- Using Anaconda behind a company proxy (<https://docs.anaconda.com/anaconda/user-guide/tasks/proxy/>)

Using the AI SDK without project templates

To install the AI SDK from the Python wheel file, simply use `pip` in a Python 3.8 environment. This will ensure the installation of any additional required Python packages.

```
pip install simaticai-1.4.0-py3-none-any.whl
```

Note that, by default, `pip` installs the latest available version of the required packages that are compatible with the AI SDK and any other packages that might already be installed. If you want to ensure that you use the versions listed in `Readme_OSS`, you can apply the appropriate constraint during installation as follows:

```
pip install simaticai-1.4.0-py3-none-any.whl -c constraints.txt
```

Note that this increases the probability that `pip` will not be able to resolve all applicable restrictions or that an older package and versions with security issues will be installed.

To use the AI SDK from your Python code, you must import modules from the `simaticai` namespace. For more information, refer to the User Guide, AI SDK API reference, or project templates.

Using the AI SDK with project templates

Standalone project template zip packages contain a prepared working directory that includes notebooks and sources. You can use them in a Python or Jupyter Lab environment of your choice. The prerequisites are as follows:

- Python 3.8, installed either natively or via Conda.
- A notebook editor such as Jupyter Notebook, Jupyter Lab, or Visual Studio Code.

We strongly recommend that you set up a separate Python environment specifically for the project template, as described in the README file. The notebooks in the project templates assume a dedicated Python environment with a predefined name and a matching kernel name. For example, the State Identifier project template uses the environment name `state_identifier`.

You can use your preferred Python environment manager to create the Python environment. Below we provide the commands for Conda and Python `venv` using the State Identifier project template, as an example. For other project templates, you must replace the name `state_identifier` as described in the corresponding README file.

```
# create a Conda environment including Python and activate it
conda create -n state_identifier python=3.8.16
conda activate state_identifier
```

```
# create a Python virtual environment in Linux and activate it
python -m venv ~/venv/state_identifier
source ~/venv/state_identifier/bin/activate
```

```
# create a Python virtual environment in Windows and activate it
python -m venv %USERPROFILE%\venv\state_identifier
USERPROFILE%\venv\state_identifier\Scripts\activate.bat
```

Once the environment is created and activated, you must register it as an interactive Python kernel to make it accessible within your notebook editor. This is usually achieved with the following commands:

```
# install and register interactive Python kernel
python -m ipykernel install --user --name state_identifier
--display-name "Python (state_identifier)"
```

Now your Python environment is ready to be used for the project template. Extract the project template from its package, change the working directory to the extracted project folder, and execute the command that follows:

```
# install packages required for the template including the AI SDK
and ipykernel
pip install ipykernel -r requirements.txt -f <directory path
containing simaticai wheel file>
```

Note that you need to specify a path to the directory containing the AI SDK wheel file, not a path to the wheel file itself.

Once the required packages are installed, you can explore and execute them in your notebook editor.

Please make sure that you select the appropriate interactive Python kernel to execute the notebooks in this example: `Python (state_identifier)`.

Note that by default, `pip` installs the latest available version of the required packages that are compatible with the AI SDK and the project template. If you want to make sure to use the versions that are listed in `Readme_OSS`, you can apply the appropriate constraint during installation as shown below:

```
pip install ipykernel -r requirements.txt -f <directory path  
containing simaticai wheel> -c constraints.txt
```

Note that this increases the probability of `pip` installings an older package version that might contain security vulnerabilities.

Using AI Software Development Kit

4.1 Training data preparation

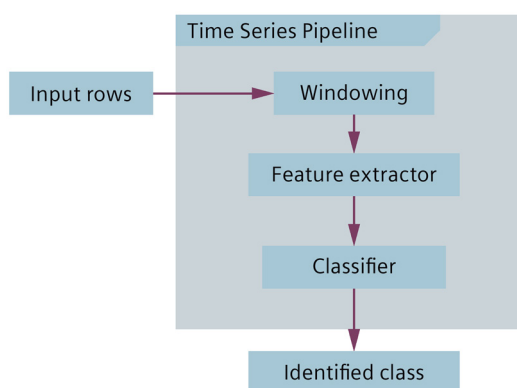
Preparing data for model training is mostly out-of-scope of the AI SDK. Prepared example datasets are available for the project templates. You can try out these templates without requiring data acquisition. Each project template includes a notebook to guide you through downloading a sample dataset.

Processing time series data

The State Identifier project template provides basic building blocks for building ML models that process time series of aligned signals. Aligned signals mean that the input of the processing pipeline consists of rows, containing a value for each signal. For example, a row consisting of 3 signals and a time stamp would look like this:

Time stamp	var1	var2	var3
09:50:23	1.2	202	25

The building blocks help you create a time series pipeline that processes a stream of such rows according to the following pattern:



The roles of the piping elements are as follows:

- "Windowing" accumulates a given number of input rows in a processing window.
- The "feature extractor" calculates several characteristics for each window. A feature is a mathematical value calculated from the values in the window.
- The "classifier" is the actual machine learning model that predicts a class for each window, based on the extracted characteristics.

Most pipelines contain other processing elements, such as imputers to fill missing values, or scalers that map an input to a predefined range.

To train the classifier in such a pipeline, the input data must undergo the preprocessing steps during the training process.

Therefore, this processing pipeline must be defined as a part of data preparation before training. This is where the building blocks in the State Identifier project template play a crucial role.

These building blocks are based on the widely used machine learning Python package `scikit-learn`. Scikit-learn provides a framework for defining pipelines that allows you to combine data transformers with classifiers or other kinds of estimators. The building blocks can be found in the `src/pipeline.py` file in the State Identifier project template. The commonly used libraries are:

- `WindowTransformer`, which transforms a series of input rows into a series of row-based windows
- `FeatureTransformer`, which transforms a window of rows into the feature values according to user-defined functions

In addition to these transformers, there is a transformer called `FillMissingValues`, which performs input data correction for simple cases. For more advanced cases, you should use a more sophisticated imputer to correct your input.

For more details and concrete examples, refer to the training notebooks in the State Identifier project template.

Mapping predicted classes of data windows to data points

As described in the previous chapter, time series data is typically classified on a window-by-window basis. This means that the class of a single data row is not defined on its own. Nevertheless, there are cases where it is convenient to map the classes defined window by window to the data points themselves. For example, if you want to visualize the data points according to their classification by color-coding the data points with the class.

The State Identifier project template provides the utility function `back_propagate_labels` in the file `src/pipeline.py` to perform this mapping. For more details and concrete examples, please refer to the training notebooks in the State Identifier project template.

4.2 Training models

The AI SDK does not restrict how you train your models and save the trained model. You can use the training notebooks in the project templates as examples. The project templates include examples using scikit-learn and TensorFlow.

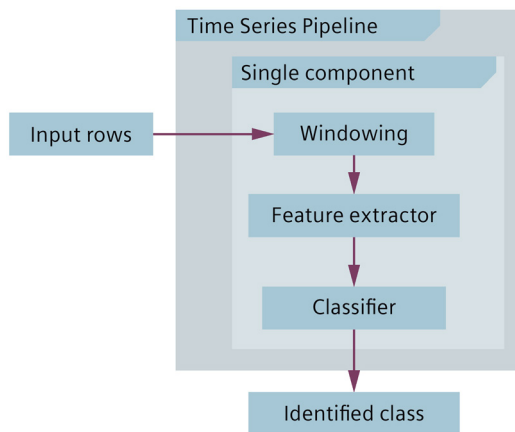
Some ML frameworks, such as TensorFlow, use their own format for storing trained models. Other frameworks, such as scikit-learn, rely on persistent Python runtime objects. In the latter case, you need to ensure that the same versions of Python libraries exist when the objects are stored after training and then retrieved in the AI Inference Server. The packaging feature of the AI SDK supports it by requiring exact version specifiers for required Python packages in a pipeline package.

4.3 Packaging models as an inference pipeline

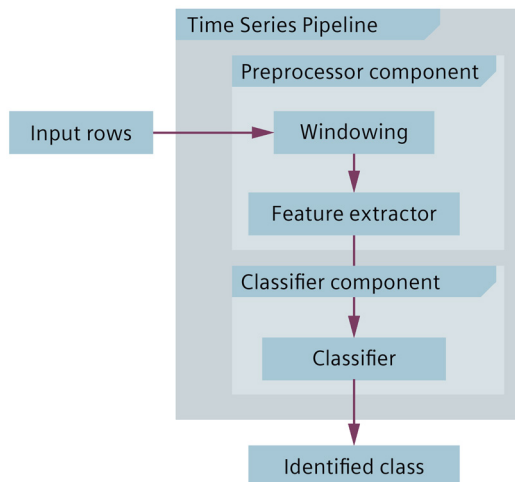
The AI SDK provides the functionality to create a pipeline configuration package that encapsulates trained models. These models can be converted to an edge configuration package using the AI SDK. Then it can be uploaded and run on the AI Inference Server on an Industrial Edge Device. The related functions can be found within the `simaticai.deployment` module.

Single or multiple components

From a deployment perspective, the inference pipeline can consist of one or more components. This is independent of the logical structure of the inference pipeline. For example, you can package a typical time series pipeline that consists of multiple scikit-learn pipeline elements into a single pipeline component for deployment:



Alternatively, you can deploy the same pipeline as two components:



To keep the deployment simple and less error-prone, you should deploy your inference pipeline with as few components as possible.

In many cases, a single component is sufficient. However, there may be reasons why you might consider using separate components, such as:

- You need a different Python environment for different parts of your processing – for instance, you have components that require conflicting package versions.
- You want to exploit parallelism between components without implementing multithreading.
- You want to modularize your pipeline and build it from a pool of component variants that you can flexibly combine.

Creating an inference pipeline package

The AI SDK allows you to create pipeline components implemented in Python and assemble linear pipelines from one or more such components.

The API is designed to anticipate future possible types of components that could be based on a technology other than Python, such as ONNX or native TensorFlow serving. However, only Python is currently supported.

The workflow for creating an inference pipeline package is as follows:

1. Write the Python code that encapsulates your trained model as an inference pipeline component.
2. Define the pipeline component.
3. Repeat the above steps if you have multiple components.
4. Configure the pipeline.
5. Save the pipeline configuration in a pipeline configuration package.

Creating pipeline components implemented in Python

Implementing an inference pipeline component in Python is a comprehensive topic in itself and will be described in detail in the next chapter, "Guideline for writing pipeline components (Page 26)".

A component consists of files and metadata.

Files contain:

- Python scripts
- trained models

Metadata includes:

- component name and component version
- required Python version and Python packages
- input and output variables
- the number of parallel executors
- the entrypoint

You can create your own arrangement of project files. We recommend that you follow the project templates for the AI SDK. Here the source code and stored trained models are

organized in a predefined structure. If you keep the same relative structure on the AI Inference Server, you can use the same relative references from the source code to the stored models or other files.

Put together all the files for the components. Usually, there should be at least one Python script for the entrypoint, the inference wrapper, and the saved model. Create the pipeline component by running a Python script or notebook that provides the following functionality:

- creates a Python component object with a specific name, component version, and required Python version
- defines required Python packages
- defines input and output variables
- defines custom metrics
- defines the number of parallel executors
- adds Python scripts and saved models
- defines the entrypoint under the Python scripts

All this takes place with the corresponding functionality of the `simaticai.deployment` module. For concrete examples, refer to the packaging notebooks in the project templates. Please refer to the AI SDK API reference manual for more information and advanced options.

Consider the following limitations:

- The AI SDK allows you to select a required Python version that is supported by different versions of AI Inference Server.
- Make sure you select a Python version that is supported by the version installed on your Industrial Edge target device. For the current AI SDK version, this is Python version 3.8.
- The required Python packages must either be added as wheel files to the pipeline component or be available for download via `pip` for the target Inference Server.
- The entrypoint script must be in the root folder of the package in the current AI SDK version.
- AI Inference Server supports a maximum of 8 parallel executors.

Configuring and saving the pipeline

After you create the component(s), you must combine them into a pipeline using a `Pipeline` object.

If the pipeline only consists of one single component, it has the same input variables and output variables as its single component. You only need to specify a pipeline name and version from which the file name is derived when you save the package. For example, refer to the end of the package creation notebooks in the project templates.

To create a linear pipeline of multiple components, you can still mostly rely on the constructor of `Pipeline`, which attempts to automatically connect the components passed as a list:

- connecting pipeline input to the first component
- connecting inputs and outputs of subsequent components if the variable name matches
- connecting the last component to the pipeline output

In general, you should pass data from one component to another component in a single variable of type `String` and serialize and deserialize any data you have through a string.

The low-level methods of the `Pipeline` class allow you to connect any components, pipeline inputs and outputs. However, the AI SDK cannot guarantee that the result will behave as intended on the AI Inference Server.

For information about pipeline input and output with different data types and defining custom metrics, refer to the Guideline for writing runtime components (Page 26). It describes, how input and output data is passed between the AI Inference Server and your entrypoint. It also explains special considerations that apply to a continuous stream of time series data or for bulk data.

Whether you created the pipeline with a single constructor call or with low-level methods, you must save it for the creation of the pipeline. This step creates the pipeline configuration package as a .zip file and leaves the contents of the .zip file in the file system. You can explore it to troubleshoot or see how your package creation calls are reflected in the contents of files and directories.

Pipeline packages are identified by their package ID and version attributes, and are grouped by package ID in the AI Inference Server and other Edge applications.

When saving a pipeline – with the `save()` method – you can specify a package ID in a UUID 4-compliant format, or an automatically generated one is assigned.

If no package ID is defined in the `save()` method, and AI SDK finds an already assigned package ID in a previously generated and similarly named package, the package ID found in the latest package is used.

AI SDK automatically assigns and increments the version number of a pipeline each time a package is saved, unless a new package ID is assigned in the `save()` method, or an explicit version number without a package ID is defined in the `save()` method, or in the pipeline constructor.

Restrictions:

- You cannot overwrite a previously saved package with the same package ID if the package ID is explicitly assigned in the `save()` method
- Existing packages without a package ID will be overwritten
- If a new package ID is assigned to an existing version of the package, the old one will be overwritten
- If no predecessor of a package is found, AI SDK assigns version 1 to the created package
- The version defined in the `save()` method takes precedence over the version assigned at the constructor level

Pipeline parameters

Advanced use cases might require a modification of the pipeline behavior after deployment, for example by changing the parameters of the AI model. For this reason, AI SDK allows you to define pipeline parameters.

In many respects, pipeline parameters are similar to pipeline inputs. But pipeline parameters are handled separately and treated specially. Unlike input variables, pipeline parameters must have a default value, which the parameter takes initially after deployment. Therefore, a pipeline parameter's value is always defined.

Depending on the configuration, a pipeline parameter might be changed interactively via the user interface of the AI Inference Server or can also be connected to an MQTT topic like an input variable. In the latter case, the pipeline can receive parameter updates from other system components via the External Databus.

The pipeline parameters apply to all components. This means that in a pipeline, all components with parameters must be ready to receive parameter updates. A pipeline component updates only relevant parameters for the specified components.

For details on how to define pipeline parameters and how to handle parameter updates in the pipeline components, refer to "Guideline for writing pipeline components (Page 26)". For a complete code example that shows how to define and use pipeline parameters, refer to the State Identifier project template.

Parallel execution

By default, pipeline components process the inputs sequentially, within the same Python interpreter context. To increase the throughput of the component, you can instruct the AI Inference Server to run multiple instances of a pipeline component and distribute the inputs among them. This way you can exploit the parallelism available in most multi-core CPUs.

If you specify parallel component execution, every instance will be initialized separately and will receive only a fraction of the inputs. Therefore, not all components are suitable for parallel execution.

For example, the single component of the pipeline given in the State Identifier project template cannot be executed by parallel instances because the component must process inputs sequentially, one by one to form windows from the data.

Theoretically, you could separate the State Identifier into two components, the first component forming the windows and the second component calculating the features and prediction. Then, the second component could be run in multiple parallel instances, as it does not have to keep previous inputs to calculate the output. (It is another question whether this complexity is worthwhile in a specific use case.)

In contrast, the component given in the Image Classification project template can be executed by parallel instances out of the box. It is practically stateless, as the key global state the component uses is the model loaded during initialization. Otherwise, the component needs only the current input to calculate the output.

Note that with parallel component execution, there is no guarantee that the outputs are produced in the same order as the corresponding inputs arrive. It might happen that one instance overtakes another even if the raw CPU time required for all inputs is about the same. The component instances compete for CPU cores with other applications running on the Industrial Edge device.

You can predefine the number of parallel component instances using the AI SDK `PythonComponent.set_parallel_steps()` function. This setting can be overridden on the AI Inference Server user interface.

4.4 Testing the pipeline configuration package locally

Once you created your pipeline configuration package, test it before converting to the Edge configuration package. Only then deploy the package to the AI Inference Server.

The advantages of local testing are the following:

- You can identify many potential issues more quickly because you don't need to go through a deployment cycle.
- You can diagnose and troubleshoot problems much more easily because you can inspect artifacts in your development environment.
- You can validate your fixes faster and move on to other issues that had been blocked from emerging by previous issues.
- You can easily include the local pipeline tests in the test automation of your build process.

2 tools for local testing

You can apply state-of-the-art software engineering practices such as unit testing and test-driven development.

This means that ideally, you already have automated unit testing or even integration testing that ensures that the Python code and stored models work in isolation as expected. This helps you localize errors when you assemble these parts and integrate them as a pipeline configuration package.

The AI SDK package `simaticai.testing` provides 2 tools for local testing:

- A pipeline validator that performs static validation of the package for the availability of required Python packages.
- A pipeline runner that allows you to simulate the execution of your pipeline in your Python environment.

Note, that all these testing features apply to pipeline configuration packages, not Edge configuration packages. You must use it before you convert your pipeline configuration package to an Edge configuration package using the AI SDK.

Since the conversion itself is done automatically, most of the potential issues are already present in the package before the conversion, thus a post-conversion verification would only delay the identification of these issues.

Static validation of a pipeline package

You can pass your pipeline configuration package to the `validate_pipeline_dependencies` function in the `simaticai.testing.pipeline_validator` submodule to perform static checks. These checks include:

- Verifying that the Python version required in the package is supported by a known version of the AI Inference Server.
- Verifying that all required Python packages are either included in the pipeline package itself or available on `pypi.org` for the target platform.

For specific programming details, refer to the AI SDK API reference manual.

Local execution of a packaged pipeline

The `LocalPipelineRunner` class in the `simaticai.testing.pipeline_runner` submodule can be used to locally mimic the behavior of the AI Inference Server for loading and running inference pipelines. This is a quick and easy way to find programming or configuration errors before deploying the package.

The local pipeline runner simulates the server environment as follows:

1. It unpacks the pipeline components into a test folder, similar to what would happen in the AI Inference Server.
2. It creates a separate Python virtual environment for each component.
3. It installs the required Python packages from the wheel files if provided in the package or by `pypi.org`.
4. It installs the mock of `log_module` (refer to "Mocking the logger of AI Inference Server (Page 24)")
5. It updates pipeline parameters if applicable.
6. It feeds the pipeline with input data by triggering the entrypoints of the components accordingly.
7. It collects the sequence of pipeline outputs for a given sequence of pipeline inputs.

You can also use the local pipeline runner to drive your pipeline component by component. You can feed individual components with inputs and verify the output produced.

If the pipeline contains parameters, the pipeline uses the default values for the parameters. You can also change the parameter values using the `update_parameters()` method. This allows you to test your pipeline with different parameters.

Note

You can only use the `update_parameters()` method before calling `run_component` or `run_pipeline()`, but you cannot change pipeline parameters while these methods are running.

From a testing strategy and risk-based testing perspective, we recommend that you validate the business logic within the pipeline components in unit tests as you would with any ordinary Python program and use the local pipeline runner to cover test risks such as the following:

- A mismatch between pipeline and component input and output variable names
- Required Python packages not covered by `requirements.txt`
- Missing source or other files from the package
- An interface mismatch between subsequent pipeline components
- The entrypoint script cannot process input data due to a mismatch in the data format
- The entrypoint script generates output data in the wrong format
- For some reason, the pipeline does not work consistently as intended

A crucial point for making the local test faithful concerning data input and output formats is to understand how data connections work in the AI Inference Server. The following data connection types are straightforward:

- **Databus**
The Databus is a distributed application that runs on individual Industrial Edge Devices and facilitates access to the data of the field devices. If required, you can configure your own data points.
- **External Databus**
You can use the External Databus to connect remote clients to the Industrial Edge Device via MQTT. The data transmitted to the External Databus is automatically transferred to the Databus and can be used by other apps within the Databus.
- **IE Vision Connector**
IE Vision Connector connects to every "Generic Interface for CAMeras" that supports image processing systems and makes image or video data available over the standard data bus or a high-throughput data bus. The IE Vision Connector has a user interface that offers options for configuring camera-specific parameters according to the "GEN<I>CAM" standard. It forms the image frame/live stream for the user to view.

For these data connection types, the AI Inference Server passes the MQTT payload string directly as the value of the connected pipeline input variable. In many use cases where you use this data connection type, your pipeline has a single input variable of type string. This means that you need to pass a Python dictionary to the local pipeline runner with each individual element.

For example, if you take the pipeline from the Image Classification project template, you have a single input variable `vision_payload`. To run your pipeline on two consecutive input images, you must call the pipeline runner as follows:

```
pipeline_input1 = { 'vision_payload': mqtt_payload1 }
pipeline_input2 = { 'vision_payload': mqtt_payload2 }
pipeline_output = runner.run_pipeline([pipeline_input1,
pipeline_input2])
```

For a complete code example that shows how to feed a pipeline with a single string input variable in a local test, refer to the Local Pipeline Test Notebook in the Image Classification project template.

The SIMATIC S7 Connector data connection type requires a higher level of effort. This connector is typically used in time series use cases. Using this connection, the AI Inference Server processes the MQTT payload used by the S7 Connector and only passes on the values of the PLC variables, but not the metadata. So, if you intend to use your pipeline with the S7 Connector, you need to feed it with dictionaries holding the PLC tag values.

Taking the pipeline from the State Identifier project template for example, you have input variables `ph1`, `ph2` and `ph3`, that should be used with the SIMATIC S7 Connector data connection type. To replicate how the AI Inference Server feeds the pipeline, you must call the pipeline runner as follows:

```
pipeline_input1 = {'ph1': 4732.89, 'ph2': 4654.44, 'ph3': 4835.02}
pipeline_input2 = {'ph1': 4909.13, 'ph2': 4775.16, 'ph3': 4996.67}
pipeline_output = runner.run_pipeline([pipeline_input1,
pipeline_input2])
```

For a complete code example that shows how to feed a pipeline with an input line of PLC tag values in a local test, see the Local Pipeline Test Notebook in the State Identifier project template.

Restrictions of local pipeline execution

The local runner works with batches of input data and processes the whole input batch component by component. In the case of a sequence of pipeline inputs, the entire sequence is first processed by the first component, and only then is the output of the first component processed by the second component.

This is different from the runtime environment on the AI Inference Server, where the components in the pipeline potentially start consuming input as soon as the preceding component has produced output.

You cannot fully test input and output data formats, as these depend on the data connection settings of the AI Inference Server, and you must provide the local runner with the input data in the representation that matches the output side of the connector. This means that if your assumptions on the data connection settings or the resulting data formats are wrong, your tests will also provide misleading results. The local runner can only simulate linear pipelines: where the pipeline input variables are only used by one component, each component uses only the outputs of the previous components, and the pipeline output only consists of variables from the last component.

Furthermore, the results obtained in local tests are not fully representative of the AI Inference Server, including but not limited to the following aspects:

- The local version of Python may be different from that in the AI Inference Server.
- The local architecture may be different, resulting in different builds of imported Python packages being used.
- The local runner executes only one instance of the Python code, regardless of the parallelism settings in the configuration.

Despite all these limitations, we recommend testing your pipeline locally before deployment. This will most likely save you more time than skipping this step.

4.5 Mocking the logger of the AI Inference Server

The Python environment on the AI Inference Server injects a Python module named `log_module` that the Python scripts can use for logging on the server. To be able to run the same code in a local development environment on a PC, the AI SDK provides a mock of `log_module` in a wheel, which you can install, import, and use in the same way. This wheel file must not be included in the pipeline package dependencies.

4.6 Deploy the packaged inference pipeline for AI@Edge

After you create and test your pipeline configuration package, you must convert it to an Edge configuration package, so it could be deployed to the AI Inference Server.

Conversion is required because, while a pipeline configuration package defines the inputs, outputs, and inner workings of an inference pipeline, it does not contain all the components required to run in the AI Inference Server. To make it complete for deployment on the AI Inference Server, the pipeline configuration package must be converted to an Edge configuration package.

Amongst other things, the conversion ensures that all the necessary Python packages for the target platform are included. If any of the required packages, including transitive dependencies, are not included in the pipeline configuration package for the target platform, they will be downloaded from `pypi.org`.

The conversion function is available in AI SDK both as a Python function and a CLI command. Please refer to the details of the function `convert_package` in module `simaticai.deployment` in the AI SDK API reference manual.

An Edge configuration package can be deployed to the AI Inference Server:

- via AI Model Manager,
- via AI Inference Server's API,
- uploaded directly to AI Inference Server via UI.

For more information, refer to AI Inference Server and AI Model Manager documentation.

4.7 Create a delta package and deploy it to AI@Edge

The amount of time taken by deployment strongly correlates with the size of the Edge configuration package. To reduce it, the AI SDK provides the functionality to create a delta pipeline package. A delta package contains only the files that are updated or newly added compared to the original version.

You can use function `create_delta_package` in module `simaticai.deployment` or the corresponding CLI command. For more details, refer to the AI SDK API reference manual.

Note

The delta configuration package can be deployed in the same way as the Edge configuration package. The original Edge configuration package must be deployed before the delta configuration package.

Guideline for writing pipeline components

AI Inference Server is an Industrial Edge application designed to execute your ML models on an Industrial Edge Device. The interface between your ML model and AI Inference Server is a Python script that consumes incoming data, processes it and creates an output response. This guideline explains the workflow for defining a pipeline component using a Python script.

5.1 Component definition

In this context, a component means a pipeline step that which consumes input data, processes that data by using a model, and produces the output data. The model can mean, in a narrower sense, an ML model such as a neural network or a random forest algorithm, or simply an aggregator or other pre- or postprocessing logic. In every case, a Python script, hereinafter referred to as the "entrypoint", establishes the connection between the model and the AI Inference Server. The server needs to receive information about what Python environment is required to execute the code, including the required Python packages or file resources.

Essential information for the execution of the code

The most essential information for the execution of the code is:

- The Python script that receives the input data
- The Python version required to run the Python script
- The input and output variables of the component

Example

The following code shows how to define component settings. The created configuration can be checked in the `pipeline-config.yml` which can be found in the Examples (Page 45) section. Please note that this code is only used to create the pipeline configuration package, but it is not contained in the package itself.

```
# create_pipeline_config_package.py
from simaticai import deployment

# defining basic properties of the pipeline component
# AI Inference Server version 1.4 supports Python 3.8
component = deployment.PythonComponent(name='classifier',
version='1.0.0', python_version='3.8')

# defining entrypoint Python script
component.add_resources("../src", "entrypoint.py")
component.set_entrypoint('entrypoint.py')
```

```
# defining input variable of component
component.add_input(name='input_1', _type='Double')
component.add_input(name='input_2', _type='Double')

# defining output variable of component
component.add_output(name='class_label', _type='Integer')
component.add_output(name='confidence', _type='Double')
```

In this example, the code uses a pre-trained scikit-learn model that is stored in a joblib file. The file acts as a resource file in this model. For more details about resource files, see the File resources section. The code also uses external Python modules (Page 33) that must be deployed and installed on the AI Inference Server.

```
# adding stored scikit-learn model as a resource file
component.add_resources('.', 'models/classifier-model.joblib')

# adding python dependency scikit-learn with version==1.0.1
component.add_dependencies([('scikit-learn', '1.0.1')])
```

With this configuration, the AI Inference Server collects data for `input_1` and `input_2`. When the data is available, the server wraps it into a data payload and calls the `process_input()` function in `entrypoint.py`. Once the data is processed and the `class_label` and `confidence` results are calculated, the function generates a return value.

5.2 The endpoint

AI Inference Server itself receives the data payload from the input data connection. With each input, the AI Inference Server triggers the `process_input(data: dict) -> dict` function in the endpoint module. After `process_input()` returns, the server forwards the output to the next pipeline component, or emits it as pipeline output over the output data connection.

Example

```
# endpoint.py
import sys

from pathlib import Path

# when you import from source, the parent folder of the module
# ('./src') must be added to the system path
sys.path.insert(0, str(Path('./src').resolve()))

from my_module import data_processor # should be adapted to your
code

def process_input(data: dict) -> dict:
    return data_processor.process_data(data["input_1"],
data["input_2"])
```

In this case, it is assumed that business logic is encapsulated in `data_processor.process_data()`. You can place the code into your package and modify only the reference to your data processor.

5.3 Input data

AI Inference Server wraps the acquired input values into a dictionary and passes them to `process_input()` as a single parameter. Each input variable is represented as a separate element in the dictionary, for example:

```
{"input_1": 123.51, "input_2": 47.02}
```

If you have multiple inputs, the `process_input()` might be triggered with incomplete data. If inter-signal alignment is enabled, the missing input variables are `None` in the dictionary.

```
{"input_1": 123.51, "input_2": None}
```

Without inter-signal alignment, inputs are passed to `process_input()` one by one, variable by variable, so that the dictionary contains only one element for each call.

```
{"input_1": 123.51}
```

For further details, refer to the Processing time series of signals (Page 43).

5.3.1 Variable types

Let's take another look and check how you defined the input variable `input_1`

```
# defining input variable
component.add_input(name= 'input_1', _type='Double')
```

You defined it as `Double`, which is a data type of AI Inference Server.

The `process_input()` function receives the inputs converted to a Python data type, which is a `float` for AI Inference Server type `Double`.

In general, you need to define the types of input and output variables as AI Inference Server types, but the Python script should use the appropriate Python type. The match between AI Inference Server data types and Python data types is shown in the following table. The table also shows which data type is supported by which data connection in AI Inference Server version 1.5.0.

AI Inference Server	Python	Databus	S7 Connector	Vision Connector	ZMQ
Bool	bool		I/O		
Integer	int		I/O		
Double	float		I/O		
String	str	I/O	I/O	input	
Object	dict			input	output
Binary	bytes				I/O
ImageSet	dict			I/O	

External Databus connections support the same data types as Databus.

5.3.2 Restrictions on type Object

AI Inference Server version 1.5 imposes restrictions on the dictionaries returned by the inference wrappers for output variables of type `Object`. The returned dictionary must contain a metadata string and a binary sequence. The metadata and the binary sequence can have any key in the `dict` but must be of type `str` and `bytes` respectively. For details, refer to the section "Returning the result (Page 36)".

Note

The structure of dictionaries that is received as pipeline input is different from the dictionary structure that is required as component output. For details, see the section "Processing images (Page 40)".

5.3.3 Restrictions on type Binary

Currently, the "binary" data format can only be used as pipeline input and output with the ZMQ connector.

However, it can be used as an intermediate format between pipeline steps without any limitations.

5.3.4 Custom data formats

To connect your pipeline to a custom application with its own data format, you can take one of the following methods:

- Use `String` and connect input or output through Databus or External Databus. In this case, you can use any text data format, such as JSON, XML, CSV, or any combination of these.
- Use `Object` and connect output via ZMQ. In this case, the AI Inference Server converts the metadata dictionary into a JSON string and passes it to the receiver together with the binary contents in a multi-part ZMQ message. For more details, refer to the AI Inference Server Function Manual (<https://support.industry.siemens.com/cs/ww/en/view/109822331>).

Specific variable types for images

AI Inference Server supports receiving URL-encoded images via MQTT. The payload type is `str` and can be extracted into a PIL image as follows:

```
# define input
component.add_input("image", "String")

# extract payload
def process_input(payload: dict):
    url_encoded_image = payload["image"]
    with urlopen(url_encoded_image) as response:
        assert response.headers["Content-type"] in ["image/png",
"image/jpeg"]
        image_bytes = response.read()
        pil_image =
Image.open(io.BytesIO(image_bytes)).resize(IMAGE_SIZE)
```

Another supported type for images is `Object` which can be used to receive or send images via ZMQ.

If the "input variable" is defined with type `Object`, the AI Inference Server takes the image from ZMQ and creates a specific payload format. In your code, this format can be processed and extracted into a PIL Image. A specific code example can be found in the Image Classification project example in the "Examples (Page 45)" chapter.

```
# define input
component.add_input("image", "Object")

# Object input format
payload = { "image":
    {
        "resolutionWidth": image.width,
        "resolutionHeight": image.height,
        "mimeType": ["image/raw"],
        "dataType": "uint8",
        "channelsPerPixel": 3,
        "image": _swap_bytes(image.tobytes())
    }
}
```

When the "output variable" is defined with type `Object`, the output must be provided in a specific format. In your code, a dictionary must be created with a `string` and a `bytes` field. They must contain the width and height information in a JSON string and the `UINT8` bytes of the raw image. A concrete code example can be found in the project example `Image Classification`.

```
# define output
component.add_output("image_with_filter", "Object")

# Object output format
return {
    "image_with_filter": {
        "metadata": json.dumps( {
            "resolutionWidth": image.width,
            "resolutionHeight": image.height
        }
    ),
    "bytes": image.tobytes()
}
```

The most commonly supported data format is "Binary", that is used to receive or send a byte array over ZMQ.

If an input variable is defined as "Binary", the AI Inference Server provides it as a Python dictionary, where the variable name is the key, and the value is the binary data provided as the Python type "bytes".

A specific code example can be found in the `Image Classification` project example in the "Examples (Page 45)" chapter.

```
# definition of input
component.add_input("image", "Binary")

# Binary input format
with open('image.png', 'rb') as f:
    binary = f.read()
    payload = { "image": binary }
...

# Decode a PIL image from Binary data
image = Image.open(io.BytesIO(binary))
...
```

If an output variable is defined with type "Binary" the output must be provided as a "bytes" value in the returned dictionary.

```
# output definition
component.add_output("processed_image", "Binary")

# Binary output format from a PIL image
membuf = io.BytesIO()
image.save(membuf, format="png")
return {
    "processed_image": membuf.getvalue()
}
```

5.3 Input data

ImageSet data type allows receiving multiple images, along with their format, dimension information, and metadata.

Example of processing an incoming ImageSet in Python:

```
# Define input
component.add_input("image_set", "ImageSet")

# Handle incoming image(s)
def process_input(data: dict):
    image_set = data['image_set']
    for image_data in image_set['image_list']:
        process_image_data(image_data['image'])
    # ...
```

Example of producing an ImageSet output in Python:

```
# Define output
component.add_output("image_set", "ImageSet")
# Assemble an ImageSet object
import json
def process_input(data):
    # ...
    image_set: {
        "version": "1",
        "camera_id": "...",
        "timestamp": "2023-08-08T09:11:12.000Z",
        "metadata": json.dumps({
            "key1": "value1",
            "key2": "value2",
            # ...
        }),
        "image_list": [{
            "id": "...",
            "width": 640,
            "height": 480,
            "format": "<GeniCam image format>",
            "timestamp": "2023-08-08T09:11:12.000Z",
            "metadata": json.dumps({
                "key1": "value1",
                "key2": "value2",
                # ...
            }),
            "image": b"..."
        }, {
            # ...
        }]
    }
    return {
        "image_set": image_set
    }
```


5.4 Processing data

The main part of your Python script is the logic for calculating the output from the input. This is done by your Python code, which can use configuration files and persistent ML models in a well-defined Python environment. To define these resources, the `PythonComponent` class of the `simaticai.deployment` module is used to add dependencies or additional files. These files are included in the configuration package and extracted in the AI Inference Server as follows:

- Dependencies are installed on the server via `pip`.
- Additional files are copied into the component directory.

5.5 Python dependencies

The AI Inference Server executes every component of a pipeline in an isolated Python virtual environment. For each component, you must specify which Python packages are required by the Python scripts in that component, including the Python packages required to load persistent Python objects.

Adding Python dependencies

The Python dependencies of a component can be added in two ways:

- As a standard wheel file or as a zip/tar file that contains standard wheel files

In both cases, the packages, which can be precompiled wheel files or pure Python source distributions, are added to the `component.dependencies` dictionary and binaries are zipped into the configuration package.

AI Inference server supports only installing source distributions that contain only Python source code.

...

```
component.add_python_packages('../packages/my_module-0.0.1-py3-any-any.whl')
```

```
component.add_python_packages('../packages/MyPackages.zip')
```

```
component.add_python_packages('../packages/my_source_module-0.0.2.tar.gz')
```

...

- By name using a list that contains the names of the Python modules

In this case, the method searches for the module in the current Python environment and adds the package with its version and all of its transitive dependencies.

```
component.add_dependencies(['numpy', 'scikit-learn'])
```

Dependencies can be added by name and version using a list that contains corresponding tuples. When the component is saved, it will perform a check if all specified dependencies can be installed together. Transitive dependencies will also be downloaded.

5.5 Python dependencies

```
...  
component.add_dependencies([('pandas', '1.3.0'), ('pyarrow',  
    '3.0.0')])  
...
```

Dependencies added to a component are installed on the AI Inference Server with the defined version and can be imported into your Python code during execution.

```
# entrypoint.py or data_processor.py  
import numpy as np  
import pandas as pd  
...
```

Download from non-public repository

Dependencies from non-public repositories can be downloaded by specifying an extra index URL at the beginning of the `requirements.txt` file.

```
...  
extra-index-url https://<API_KEY>@your/private/repository  
...
```

Please be aware that if a package is also available in a public repository, `pip` may download it from there and not look for it in the private repository, which may pose a cybersecurity risk.

It is recommended to only use trusted private repositories, pin the version of the package, and check if a package with the same properties already exists on <https://pypi.org/simple>.

If you want to change the default <https://pypi.org/simple> package index, you can do it by using an index URL at the beginning of the `requirements.txt` file. This allows you to download dependencies exclusively from a private repository.

```
...  
index-url https://<API_KEY>@your/private/repository  
...
```

5.6 File resources

File resources can be of any file type required to execute the Python code, including the Python sources themselves, such as configuration file, static data, or trained AI models stored in `joblib` or `pickle` format.

Adding resources

In order for the configuration package to transfer these files to the server environment, you must specify them using the `add_resources(base_dir, resources)` method, as shown below:

```
# the method adds 'prediction_model.joblib' from the '../models' directory file to the
component

# and the file will be extracted on the server into the component folder under the 'models'
directory

component.add_resources(base_dir="..",
resources="models/prediction_model.joblib")

# same way we define a file 'model-config.yml' to bring into the 'config' directory

component.add_resources(base_dir="..", resources="config/model-
config.yml")
```

Once the pipeline is imported into the AI Inference Server and the component is installed, the files in the server file system are available in the component directory and can be accessed by the Python scripts:

```
# data_processor.py

import yaml
import joblib

from pathlib import Path

# Our goal is to have an identical relative path to the resources in
the source repository and on the server.

base_dir = Path(__file__).parents[1]

# file 'model-config.yml' is extracted into the 'config' directory
config_path = base_dir / "config/model-config.yml"

model_config = yaml.load(config_path)

# file 'prediction_model.joblib' is extracted into the 'models'
directory

model_path = base_dir / "models/prediction_model.joblib"

with open(model_path, "rb") as model_file:

    model = joblib.load(model_file)
```

As loading files can be time-consuming, it is recommended to load files and ML models into memory at initialization time of your Python code and not during the call to `process_input()`. The entrypoint `process_input()` should focus on processing the incoming data as quickly as possible. We highly suggest initializing the objects that are used in this code at the beginning of the script, and then using them in the functions invoked by `process_input`.

Please be aware that this approach can result in a massive memory load, so you have to make a trade-off between memory consumption or CPU load and response time.

After loading, the model is ready to be used to process the input data. In simple cases, this can be done directly in the entrypoint script. In the given example, we have factored this out into a module to illustrate how another module can be called from the entrypoint.

```
# data_processor.py
def process_data(width, height):
    data=[width, height]

    class_label, confidence=model.predict(data)

    return{"class_label": class_label, "confidence": confidence}
```

5.7 Returning the result

If you want to return the results after processing the input data, you must return them in a dictionary. The keys should be the variable names of the component's outputs. In the example in File resources (Page 35), the `process_data()` function returns such a dictionary. It can be directly returned from `process_inputs()` as well. The dictionary contains an integer for `class_label` and a floating-point value that represents the `confidence` of the prediction.

If there is no output for a particular call to `process_input()`, you should return `None`. As a result, the AI Inference Server does not trigger the next component in the pipeline, or if it is the last component of the pipeline, the pipeline does not emit any output from that component. A specific use case can be found under "Use cases (Page 40)".

Returning Object

AI Inference Server version 1.5 restricts the type of dictionary that a pipeline component can return in an output variable of type `Object`. The dictionary must hold a metadata string and a binary sequence. The metadata and the binary sequence can have an arbitrary key in the `dict` but must be of type `str` and `bytes` respectively. This means that any dictionary with two elements will work if one of the elements is a `str` and the other is a `bytes` type.

For example, if you want to pass a processed version of the input image to a later component or to ZMQ, you can do it as follows:

```
# define output
component.add_output("processed_image", "Object")

# Object output format
return { "processed_image": {
    "metadata": json.dumps( {
        "mode": image.mode,
        "width": image.width,
        "height": image.height
    }
    ),
    "bytes": image.tobytes()
  }
}
```

Note

You cannot pass a dictionary received as pipeline input as component output because the structures of these dictionaries are different.

In the receiver pipeline component, you can decode the image as follows:

```
# define input
component.add_input("processed_image", "Object")
# construct PIL Object from metadata and binary data
def process_input(data: dict):
    metadata = json.loads(data['processed_image']['metadata'])
    image_data = data['processed_image']['bytes']
    mode = metadata['mode']
    width = metadata['width']
    height = metadata['height']
    image = Image.frombytes(mode, (width, height), image_data)
```

5.7.1 Returning Binary data

AI Inference Server version 1.5.0 allows data to be returned in "Byte" format as an output variable, which is defined as "Binary" in the pipeline configuration.

To pass binary data, such as an image, between components or as pipeline output, you can do the following:

```
# definition of outputs
component.add_output("prediction", "String")
component.add_output("processed_image", "Binary")

# Binary output format from a PIL image
membuf = io.BytesIO()
```

5.8 Adding custom metrics

```
image.save(membuf, format="png")
return {
    "prediction": str(prediction),
    "processed_image": membuf.getvalue()
}
```

In the receiver pipeline component, you can decode the image as follows:

```
# definition of input
component.add_input("processed_image", "Binary")
# construct PIL object from metadata and binary data
def process_input(data: dict):
    image_data = data['processed_image']
    image = Image.open(io.BytesIO(image_data))
```

5.8 Adding custom metrics

You can implement any model metrics as component metrics that you can use to evaluate the performance of the model in the AI Model Monitor. The pipeline automatically generates the metrics as outputs that are automatically mapped to the required Databus topics. In the AI Inference Server, you only need to select Databus as the data connection for these metric outputs.

A custom metric must be defined for the component as follows:

```
component.add_metric("ic_probability")
```

Note

The metric name must be prefixed and must contain an underscore (_), because the prefix is used to group custom metrics on the dashboard.

An output with the same name can be returned from the inference wrapper as follows:

```
def process_input(data: dict):

    prediction, metric_value = predict(data)

    return {
        "prediction": prediction,
        "ic_probability": json.dumps({"values": metric_value}),
    }
```

Once the pipeline is created, it collects the metrics from all components and delivers them as pipeline outputs. The AI Inference Server can continue to use them as output. The custom metric is displayed in the AI Inference Server as output with a pre-configured topic that needs to be connected to the Databus.

Note

You can also add custom metrics to a monitoring component provided by the AI SDK Monitoring Extension.

5.9 Pipeline parameters

If a component is used in a pipeline with parameters, the component must provide an `update_parameters()` function to handle parameter updates. AI Inference Server calls `update_parameters()` at least once after the pipeline has been started. However, before passing the first input to the `update_parameters()` component, it can be called again if pipeline parameters are changed while the pipeline is running. AI Inference Server ensures that calls to `update_parameters()` and `process_input()` do not occur at the same time concurrently.

Parameters to be changed on the user interface

Use individual parameters if you intend to let the operator to change pipeline parameters interactively in the AI Inference Server user interface. For each parameter, you must define the name, default value and type.

```
# define individual parameters for the pipeline
pipeline.add_parameter('windowSize', 300, 'Integer')
pipeline.add_parameter('windowStepSize', 75, 'Integer')
```

In this way, the AI Inference Server provides individual input fields for each parameter on its user interface. In the handler function, the parameters can be retrieved as individual dictionary elements from the function parameters.

```
# entrypoint.py
def update_parameters(params: dict):
    windowSize = params['windowSize']
    windowStepSize = params['windowStepSize']
```

Parameters to be changed via MQTT

If you need to change pipeline parameters programmatically by sending MQTT messages, use a single composite parameter of type `String` and combine multiple parameters into JSON. For the AI Inference Server to provide parameter values from an MQTT topic, you must enable this function by passing `True` as an additional, fourth argument.

```
# define compound JSON parameter for the pipeline
pipeline.add_parameter('windowing', json.dumps({'windowSize': 300,
'stepSize': 75}), 'String', True)
```

5.10 Use cases

Therefore, the AI Inference Server allows you to map this parameter to an MQTT topic, which is similar to the method of mapping input and output variables. In the handler function, the parameters can be retrieved by first unfolding the JSON in the individual formal pipeline parameter into a dictionary, then they can be accessed individually.

```
def update_parameters(params: dict):
    windowing = json.loads(params['windowing'])
    windowSize = windowing['windowSize']
    windowStepSize = windowing['stepSize']
```

5.10 Use cases

The following sections provide guidance on how to use inputs and outputs as intended in selected typical ML use cases.

5.10.1 Processing images

AI Inference Server version 1.5 supports image input from the Vision Connector application. The connection can be made via Databus or ZMQ. Depending on the connection, the images are represented either as input `Strings` or input `Objects`.

Using ZMQ connection

For use cases with large images or high input rates, use a ZMQ connection because the Databus is not designed to handle large quantities of data. For image input via ZMQ, you need to define the input variable as `Object` as follows:

```
# define input
component.add_input('vision_payload', 'Object')
```

Currently, the MIME type, the data type, and the number of channels are limited to the values defined in the example. AI Inference Server converts the Vision Connector payload received via ZMQ into a Python imagery data dictionary with the following entries:

```
# Object input format
image_data = { 'image':
    {
        'resolutionWidth': 640,
        'resolutionHeight': 480,
        'mimeType': 'image/raw',
        'dataType': 'uint8',
        'channelsPerPixel': 3,
        'image': 'placeholder for binary image contents'
    }
}
```


You can extract the binary image into a PIL image:

```
from PIL import Image

def process_input(data: dict):
    image_data = data['vision_payload']

    assert image_data['dataType'] == 'uint8' and
    image_data['channelsPerPixel'] == 3

    width = image_data['resolutionWidth']
    height = image_data['resolutionHeight']

    # image received with 'BGR' byte order
    return Image.frombytes('RGB', (width, height), image_data['image'],
    'raw', 'BGR')
```

Sending multiple images as ImageSet

If you want to send multiple images in a single payload, you should use the ImageSet data type. Refer to Image Classification project template for a complete example for MQTT and ZMQ connection variants.

```
...
# define input
component.add_input('vision_payload', 'String')
...
```

AI Inference Server passes through the payload provided by the Vision Connector as a dict, with the following fields:

Example of producing an ImageSet payload

```
payload = {
    "version": "1",
    "camera_id": "...",
    "timestamp": "2023-08-08T09:11:12.000Z",
    "metadata": json.dumps({
        "key1": "value1",
        "key2": "value2",
        # ...
    }),
    "image_list": [{
        "id": "...",
        "height": 480,
        "format": "<GeniCam image format>",
        "timestamp": "2023-08-08T09:11:12.000Z",
        "metadata": json.dumps({
            "key1": "value1",
            "key2": "value2",
            # ...
        }),
        "image": b"..."
    }],
    {
```

```

        # ...
    }]
}

```

To extract the images of an image set:

```

...
from PIL import Image

def process_input(data: dict):
    payload = data['vision_payload']
    pil_images = [Image.frombytes(image["format"], (image["width"],
image["height"]), image["image"]) for image in
image_set['image_list']]
...

```

Using Databus connection

For use cases with small images and low input rates, you can use Databus. In this case, you need to define the type of component input variable for passing the image as a `String` that corresponds to a Python `str` in the input dictionary.

```

# define input
component.add_input('vision_payload', 'String')

```

AI Inference Server passes through the payload provided by the Vision Connector directly as a string. This string is a JSON file that contains metadata and the image itself in data URL-encoded form as follows:

```

# example image in Vision Connector MQTT JSON format
{
    'timestamp': '2022-02-23T09:29:45.276338',
    'sensor_id': 'a204dba4-274e-43ce-9a71-55de9e715e72',
    'image':
'data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAASwAAAGQCAIA...QmCC',
    # note this URL encoded string is truncated
    'status':
    {
        'genicam_signal': {'code': 3}
    }
}

```

You can extract the URL-encoded image into a PIL image object as follows:

```

# extract payload
from urllib.request import urlopen
from PIL import Image

def process_input(data: dict):
    payload = json.loads(data['vision_payload'])
    url_encoded_image = payload['image']
    with urlopen(url_encoded_image) as response:
        assert response.headers['Content-type'] in ['image/png',
'image/jpeg']
    image_bytes = response.read()
    return Image.open(io.BytesIO(image_bytes))

```

Refer to Image Classification project template for a complete example of MQTT and ZMQ connection variants.

For more details, refer to the Vision Connector User Guide.

5.10.2 Processing time series of signals

For time series use cases, the situation is typically as follows. You have several signals, that you want to sample at a regular rate, and you want to feed your ML model with a time window of multiple samples at once. For example, your model might be designed to expect a window of 5 samples for 3 variables.

Time stamp	var1	var2	var3
09:50:23	1.2	202	25
09:50:28	1.3	230	5
09:50:33	1.2	244	18
09:50:38	1.2	244	18
09:50:43	1.2	244	18

Ideally, each signal is a variable that is read by the Industrial Edge Databus, as configured in the AI Inference Server. Often, the signals from PLCs are captured using the Industrial Edge S7 Connector, which samples these signals from given PLC tags and makes them available on the Databus.

Unfortunately, data points for the signals do not necessarily arrive at a regular rate or synchronously. By default, your Python script is usually called with a single variable and the others are `None`. However, the ML model expects an entire matrix of multiple variables in an entire time window.

Time stamp	var1	var2	var3
09:50:28	None	230	None

To ensure the regular rate and the synchronicity of inputs, the AI Inference Server supports inter-signal alignment. You can specify a time interval and receive the inputs for all variables stimulated in that interval. In our example, you specify 5 seconds as the time interval and receive inputs such as:

Time stamp	var1	var2	var3
09:50:28	1.3	230	5

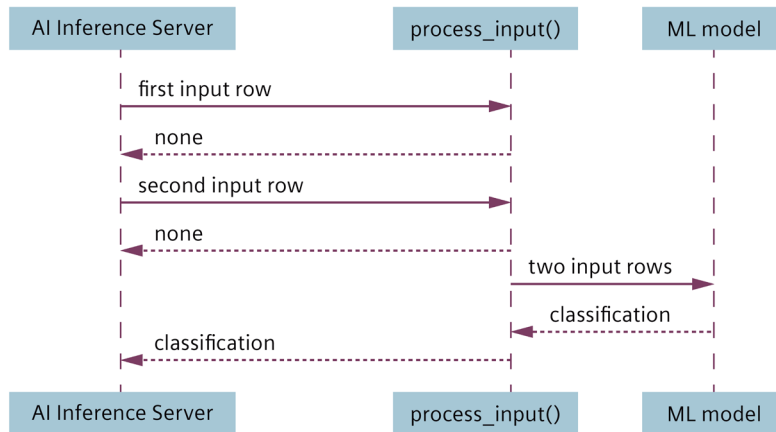
As there is no guarantee that the data source will deliver a data point in each interval, there is still a possibility that some values are missing and set to `None` in an input row. However, if the sample rate of inter-signal alignment does not exceed the data rate of the sources, your Python script will mostly provide complete rows of data.

For details of inter-signal alignment, refer to the AI Inference Server user's manual.

Refer to the packaging notebook in the State Identifier project template and the AI SDK API reference to specify inter signal alignment to be applied to the input of an ML pipeline when it is packaged for deployment to the AI Inference Server.

However, rows are not yet enough to feed time series ML models when they need data windows consisting of multiple rows. Since the AI Inference Server does not support accumulating windows, the Python script must take over this task. The key point of the server's script interface here is that not all inputs in your Python script result in an output because the ML model can only produce output if the received input has just completed a data window that can be passed to the model to calculate an output.

As described above in Returning the result (Page 36), the script can return `None` while it accumulates input, and the model cannot calculate a value for the output. For reasons of compactness, the following diagram shows this for a window size of two.



This can be even more complex in real life, depending on whether the windows are overlapping or not. For a concrete implementation of such accumulation logic, refer to the Python script provided in the State Identifier project template.

Note that you cannot use parallel component execution if your component relies on building up windows from subsequent data points, as the data points would be distributed to different instances of the component. You can, however, separate the aggregation of data windows and CPU-intensive processing of the windows into a component each, and enable parallel execution for the latter only.

Even in this case, there is no guarantee that the processing of the windows finishes in the original sequence of the data. If that is essential, you should supply the windows with a sequence ID in the aggregating component that you pass on to the output of the processing component. That way the consumer of the pipeline can recreate the original sequence.

5.10.3 Processing batch data

In batch use cases, such as the classification of discretely manufactured items, the input data is often available in a single packet on the Databus in a text representation such as a JSON structure or CSV table. For example, data is provided through External Databus as a textual payload. The Industrial Edge Vision Connector also delivers images embedded in a JSON through Databus.

Such data is treated as a single variable of type `String`, which is passed then to `process_input()` as a dictionary with a single element. You need to process this input string according to which representation it uses, for example, `json.loads()` for JSON or a combination of `splitlines()` and `csv.reader()` for CSV.

For a concrete code example that shows how to process a single input variable with a JSON structure refer to the Image Classification project template.

5.11 Examples

Component definition example

The following script creates an example pipeline with a single component, two inputs and two outputs.

```
from simaticai import deployment

# defines basic properties of the pipeline step
component = deployment.PythonComponent(name='classifier',
version='1.0.0', python_version='3.8')

component.add_input(name='input_1', _type='Double')
# defines input variable
component.add_input(name='input_2', _type='Double')
# defines input variable
component.add_output(name='class_label', _type='Integer')
# defines output variable
component.add_output(name='confidence', _type='Double')
# defines output variable
component.add_resources("../src/", "entrypoint.py")
component.set_entrypoint('entrypoint.py')

pipeline = deployment.Pipeline.from_components([component],
name='Example', version='1.0.0')

pipeline_package_path = pipeline.save('../packages')
```

The above code generates a `pipeline-config.yml` that contains, among others, the following:

```
# pipeline-config.yml
components:
  name: classifier
  entrypoint: entrypoint.py
  version: 1.0.0
  runtime:
    type: python
    version: 3.8
  inputType:
    - name: input_1
      type: Double
```

5.11 Examples

```

- name: input_2
  type: Double
outputType:
- name: class_label
  type: Integer
- name: confidence
  type: Double

```

The pipeline looks as follows:

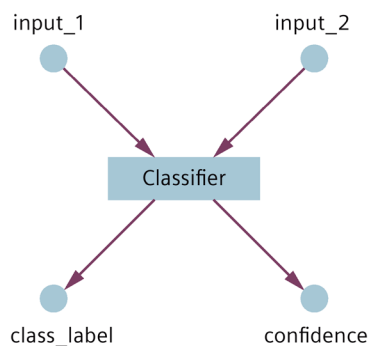


Image Classification

The following script creates an image classification pipeline that consists of a single component. The pipeline processes images embedded in JSON strings and produces a classification result as a string. This example is detailed in the Image Classification project template.

```

from simaticai import deployment

# create pipeline component and define basic properties
component = deployment.PythonComponent(name='inference',
version='1.0.0', python_version='3.8')
component.add_input('vision_payload', 'String')
# define a single input variable
component.add_output('prediction', 'String')
# define a single output variable
component.add_resources('.', 'entrypoint.py')
# add Python script
component.set_entrypoint('entrypoint.py')
# define the above script as entrypoint
component.add_resources('.', 'src/vision_classifier_tflite.py')

```

```
# add classifier script used by entrypoint
component.set_requirements("../runtime_requirements_tflite.txt")
# define required Python packages
component.add_resources('.',
    'models/classification_mobilnet.tflite')
# add saved model used in classifier
component.set_parallel_steps(2) # set the number of parallel executors
# create and save a pipeline consisting of a single component
pipeline = deployment.Pipeline.from_components([component],
    name='Image_TFLite_package', version='1.0.0')
pipeline_package_path = pipeline.save('../packages')
# convert pipeline configuration package to edge configuration
package
deployment.convert_package(pipeline_package_path)
```

The above code generates a `pipeline_config.yml` that contains, among other things:

```
dataFlowPipeline:
  components:
    - entrypoint: ./entrypoint.py
      inputType:
        - name: vision_payload
          type: String
      name: inference
      outputType:
        - name: prediction
          type: String
      runtime:
type: python
  version: '3.8'
  version: 1.0.0
pipelineDag:
  - source: Databus.vision_payload
    target: inference.vision_payload
  - source: inference.prediction
    target: Databus.prediction
```

5.11 Examples

```
pipelineInputs:
  - name: vision_payload
    type: String
pipelineOutputs:
  - name: prediction
    type: String
dataFlowPipelineInfo:
  dataFlowPipelineVersion: 1.0.0
  projectName: Image_TFLite_package
```

The saved pipeline configuration package contains the files listed below. The main folder contains the YAML files that describe the pipeline. The `inference` subfolder contains the files that belong to this component.

```
Image_TFLite_package_1.0.0/pipeline_config.yml
Image_TFLite_package_1.0.0/datalink_metadata.yml
Image_TFLite_package_1.0.0/inference/entrypoint.py
Image_TFLite_package_1.0.0/inference/requirements.txt
Image_TFLite_package_1.0.0/inference/src/vision_classifier_tflite.py
Image_TFLite_package_1.0.0/inference/models/classification_mobilnet.
tflite.py
```


5.12 Writing components for earlier versions of the AI Inference Server

AI Inference Server versions up to 1.1 require the entrypoint to define a function named as `run()` instead of `process_input()`. Not only is the function name different, the component inputs and outputs are passed differently too.

AI Inference Server version 1.1 passes the input variables as a JSON string, which you must convert to a dictionary. On the output side, you must also pass the outputs as a JSON string, and embed it all into a dictionary with a `ready` flag.

The following code example shows how to wrap `process_input()` into a `run()` function compatible with AI Inference Server 1.1.

```
# entrypoint.py

...

# compatibility run() wrapper for process_input()
def run(data: str) -> dict:
    input_data = json.loads(data)

    result = process_input(input_data)

    if result is None:
        answer = {"ready": False, "output": None}
    else:
        answer = {"ready": True, "output": json.dumps(result)}
    return answer
```