Template for Application Repositories

Manual

IO-Aero Team

Nov 10, 2024

Table of Contents

Chapter 1

General Documentation

This section contains the core documentation for setting up and starting with IO-TEMPLATE-APP. It covers everything from installation to basic and advanced configurations.

1.1 Introduction

TODO

1.2 Requirements

The required software is listed below. Regarding the corresponding software versions, you will find the detailed information in the Release Notes.

1.2.1 Operating System

The supported operating system is Ubuntu with the Bash shell.

1.2.2 Python

This project utilizes Python from version 3.10, which introduced significant enhancements in type hinting and type annotations. These improvements provide a more robust and clear definition of function parameters, return types, and variable types, contributing to improved code readability and maintainability. The use of Python 3.12 ensures compatibility with these advanced typing features, offering a more structured and error-resistant development environment.

1.2.3 Docker Desktop

The project employs PostgreSQL for data storage and leverages Docker images provided by PostgreSQL to simplify the installation process. Docker Desktop is used for its ease of managing and running containerized applications, allowing for a consistent and isolated environment for PostgreSQL. This approach streamlines the setup, ensuring that the database environment is quickly replicable and maintainable across different development setups.

1.2.4 Miniconda

Some of the Python libraries required by the project are exclusively available through Conda. To maintain a minimal installation footprint, it is recommended to install Miniconda, a smaller, more lightweight version of Anaconda that includes only Conda, its dependencies, and Python.

By using Miniconda, users can access the extensive repositories of Conda packages while keeping their environment lean and manageable. To install Miniconda, follow the instructions provided in the `scripts` directory of the project, where the operating system-specific installation script named

`run_install_miniconda` is available for Ubuntu (Bash shell).

Utilizing Miniconda ensures that you have the necessary Conda environment with the minimal set of dependencies required to run and develop the project efficiently.

1.2.5 DBeaver Community - optional

DBeaver is recommended as the user interface for interacting with the PostgreSQL database due to its comprehensive and user-friendly features. It provides a flexible and intuitive platform for database management, supporting a wide range of database functionalities including SQL scripting, data visualization, and import/export capabilities. Additionally, the project includes predefined connection configurations for DBeaver, facilitating a hassle-free and streamlined setup process for users.

1.3 Installation

1.3.1 Python

The project repository contains a `scripts` directory that includes operating system-specific installation scripts for Python, ensuring a smooth setup across various environments.

- **Ubuntu**: For users on Ubuntu, the `run_install_python.sh` script is provided. This Bash script is created to operate within the default shell environment of Ubuntu, facilitating the Python installation process.

1.3.2 AWS Command Line Interface

Within the project's `scripts` directory, you will find a set of scripts specifically designed for the installation of the AWS Command Line Interface (AWS CLI). These scripts facilitate the installation process on different operating systems, ensuring a consistent and reliable setup.

- **Ubuntu**: Ubuntu users should utilize the `run_install_aws_cli.sh` script. This script is a Bash script that simplifies the AWS CLI installation on Ubuntu systems by setting up the necessary repositories and installing the CLI via apt-get.

1.3.3 Miniconda

The `scripts` directory includes a collection of operating system-specific scripts named `run_install_miniconda` to streamline the installation of Miniconda. These scripts are designed to cater to the needs of different environments, making the setup process efficient and user-friendly.

- **Ubuntu Bash Shell**: Ubuntu users can take advantage of the `run_install_miniconda.sh` script. This Bash script is intended for use within the Ubuntu terminal, encapsulating the necessary commands to install Miniconda seamlessly on Ubuntu systems.

1.3.4 Docker Desktop

The `scripts` directory contains scripts that assist with installing Docker Desktop on Ubuntu, facilitating an automated and streamlined setup.

- **Ubuntu**: The `run_install_docker.sh` script is available for Ubuntu users. This Bash script sets up Docker Desktop on Ubuntu systems by configuring the necessary repositories and managing the installation steps through the system's package manager.

1.3.5 DBeaver - optional

DBeaver is an optional but highly recommended tool for this software as it offers a user-friendly interface to gain insights into the database internals. The project provides convenient scripts for installing

DBeaver on Ubuntu.

- **Ubuntu**: For Ubuntu users, the `run_install_dbeaver.sh` script facilitates the installation of DBeaver. This Bash script automates the setup process, adding necessary repositories and handling the installation seamlessly.

1.3.6 Python Libraries

The project's Python dependencies are managed partly through Conda and partly through pip. To facilitate a straightforward installation process, a Makefile is provided at the root of the project.

- **Development Environment**: Run the command `make conda-dev` from the terminal to set up a development environment. This will install the necessary Python libraries using Conda and pip as specified for development purposes.
- **Production Environment**: Execute the command `make conda-prod` for preparing a production environment. It ensures that all the required dependencies are installed following the configurations optimized for production deployment.

The Makefile targets abstract away the complexity of managing multiple package managers and streamline the environment setup. It is crucial to have both Conda and the appropriate pip tool available in your system's PATH to utilize the Makefile commands successfully.

1.4 Configuration IO-TEMPLATE-APP

1.4.1 .act_secrets

This file controls the secrets of the `make action` functionality. This file is not included in the repository. The file `.act_secrets_template` can be used as a template.

The customisable entries are:

Parameter	Description
GLOBAL_USER_EMAIL	The global email address for GitHub

Examples:

```
GLOBAL_USER_EMAIL=a@b.com
```

1.4.2 .settings.io_aero.toml

This file controls the secrets of the application. This file is not included in the repository. The file `.settings.io_aero_template.toml` can be used as a template.

The customisable entries are:

Parameter	Description
postgres_password	Password of the database user
postgres_password_admin	Password of the database administrator

The secrets can be set differently for the individual environments (`default` and `test`).

Examples:

```
[default]
postgres_password = "..."
postgres_password_admin = "..."

[test]
```

```
postgres_password = "postgres_password"
postgres_password_admin = "postgres_password_admin"
```

1.4.3 settings.io_aero.toml

This file controls the behaviour of the application.

The customisable entries are:

Parameter	Description
check_-value	`default` for productive operation, `test` for test operation
is_verbose	Display progress messages for processing

The configuration parameters can be set differently for the individual environments (`default` and `test`).

Examples:

```
[default]
check_value = "default"
is_verbose = true

[test]
check_value = "test"
```

1.5 Configuration Logging

In **IO-TEMPLATE-APP** the Python standard module for logging is used - details can be found here.

The file `logging_cfg.yaml` controls the logging behaviour of the application.

Default content:

```
version: 1

disable_existing_loggers: False

formatters:
  simple:
    format: "%(asctime)s [%(name)s] [%(module)s.py  ] %(levelname)-5s
 %(funcName)s:%(lineno)d %(message)s"
  extended:
    format: "%(asctime)s [%(name)s] [%(module)s.py  ] %(levelname)-5s
 %(funcName)s:%(lineno)d \n%(message)s"

handlers:
  console:
    class: logging.StreamHandler
    level: INFO
    formatter: simple
  file_handler:
    class: logging.FileHandler
    level: INFO
    filename: logging_io_aero.log
    formatter: extended

root:
  level: DEBUG
  handlers: [ console, file_handler ]
```

1.6 First Steps

To get started, you'll first need to clone the repository, which contains essential scripts for various operating systems. After cloning, you will use these scripts to install the necessary foundational software. Finally, you will complete the repository-specific installation to set up your environment correctly. Detailed instructions for each of these steps are provided below.

1.6.1 Cloning the Repository

Start by cloning the *io-template-app* repository. This repository contains essential scripts and configurations needed for the project.

```
git clone https://github.com/io-aero/io-template-app
```

1.6.2 Install Foundational Software

Once you have successfully cloned the repository, navigate to the cloned directory.

To set up the project on an Ubuntu system, the following steps should be performed in a terminal window within the repository directory:

a. Grant Execute Permission to Installation Scripts

Provide execute permissions to the installation scripts:

```
chmod +x scripts/*.sh
```

b. Install Python and pip

Run the script to install Python and pip:

```
./scripts/run_install_python.sh
```

c. Install AWS Command Line Interface

Execute the script to install the AWS CLI:

```
./scripts/run_install_aws_cli.sh
```

d. Install Miniconda and the Correct Python Version

Use the following script to install Miniconda and set the right Python version:

```
./scripts/run_install_miniconda.sh
```

e. Install Docker Desktop

This step is not required for WSL (Windows Subsystem for Linux) if Docker Desktop is installed in Windows and is configured for WSL 2 based engine.

To install Docker Desktop, run:

```
./scripts/run_install_docker.sh
```

f. Install Terraform

To install Terraform, run:

```
./scripts/run_install_terraform.sh
```

g. Optionally Install DBeaver

If needed, install DBeaver using the following script:

```
./scripts/run_install_dbeaver.sh
```

h. Close the Terminal Window

Once all installations are complete, close the terminal window.

1.6.3 Repository-Specific Installation

After installing the basic software, you need to perform installation steps specific to the *io-template-app* repository. This involves setting up project-specific dependencies and environment configurations. To perform the repository-specific installation, the following steps should be performed in a command prompt or a terminal window (depending on the operating system) in the repository directory.

1.6.4 Setting Up the Python Environment

To begin, you'll need to set up the Python environment using Miniconda, which is already pre-installed. You can use the provided Makefile for managing the environment.

a. For production use, run the following command:

```
make conda-prod
```

b. For software development, use the following command:

```
make conda-dev
```

These commands will create and configure a virtual environment for your Python project, ensuring a clean and reproducible development or production environment. The virtual environment is automatically activated by the Makefile, so you don't need to activate it manually.

1.6.5 Minor Adjustments for GDAL

The installation of the GDAL library requires the following minor operating system-specific adjustments:

In Ubuntu, the GDAL library must be installed as follows:

```
sudo apt-get install gdal-bin libgdal-dev
```

1.6.6 System Testing with Unit Tests

If you have previously executed *make conda-dev*, you can now perform a system test to verify the installation using *make test*. Follow these steps:

a. Run the System Test:

Execute the system test using the following command:

```
make tests
```

This command will initiate the system tests using the previously installed components to verify the correctness of your installation.

b. Review the Test Results:

After the tests are completed, review the test results in the terminal. Ensure that all tests pass without errors.

If any tests fail, review the error messages to identify and resolve any issues with your installation.

1.6.7 Downloading Database Files (Optional)

Database files can be downloaded from the IO-Aero Google Drive directory *io_aero_data/TO DO /database/TO DO* to your local repository directory *data*. Before extracting, if a *postgres* directory exists within the *data* directory, it should be deleted.

Follow these steps to manage the database files:

a. Access the IO-Aero Google Drive Directory:

Navigate to the IO-Aero Google Drive and locate the directory *io_aero_data/TO DO/database/TO DO*.

b. Download Database Files:

Download the necessary database files from the specified directory to your local repository directory *data*.

c. Delete Existing *postgres* Directory (if present):

If a directory named *postgres* already exists within the *data* directory, you should delete it to avoid conflicts.

d. Extract Database Files:

The downloaded database files are in an archive format (ZIP) and should be extracted in the *data* directory. After completing these steps, the database files should reside in the *data* directory of your local repository and will be ready for use.

1.6.8 Creating the Docker Container with PostgreSQL DB

To create the Docker container with PostgreSQL database software, you can use the provided *run_io_template_app* script. Depending on your operating system, follow the relevant instructions below:

```
./scripts/run_io_template_app.sh s_d_c
```

This command will initiate the process of creating the Docker container with PostgreSQL database software.

1.7 Advanced Usage

TODO

Development

2.1 Makefile Documentation

This document provides an overview of the Makefile used to support the development process for Python applications. It describes each target, its purpose, and the tools involved.

2.1.1 Makefile Functionalities

General Information

- **MODULE**: The name of the main application module, here set as `iotemplateapp`.
- **PYTHONPATH**: Specifies the paths included in the Python environment, covering `docs`, `scripts`, and `tests`.
- **ARCH** and **OS**: Set up to detect the operating system and architecture to configure `DOCKER2EXE` settings.
- **ENV_FOR_DYNACONF**: Set to `test` for the Dynaconf environment.
- **LANG**: Language encoding set to `en_US.UTF-8`.

Available Targets

Each target in the Makefile is documented below:

Development and CI

- **`help`**: Lists available targets and their descriptions.
- **`dev`**: Runs code formatting, linting, and tests.
- **`docs`**: Generates documentation with Sphinx.
- **`everything`**: Runs `dev` and `docs` targets for pre-checkin verification.
- **`final`**: A full workflow, running code formatting, linting, documentation generation, and tests.
- **`pre-push`**: Prepares code for pushing by formatting, linting, running tests, incrementing version, and building documentation.

Code Formatting and Linting

- **`format`**: Formats code using `Black` and `docformatter`.
- **`lint`**: Runs a suite of linters, including `ruff`, `Bandit`, `Vulture`, `Pylint`, and `Mypy`.
- **`black`**: Formats Python code for consistency.

- **`docformatter`**: Formats docstrings to comply with PEP 257.
- **`ruff`**: Runs an optimized linter and formatter.

Testing

- **`tests`**: Runs all tests with `pytest`.
- **`pytest-ci`**: Installs `pytest` dependencies, then runs tests.
- **`pytest-first-issue`**: Runs `pytest` but stops at the first failure.
- **`pytest-ignore-mark`**: Runs all tests excluding those marked with `no_ci`.
- **`pytest-issue`**: Runs only tests marked with `issue`.
- **`pytest-module`**: Runs tests on a specific module.

Static Analysis and Security

- **`bandit`**: Checks for common security issues.
- **`vulture`**: Detects unused code.
- **`mypy`**: Performs static type checking.
- **`mypy-stubgen`**: Generates stub files for type hinting.

Environment and Version Management

- **`conda-dev`**: Sets up a development Conda environment.
- **`conda-prod`**: Sets up a production Conda environment.
- **`next-version`**: Increments the project's version.
- **`version`**: Displays versions of installed dependencies.

Documentation

- **`sphinx`**: Generates HTML and PDF documentation with Sphinx.

Docker

- **`docker`**: Builds a Docker image and prepares executables using `docker2exe`.

2.1.2 How to Use the Makefile

1. **Install Dependencies**: Make sure all tools required by the Makefile are installed in your environment.
2. **Common Commands**:
   - `make dev`: Runs the development workflow, including formatting, linting, and testing.
   - `make docs`: Generates and verifies the documentation.
   - `make everything`: Runs a comprehensive check (development and documentation).
   - `make pre-push`: Prepares code for committing to the repository by ensuring code quality and documentation.
   - `make conda-dev` or `make conda-prod`: Creates Conda environments for development or production.
3. **Testing Specifics**:
   - `make tests`: Runs all tests.

- `make pytest-module TEST-MODULE=<module>`: Runs tests for a specific module by setting the `TEST-MODULE` variable.

2.1.3 Tool Analysis

Summary of Tools Used

Each tool and its relevance to the development process is explained below:

1. **Act (`nektos/act`)**: Allows GitHub Actions to run locally, facilitating CI/CD testing without needing GitHub's infrastructure. This can be helpful for testing workflows before pushing code to remote repositories.
2. **Bandit**: Focuses on identifying security issues in Python code. This tool is essential for ensuring the security of the application.
3. **Black**: A widely-used code formatter that enforces a consistent coding style, improving readability and reducing merge conflicts.
4. **Compileall**: Byte-compiles all Python scripts, which can help with performance optimization and testing compiled code for syntax validity.
5. **Conda**: Manages isolated environments with different dependencies, ensuring compatibility across development, testing, and production setups.
6. **Coveralls**: Reports test coverage to coveralls.io, which is essential for understanding code coverage metrics in CI/CD pipelines.
7. **Docformatter**: Formats docstrings to comply with PEP 257, enhancing readability and consistency in documentation.
8. **Docker** and **docker2exe**: Docker is essential for containerizing applications. `docker2exe` helps create executables from Docker images, which is useful for building standalone applications.
9. **Mypy**: Provides static type checking, reducing runtime errors and improving code clarity.
10. **Pylint**: Performs a comprehensive linting process, identifying code smells and enforcing PEP 8 compliance.
11. **Pytest**: Runs tests in an organized manner, and various configurations allow different testing scopes, from unit tests to CI-oriented tests.
12. **Ruff**: A fast Python linter with a broad range of checks, offering a high-performance alternative to other linters.
13. **Sphinx**: Generates documentation from docstrings and reStructuredText, providing an essential resource for developers and end-users.
14. **Vulture**: Detects unused code, which helps keep the codebase clean and optimized.

Necessity and Redundancy

- **Necessary Tools**: All tools involved contribute to either code quality, security, testing, or documentation, making them generally valuable for a robust development environment.
- **Potential Redundancies**:
  - Both `ruff` and `pylint` serve similar linting purposes. If speed is a concern, `ruff` alone might be sufficient for linting.
  - `Compileall` may be optional unless there's a specific need for bytecode testing or distribution.

Recommendations

1. **Consolidate Linting**: Depending on project requirements, you might choose `ruff` over `pylint` to streamline the linting process.
2. **Assess Docker Usage**: Ensure that `docker2exe` is necessary for the application's deployment process, as it may add complexity to Docker builds.

This Makefile provides a comprehensive setup for Python application development, encompassing formatting, linting, testing, security analysis, documentation, and deployment. Each target streamlines common tasks, aiding both local development and CI/CD workflows.

API Documentation

Here, you will find detailed API documentation, which includes information about all modules within the IO-TEMPLATE-APP, allowing developers to understand the functionalities available.

3.1 iotemplateapp

3.1.1 iotemplateapp package

Submodules

iotemplateapp.glob_local module

Global constants and variables.

iotemplateapp.glob_local.**ARG_TASK = 'task'**
A constant key used to reference the 'task' argument in function calls and command line arguments throughout the software.

Type str

iotemplateapp.glob_local.**ARG_TASK_CHOICE = ''**
Initially set to an empty string, this variable is intended to hold the user's choice of task once determined at runtime.

Type str

iotemplateapp.glob_local.**ARG_TASK_VERSION = 'version'**
A constant key used to reference the 'version' argument for tasks, indicating the version of the task being used.

Type str

iotemplateapp.glob_local.**CHECK_VALUE_TEST = True**
A boolean indicating whether the check value from io_settings is 'test'.

Type bool

iotemplateapp.glob_local.**FATAL_00_926 = "FATAL.00.926 The task '{task}' is invalid"**
Error message template indicating that the specified task is invalid.

Type str

iotemplateapp.glob_local.**INFO_00_004 = 'INFO.00.004 Start Launcher'**
Information message indicating the start of the launcher.

Type str

iotemplateapp.glob_local.**INFO_00_005 = "INFO.00.005 Argument '{task}'='{value_task}"'**
Information message indicating the value of a specific argument in the launcher.

Type str

iotemplateapp.glob_local.**INFO_00_006 = 'INFO.00.006 End Launcher'**
Information message indicating the end of the launcher.

Type str

iotemplateapp.glob_local.**INFO_00_007 = "INFO.00.007 Section: '{section}' - Parameter: '{name}'='{value}"'**
Information message indicating the value of a specific configuration parameter.

Type str

iotemplateapp.glob_local.**INFORMATION_NOT_YET_AVAILABLE = 'n/a'**
Placeholder indicating that information is not yet available.

Type str

iotemplateapp.glob_local.**IO_TEMPLATE_APP_VERSION = '9.9.9'**
The current version number of the IO-Aero template application.

Type str

iotemplateapp.glob_local.**LOCALE = 'en_US.UTF-8'**
Default locale setting for the system to 'en_US.UTF-8', ensuring consistent language and regional format settings.

Type str

iotemplateapp.templateapp module

IO-TEMPLATE-APP interface.

iotemplateapp.templateapp.**ARG_TASK = ''**
Placeholder for the command line argument 'task'.

Type str

iotemplateapp.templateapp.**check_arg_task** (*args: Namespace*) → None
Check the command line argument: -t / –task.
Args:

args (argparse.Namespace): Command line arguments.

iotemplateapp.templateapp.**get_args** ( ) → None
Load the command line arguments into the memory.

iotemplateapp.templateapp.**progress_msg** (*msg: str*) → None
Create a progress message.
Args:

msg (str): Progress message.

iotemplateapp.templateapp.**progress_msg_time_elapsed** (*duration: int, event: str*) → None
Create a time elapsed message.
Args:

duration (int): Time elapsed in ns. event (str): Event description.

iotemplateapp.templateapp.**terminate_fatal** (*error_msg: str*) → None
Terminate the application immediately.
Args:

error_msg (str): Error message.

iotemplateapp.templateapp.**version** ( ) → str
Return the version number of the IO-XPA-DATA application.

Returns **str**

Return type The version number of the IO-XPA-DATA application.

Module contents

IO-TEMPLATE-APP.

Tool Evaluations

4.1 Tool Evaluations

4.1.1 Python Package Manager Alternatives

When managing Python packages, you have several tools to choose from, each with its own strengths and weaknesses. Here's a comparison of four popular package managers—**Mamba**, **Vu**, **Poetry** (with a system package manager), and **Pip with Virtual Environments (Pip + venv)**—to help you decide which one best suits your needs.

1. Mamba

Overview: Mamba is a free, open-source package manager designed as a faster, drop-in replacement for Conda. It's optimized to resolve dependencies quickly, especially useful in large environments with complex dependencies like `GDAL` and `PDAL`.

Pros:

- **Performance**: Mamba's dependency resolution and installation speeds are significantly faster than Conda's, thanks to its C++ core.
- **Full Conda Compatibility**: Mamba is fully compatible with Conda environments, channels (e.g., Conda-Forge), and configuration files, allowing you to use it interchangeably with Conda.
- **Non-Python Dependency Support**: Mamba handles complex non-Python dependencies smoothly, making it highly suitable for packages like `GDAL` and `PDAL`.

Cons:

- **Young Ecosystem**: While Mamba has a growing user base, it's still newer than Conda, which might mean slightly less extensive community support for troubleshooting.

Suitability: **Highly recommended**. Mamba provides all the functionality of Conda with much better performance and is free, making it ideal for scientific projects with complex dependencies.

---

2. Vu

Overview: Vu is a relatively new package manager developed as a fast, dependency-resolving alternative with a specific focus on machine learning, data science, and applications requiring complex dependencies.

Pros:

- **High Performance**: Vu is designed with fast dependency resolution in mind, competing with

Mamba in terms of speed.

- **Built for Complex Environments**: Vu emphasizes support for machine learning and scientific libraries, which include dependencies like `GDAL`, `PDAL`, and others common in data science.
- **Free and Open Source**: Vu is completely free to use, targeting scientific and academic users with a focus on performance.

Cons:

- **Ecosystem and Community**: Vu is relatively new and not as widely adopted as Conda/Mamba, which can limit the availability of community support, tutorials, and resources.
- **Compatibility**: Vu is still building out its compatibility with certain ecosystem features (e.g., all Conda channels), which could create minor compatibility issues in larger projects.

Suitability: **Recommended for experimentation** if you're looking for speed and can work around potential ecosystem limitations. Vu's focus on scientific dependencies makes it a promising choice for data science projects, though its ecosystem is less mature than Mamba's.

---

3. Poetry (with a System Package Manager)

Overview: Poetry is a Python package manager focused on dependency management, versioning, and publishing. It's lightweight and often faster than Conda for Python-only projects but lacks native support for non-Python dependencies.

Pros:

- **Efficient for Python-Only Projects**: Poetry's dependency resolver is fast and well-suited to pure Python projects.
- **Standardized Configuration**: Uses `pyproject.toml`, which is now part of the official Python packaging specification.
- **Free and Open Source**: Poetry is fully free and widely adopted.

Cons:

- **Limited Support for Non-Python Dependencies**: For non-Python libraries like `GDAL` and `PDAL`, you'd need to install dependencies manually using a system package manager like `apt` or `brew`.
- **Complexity with Mixed Dependencies**: Managing both Poetry and a system package manager can complicate the setup, especially for large projects.

Suitability: **Recommended only for Python-centric projects**. If your projects often require `GDAL`, `PDAL`, or other complex dependencies, Poetry will be challenging to configure and maintain.

---

4. Pip with Virtual Environments (Pip + venv)

Overview: Pip with `venv` (or `virtualenv`) is the standard for managing Python packages and environments. It works well for simpler projects but has limitations with scientific libraries that require complex non-Python dependencies.

Pros:

- **Wide Compatibility**: Pip works directly with PyPI, making it compatible with a broad range of Python packages.
- **Standard and Lightweight**: Pip and `venv` are standard in Python, easy to set up, and don't add external dependencies to your workflow.
- **Free and Widely Supported**: Pip and `venv` are built into Python, with extensive community

support.

Cons:

- **Limited Dependency Resolution**: Pip lacks Conda/Mamba's dependency resolver, which can cause conflicts with complex dependencies.
- **No Non-Python Dependency Management**: Pip cannot natively install packages with complex non-Python dependencies, such as `GDAL`, without requiring additional system-level installations.

Suitability: **Not recommended** for projects with complex dependencies like `GDAL` and `PDAL`. Pip with `venv` may be insufficient unless you have a reliable way to handle system dependencies.

Summary and Recommendation

Tool	Non-Python Dependencies	Speed	Ecosystem Support	Suitability
Mamba	Excellent	Excellent	High	Highly Recommended
Vu	Good	Excellent	Moderate	Recommended for Testing
Poetry	Limited	Good	High	Limited to Python-only
Pip + venv	Poor	Moderate	High	Not Recommended

Recommendation: Given your requirements, **Mamba** remains the best choice for performance, compatibility, and non-Python dependency support, providing a seamless transition from Conda with faster speeds. **Vu** is a promising alternative that may suit projects where maximum performance is critical and dependency requirements align closely with Vu's supported ecosystem, but its immaturity might pose occasional compatibility issues.

4.1.2 Ruff vs. Black: A Comparison

When it comes to Python code quality and formatting, **Ruff** and **Black** serve distinct but sometimes overlapping purposes. Understanding their functionalities and how they complement each other can help you decide whether to use one, both, or a combination of both in your development workflow.

What is Black?

Black is an uncompromising Python code formatter. Its primary goal is to enforce a consistent coding style by automatically formatting your code according to its predefined rules. Black focuses solely on formatting, ensuring that your code adheres to a specific style without requiring manual adjustments.

Key Features of Black:

- **Consistent Formatting:** Automatically formats code to follow the Black style guide.
- **Minimal Configuration:** Black intentionally offers limited configuration options to reduce debates over style preferences.
- **Deterministic Output:** Given the same input, Black will always produce the same output, making it reliable for automated formatting.

Example Usage:

```
black your_script.py
```

What is Ruff?

Ruff is a fast Python linter written in Rust. It aims to provide a comprehensive set of linting rules to catch errors, enforce coding standards, and improve code quality. While Ruff includes some formatting capabilities, its primary focus is on linting rather than formatting.

Key Features of Ruff:

- **Extensive Linting Rules:** Covers a wide range of linting checks, including PEP8 compliance, complexity analysis, unused imports, and more.
- **Performance:** Written in Rust, Ruff is designed to be extremely fast, making it suitable for large codebases.
- **Configurable:** Offers extensive configuration options to enable or disable specific rules according to your project's needs.
- **Some Formatting Support:** Includes basic formatting fixes, but not as comprehensive as dedicated formatters like Black.

Example Usage:

```
ruff check your_script.py
```

Ruff vs. Black: Overlaps and Differences

While both tools aim to improve code quality, their primary functions differ:

Feature	Black	Ruff
Primary Purpose	Code formatting	Code linting and error checking
Formatting Capability	Comprehensive, style-driven formatting	Basic formatting fixes, not as extensive as Black
Linting Rules	Minimal (focused on formatting)	Extensive, covering various linting aspects
Performance	Fast, but not as fast as Ruff	Extremely fast, optimized for speed
Configuration	Minimal (few formatting options)	Highly configurable with many options
Integration	Easily integrates with IDEs and CI/CD pipelines for formatting	Integrates well for linting, can complement other tools

Should You Use Both Ruff and Black?

Yes, using both Ruff and Black can be highly beneficial, as they complement each other by covering different aspects of code quality:

1. **Black for Formatting:** Let Black handle the automatic formatting of your code. It ensures that your codebase maintains a consistent style without manual intervention.
2. **Ruff for Linting:** Use Ruff to perform comprehensive linting checks. It can catch potential errors, enforce coding standards, and identify code smells that Black does not address.

Benefits of Using Both:

- **Comprehensive Coverage:** You get the best of both worlds—consistent formatting from Black and thorough linting from Ruff.
- **Improved Code Quality:** Combining both tools helps maintain high code quality by addressing both stylistic and functional aspects.
- **Efficiency:** Ruff's speed ensures that linting checks are fast, while Black's deterministic formatting keeps your workflow smooth.

How to Integrate Ruff and Black Together

To seamlessly integrate Ruff and Black into your workflow, you can configure Ruff to defer formatting responsibilities to Black. This setup allows Ruff to focus solely on linting while Black handles all formatting tasks.

Steps to Integrate Ruff and Black:

1. **Install Both Tools:**

```
pip install black ruff
```

2. **Configure Ruff to Use Black for Formatting:** Create a `pyproject.toml` file in your project root (if you don't have one already) and add the following configuration:

```
[tool.ruff]
select = ["E", "F", "W", "C", "I", "B"]  # Customize based on your needs
extend-select = ["B"]  # Include Black-compatible rules
line-length = 88  # Black's default line length
formatter = "black"  # Delegate formatting to Black
```

3. **Set Up Pre-commit Hooks (Optional but Recommended):** Using pre-commit hooks ensures that code is automatically formatted and linted before commits.

   - **Install pre-commit:**

```
pip install pre-commit
```

   - **Create a `.pre-commit-config.yaml` File:**

```
repos:
  - repo: https://github.com/psf/black
    rev: 23.3.0  # Use the latest stable version
    hooks:
      - id: black
        language_version: python3

  - repo: https://github.com/charliermarsh/ruff-pre-commit
    rev: v0.0.241  # Use the latest stable version
    hooks:
      - id: ruff
        args: ["--fix"]
```

   - **Install the Pre-commit Hooks:**

```
pre-commit install
```

   With this setup, every time you make a commit, Black will format your code, and Ruff will perform linting checks automatically.

Conclusion

While **Ruff** and **Black** have some overlapping functionalities, they are designed to address different aspects of Python code quality:

- **Black** is specialized in formatting your code consistently and automatically.
- **Ruff** excels in providing fast and comprehensive linting, catching potential issues beyond just formatting.

Using both tools in tandem allows you to maintain a clean, consistent, and error-free codebase efficiently. By delegating formatting to Black and utilizing Ruff for thorough linting, you can streamline your development workflow and enhance overall code quality.

4.1.3 Ruff vs. Pylint: A Comparison

`Ruff` does not completely cover all of `Pylint`'s functionality. While `Ruff` is exceptionally fast and supports a wide range of linting rules, it lacks some of the deeper, analysis-based checks that `Pylint` performs. Here's a comparison of key areas where their functionality overlaps and differs:

1. Supported Rules and Coverage

- **Ruff**: Focuses on speed and provides broad coverage for stylistic, formatting, and common error checks (e.g., variable naming, unused imports, unused variables, type annotations, and simple logical checks).
- **Pylint**: Offers more extensive checks, including object-oriented checks (e.g., method signatures, class inheritance issues), control flow analysis, and specific code smells (e.g., too many arguments, cyclomatic complexity, and duplicate code). `Pylint` is also known for its configurable thresholds for complexity metrics and custom rules.

2. Error Types and Depth of Analysis

- **Ruff**: Targets primarily stylistic and performance issues, along with common errors that can be statically analyzed quickly. It doesn't perform complex flow analysis, which can identify issues like potential bugs from misused variable scopes.
- **Pylint**: Performs in-depth checks, including flow analysis, class and method structure validations, and advanced bug detection (e.g., unhandled exceptions and unreachable code).

3. Configuration and Extensibility

- **Ruff**: Relatively lightweight configuration, generally configured through a `pyproject.toml` file or similar. Its primary focus is on quick, standard linting with minimal setup.
- **Pylint**: Highly configurable, allowing users to enable or disable specific checks, set strictness levels, and define custom thresholds. `Pylint` also supports plugins, enabling custom rule sets to be added, which can be essential for enforcing project-specific standards.

4. Performance

- **Ruff**: Designed for speed, making it much faster than `Pylint`, especially on larger codebases.
- **Pylint**: Known for being slower, as it performs more comprehensive checks and deeper analysis.

5. Summary of Overlap and Differences

Feature	Ruff	Pylint
Style Checks	?	?
Error Detection	?	?
Complexity Checks	?	?
Flow Analysis	?	?
OOP-Specific Checks	?	?
Speed	?	?
Configurability	Basic	Extensive
Custom Plugins	Limited	Extensive

When to Use Ruff, Pylint, or Both

- **Ruff**: Best for fast feedback, stylistic checks, and general error detection in CI/CD pipelines where speed is essential.
- **Pylint**: More suitable for thorough code quality analysis, complex projects, or when deeper insight into code structure and design is required.
- **Both**: Many teams use `Ruff` for quick feedback and `Pylint` as a secondary step for in-depth analysis, especially in pre-commit hooks or CI pipelines.

In conclusion, while `Ruff` covers many basic checks similar to `Pylint`, it does not replace `Pylint`

entirely in terms of depth and customization.

4.1.4 Tool Coverage by Ruff

Understanding Ruff

Ruff is a fast and efficient Python linter written in Rust. It aims to provide comprehensive linting capabilities by integrating multiple linting functionalities into a single tool. Ruff can handle many tasks traditionally managed by separate tools, offering both speed and versatility.

Key Features of Ruff:

- **Linting:** Detects syntax errors, code smells, and stylistic issues.
- **Import Sorting:** Manages and sorts imports, effectively replacing tools like `isort`.
- **Fixes:** Automatically fixes certain linting issues.
- **Performance:** Extremely fast, making it suitable for large codebases.

Tools You Can Remove When Using Ruff

1. `isort`

- **Purpose of `isort`:** Specifically sorts and organizes import statements in Python files.
- **Ruff's Replacement Capability:** Ruff includes built-in support for import sorting, effectively handling the tasks performed by `isort`.
- **Action: Remove `isort`** from your Makefile.

```
## format:              Format the code with Black and docformatter.
format: black docformatter
```

(Removed `isort` from the `format` target)

2. `pylint` (Optional)

- **Purpose of `pylint`:** A comprehensive linter that checks for errors, enforces a coding standard, and looks for code smells.
- **Ruff's Replacement Capability:** Ruff covers a substantial portion of `pylint`'s functionality with improved performance. However, `pylint` offers some advanced checks and reporting that Ruff may not cover entirely.
- **Decision:**
  - **If Ruff Meets Your Needs: Remove `pylint`** for a simpler setup.
  - **If You Require Advanced Checks: Keep `pylint`** alongside Ruff.

```
## lint:                Lint the code with ruff, Bandit, vulture, Pylint and
 Mypy.
lint: ruff bandit vulture mypy
```

(Optionally remove `pylint` if not needed)

3. `docformatter`

- **Purpose of `docformatter`:** Formats docstrings to follow PEP 257 standards.
- **Ruff's Replacement Capability:** Ruff does **not** handle docstring formatting.
- **Action: Keep `docformatter`** in your Makefile.

4. `isort` Replacement Confirmed

- As mentioned, Ruff replaces `isort`. No additional action needed beyond removal.

Tools to Retain When Using Ruff

1. `bandit`

- **Purpose:** Identifies common security issues in Python code.
- **Ruff's Capability:** Ruff does **not** focus on security analysis.
- **Action: Keep `bandit`** for security linting.

2. `vulture`

- **Purpose:** Detects dead code in Python projects.
- **Ruff's Capability:** Ruff does **not** specialize in dead code detection.
- **Action: Keep `vulture`** for identifying unused code.

3. `black`

- **Purpose:** A code formatter that enforces a consistent coding style.
- **Ruff's Capability:** While Ruff can fix some formatting issues, it does **not** fully replace `black`.
- **Action: Keep `black`** for comprehensive code formatting.

4. `mypy`

- **Purpose:** Performs static type checking in Python.
- **Ruff's Capability:** Ruff does **not** perform type checking.
- **Action: Keep `mypy`** for type safety and static analysis.

5. `coveralls`

- **Purpose:** Uploads coverage data to Coveralls.io for test coverage reporting.
- **Ruff's Capability:** Ruff does **not** handle test coverage.
- **Action: Keep `coveralls`** for coverage reporting.

6. `sphinx`

- **Purpose:** Generates documentation from source code.
- **Ruff's Capability:** Ruff does **not** handle documentation generation.
- **Action: Keep `sphinx`** for creating and managing documentation.

7. `conda` Environments

- **Purpose:** Manages project dependencies and environments.
- **Ruff's Capability:** Ruff does **not** manage environments.
- **Action: Keep `conda`** for environment management.

8. `docker`, `compileall`, `next-version`, etc.

- **Purpose:** Various tasks like building Docker images, compiling Python scripts, and version management.
- **Ruff's Capability:** Ruff does **not** handle these tasks.
- **Action: Keep these tools** as needed for your project workflow.

Revised Makefile Example

Based on the above analysis, here's how you can adjust your Makefile to remove unnecessary tools when using Ruff:

```
.DEFAULT_GOAL := help

MODULE=iotemplateapp
PYTHONPATH=${MODULE} docs scripts tests

ARCH:=$(shell uname -m)
OS:=$(shell uname -s)

ifeq (${OS},Linux)
    DOCKER2EXE_DIR=linux-amd64
    DOCKER2EXE_SCRIPT=sh
    DOCKER2EXE_TARGET=linux/amd64
else ifeq (${OS},Darwin)
    DOCKER2EXE_SCRIPT=zsh
    ifeq ($(ARCH),arm64)
        DOCKER2EXE_DIR=darwin-arm64
        DOCKER2EXE_TARGET=darwin/arm64
    else ifeq ($(ARCH),x86_64)
        DOCKER2EXE_DIR=darwin-amd64
        DOCKER2EXE_TARGET=darwin/amd64
    endif
endif

export ENV_FOR_DYNACONF=test
export LANG=en_US.UTF-8

## ====================================================================================
## make Script       The purpose of this Makefile is to support the whole
##                   software development process for an application. It
##                   contains also the necessary tools for the CI activities.
##                   ------------------------------------------------------------
##                   The available make commands are:
## ------------------------------------------------------------------------------------
## help:               Show this help.
## ------------------------------------------------------------------------------------
## action:             Run the GitHub Actions locally.
action: action-std
## dev:                Format, lint and test the code.
dev: format lint tests
## docs:               Check the API documentation, create and upload the user
 documentation.
docs: sphinx
## everything:         Do everything precheckin
everything: dev docs
## final:              Format, lint and test the code and create
 the documentation.
final: format lint docs tests
## format:             Format the code with Black.
format: black docformatter
## lint:               Lint the code with ruff, Bandit, vulture, and Mypy.
lint: ruff bandit vulture mypy
## pre-push:           Preparatory work for the pushing process.
pre-push: format lint tests next-version docs
## tests:              Run all tests with pytest.
tests: pytest
## ------------------------------------------------------------------------------------

help:
```

```
    @sed -ne '/@sed/!s/## //p' ${MAKEFILE_LIST}

# Run the GitHub Actions locally.
# https://github.com/nektos/act
# Configuration files: .act_secrets & .act_vars
action-std:         ## Run the GitHub Actions locally: standard.
    @echo "Info **********  Start: action ************************************"
    @echo "Copy your .aws/credentials to .aws_secrets"
    @echo "--------------------------------------------------------------------"
    act --version
    @echo "--------------------------------------------------------------------"
    act --quiet \
        --secret-file .act_secrets \
        --var IO_LOCAL='true' \
        --verbose \
        -P ubuntu-latest=catthehacker/ubuntu:act-latest \
        -W .github/workflows/github_pages.yml
    act --quiet \
        --secret-file .act_secrets \
        --var IO_LOCAL='true' \
        --verbose \
        -P ubuntu-latest=catthehacker/ubuntu:act-latest \
        -W .github/workflows/standard.yml
    @echo "Info **********  End:   action ************************************"

# Bandit is a tool designed to find common security issues in Python code.
# https://github.com/PyCQA/bandit
# Configuration file: none
bandit:             ## Find common security issues with Bandit.
    @echo "Info **********  Start: Bandit ************************************"
    @echo "PYTHONPATH=${PYTHONPATH}"
    @echo "--------------------------------------------------------------------"
    bandit --version
    @echo "--------------------------------------------------------------------"
    bandit -c pyproject.toml -r ${PYTHONPATH}
    @echo "Info **********  End:   Bandit ************************************"

# The Uncompromising Code Formatter
# https://github.com/psf/black
# Configuration file: pyproject.toml
black:              ## Format the code with Black.
    @echo "Info **********  Start: black *************************************"
    @echo "PYTHONPATH=${PYTHONPATH}"
    @echo "--------------------------------------------------------------------"
    black --version
    @echo "--------------------------------------------------------------------"
    black ${PYTHONPATH}
    @echo "Info **********  End:   black *************************************"

# Byte-compile Python libraries
# https://docs.python.org/3/library/compileall.html
# Configuration file: none
compileall:         ## Byte-compile the Python libraries.
    @echo "Info **********  Start: Compile All Python Scripts ******************"
    python3 --version
    @echo "--------------------------------------------------------------------"
    python3 -m compileall
    @echo "Info **********  End:   Compile All Python Scripts ******************"

# Miniconda - Minimal installer for conda.
# https://docs.conda.io/en/latest/miniconda.html
# Configuration file: none
```

```
conda-dev:          ## Create a new environment for development.
    @echo "Info **********  Start: Miniconda create development environment *****"
    conda config --set always_yes true
    conda --version
    echo "PYPI_PAT=${PYPI_PAT}"
    @echo "-------------------------------------------------------------------------"
    conda env remove -n ${MODULE} 2>/dev/null || echo "Environment '${MODULE}' does
 not exist."
    conda env create -f config/environment_dev.yml
    @echo "-------------------------------------------------------------------------"
    conda info --envs
    conda list
    @echo "Info **********  End:   Miniconda create development environment *****"

conda-prod:         ## Create a new environment for production.
    @echo "Info **********  Start: Miniconda create production environment ******"
    conda config --set always_yes true
    conda --version
    @echo "-------------------------------------------------------------------------"
    conda env remove -n ${MODULE} 2>/dev/null || echo "Environment '${MODULE}' does
 not exist."
    conda env create -f config/environment.yml
    @echo "-------------------------------------------------------------------------"
    conda info --envs
    conda list
    @echo "Info **********  End:   Miniconda create production environment ******"

# Requires a public repository !!!
# Python interface to coveralls.io API
# https://github.com/TheKevJames/coveralls-python
# Configuration file: none
coveralls:          ## Run all the tests and upload the coverage data to
 coveralls.
    @echo "Info **********  Start: coveralls ***********************************"
    pytest --cov=${MODULE} --cov-report=xml --random-order tests
    @echo "-------------------------------------------------------------------------"
    coveralls --service=github
    @echo "Info **********  End:   coveralls ***********************************"

# Formats docstrings to follow PEP 257
# https://github.com/PyCQA/docformatter
# Configuration file: pyproject.toml
docformatter:       ## Format the docstrings with docformatter.
    @echo "Info **********  Start: docformatter ********************************"
    @echo "PYTHONPATH=${PYTHONPATH}"
    @echo "-------------------------------------------------------------------------"
    docformatter --version
    @echo "-------------------------------------------------------------------------"
    docformatter --in-place -r ${PYTHONPATH}
#   docformatter -r ${PYTHONPATH}
    @echo "Info **********  End:   docformatter ********************************"

# Creates Docker executables
# https://github.com/rzane/docker2exe
# Configuration files: .dockerignore & Dockerfile
docker:             ## Create a docker image.
    @echo "Info **********  Start: Docker **************************************"
    @echo "OS                =${OS}"
    @echo "ARCH              =${ARCH}"
    @echo "DOCKER2EXE_DIR    =${DOCKER2EXE_DIR}"
    @echo "DOCKER2EXE_SCRIPT=${DOCKER2EXE_SCRIPT}"
    @echo "DOCKER2EXE_TARGET=${DOCKER2EXE_TARGET}"
    @echo "-------------------------------------------------------------------------"
```

```
    docker ps -a
    @echo "-------------------------------------------------------------------------"
    @sh -c 'docker ps -a | grep -q "${MODULE}" && docker rm --force ${MODULE} ||
 echo "No existing container to remove."'
    @sh -c 'docker image ls | grep -q "${MODULE}" && docker rmi --force
 ${MODULE}:latest || echo "No existing image to remove."'
    docker system prune -a -f
    docker build --build-arg PYPI_PAT=${PYPI_PAT} -t ${MODULE} .
    @echo "-------------------------------------------------------------------------"
    rm -rf app-${DOCKER2EXE_DIR}
    mkdir app-${DOCKER2EXE_DIR}
    chmod +x dist/docker2exe-${DOCKER2EXE_DIR}
    ./dist/docker2exe-${DOCKER2EXE_DIR} --name ${MODULE} \
                                  --image ${MODULE}:latest \
                                  --embed \
                                  -t ${DOCKER2EXE_TARGET} \
                                  -v ./data:/app/data \
                                  -v ./logging_cfg.yaml:/app/logging_cfg.yaml \
                                  -v ./settings.io_aero.toml:/app
/settings.io_aero.toml
    mkdir app-${DOCKER2EXE_DIR}/data
    mv dist/${MODULE}-${DOCKER2EXE_DIR} app-${DOCKER2EXE_DIR}/${MODULE}
    chmod +x app-${DOCKER2EXE_DIR}/${MODULE}
    cp logging_cfg.yaml                          app-${DOCKER2EXE_DIR}/
    cp run_iotemplateapp.${DOCKER2EXE_SCRIPT}    app-${DOCKER2EXE_DIR}/
    chmod +x app-${DOCKER2EXE_DIR}/*.${DOCKER2EXE_SCRIPT}
    cp settings.io_aero.toml                     app-${DOCKER2EXE_DIR}/
    @echo "Info **********  End:   Docker ************************************"

# isort your imports, so you don't have to.
# https://github.com/PyCQA/isort
# Configuration file: pyproject.toml
# Removed `isort` as Ruff handles import sorting.

# Mypy: Static Typing for Python
# https://github.com/python/mypy
# Configuration file: pyproject.toml
mypy:               ## Find typing issues with Mypy.
    @echo "Info **********  Start: Mypy **************************************"
    @echo "PYTHONPATH=${PYTHONPATH}"
    @echo "-------------------------------------------------------------------------"
    mypy --version
    @echo "-------------------------------------------------------------------------"
    mypy ${PYTHONPATH}
    @echo "Info **********  End:   Mypy **************************************"

mypy-stubgen:       ## Autogenerate stub files.
    @echo "Info **********  Start: Mypy **************************************"
    @echo "MODULE=${MODULE}"
    @echo "-------------------------------------------------------------------------"
    rm -rf out
    stubgen --package ${MODULE}
    cp -f out/${MODULE}/* ./${MODULE}/
    rm -rf out
    @echo "Info **********  End:   Mypy **************************************"

next-version:       ## Increment the version number.
    @echo "Info **********  Start: next_version **********************************"
    @echo "PYTHONPATH=${PYTHONPATH}"
    @echo "-------------------------------------------------------------------------"
    python3 --version
    @echo "-------------------------------------------------------------------------"
```

```
    python3 scripts/next_version.py
    @echo "Info **********  End:   next version **********************************"

# Pylint is a tool that checks for errors in Python code.
# https://github.com/PyCQA/pylint/
# Configuration file: .pylintrc
# Removed `pylint` if not needed; otherwise keep it.

# pytest: helps you write better programs.
# https://github.com/pytest-dev/pytest/
# Configuration file: pyproject.toml
pytest:             ## Run all tests with pytest.
    @echo "Info **********  Start: pytest ***************************************"
    @echo "CONDA      =${CONDA_PREFIX}"
    @echo "PYTHONPATH=${PYTHONPATH}"
    @echo "--------------------------------------------------------------------------"
    pytest --version
    @echo "--------------------------------------------------------------------------"
    pytest --dead-fixtures tests
    pytest --cache-clear --cov=${MODULE} --cov-report term-missing:skip-covered
 --cov-report=lcov -v tests
    @echo "Info **********  End:   pytest ***************************************"

pytest-ci:          ## Run all tests with pytest after test tool installation.
    @echo "Info **********  Start: pytest ***************************************"
    @echo "CONDA      =${CONDA_PREFIX}"
    @echo "PYTHONPATH=${PYTHONPATH}"
    @echo "--------------------------------------------------------------------------"
    pip3 install pytest pytest-cov pytest-deadfixtures pytest-helpers-namespace
 pytest-random-order
    @echo "--------------------------------------------------------------------------"
    pytest --version
    @echo "--------------------------------------------------------------------------"
    pytest --dead-fixtures tests
    pytest --cache-clear --cov=${MODULE} --cov-report term-missing:skip-covered -v
 tests
    @echo "Info **********  End:   pytest ***************************************"

pytest-first-issue: ## Run all tests with pytest until the first issue occurs.
    @echo "Info **********  Start: pytest ***************************************"
    @echo "CONDA      =${CONDA_PREFIX}"
    @echo "PYTHONPATH=${PYTHONPATH}"
    @echo "--------------------------------------------------------------------------"
    pytest --version
    @echo "--------------------------------------------------------------------------"
    pytest --cache-clear --cov=${MODULE} --cov-report term-missing:skip-covered -rP
 -v -x tests
    @echo "Info **********  End:   pytest ***************************************"

pytest-ignore-mark: ## Run all tests without marker with pytest."
    @echo "Info **********  Start: pytest ***************************************"
    @echo "CONDA      =${CONDA_PREFIX}"
    @echo "PYTHONPATH=${PYTHONPATH}"
    @echo "--------------------------------------------------------------------------"
    pytest --version
    @echo "--------------------------------------------------------------------------"
    pytest --dead-fixtures -m "not no_ci" tests
    pytest --cache-clear --cov=${MODULE} --cov-report term-missing:skip-covered
 --cov-report=lcov -m "not no_ci" -v tests
    @echo Info **********  End:   pytest ***************************************

pytest-issue:       ## Run only the tests with pytest which are marked with
 'issue'.
```

```
    @echo Info **********  Start: pytest ************************************
    @echo CONDA      =${CONDA_PREFIX}
    @echo PYTHONPATH=${PYTHONPATH}
    @echo "----------------------------------------------------------------------"
    pytest --version
    @echo "----------------------------------------------------------------------"
    pytest --dead-fixtures tests
    pytest --cache-clear --capture=no --cov=${MODULE} --cov-report
 term-missing:skip-covered -m issue -rP -v -x tests
    @echo "Info **********  End:   pytest ************************************"

pytest-module:      ## Run test of a specific module with pytest.
    @echo "Info **********  Start: pytest ************************************"
    @echo "CONDA      =${CONDA_PREFIX}"
    @echo "PYTHONPATH=${PYTHONPATH}"
    @echo "TESTMODULE=tests/${TEST-MODULE}.py"
    @echo "----------------------------------------------------------------------"
    pytest --version
    @echo "----------------------------------------------------------------------"
    pytest --cache-clear --cov=${MODULE} --cov-report term-missing:skip-covered -v
 tests/${TEST-MODULE}.py
    @echo "Info **********  End:   pytest ************************************"

# https://github.com/astral-sh/ruff
# Configuration file: pyproject.toml
ruff:               ## An extremely fast Python linter and code formatter.
    @echo "Info **********  Start: ruff **************************************"
    ruff --version
    @echo "----------------------------------------------------------------------"
    ruff check --fix
    @echo "Info **********  End:   ruff **************************************"

sphinx:             ## Create the user documentation with Sphinx.
    @echo "Info **********  Start: sphinx ************************************"
    sphinx-apidoc --version
    sphinx-build --version
    @echo "----------------------------------------------------------------------"
    sudo rm -rf docs/build/*
    sphinx-apidoc -o docs/source ${MODULE}
    sphinx-build -M html docs/source docs/build
    sphinx-build -b rinoh docs/source docs/build/pdf
    @echo "Info **********  End:   sphinx ************************************"

version:            ## Show the installed software versions.
    @echo "Info **********  Start: version ***********************************"
    python3 --version
    pip3 --version
    @echo "Info **********  End:   version ***********************************"

# Find dead Python code
# https://github.com/jendrikseipp/vulture
# Configuration file: pyproject.toml
vulture:            ## Find dead Python code.
    @echo "Info **********  Start: vulture ***********************************"
    @echo "PYTHONPATH=${PYTHONPATH}"
    @echo "----------------------------------------------------------------------"
    vulture --version
    @echo "----------------------------------------------------------------------"
    vulture ${PYTHONPATH}
    @echo "Info **********  End:   vulture ***********************************"

## ============================================================================
```

Key Changes:

1. **Removed `isort`:**
   - **Before:**

```
## format:              Format the code with Black and docformatter.
format: isort black docformatter
```

   - **After:**

```
## format:              Format the code with Black.
format: black docformatter
```

   - **Reason:** Ruff handles import sorting, making `isort` redundant.
2. **Optional Removal of `pylint`:**
   - **Before:**

```
lint: ruff bandit vulture pylint mypy
```

   - **After (if removing `pylint`):**

```
lint: ruff bandit vulture mypy
```

   - **Reason:** Ruff covers many linting tasks previously handled by `pylint`. However, evaluate if Ruff's linting suffices for your project's needs before removing `pylint`.
3. **Adjusted `format` Target:**
   - **Removed `isort` from the `format` target as Ruff handles import sorting.**

Other Considerations:

- **`docformatter` is retained** because Ruff does not handle docstring formatting.
- **Security (`bandit`), dead code detection (`vulture`), type checking (`mypy`), and documentation (`sphinx`)** remain essential and should be kept.
- **`black` remains for comprehensive code formatting** that Ruff does not fully replace.

Final Recommendations

1. **Evaluate Ruff's Coverage:**
   - Before removing tools like `pylint`, ensure that Ruff's linting capabilities meet your project's requirements. Run Ruff extensively to verify that it catches all necessary issues.
2. **Maintain Essential Tools:**
   - **Security:** Keep `bandit` to ensure your code remains secure.
   - **Dead Code Detection:** Retain `vulture` to identify and remove unused code.
   - **Type Checking:** Continue using `mypy` for type safety.
   - **Documentation:** Keep `sphinx` and `docformatter` for maintaining and formatting documentation.
3. **Simplify Your Workflow:**
   - Removing redundant tools like `isort` can streamline your Makefile and reduce maintenance overhead.
   - Ensure that the remaining tools are well-integrated and complement each other to main-

tain code quality and consistency.

4. **Continuous Integration (CI):**
   - Update your CI pipelines to reflect the changes in the Makefile, ensuring that only the necessary tools are invoked during automated checks.

5. **Documentation and Team Alignment:**
   - Document the changes to the Makefile and inform your team to ensure everyone is aware of the new setup.
   - Provide guidelines on how to use Ruff alongside the retained tools for optimal code quality.

About

This section provides additional context and legal information about IO-TEMPLATE-APP, including release notes and licensing details.

5.1 Release Notes

5.1.1 Version 2.0.7

Release Date: 05.11.2024

Applied Software

Software	Version	Remark	Status
Docker	27.3.1		
Miniconda	24.9.2		
Python	3.12.7		

5.2 End-User License Agreement

5.2.1 End-User License Agreement (EULA) of IO-Aero Software

This End-User License Agreement ("**EULA**") is a legal agreement between you and **IO-Aero**.

This **EULA** agreement governs your acquisition and use of our **IO-Aero Software** ("Software") directly from **IO-Aero** or indirectly through a **IO-Aero** authorized reseller or distributor (a "Reseller").

Please read this **EULA** agreement carefully before completing the installation process and using the **IO-Aero Software**. It provides a license to use the **IO-Aero Software** and contains warranty information and liability disclaimers.

If you register for a free trial of the **IO-Aero Software**, this **EULA** agreement will also govern that trial. By clicking "accept" or installing and/or using the **IO-Aero Software**, you are confirming your acceptance of the Software and agreeing to become bound by the terms of this **EULA** agreement.

If you are entering into this **EULA** agreement on behalf of a company or other legal entity, you represent that you have the authority to bind such entity and its affiliates to these terms and conditions. If you do not have such authority or if you do not agree with the terms and conditions of this **EULA** agreement, do not install or use the Software, and you must not accept this **EULA** agreement.

This **EULA** agreement shall apply only to the Software supplied by **IO-Aero** herewith regardless of whether other software is referred to or described herein. The terms also apply to any **IO-Aero** updates, supplements, Internet-based services, and support services for the Software, unless other

terms accompany those items on delivery. If so, those terms apply.

License Grant

IO-Aero hereby grants you a personal, non-transferable, non-exclusive licence to use the **IO-Aero Software** on your devices in accordance with the terms of this **EULA** agreement.

You are permitted to load the **IO-Aero Software** (for example a PC, laptop, mobile or tablet) under your control. You are responsible for ensuring your device meets the minimum requirements of the **IO-Aero Software**.

You are not permitted to:

- Edit, alter, modify, adapt, translate or otherwise change the whole or any part of the Software nor permit the whole or any part of the Software to be combined with or become incorporated in any other software, nor decompile, disassemble or reverse engineer the Software or attempt to do any such things
- Reproduce, copy, distribute, resell or otherwise use the Software for any commercial purpose
- Allow any third party to use the Software on behalf of or for the benefit of any third party
- Use the Software in any way which breaches any applicable local, national or international law
- use the Software for any purpose that **IO-Aero** considers is a breach of this **EULA** agreement Intellectual Property and Ownership

IO-Aero shall at all times retain ownership of the Software as originally downloaded by you and all subsequent downloads of the Software by you. The Software (and the copyright, and other intellectual property rights of whatever nature in the Software, including any modifications made thereto) are and shall remain the property of **IO-Aero**.

IO-Aero reserves the right to grant licences to use the Software to third parties.

Termination

This **EULA** agreement is effective from the date you first use the Software and shall continue until terminated. You may terminate it at any time upon written notice to **IO-Aero**.

It will also terminate immediately if you fail to comply with any term of this **EULA** agreement. Upon such termination, the licenses granted by this **EULA** agreement will immediately terminate, and you agree to stop all access and use of the Software. The provisions that by their nature continue and survive will survive any termination of this **EULA** agreement.

Governing Law

This **EULA** agreement, and any dispute arising out of or in connection with this **EULA** agreement, shall be governed by and construed in accordance with the laws of the United States.

Indices and tables

- genindex
- modindex

6.1 Repository

Link to the repository for accessing the source code and contributing to the project:
IO-TEMPLATE-APP GitHub Repository

6.2 Version

This documentation is for IO-TEMPLATE-APP version 1.4.8.

i

A

C

F

G

I

L

M

P

T

V