Earth Observation & Weather Data Federation with AI Embeddings

D3.1: AI-COMPRESSOR CONTAINERIZATION CICD PIPELINE

Revision: v 3.0

D3.1: AI-compressor containerization CICD pipeline

Work package	WP3
Task	3.1
Due date	31/07/2024
Submission date	26/07/2024
Deliverable lead	IBM Research
Version	3
Authors	Romeo Kienzler (IBM Research)
Reviewers	Gabriele Cavallaro (FZJ), Jonas Hurst (WWU)
Abstract	This deliverable presents a CI/CD pipeline system developed using the GitLab service provided by Helmholtz and the Linux Foundation's CLAIMED framework. The pipeline is designed to automate the compilation of the AI-compressor into a reusable operator, ensuring its applicability across various containerized and non-containerized execution environments in globally distributed data centres using heterogeneous software stacks. This automation facilitates project partners and future third-party users in effortlessly deploying the AI-compressor in its correct version and capability on their respective datasets. The implementation aims to streamline the integration process and enhance the usability of the AI-compressor in diverse computational settings.
Keywords	AI-compressors, OpenEO, CICD pipeline

Document Revision History

Version	Date	Description of change	List of contributor(s)
1	28/06/2024	1st edit	Romeo Kienzler (IBM Research)
2	25/07/2024	Incorporating PO's feedback	Romeo Kienzler (IBM Research)
3	29/07/2024	Incorporating reviewers Feedback	Romeo Kienzler (IBM Research), Jonas Hurst (WWU), Gabriele Cavallaro (FZJ)
4	25/09/2024	Incorporating reviewers Feedback	Romeo Kienzler (IBM Research), Jonas Hurst (WWU), Gabriele Cavallaro (FZJ)

Funded by the European Union

EUSPA

Grant Agreement No: 101131841 | **Topic:** HORIZON-EUSPA-2022-SPACE-02-55
Call: HORIZON-EUSPA-2022-SPACE | **Type of action:** HORIZON-RIA

DISCLAIMER

Embed2Scale (Earth Observation & Weather Data Federation With AI Embeddings) project is funded by the EU's Horizon Europe program under Grant Agreement number 101131841. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them. This work has also received funding from the Swiss State Secretariat for Education, Research and Innovation (SERI) and the UK Research and Innovation (UKRI).

COPYRIGHT NOTICE

Project funded by the European Commission in the Horizon Europe Programme		
Nature of the deliverable:	R	
Dissemination Level		
PU	Public, fully open, e.g. web (Deliverables flagged as public will be automatically published in CORDIS project's page)	☐
SEN	Sensitive, limited under the conditions of the Grant Agreement	
Classified R-UE/ EU-R	*EU RESTRICTED under the Commission Decision No2015/ 444*	
Classified C-UE/ EU-C	*EU CONFIDENTIAL under the Commission Decision No2015/ 444*	
Classified S-UE/ EU-S	*EU SECRET under the Commission Decision No2015/ 444*	

** R: Document, report (excluding the periodic and final reports)*
DEM: Demonstrator, pilot, prototype, plan designs
DEC: Websites, patents filing, press & media actions, videos, etc.
DATA: Data sets, microdata, etc.
DMP: Data management plan
ETHICS: Deliverables related to ethics issues.
SECURITY: Deliverables related to security issues
OTHER: Software, technical diagram, algorithms, models, etc.

TABLE OF CONTENTS

EXECUTIVE SUMMARY

The project involves the creation of a continuous integration and continuous deployment (CICD) pipeline framework using GitLab pipelines. A key feature is the integration of the C3 compiler library from the Linux Foundation, which transforms any code into reusable and portable components.

Key Innovations

- Containerless Execution: To meet the project's requirement for containerless execution, the C3 compiler has been extended. This extension allows the creation of containerless components, enhancing the flexibility and portability of the system.
- Universal Compatibility: The components generated by the extended C3 compiler can now run seamlessly on any system architecture, including bare metal, virtual machines (VMs), Slurm, PBS, LSF, Docker, Kubernetes, and OpenShift environments. This universal compatibility is crucial for maintaining reproducibility, ensuring that every partner and eventual third-party consumer can run identical code, on different execution environments.

Benefits

- Universal Compatibility:
  Our solution ensures that code can be executed consistently across various platforms, fostering reproducibility and collaboration.
- Simplified Deployment:
  The CI/CD pipeline and containerless components streamline the deployment process.
- Open Standards Adherence:
  Leveraging OpenEO promotes interoperability and accessibility, as the AI-Compressor is used to create virtual data layers on top of raw data made available through OpenEO and STAC to remote consumers.

Funded by the European Union

ABBREVIATIONS

CLAIMED	Component Library for AI, Machine Learning, ETL and Data Science
C3	Claimed Component Compiler
LFAI	Linux Foundation Artificial Intelligence
CICD	Continuous Integration Continuous Delivery
MLX	Machine Learning Exchange, an ML asset catalog
VCS	Version Control System

INTRODUCTION AND MOTIVATION

In the fast-paced world of AI (foundation) model development, like for producing an "AI compressor," delivering high-quality models quickly and consistently is crucial. Continuous Integration and Continuous Delivery (CI/CD) is a transformative approach that addresses these challenges directly.

CI/CD automates the process of integrating code changes, running tests, and deploying models/making them available to partners. This enables us to release updates and new model features more rapidly and confidently, while maintaining compatibility through versioning. Here's why CI/CD is a perfect fit for E2S:

1. Faster, More Efficient Development:
   Automated pipelines optimize the build, test, and deployment processes, significantly accelerating the delivery of new models and updates. By removing manual steps, CI/CD reduces human error and allows developers and researchers to focus on innovation rather than repetitive tasks.
2. Separation of concerns:
   The AI researchers are liberated from understanding a large amount of software engineering details whereas the CI/CD engineer doesn’t need to understand large amounts of details of the AI model development process.
3. Continuous Improvement and Innovation:
   CI/CD fosters a culture of continuous improvement by promoting frequent updates and iterative development on the AI/Compressor. This approach supports rapid experimentation and innovation, including assessment of model performance, keeping our project at the forefront of AI technology.
4. Embracing open standards:
   Leveraging open-source tools and community contributions, such as those from GitLab, CLAIMED and MLX, ensures our CI/CD processes remain aligned with the latest best practices and advancements.

In summary, CI/CD is not just about automating tasks; inspired by traditional software engineering, it's about revolutionising how we develop, test, and deliver AI models. It ensures that we can meet the demands of modern AI model development by enhancing collaboration, efficiency, quality, scalability, and innovation. By adopting CI/CD, we position ourselves to deliver high-quality AI models more swiftly and reliably, ultimately driving the success of E2S.

ARCHITECTURAL DECISIONS

This section gives a short overview on the architectural decisions made for this deliverable.

• SELECTION OF CICD FRAMEWORK

There are numerous CI/CD tools available in the market, each offering distinct features and functionalities tailored to different needs. Among these, GitLab, generously provided to us at no cost by Helmholtz, stands out due to its extensive and versatile capabilities and OpenSource core. Its range of features and functionalities not only meets but exceeds our project's requirements, making it an ideal choice.

One of the key advantages of GitLab is its use of a straightforward YAML file to define CI/CD pipelines. This simplicity not only streamlines the setup process but also makes it easier to migrate these pipelines to other CI/CD systems if needed. Furthermore, GitLab functions as a powerful git code versioning tool, ensuring that code integration is seamless and efficient. Every time there is a change in the codebase within GitLab, it automatically triggers a new CI/CD pipeline run. This process results in the creation of a new version of the deliverable, ensuring that our deployments are always up-to-date with the latest changes.

Moreover, as an open-source platform, GitLab benefits from the contributions of a large and active community. This extensive network of developers and users continuously enhances the tool, adding new features and improving existing ones, which ensures that GitLab remains at the forefront of CI/CD technology. The ongoing innovation and improvements from the community provide us with a robust and reliable tool that evolves with the industry standards.

Considering these significant advantages, we have decided to adopt GitLab for our project. Its comprehensive capabilities, ease of use, seamless integration, and the continuous support from the open-source community make it the best choice to meet our CI/CD needs.

• SELECTION OF COMPONENT COMPILER

CLAIMED C3 is highly regarded as one of the most advanced component compilers available in the market today. Its sophisticated plugin system stands out, allowing it to target a diverse array of runtime engines. These include popular choices like SLURM, PBS, LSF, Kubernetes Jobs, Kubeflow, Argo, Tekton, Airflow, and the Common Workflow Language (CWL). This flexibility ensures that CLAIMED C3 can seamlessly integrate into various computational environments, making it an incredibly versatile tool for our needs.

One of the most notable features of CLAIMED C3 is its support for grid computing style parallelization right out of the box. This eliminates the need for a dedicated coordinator, as the only requirement is a shared filesystem, COS (Cloud Object Storage), or database. This built-in parallelization capability simplifies the setup process and reduces the overhead typically associated with coordinating multiple computational tasks.

Moreover, CLAIMED C3 automates the generation and embedding of essential code and libraries into the worker components. This includes the coordinator code, as well as parameter servers and logging code. By automatically incorporating these elements, CLAIMED C3 streamlines the overall workflow, enhancing efficiency and reducing the potential for errors. This level of automation allows

developers to focus more on the core aspects of their work rather than on the intricate details of task coordination and logging.

Considering these substantial benefits, we have decided to adopt CLAIMED C3 for our project. Its advanced features, versatile plugin system, open source license and built-in parallelization support make it an ideal choice to meet our computational needs. The automation of coordinator, parameter server, and logging codes further enhances its appeal, ensuring a more efficient and streamlined workflow.

CLAIMED was developed and donated to the Linux Foundation by IBM, which ensures that string support is available for this framework.

• EXTENSIONS MADE TO CLAIMED

■ Containerless operators and containerless bootstrapping (compiling)

During the course of the project, the CLAIMED framework's compiler has been extended to not only generate container images but also create containerless operators by packaging Python environments (venv) and code into a single, portable, and versioned archive. This allows for greater flexibility, as it enables deploying operators without the need for container orchestration. The framework achieves this by zipping both the Python virtual environment and the source code into a compact file, making it easier to manage and distribute components across various runtime environments on different (HPC) data centers around the globe, simplifying portability and reproducibility for machine learning and data science workflows.

The CLAIMED framework now supports both containerized and containerless deployments, offering flexibility for varied infrastructure. A key innovation is the creation of a containerless bootstrapper, which enables building and deploying operators without relying on container technology. This bootstrapper allows these containerless operators to be created directly in non-containerized environments, such as Helmholtz Gitlab, enhancing usability and broadening its scope across different computational infrastructures. This feature is crucial for environments with strict containerization constraints or legacy HPC computing environments, promoting seamless integration and deployment, allowing the same code base to be replicated among all partners.

■ Extending the CLAIMED CLI to support containerless operators

In this project, the CLAIMED CLI has been extended to support execution of the containerless operators. It is crucial to notice that the syntax has not changed, therefore, the call to a CLAIMED operator, here, especially calls to the AI-Compressor or any other auxiliary pre/post processing operators syntactically stay the same by simply changing the source from a container registry (e.g., docker.io) to the keyword containerless (see below). Also the operator download/deployment semantics stay the same. When using containerized operators, the container image got pulled (and cached) from the container registry, taking updates into account (issuing another pull). The same holds for the containerless operators where updated versions on the containerless operator registry cause the CLAIMED CLI to seamlessly download the newest version of the operator (unless an exact version of the operator is pinned by specifying the :version_tag instead of :latest after the operator's name).

• SELECTION OF COMPONENT REGISTRY

To keep our storage solution simple, cost-effective, and scalable, we have decided to use Cloud Object Storage (COS) for storing our components, e.g. different versions and modalities of the AI-Compressor as well as all workflow steps necessary for pre- and postprocessing. This choice provides us with a flexible and reliable storage system that can easily scale with our needs.

In addition to storing the components, we will generate metadata as part of D3.4, ensuring that our storage is properly organized and managed. This metadata will play a crucial role in maintaining order and facilitating easy access to the stored components.

For versioning, we will adopt a system inspired by the versioning used in Docker container images. By managing versions through file names, we can easily track and retrieve different versions of our components. This method not only simplifies the versioning process but also ensures that our storage system remains efficient and scalable, allowing us to manage our components effectively without unnecessary complexity. Please note that versioning via a VCS like Git is not feasible because a single version of the compiled AI-Compressor exceeds 100s of megabytes. Nevertheless, any compiled AI-Compressor component can always be rebuild by the CI/CD pipeline on request, as the source artifacts are versioned through Git.

At a later stage, in D3.3, we plan to adopt MLX, the Linux Foundation AI Machine Learning Exchange. MLX is a state-of-the-art catalogue designed to manage and share machine learning models, datasets, data operators, and more. As a leading platform in the field, MLX provides an organized and comprehensive repository that supports the entire machine learning lifecycle.

By integrating MLX into our workflow, we will benefit from its advanced capabilities for cataloguing and retrieving machine learning assets. This adoption will enhance our ability to manage complex data and model dependencies, facilitate collaboration, and ensure that our machine learning resources are easily accessible and reusable. MLX's robust features will help us maintain a cutting-edge approach to machine learning, keeping us aligned with industry standards and best practices.

■ Conclusion

Deploying a containerless operator using CLAIMED is simpler because it only requires unzipping the pre-packaged archive, which includes both the Python environment and the code. Afterward, it can be directly executed without additional setup. This approach streamlines deployment, as users only need to install CLAIMED via pip on a Python-enabled compute node. The CLAIMED CLI takes care of these steps such that using a containerless claimed operator boils down to the following single steps (on a python enabled compute node):

claimed –component containerless/e2s/ai-compressor:latest –param1 value1 …

Again, if the operator is not available on the compute node, it gets automatically downloaded and cached. Each subsequent call will check on the operator registry if there is a new version available (and downloads new versions automatically).

Please note that the claimed framework needs to be installed only once per compute node using the following command:

pip install claimed

In contrast, deploying a typical Python application requires numerous steps, such as setting up virtual environments, manually installing dependencies, managing conflicting versions, and

configuring system paths. These tasks add complexity and can introduce errors, particularly when dependencies change over time. An example of state-of-the-art deployment steps (on a python enabled compute node) are:

1. Create a virtual environment:
   python -m venv .venv
2. Activate the environment
   source ./venv/bin/activate
3. Install the dependencies
   pip install -r requirements.txt
4. Execute the applications (invocation might differ due to lack of standards)
   python ai-compressor.py –param1 value1 …

Step three is a very error prone step as it heavily depends on the build environment. As CLAIMED operators are pre-build - for E2S in the gitlab environment - this potential source of error is removed. Figure 1 shows an example of one of the various errors potentially occurring during step three, in this case the error was caused by missing a C++ compiler in the correct version on the build system.

```
2024-09-26 10:56:07.682 [info]   error: subprocess-exited-with-error

× Building wheel for compressai (pyproject.toml) did not run successfully.
│ exit code: 1
╰─> [144 lines of output]
    fatal: not a git repository (or any parent up to mount point /)
    Stopping at filesystem boundary (GIT_DISCOVERY_ACROSS_FILESYSTEM not set).
    running bdist_wheel
    running build
    running build_py
    creating build/lib.linux-x86_64-cpython-310/compressai
    copying compressai/__init__.py -> build/lib.linux-x86_64-cpython-310/compressai
    copying compressai/version.py -> build/lib.linux-x86_64-cpython-310/compressai
    creating build/lib.linux-x86_64-cpython-310/compressai/datasets
    copying compressai/datasets/__init__.py -> build/lib.linux-x86_64-cpython-310/compressai/datasets
    copying compressai/datasets/image.py -> build/lib.linux-x86_64-cpython-310/compressai/datasets
    copying compressai/datasets/pregenerated.py -> build/lib.linux-x86_64-cpython-310/compressai/datasets
    copying compressai/datasets/rawvideo.py -> build/lib.linux-x86_64-cpython-310/compressai/datasets
    copying compressai/datasets/video.py -> build/lib.linux-x86_64-cpython-310/compressai/datasets
    copying compressai/datasets/vimeo90k.py -> build/lib.linux-x86_64-cpython-310/compressai/datasets
    creating build/lib.linux-x86_64-cpython-310/compressai/entropy_models
    copying compressai/entropy_models/__init__.py -> build/lib.linux-x86_64-cpython-310/compressai/entropy_models
    copying compressai/entropy_models/entropy_models.py -> build/lib.linux-x86_64-cpython-310/compressai/entropy_models
    creating build/lib.linux-x86_64-cpython-310/compressai/layers
```

Figure 1: An example of an error occurring during installation of requirements of the AI-compressor during the very error prone requirements installation step.

CLAIMED's containerized deployment using Docker or Podman is equally simple. Operators are lazily downloaded and cached from the operator/model registry (e.g., a container registry), allowing for smooth operation with just a local Docker installation, further reducing the burden of manual configuration. This approach centralizes dependency management, making it faster and more reliable, especially since the invocation of a CLAIMED containerized operator is similar to a non-containerized one:

claimed –component docker.io/e2s/ai-compressor:latest –param1 value1 …

THE CICD PIPELINE EXPLAINED

There is a concept of a portable application package, referred to here as an operator or component, which is designed to run across different data centres in heterogeneous execution environments. This aligns with the principle of "write once, run everywhere," meaning that a single code file—along with any number of dependent packages and source files—can be transformed into such an operator.

By adhering to this principle, we can ensure that our applications are highly portable and versatile. This approach allows a single codebase to be deployed and executed consistently across various platforms and environments without the need for modification. The ability to create an operator from a code file, regardless of its dependencies, simplifies the deployment process and enhances the scalability and flexibility of our applications. This methodology ensures that our solutions remain robust and adaptable, capable of leveraging diverse computational resources efficiently.

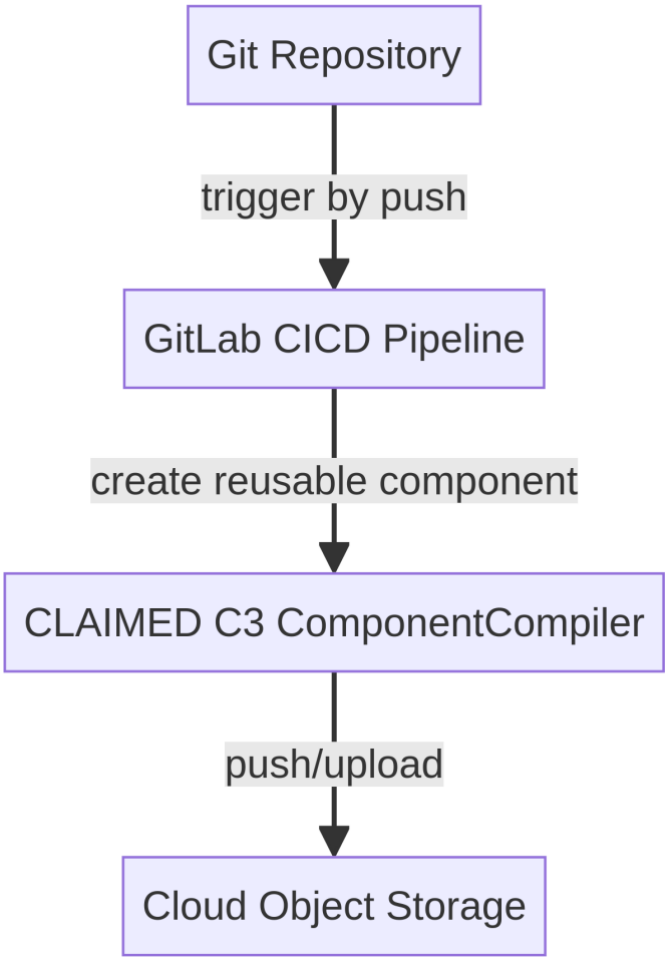

Figure 1: The CICD pipeline creating a reusable, portable component from any source file pushed to the repository

The CI/CD pipeline is designed to create a reusable, portable component from any source file pushed to the repository. When a source file is committed, the pipeline automatically initiates a series of steps to transform the file into a fully portable operator or component.

This process, as exemplified in Figure 1, involves several key stages:

1. **Source File Integration**: The pipeline detects the new or updated source file in the repository and begins the integration process.
2. **Dependency Management:** Any dependent packages or additional source files required by the main code are identified and included, ensuring the component has all necessary resources.
3. **Compilation and Packaging:** The source file, along with its dependencies, is compiled and packaged into a standalone operator. This package is designed to be portable, adhering to the "write once, run everywhere" principle. In this step, any execution environment specific metadata files (e.g. kubernetes job yaml, CWL task definition, …) are created. For containerized execution environments, a container image is build, for non-containerized execution environments, all required files, libraries and dependencies (e.g., python virtual environment folders or conda environments) are packaged into a single zip file.
4. **Testing and Validation:** The component undergoes rigorous testing to ensure it functions correctly in various environments. This includes unit tests, integration tests, and performance benchmarks.
5. **Metadata Generation:** Relevant metadata is generated and associated with the component, facilitating proper organization, management, and retrieval.
6. **Versioning:** The component is versioned using a system inspired by Docker container images, which involves appending version information to the file names. This ensures easy tracking and retrieval of different component versions.
7. **Storage and Deployment:** The final portable component is stored in our Cloud Object Storage (COS) solution, making it readily available for deployment across different data centers and heterogeneous execution environments.

By automating these steps, the CI/CD pipeline ensures that any main source file pushed to the repository can be efficiently transformed into a reusable, portable component. This approach streamlines the development process, enhances the scalability and flexibility of our applications, and supports consistent performance across various platforms.

EXAMPLE OF A RUNNING PIPELINE

As we do not have AI compressors yet, and cannot implement the entire pipeline at this point, to facilitate usage and adoption of our CI/CD system, we have created a reference pipeline that serves as a boilerplate for developing portable components. This boilerplate pipeline provides a standardized template, making it easier for developers to quickly set up and implement their own CI/CD workflows. Therefore in the following section is an example for the reference pipeline, illustrating how developers can customize it for their own projects.

THE PIPELINE DEFINITION

To define a CI/CD pipeline in GitLab, as shown in Figure 2, a .gitlab-ci.yml file needs to be placed in the root folder of the GitLab project. When changes are pushed to the repository, GitLab automatically detects this file and initiates the pipeline processes defined within it. This YAML file specifies the stages, jobs, and scripts to be executed, enabling continuous integration and delivery.

Each time code changes are committed; a new pipeline run is triggered. This pipeline compiles the component using the specified tools, such as CLAIMED C3, and ultimately deploys the compiled component to a designated storage solution like Cloud Object Storage (COS).

This reference pipeline acts as a boilerplate, providing a robust and standardized starting point for creating reusable, portable components. By adhering to this template, developers can streamline their CI/CD processes, ensuring consistency, efficiency, and ease of adoption across various projects. The reference pipeline's comprehensive approach covers all necessary stages, from initialization to deployment, making it a valuable tool for facilitating the development and deployment of portable application packages.

Benefits of the Reference Pipeline:

1. **Consistency**: Provides a standardized approach to building, testing, packaging, and deploying components.
2. **Efficiency**: Automates the CI/CD process, reducing manual intervention and potential errors.
3. **Ease of Adoption**: Simplifies the setup for new projects, enabling developers to quickly implement a reliable CI/CD pipeline.
4. **Portability**: Ensures components are portable and can be deployed across different data centers and heterogeneous execution environments.

Funded by the European Union

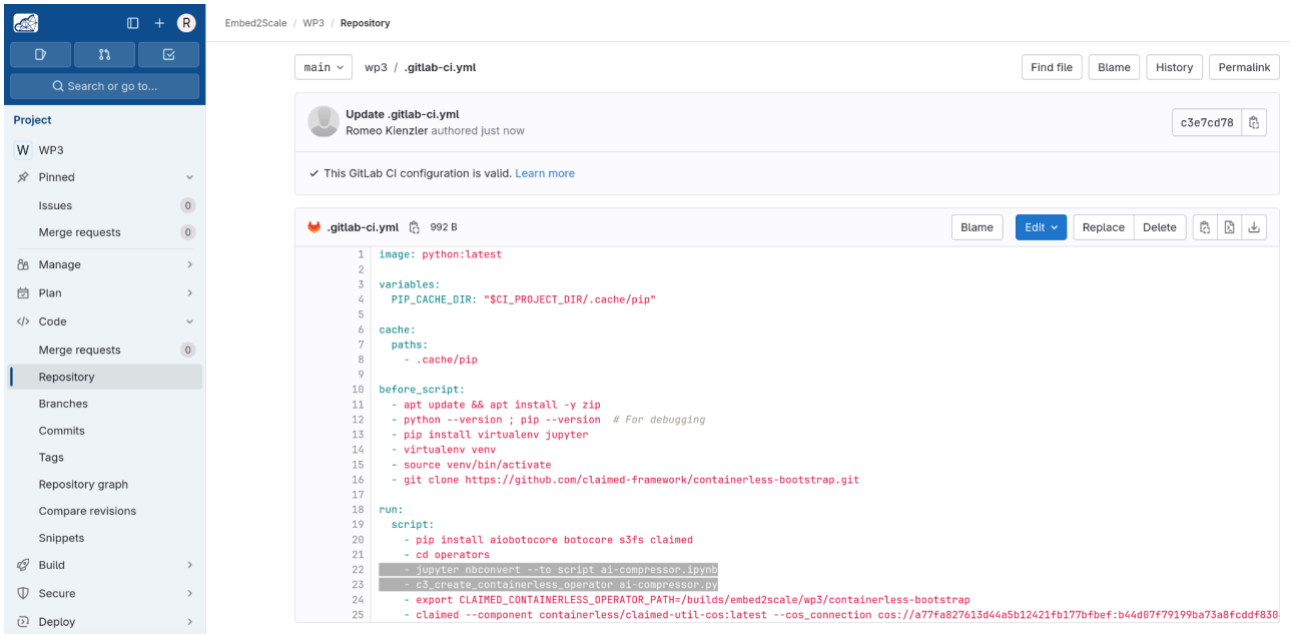

Figure 2: A simple example of how a push to the main branch will automatically build the AI-compressor and save it to the operator registry is configured using a gitlab specific CI/CD pipeline yaml file

HOW GITLAB VISUALISES PIPELINES

GitLab provides a detailed and intuitive visualization of CI/CD pipelines through its built-in pipeline view. This feature is designed to give users a comprehensive and clear graphical representation of the pipeline's progression and current status. When a pipeline is triggered, users can view a detailed diagram that outlines the various stages and jobs as defined in the .gitlab-ci.yml file.

In this view, each stage is depicted as a separate column, creating an organized and easy-to-follow layout. Within these columns, individual jobs are represented as cards. These job cards are color-coded to reflect their current status: blue for running jobs, green for those that have successfully completed, red for failed jobs, and yellow for jobs that are either manual or pending actions. This color-coding provides an at-a-glance understanding of the pipeline's state.

Moreover, users can interact with these job cards by clicking on them, which reveals detailed logs and job artifacts. This capability is crucial for developers and operations teams as it allows for precise monitoring of the pipeline's progress and the identification of any issues that may arise. The detailed logs provide insight into what has occurred during each job, making it easier to debug and resolve problems efficiently.

The comprehensive visualization offered by GitLab enhances the transparency and manageability of the CI/CD pipeline. By providing a clear view of each stage and job, it aids in workflow management, allowing teams to identify bottlenecks and streamline their processes. Additionally, the ability to access detailed logs and artifacts directly from the pipeline view simplifies troubleshooting and ensures that issues can be addressed promptly.

This visualization tool is not just about monitoring but also about improving the efficiency and reliability of the CI/CD process. By making the entire pipeline transparent and easily understandable, GitLab enables teams to maintain a smooth and consistent integration and delivery cycle. This feature is particularly valuable in complex environments where multiple stages and jobs are involved, providing a high level of control and oversight that is essential for maintaining quality and performance in continuous integration and delivery practices.

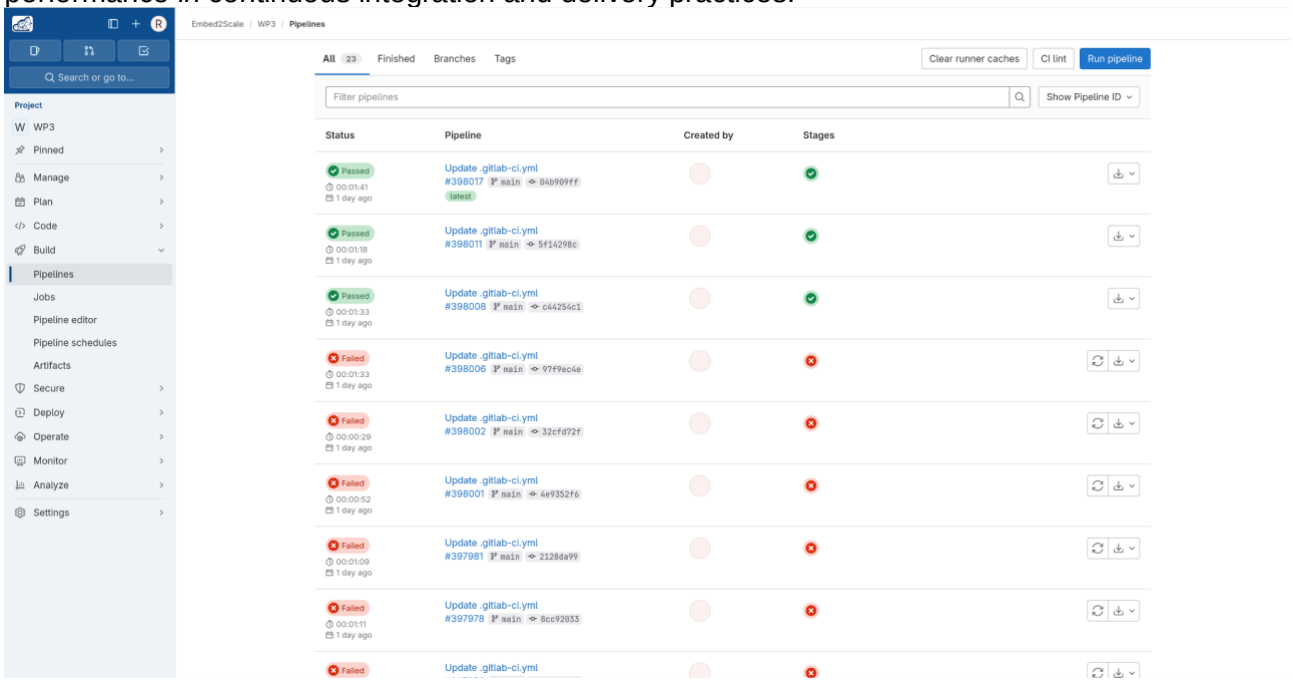

Figure 4: A pipeline is run after an operator script has been pushed to the main branch.

As illustrated in Figure 4, the pipeline definition has been modified, which in turn has automatically triggered a pipeline execution. This automated triggering mechanism is integral to maintaining continuous integration and continuous deployment (CI/CD) practices. The most recent three pipeline executions have all completed successfully, culminating in the creation of a new version of the operator within the repository. Each successful run signifies the robustness of the pipeline and its ability to handle iterative updates without failures.

This newly generated operator version is pivotal, as it will serve as the foundational component for the automated construction of various versions of the AI-Compressor. Leveraging CI/CD, the process ensures that every change to the pipeline definition or associated codebase is systematically built, tested, and deployed. The continuous integration aspect involves automatically integrating code changes from multiple contributors into a shared repository several times a day, followed by automated builds and tests. This process is critical for detecting and addressing integration issues early (e.g., drop in downstream task performance of models using the compressed features the AI-Compressor created), thereby enhancing code quality and reliability.

Furthermore, the continuous deployment component ensures that each successful pipeline run results in the automatic deployment of new operator versions, which are subsequently used for building different iterations of the AI-Compressor. This approach not only streamlines the development workflow but also significantly reduces manual intervention, thereby minimizing the risk of human error and accelerating the delivery cycle.

THE REPOSITORY

To achieve simplicity, cost-efficiency, scalability, and reliability in our storage strategy, the actual artifacts are stored in Cloud Object Storage, while their metadata is managed separately as detailed in D3.4. This bifurcation ensures a streamlined approach to data management, optimizing both performance and operational efficiency.

Storage Strategy and Version Control

The use of Cloud Object Storage allows for efficient handling of large data volumes, a critical requirement for scalable operations. Each artifact's version is meticulously embedded within the object's name, adhering to a consistent naming convention. This approach not only facilitates straightforward version tracking but also simplifies the retrieval process, allowing for precise identification and access to specific artifact versions.

Artifact Composition

The artifacts stored within Cloud Object Storage comprise the AI-Compressor codebase along with all requisite dependencies. This comprehensive inclusion is a strategic measure designed to ensure that each version of the AI-Compressor is entirely self-sufficient. By bundling the necessary libraries and modules, we mitigate potential compatibility issues that could arise from external dependency changes. This encapsulation ensures that every build and deployment process is executed with consistent and reliable components.

Technical Advantages of Cloud Object Storage:

1. Scalability
   Cloud Object Storage offers virtually unlimited storage capacity, making it an ideal solution for projects anticipating significant growth in data volume. This scalability is paramount as it allows for seamless expansion without the need for significant infrastructure overhauls, ensuring that storage limitations never hinder the project's progression. Therefore, the rolling-out process of new AI-Compressor versions will not introduce bottlenecks.
2. Durability and Availability
   Designed for high durability and availability, Cloud Object Storage guarantees that artifacts remain accessible and intact over extended periods. This reliability is crucial for maintaining the integrity of the CI/CD pipeline, ensuring that development and deployment processes can proceed uninterrupted. In addition, as COS resides on public internet, network access is seamless.
3. Cost Efficiency
   As storing AI-Compressor versions together with all their dependencies, storage requirements are above average. By leveraging a pay-as-you-go model, Cloud Object Storage provides a flexible and cost-effective alternative to traditional on-premises storage solutions.
4. Integration and Automation
   The integration capabilities of Cloud Object Storage with other cloud services enhance the overall automation and orchestration of our CI/CD pipeline. Automated workflows can interact seamlessly with stored artifacts, facilitating tasks such as retrieving specific versions, executing tests, and deploying updates without requiring manual intervention. This level of integration streamlines operations, reduces the potential for human error, and increases overall process efficiency.
5. Security and Compliance
   In addition to its operational benefits, Cloud Object Storage provides robust security features.

These features ensure that our artifacts are protected against unauthorized access. As COS is delivered as a service, operational maintenance costs are negligible.

In conclusion, the strategic decision to store actual artifacts in Cloud Object Storage, as exemplified in Figure 5, while managing their metadata separately, achieves a balance of simplicity, cost-efficiency, scalability, and reliability. By embedding version information within object names and including all necessary dependencies, we ensure consistent and reliable builds and deployments. The technical advantages provided by Cloud Object Storage, such as scalability, durability, global accessibility, and cost efficiency, significantly enhance the effectiveness and robustness of our CI/CD pipeline. This approach not only supports the project's current requirements but also positions it for sustainable long-term growth and success.

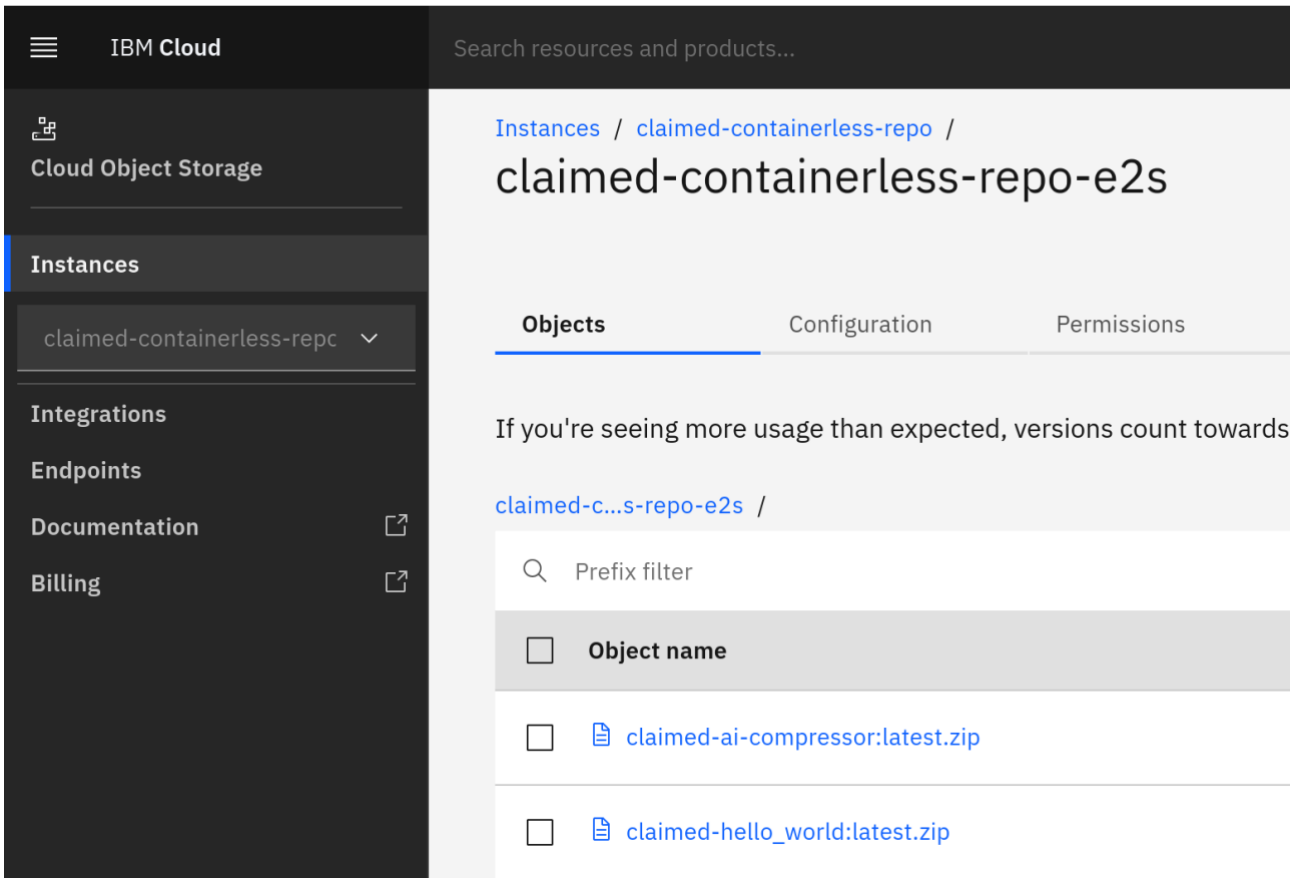

Figure 5: Web interface showing the generated operators in the IBM Cloud Object Store

HOW THE AI-COMPRESSOR CAN BE USED

Given that our CI/CD pipeline utilizes the CLAIMED framework to create reusable operators, we also benefit from CLAIMED's plug-in-based runtime support, which enhances the flexibility and scalability of our deployment processes.

The CLAIMED framework is pivotal in our CI/CD pipeline, providing a robust platform for developing and managing reusable operators. This framework's plug-in-based runtime support is particularly advantageous, enabling seamless integration and execution of operators in various environments. The modularity offered by CLAIMED allows for the extension and customization of runtime capabilities, ensuring that our operators can adapt to diverse operational requirements and constraints.

USABILITY ILLUSTRATION

To demonstrate the practicality and versatility of the CLAIMED framework, we utilize the CLAIMED Command Line Interface (CLI) to run an operator created by C3. This operator will be executed in both containerless and containerized modes, showcasing the adaptability of the framework.

Running in Containerless Mode

Running the operator in containerless mode involves executing the operator directly on the host system without the encapsulation of a container. This mode is particularly useful for environments where container overhead is undesirable or where the operator needs to interact closely with the host system resources. Using the CLAIMED CLI, the following command shown in Figure 6 can be used to run the AI-compressor in containerless mode on any laptop, HPC Slurm, PBS, LSF system:

```
claimed \
        --component containerless/claimed-ai-compressor:latest \
        --nnodes 1 \
        --nproc_per_node 1 \
        --dataset millionAID \
        --model mae_vit_compress_adapter \
        --epochs 20 \
        --warmup_epochs 1 \
        --data_path = /dcstore/e2s/datasets/millionAID/train \
        --save_every_n_epochs 2 \
        --num_workers 8 \
        --ld 1e10 \
        --path_to_pretrained_model /dcstore/e2s/models/ \
        --output_dir /dcstore/e2s/tmp/ \
        --log_dir /dcstore/e2s/tmp/ \
        --blr 1.5e-4 \
        --weight_decay 0.05 \
        --input_size 224 \
        --batch_size 256
```

Figure 6: CLI Example how the AI-Compressor can be used from a CLI without containerization

```
claimed \
        --component docker.io/e2s/claimed-ai-compressor:latest \
        --nnodes 1 \
        --nproc_per_node 1 \
        --dataset millionAID \
        --model mae_vit_compress_adapter \
        --epochs 20 \
        --warmup_epochs 1 \
        --data_path = /dcstore/e2s/datasets/millionAID/train \
        --save_every_n_epochs 2 \
        --num_workers 8 \
        --ld 1e10 \
        --path_to_pretrained_model /dcstore/e2s/models/ \
        --output_dir /dcstore/e2s/tmp/ \
        --log_dir /dcstore/e2s/tmp/ \
        --blr 1.5e-4 \
        --weight_decay 0.05 \
        --input_size 224 \
        --batch_size 256
```

Figure 7: CLI Example how the AI-Compressor can be used from a CLI using docker or podman

Please note that switching to containerized mode only needs to change the component prefix, nothing else, as shown in Figure 7. In this mode, the CLAIMED CLI manages the creation and execution of a container for the C3 operator. This encapsulation ensures that the operator runs in a controlled and predictable environment, which is particularly beneficial for distributed and cloud-based deployments. Containers also provide enhanced security by isolating the operator from the host system and other applications.

■ Benefits and Implications

The ability to run operators in both containerless and containerized modes offers several benefits:

1. Flexibility
   Operators can be deployed in various environments, adapting to specific operational needs and constraints. Through the addition of containerless compiling and execution of operators this flexibility has been increased significantly as most of the execution environments of current and potential future project partners are running in containerless environments.
2. Performance
   Containerless mode leverages the full capabilities of the host system, potentially improving performance for resource-intensive operations. The added containerless mode for operators allows the AI-Compressor to run on the compute nodes without additional overhead of containers or virtual machines but still leveraging the benefits of those virtualization technologies.

3. Portability
   The new CLAIMED compiler and CLI developed in this project ensures that operators can run consistently across different platforms, enhancing portability and reducing compatibility issues across various platforms and workload schedulers (e.g, SLURM, PBS, LSF, Kubernetes, ..)
4. Isolation
   Containerized execution provides a secure and isolated environment, minimizing the risk of conflicts and enhancing security. This means, where resource isolation is required, CLAIMED operators now can run in either a containerized environment (docker, podman, kubernetes) or inside a virtual machine in containerless mode to achieve security and isolation. Using either virtual machines, containers or resource managers like SLURM allows administrators to constrain resource requirements of the AI-compressor, while creating the embeddings.

In conclusion, the CLAIMED framework's plug-in-based runtime support significantly enhances our CI/CD pipeline's flexibility and adaptability. By demonstrating the execution of a C3 operator in both containerless and containerized modes using the CLAIMED CLI, we illustrate the practical benefits and versatility of this framework. This capability not only streamlines our deployment processes but also ensures that our operators can efficiently operate in a wide range of environments, thereby supporting the scalability and reliability of our overall system.

Figures 6 and 7 illustrate the versatility of the AI-Compressor, demonstrating how it can be effortlessly deployed in both containerized and non-containerized environments using the CLAIMED CLI.

CONCLUSION

The development of the AI-compressor containerization CI/CD pipeline, led by IBM Research, represents a significant advancement in streamlining the deployment and execution of AI models across diverse computing environments. By leveraging GitLab and the CLAIMED framework, the pipeline automates the integration, testing, and deployment processes, ensuring high reproducibility, ease of use, and universal compatibility. The innovative use of the C3 compiler and Cloud Object Storage enhances the flexibility, scalability, and reliability of the system, allowing seamless operation on various platforms, including bare metal, VMs, Slurm, PBS, LSF, Docker, Kubernetes, and OpenShift. This sophisticated approach not only accelerates development cycles but also maintains consistent performance, ultimately driving the success of the Embed2Scale project by enabling rapid and reliable AI model deployment.

• APPENDIX

Embed2Scale AI-compressor operator

For more information please visit https://www.embed2scale.eu/

```
#!pip install torch==2.4.1 torchvision==0.19.1 mmcv-full==1.5.0 compressai==1.2.4 timm tensorboard einops git+https://github.com/IBM/neural-embedding-compression.git
```

Python

```
import os
```

Python

```
nnodes = int(os.getenv('nnodes','1'))
nproc_per_node = int(os.getenv('nproc_per_node','1'))
dataset = os.getenv('dataset','millionAID')
model = os.getenv('model','mae_vit_compress_adapter')
epochs = int(os.getenv('epochs','1'))
warmup_epochs = int(os.getenv('warmup_epochs','1'))
data_path = os.getenv('data_path')
save_every_n_epochs = int(os.getenv('save_every_n_epochs','2'))
num_workers = int(os.getenv('num_workers','1'))
ld = os.getenv('ld','1e10')
path_to_pretrained_model = os.getenv('path_to_pretrained_model')
output_dir = os.getenv('output_dir')
log_dir = os.getenv('log_dir','./log')
blr = os.getenv('blr','1.5e-4')
weight_decay = float(os.getenv('weight_decay','0.05'))
input_size = int(os.getenv('input_size','224'))
batch_size = int(os.getenv('batch_size','256'))
```

Python

```
command = f"""
LOCAL_RANK="" torchrun --standalone --nnodes={nnodes} --nproc_per_node={nproc_per_node} ../../MAEPretrain_SceneClassification/main_compress.py \\
--dataset {dataset} --model {model} --epochs {epochs} --warmup_epochs {warmup_epochs} --data_path {data_path} \\
--save_every_n_epochs {save_every_n_epochs} --num_workers {num_workers} --ld {ld} --finetune {path_to_pretrained_model} --output_dir {output_dir} \\
--log_dir {log_dir} --blr {blr} --weight_decay {weight_decay} --input_size {input_size} --batch_size {batch_size}
"""
```

Python

```
os.system(command)
```

Python

Figure 8: The notebook wrapping the AI-compressor and providing an interface definition via environment variables in order for the CLAIMED compiler to read

```
(.venv) romeokienzler:notebooks$ claimed \
        --component containerless/claimed-ai-compressor:latest \
        --nnodes 1 \
        --nproc_per_node 1 \
        --dataset millionAID \
        --model mae_vit_compress_adapter \
        --epochs 20 \
        --warmup_epochs 1 \
        --data_path = /tmp/ \
        --save_every_n_epochs 2 \
        --num_workers 8 \
        --ld 1e10 \
        --path_to_pretrained_model /tmp/ \
        --output_dir /tmp/ \
        --log_dir /tmp/ \
        --blr 1.5e-4 \
        --weight_decay 0.05 \
        --input_size 224 \
        --batch_size 256
Entering containerless operation
Executing: python .//claimed-ai-compressor.latest/runnable.py --env nnodes=1 --env nproc_per_node=1 --env dataset=millionAID --env model=mae_vit_compress_adapter
--env epochs=20 --env warmup_epochs=1 --env data_path==/tmp/ --env save_every_n_epochs=2 --env num_workers=8 --env ld=1e10 --env path_to_pretrained_model=/tmp/ --env
output_dir=/tmp/ --env log_dir=/tmp/ --env blr=1.5e-4 --env weight_decay=0.05 --env input_size=224 --env batch_size=256
2024-09-26 23:03:55,145 - root - INFO - Parameter: nnodes = "1"
2024-09-26 23:03:55,145 - root - INFO - Parameter: nproc_per_node = "1"
2024-09-26 23:03:55,145 - root - INFO - Parameter: dataset = "millionAID"
2024-09-26 23:03:55,145 - root - INFO - Parameter: model = "mae_vit_compress_adapter"
2024-09-26 23:03:55,145 - root - INFO - Parameter: epochs = "20"
2024-09-26 23:03:55,145 - root - INFO - Parameter: warmup_epochs = "1"
2024-09-26 23:03:55,145 - root - INFO - Parameter: data_path = "=/tmp/"
2024-09-26 23:03:55,145 - root - INFO - Parameter: save_every_n_epochs = "2"
2024-09-26 23:03:55,145 - root - INFO - Parameter: num_workers = "8"
2024-09-26 23:03:55,145 - root - INFO - Parameter: ld = "1e10"
2024-09-26 23:03:55,145 - root - INFO - Parameter: path_to_pretrained_model = "/tmp/"
2024-09-26 23:03:55,146 - root - INFO - Parameter: output_dir = "/tmp/"
2024-09-26 23:03:55,146 - root - INFO - Parameter: log_dir = "/tmp/"
2024-09-26 23:03:55,146 - root - INFO - Parameter: blr = "1.5e-4"
2024-09-26 23:03:55,146 - root - INFO - Parameter: weight_decay = "0.05"
2024-09-26 23:03:55,146 - root - INFO - Parameter: input_size = "224"
2024-09-26 23:03:55,146 - root - INFO - Parameter: batch_size = "256"
```

Figure 9: How the parameters supplied on the CLAIMED CLI are passed to the underlying containerless AI-compressor