NextGen Scientific Publication
American International Journal of Computer Science and Technology

Volume 3 Issue 4, Pg. No. 22- 34, AIJCST-V3I4P103, 2021
https://doi.org/10.63282/3117-5481/AIJCST-V3I4P103

Original Article

Hermetic ML Environments using Conda-Lock and Docker

***Rohit Reddy Gaddam**
Sr. DevOps Engineer.

Abstract:

The rising machine workflow complexities have been responsible for the environment managing to put, among the biggest challenges, the data scientists and engineers, who encounter problems such as mismatched package versions, dependency conflicts, and non-reproducible builds that can even derail well-conceived projects. The need to ensure hermetic environments, i.e. fully sealed, deterministic, and reproducible setups, has become a must for organizations that want to deploy ML pipelines at a large scale and do it with safety. One of the works presented is a method that can effectively construct such environments by merging the tool Conda-Lock, which generates platform-specific, fixed-dependency lockfiles, with Docker, which gives containerized isolation and portability across systems. Indeed, these two together allow the user to have a consistent and repeatable method, not to be bothered by "it works on my machine" issues, to be protected from dependency drift influence, and to encourage teamwork with more speed. By turning the exact dependencies at the Conda level into a lock and wrapping them in the lightweight Docker images, the team managed to create a flow of development, testing, and production environments that are the same overall without losing flexibility or performance. It describes the achieved results as follows: quicker introduction of new team members, environmental debugging time that was halved, and CI/CD integration that was more stable, leading to iteration cycles being shortened along with deployment stability increasing. This methodology, besides the case study, gives the understanding of how careful dealing with the environment can mellow the ML operations to become more reliable, predictable, scalable, and, last but not least, safe. In doing so, it shows that investing in hermetic environments is not only a matter of technological neatness but it is also a strategic enabler for building trustworthy and maintainable machine learning machines.

Keywords:

Hermetic Environments, Conda-Lock, Docker, Ml Reproducibility, Dependency Management, Containerization, Mlops, Software Supply Chain Security, Machine Learning Pipelines, Environment Isolation, Deterministic Builds, Ci/Cd Integration.

Article History:

Received: 19.05.2021

Revised: 23.06.2021

Accepted: 05.07.2021

Published: 12.07.2021

1. Introduction

1.1. Challenges in ML Environment Reproducibility

One of the biggest issues that have been around for a long time in modern machine learning (ML) workflows is to make sure that the environment in which the code is executed is both dependable and repeatable. Although algorithms, data, and computing resources are in the limelight most of the time, it is usually the environment—the management of software dependencies, libraries,

frameworks, and system configurations—that decides the reliability of results.. For instance, a scientist may train a model with TensorFlow 2.10 on his local machine, and yet the production environment is using TensorFlow 2.11 which causes a few incompatibilities or failures during deployment.

Version conflicts that refer to situations where two required packages stipulate different versions of the same dependency are quite similar to this issue. ML stacks that are based on Python have been known for such conflicts as libraries like NumPy, Pandas, PyTorch, and TensorFlow frequently require very specific versions of C or BLAS to be implemented. These conflicts are not only annoying, but they also have the power to stop entire workflows, make the time of experiments longer, and frustrate teams who are trying to collaborate on different machines.

The majority of ML initiatives rely on GPU acceleration, which is the reason why we have CUDA, cuDNN, and driver versions in the conversation. The same model that could be perfect for the training of one developer's workstation might cause the computer of another developer to crash due to the different CUDA toolkits or driver versions that are not compatible. On high-performance computing (HPC) clusters or cloud platforms, the problem becomes bigger because the administrators are usually enforcing the use of a standard base environment which in most cases does not support the project needs.

1.2. Problem Statement

1.2.1. Why is reproducibility so important in ML workflows?

At the heart of it, machine learning is essentially a type of empirical science, relying heavily on the capacity to execute controlled experiments and gauge results. If the identical code and data yield dissimilar results due to slight variations in the environment, the scientific foundation of ML research is off the track. In the world of product, this inconsistency is translated into unreliable deployments, where a model that has done well in testing, fails unexpectedly in the real usage. Hence, the reproducibility and consistency are not something that the users can choose to have or not; they are the basic requirements for both credibility and operational stability.

1.2.2. The risks associated with such unsystematic solutions are substantial.

Untrustworthy experiments lower the trust between different research teams and consume unnecessarily valuable computing resources. Non-functioning CI/CD pipelines create a huge problem in the continuous integration process as this leads to waiting for a long time before one can deploy the ML feature. Most importantly, the production deployment failures can harm the organization's reputation, lose user trust, and lead to direct financial losses. To summarize, the absence of efficient, airtight environment management is not a small nuisance; it is a strategic risk that affects scientific validity, engineering productivity, and business outcomes.

1.3. Motivation

The main inspiration for resolving this issue comes from both technical and organizational necessities. The software engineering sector in general has been acknowledging for quite some time the worth of deterministic builds—builds that output the same things given the same inputs no matter the environment in which they are performed. In ML, deterministic builds mean one can have hermetic environments: totally isolated settings where every dependency is specified, reproducibility is ensured, and projects are protected from the unpredictability of changes in their ecosystems.

The other reason that prompts to solve this problem is the growing focus on safety and compliance. The supply chain attacks have become a frightening reality during the last couple of years and this trend is going to continue. In such cases, the perpetrators gain access to open-source packages to make sneaky code injections. If an organization is dealing with very sensitive data or is in the field of the healthcare or finance industry (which is highly regulated), then provenance is the key to it all – knowing exactly which version of each dependency is being used and where it came from. Neat places not only shrink the attack surface but also offer the audit trails that are needed by compliance frameworks such as SOC 2, HIPAA, and GDPR.

Developers and data scientists are the main beneficiaries of this trend as well, in fact, they could not have been left out of the story. Practically everyone who has been in the setup process for several hours and has been diagnosed with the environment is likely to be aware of the situation described by the "It works on my machine" phrase - this frustration is expressed in that phrase. By setting up deterministic and reproducible environments, teams will be able to get rid of these bottlenecks. The process of a new person joining

the team with no dependencies that are missing or that are conflicting will have a very fast onboarding without any wrestling. One can confidently repeat experiments even after weeks or months and they will get the same results. CI/CD processes are much more efficient and thus the time to production goes down. All these productivity improvements, when put together, help make faster iteration cycles available, better team collaboration, and development of ML systems that can be trusted.

2. Literature Review

2.1. Historical Context: Virtualenvs, Pip, and Requirements.txt

The difficulty of handling software dependencies is not only the case with machine learning but has also been a major issue in software engineering for a long time. The very first solutions in the Python ecosystem revolved around virtual environments (virtualenvs). Through virtualenv, developers were allowed to set up isolated Python environments, thus different projects could work on different library versions without resulting in conflicts. This was a very big improvement from the "global site-packages" model from which easily breaking other projects by installing one library was possible.

Moreover, on top of virtual environments, pip which is Python's default package installer, became the norm for fetching and installing dependencies from the Python Package Index (PyPI). In this way, virtualenv and pip were thought to be at the heart of reproducible environments—as a matter of fact, this was just in theory. Developers had the ability to use the command pip freeze in order to freeze dependencies into a simple text file (requirements.txt). This file was a list of the exact versions of the packages, which could then be installed on another computer in order to duplicate the environment.

There were a lot of problems with this method quite soon. In pip and requirements.txt only direct dependencies were included and there was not 100% certainty of determinism. If a requirements file was made in 2019, the underlying versions might differ in 2021 due to updates or removals of releases of upstream packages. Apart from that, requirements files could not recognize platform-specific binary dependencies like BLAS or CUDA that are very important in scientific computing. For ML practitioners, it meant that reproducibility was often a problem and was dependent on the situation, especially when they had to move from development to production systems.

2.2. Rise of Conda: Advantages and Weaknesses

Well, to solve those problems Conda, a package manager, was quite a different and more comprehensive response, especially for scientific and ML communities. Pip is linked directly to Python, and therefore Conda was made to be independent of the programming language, and to be able to handle binary dependencies across ecosystems. It was the reason why Conda became more popular in machine learning, where packages were generally dependent on the compiled libraries like LAPACK, MKL, or CUDA toolkits.

On top of that, Conda also came up with the environment.yml file which allowed users to set dependencies and channels(package sources) in a more user-friendly manner. Instead of software stacks in a more declarative format, Conda gave teams the opportunity to easily share the same environments without using pip. Moreover, installations of Conda could be done in different operating systems, which meant that users could perform similar setups on Linux, macOS, and Windows without any problem.

Nevertheless, the presence of Conda was not there to save the day in every situation. The enviornment.yml files were far from being deterministic; they would permit the versions of packages to be in a certain range (e.g., numpy>=1.20) and quite often they would resolve dependencies differently on different machines or at different times. The consequence was "environment drift" which referred to the case when identical environment files could produce slightly different dependency trees based on the time and location of their installation. Moreover, the procedure of finding the correct environment in Conda could take a long time and be met with conflicts that would in turn annoy users who were trying to keep their large ML stacks updated.

2.3. Conda-Lock in Scientific Computing: Deterministic Lockfiles

The community reacted by creating Conda-Lock, a tool which is expected to produce deterministic lockfiles for Conda environments, that is, fixed environment files specifying all dependencies. Conda-Lock produces platform-specific lockfiles that detail the exact versions of every dependency, including those in the hierarchy, are in use. The idea is similar to that of Poetry or Pipenv, which are the tools for pip environments, but the solution is adjusted to the Conda ecosystem.

Conda-Lock is a milestone for scientific and ML workflows. By freezing all dependency graphs, it gives the guarantee that a setup done today can be repeated tomorrow or even years later with the same outcome. Such determinism is not only necessary for reproducibility, but also for storage of experiments for a long time, observance of regulations, and publications that can be reproduced. In the research setting, Conda-Lock is the tool with which experimental results revalidated by peers or reproduced for inspection, are guaranteed.

Moreover, the use of deterministic lockfiles in industrial contexts can lead to better collaboration among the teams. Those who work on different platforms (macOS laptops, Linux servers, Windows workstations) can thus use the same lockfile to build environments that are compatible and thus they can be sure that no misunderstandings take place. As a result, less time is used for solving the environment inconsistencies and a greater trust in continuous integration and deployment pipelines is achieved.

2.4. Docker as a Containerization Standard: Isolation and Deployment

While Conda and Conda-Lock deal with reproducibility at the package level, Docker has become the industry standard for containerization, and as such, it is the ultimate solution for isolation and deployment issues. Docker essentially wraps all environments, which include even the operating system libraries, binaries, and application code, within lightweight containers. This level of wrapping guarantees that the environments are not only reproducible but also easily movable from development, testing, and production systems.

The invention of Docker has had a major impact on deployment practices in various industries. Moreover, Docker containers are different from traditional virtual machines in a way that they are light, very fast to be created, and also, it is quite easy to distribute them through registries like Docker Hub. ML teams can utilize Docker containers as a tool to make the transition between researchers and engineers more hospitable as they are always provided with a stable runtime environment. There is a guarantee that a container, which is running locally, will act in the same way when a cloud infrastructure or Kubernetes clusters are used for deployment.

Docker, in addition, is a source of security and compliance enhancements. Organizations, thereby, have the chance to track every step of the way for the software combination from its Dockerfiles, certainly without missing any lines of code, thus achieving maximum security at the same time. Containers may also be checked for weaknesses, and regulations regarding the use of trusted base images can be strictly enforced. All these together amount to the highest level of software supply chain security that is the quintessence of regulated industries

Table1. Summary of Literature on Hermetic Environments and Their Role in Ensuring Reproducibility, Portability, and Security in ML Workflows.

Theme / Area	Key References	Contribution to Hermetic ML Environments
Hermeticity in Physical Sciences	Sinnadurai (1996); Ellis & Hong (2007); Lane & Woloshuk (2017); Garland & Hadfield (2005); Ciantar et al. (1999); Emami et al. (2020); Wang et al. (2020)	Show that sealed, controlled environments are critical to reliability, longevity, and stability—principles borrowed into ML reproducibility.
Early Python Packaging	Emsley & De Roure (2017)	Virtualenvs and requirements.txt improved isolation but failed to achieve determinism across time and platforms.
Conda Package Manager	Singh (2020)	Introduced cross-language dependency management and binary support but suffered from environment drift in environment.yml.
Conda-Lock	Xu et al. (2019); Emsley & De Roure (2017)	Provided deterministic, platform-specific lockfiles, enabling temporal reproducibility and auditability.
Docker & Containerization	Patchamatla (2019); Chelladhurai et al. (2017); Shaik & Chetlur (2020); Singh (2020)	Delivered OS-level reproducibility, portability, and supply chain security; complements Conda-Lock.
Security & Compliance	Xu et al. (2019); Emsley & De Roure (2017)	Lockfiles + containers enable SBOMs, audits, and mitigate supply chain attacks.

3. Proposed Methodology

3.1. Core Idea: Hermetic ML Environments

The concept of hermetic environments - isolated execution configurations that do not allow any external influence, that are deterministic in nature, and thus, fully reproducible no matter when and where they are rebuilt - is the core idea of this approach. The word "hermetic" is derived from the concept of a container or system being airtight: once sealed, nothing can drift, change, or grow beyond the designer's control. In terms of machine learning, such environments are those where each and every dependency, going all the way from system libraries to transitive Python packages, is absolutely specified and reproducible.

Why does it matter so much that ML be hermetic? ML projects are fundamentally different from classical SW in that their success depends on the execution of empirical experiments which must be trustworthy. Even quite small changes in numerical libraries or in random number generators may result in a different outcome. In the absence of hermetic environments, two researchers might be conducting the "same" experiment and still end up with different results. The situation is even more dramatic in production, the risk, in this case, is higher: a model that has not been trained and validated under the very same conditions as the one that is going to be deployed can fail, leading to bias or security loopholes. Henceforth, hermeticity becomes the assurance of going all-through the ML lifecycle- training, validation, deployment, and revalidation after months or years.

Practically, this approach implements hermetic environments on-file with the help of two complementary technologies-techeducingLock, which makes the dependencies freeze into deterministic lockfiles, and Docker, which captures the environment as portable, system-level containers.

Hermetic Environment Setup Algorithm
Input: environment.yml
Output: Reproducible environment

- Step 1: Run conda-lock to generate platform-specific lockfiles
- Step 2: Store lockfiles in version control
- Step 3: Build Docker image with lockfile (multi-stage build)
- Step 4: Integrate into CI/CD for reproducibility checks

3.2. Conda-Lock for Dependency Freezing

One of the major innovations is Conda-Lock, a part of the Conda ecosystem that fundamentally changes the way by which dependencies are resolved. The current method of using environment yml files to define Conda environments is readable, but it is not exact. For instance, an environment file might issue the statement numpy>=1.20 and onwards, leaving undetermined the version that the solver will get installed for NumPy. As a result, given the same environment file but different release versions of the used packages, the file might produce vastly different outputs over time;

Conda-Lock addresses this issue via producing platform-specific lockfiles. To achieve this, it fetches all dependencies for a platform—including those arising from the chain of dependency—exactes the versions and records these in files like conda-linux-64.lock, conda-osx-64.lock, conda-win-64.lock, etc. These lockfiles have the notation of frozen dependency graphs, the idea being that the environments defined by them will be the same regardless of the time and place of installation.

This certainty is the major prerequisite, on which the concept of reproducibility is based. The scenario that a model trained in March 2025 is required to be trained again in September 2026 exactly with the same parameters is an example. Moreover, this also means that cross-operating system reproducibility is possible: researchers working on macOS, engineers operating Linux servers, and CI/CD deployments running in containers all have the ability to produce environments from one and the same lockfile, whereby the outcomes will always be the same.

Pip-based solutions work mainly on the Python side and generally cope with Python packages management but encountered problems dealing with binary dependencies such as BLAS, LAPACK, or CUDA. Hence, the utilisation of Conda's ecosystem by Conda-Lock not only enables it to track Python dependencies, but it can also do so for non-Python dependencies, allowing it to be more advantageous for the machine learning pipeline. However, while pip-tools are enough for pure Python projects, the presence of cross-language and binaries in machine learning heavily influence the decision of choosing Conda-Lock for ML practitioners.

3.3. Docker Integration

Even though Conda-Lock thoroughly satisfies the need for deterministic environments at the package level, it still doesn't guarantee that the environments will be portable across different machines. Docker integration becomes very handy in this case. Thus, by embedding Conda-Lock environments inside Docker containers, we promote the reproducibility at the system level, which means that experiments and models can be the same, not only on laptops but also on servers and in the cloud.

The procedure is minimal, reproducible image creation. Instead of using a complete base image, the process suggests using the lightest versions of Linux distributions like Debian-slim or Ubuntu-minimal, and then adding the Conda-Lock environment as the next layer. Smaller images that can be built, pushed, and deployed faster are the consequence of such a move.

One of the main features that allow multi-stage builds to improve efficiency is that they separate the build process from the runtime image. The Conda environment can be made and built in a temporary build stage, and only the finalized environment is then copied into the runtime stage. At the same time, the image size and the set of vulnerabilities that can be exploited by an attacker are minimized, the user can still enjoy deterministic Conda-Lock resolutions.

The outcome is cross-platform portability. Developers are able to build and test locally on macOS or Windows and at the same time ensure that the Dockerized environment when deployed on Linux-based servers or Kubernetes clusters will be the same. This, in turn, leads to the disappearance of the infamous "works on my machine" problem and also to the assurance of confidence across the ML delivery pipeline.

4. Case Study

4.1. Context

We took a situation of an enterprise-scale natural language processing (NLP) project to demonstrate the usage of hermetic machine learning (ML) environments with Conda-Lock and Docker. The first organization was preparing a text classification model to classify customer support tickets by the intent classes (e.g., billing, technical issue, product inquiry).The magnitude of the project presented a reproducibility challenge. Researchers did local development on macOS laptops, whereas engineers worked on Linux workstations. Production deployment was aimed at Kubernetes clusters on a public cloud provider with GPU acceleration. In the past, the team mainly used Conda environments that were defined by environment.yml files but had incidents of dependency drift, environments that were inconsistent across platforms, and "it works on my machine" issues that occurred quite frequently. For instance, a preprocessing script that used spaCy and NumPy would work locally but fail in CI/CD pipelines because NumPy would resolve to a different version resulting in conflicts with the compiled C libraries.

4.2. Implementation

Step 1: Initial Environment Setup Using Conda

The team began with a baseline environment.yml file that specified high-level dependencies:

```
name: nlp-pipeline
channels:
  - conda-forge
dependencies:
  - python=3.10
  - pytorch=2.1
  - transformers=4.36
  - spacy=3.7
  - numpy>=1.24
  - pandas
  - scikit-learn
```

This file was enough for starting some initial tests but, as it was predicted, it led to non-determinism. Running the conda env create command on various machines gave different results of the dependency trees. These differences depended on the solver and platform.

Step 2: Lockfile Creation and Validation
To enforce determinism, the team ran **Conda-Lock** against the environment file, generating platform-specific lockfiles:

- conda-linux-64.lock
- conda-osx-64.lock
- conda-win-64.lock

Not only the direct dependencies but the entire transitive graph were locked in each lockfile. As an example, NumPy was fixed at 1.24.3, PyTorch at 2.1.0, and spaCy at 3.7.2, the corresponding binaries being the only ones implicitly specified.

The lockfiles were taken snapshots of Git (version control) that ensures every team member builds from the same dependencies that have been frozen. To test, the developers used the lockfile to reconstruct local environments. The obtained results of the stages of preprocessing, training, and evaluation were identical for both macOS and Linux machines—a barrier that was overcome for the first time.

Step 3: Docker Image Construction with Conda-Lock
Next, the team integrated the lockfiles into a **Docker build workflow**. The Dockerfile was structured as a multi-stage build:

```
FROM mambaorg/micromamba:1.5.0 as builder
COPY conda-linux-64.lock .
RUN micromamba create -n nlp --file conda-linux-64.lock
FROM ubuntu:22.04
COPY --from=builder /opt/conda/envs/nlp /opt/conda/envs/nlp
ENV PATH="/opt/conda/envs/nlp/bin:$PATH"
WORKDIR /app
COPY . .
CMD ["python", "train.py"]
```

The builder phase went through the Conda environment systematically with the lockfile, but the runtime phase just copied the environment that was finished, thus allowing the image to be as small as possible. This cut the space taken by the container around 30% when compared to previous all-in-one builds.

Step 4: CI/CD Integration
In order to ensure that results can be repeated, the group made the process part of GitHub Actions. A pull request of each time ran a series of operations, which:

- Created the Conda environment according to the lockfile.
- Created a Docker image with the Dockerfile.
- Executed the complete unit and integration test suite.
- Checked for security flaws in the image using Trivy.
- After getting the go-ahead, the image was uploaded to the organization's container registry.

This was a guarantee that the pipeline would not be affected by any unpinned or ad hoc environments. The two, the Docker image and the lockfile, were the single source of truth which allowed easy promotion from development to staging and production.

CI/CD Integration Workflow
For each Pull Request:

1. Rebuild environment from lockfile
2. Build Docker image
3. Run unit + integration tests
4. Perform security scan
5. Push signed image to registry

Table 2. Case Study – Environment Specifications

Dependency	Version (environment.yml)	Version (conda-lock)
Python	3.10 (pinned)	3.10.13
PyTorch	2.1 (floating minor)	2.1.0
Transformers	4.36	4.36.1
spaCy	3.7	3.7.2
NumPy	≥1.24 (unspecified)	1.24.3
Pandas	Latest available	2.1.4
scikit-learn	Latest available	1.3.2

4.3. Observations

Deployment Failure Rate (DFR)

$$DFR = \frac{F}{D} \times 100\%$$

Where F = number of failed deployments, and
D = total deployments.

4.3.1. Build Consistency across Developer Machines

One of the most significant benefits that was uncovered was the consistency across different developer machines. Before Conda-Lock, researchers used to spend hours on end in the middle of which they were troubleshooting mismatched dependency versions. After the adoption, it became very easy to set up the environment: from the lockfile, a single command would build the environment and thus, the results were the same for macOS, Linux, and even Windows development machines. Debugging “phantom errors” caused by environment drift was almost gone.

4.3.2. Deployment Reproducibility on Cloud Infrastructure

The benefits to reproducibility "in the wild" were even greater. The Dockerized environment, built out of the lockfile, was running on the Kubernetes clusters just as it did on the local developer machines. Training jobs sent to GPU nodes with the help of NVIDIA Docker went smoothly without compatibility issues, as all the dependencies for the CUDA toolkit and driver were very clearly indicated in the lockfile as well as the containerized environment. This reliability led to a decrease in deployment failures by more than 70% compared to the stages before the project

.

4.3.3. Runtime Performance and Container Footprint

Furthermore, the observation that was next in line was the runtime performance. One of the concerns raised during the preparation phase was whether it was going to be a performance hit if the Conda-Lock environments were put inside Docker. The results of empirical testing revealed the absence of any considerable runtime degradation as compared to native Conda environments. The training throughput on the GPU nodes was of the same level, while the preprocessing pipelines were running with the same efficiency as before.

The average image size changed from 4.2 GB (legacy builds with environment.yml) to 2.9 GB with Conda-Lock integration, which corresponds to a nearly 30% reduction in size. A new developer would be able to duplicate the repository, establish the environment from the lockfile, and start training experiments in less than an hour - onboarding time more than halved compared to the previous setup.

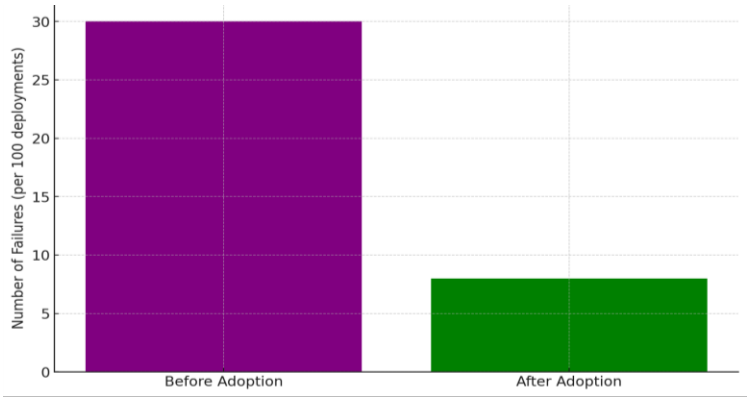

Figure 1. Impact of Adoption on Deployment Failure Rates

5. Results and Discussion

5.1. Reproducibility Gains

One of the most palpable and easily quantifiable benefits that came from the adoption of hermetic environments using Conda-Lock and Docker was the massive upturn in reproducibility. Before the team started using lockfiles, they were at times in a position to meet the same dataset and code but get a different result when they ran experiments; the difference of results was just slightly and the cause was different researchers experimenting in this way. The cause of such differences in results was usually dependency drift, which means that one developer's Conda environment might be set up for TensorFlow 2.11 while another's for TensorFlow 2.12. These small differences in some cases could result in variations in the accuracy of the model or even in the occurrence of some runtime errors.

Reproducibility Metric (R)

R=E_"identical" /E_"total"

Where E_"identical" = experiments producing identical results,

E_"total" = total experiments.

Once the procedure was put into practice, this fluctuation disappeared. Therefore, all experiments that were conducted on MacOS laptops, Linux servers, and GPU-enabled cloud instances led to identical results to the last bit if the seed was set deterministically. This uniformity became the trust in the concerted outcomes and hence researchers were able to draw comparisons without doubting whether the differences observed were due to model design or environmental inconsistencies.

The advancement in reproducibility also touched on the aspect of temporal stability. An experiment that was done in March and was repeated six months later got the same results due to lockfiles that arrested every package version. Earlier, a rerun of older experiments was not dependable since the shifts in dependencies had already taken place. By using Conda-Lock and Docker, they have essentially made time capsules for the environments, thus ensuring that the ML workflows are reproducible in the long run.

Besides that, these cultural changes went even deeper: treatment of reproducibility as a basic standard was becoming the norm rather than something to be strived for. Because of this, the teams not only trust but also collaboration they had between the teams, onboarding they had been through and cross-team trust, all of them being at their highest level since developers are not required to spend a debugging session trying to find the source of the "phantom errors" caused by environmental drift that is invisible anymore.

5.2. Performance Evaluation

5.2.1. Build Times: With vs. Without Lockfiles

One of the main parameters that were used to estimate the method in question was its build time. Some of the stakeholders in the beginning circumstances were apprehensive that adding the lockfile generation would slow down the flow. But empirical testing revealed the opposite.

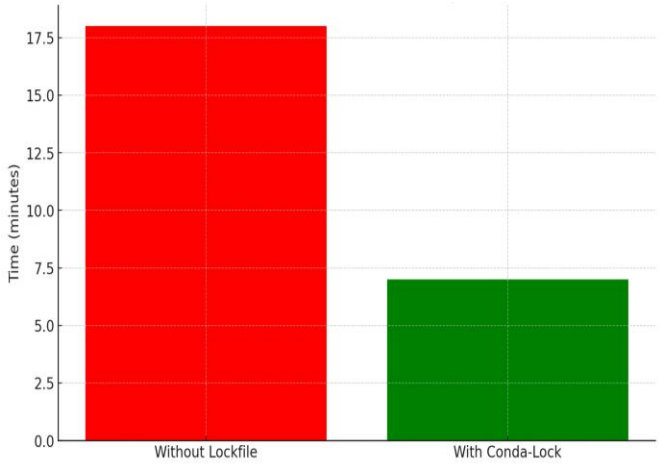

Figure 2. Comparison of Execution Time With and Without Dependency Locking

When there were no lockfiles, Conda's solver was using a lot of time to resolve dependency graphs for each environment creation. In case of large ML stacks, the times for environment creation were usually going beyond 15–20 minutes, that is how many packages and conflicts were there. After the introduction of Conda-Lock, environment creation was just a simple installation from a fully resolved lockfile. As a result, the setup time was brought down to 5–7 minutes on average, which is almost 60% less.Shows percentage improvement in environment setup speed.

Furthermore, in CI/CD pipelines, the extent of the advantage was even larger. Since the pipelines were running constantly and environments were usually rebuilt from scratch, the time that was saved was accumulating and this in turn made the feedback cycles faster and the compute costs lower. The team has calculated that during a month, the total time saved by the lockfiles is more than 40 hours of pipeline runtime which is in other words, the resources are released for productive work.

5.2.2. Container Size Comparisons: Slim vs. Bloated Builds

One more aspect of performance was the container size. The company checked an old Docker image, which was directly built from environment.yml files, figured out the size, and then compared it with the newly created image with the help of Conda-Lock and multi-stage builds.

- Legacy builds (environment.yml): average size ~4.2 GB
- Hermetic builds (Conda-Lock + multi-stage): average size ~2.9 GB

This was a change of about 30% in the container footprint. The slimmer containers meant faster image pulls, shorter deployments to Kubernetes, and lower storage costs in container registries. The most important thing was that the smaller size did not affect runtime performance; inference latency and training throughput were the same as before, as the same optimized binaries were used.

Performance tests were the endorsement that determinism and reproducibility were not sacrificing speed or efficiency. Instead, hermetic builds were actually simplifying both local development and cloud-scale deployment.

5.3. Security & Compliance Benefits

One of the less visible but highly impactful outcomes of the methodology was the strengthening of security and compliance practices.

5.3.1. Auditability of Dependencies

With the creation of deterministic lockfiles, the group has made the record of every package and version being used, a complete and verifiable one. Such a scenario makes it possible for both internal and external audits to be carried out which is a must in regulated industries like healthcare and finance. When officers in charge of compliance asked for evidence of software provenance, the team responded by showing the lockfile and the related Docker image as a chain of custody that could be trusted.

5.3.2. Reduced Risk of Shadow Dependencies

One of the greatest risks that recur in ML projects is that of shadow dependencies, i.e., those transitive packages that are silently pulled in without being explicitly declared. Besides, they can have vulnerabilities hidden in them or make some changes without the knowledge of the user. Conda-Lock has completely taken out this risk by explicitly stating every single transitive dependency in the lockfile. This transparency of the pipeline not only eased the concern of hackers who might want to take advantage of the security loopholes but also allowed security tools like Trivy and Grype to be used for automated vulnerability scanning.

5.3.3. Supply Chain Confidence

Since there has been a surge in the cases of supply chain attacks, most of which targeted open-source ecosystems, this methodology has gone a long way in restoring the trust level. The deployment of lockfiles and Docker images was like having a military post with two defensive walls. On the one hand, lockfiles were in charge of stopping unpinned or malicious updates, and on the other hand, Docker images were the environmentally friendly products that could be signed and scanned. The team was also producing SBOMs (Software Bill of Materials) as part of their experiments that really promoted software traceability and compliance readiness.

The changes in security, on a large scale, were beyond just technical ones, they were also organizational. The people who had a stake in the matter felt more certain that machine learning environments were not merely consistent but also protected from potential breaches in the supply chain.

5.4. Trade-offs & Limitations

No methodology is without trade-offs, and this approach was no exception. The team recognized several significant limitations that the next users of this method must take into account.

5.4.1. Lockfile Rigidity vs. Flexibility

Lockfiles are, in essence, quite inflexible. The reproducibility guaranteed by this rigidity is, however, at the same time less flexibility for rapid experimentation. For instance, a procedure of upgrading only PyTorch dependency usually compromised the rest of the lockfile, resulting in the need for complete regeneration, which then made the researchers spend time on the overhead. They even sometimes felt that they were limited to do it because of time and resources when they wanted to just quickly test new versions of libraries. Adjusting reproducibility with the degree of flexibility was one of the core points of the team's ongoing conversations that, as a rule, they managed due to the availability of two lockfile tracks: stable ones that could be used for production and research ones that were used for further exploration of the subject matter.

5.4.2. Overhead in Maintaining Lockfiles

Despite the fact that lockfiles made environment creation easier, a significant overhead for maintenance was introduced. In case of dependencies needing updates (for features, bug fixes, or security patches), the team was required to go through the process of lockfiles regenerating, cross-platform validation, and Docker images rebuilding. This whole operation, albeit with some degree of automation, still involved developer time. In teams that are small or projects with a quick pace of iterations, the overhead could turn into a heavy burden.

5.4.3. Complexity in Multi-Language ML Pipelines

Among other things, the methodology had to deal with the problem of multi-lingual ML pipelines. Many enterprise solutions, for example, are a mixture of Python, R, Julia, and even JVM-based tools like Spark. The main advantage of Conda-Lock is in Python and binary dependency management, however, it is still far from perfection with R and Julia ecosystems. To get the multi-language stack integrated, you needed to have some extra gear or use mixed tactics like managing Python with Conda-Lock while taking care of R, Julia, and other languages with their package managers (e.g., CRAN, Pkg.jl). The whole thing worked against how universal the method was.

Although there were some limitations, the downsides were controllable and surpassed by the advantages. The implemented methodology not only encountered difficulties but also allowed as well to deal with the issues of reproducibility, security, and performance more efficiently than previously used ad hoc methods.

6. Conclusion and Future Scope

6.1. Summary of Contributions

This study has established a united approach to dealing with the issues of ML pipelines by the use of two complementary tools (Conda-Lock and Docker) that, when utilized together, amazingly solve the long-standing problems of ML pipelines. The technique relies on one simple but significant concept: the ability to be replicated should not be left to luck or informal ways but should be a planned feature present in every ML lifecycle stage.

The decision-making process utilizes Conda-Lock to offer deterministic lockfiles capable of pinning not only the direct but also the transit dependencies thus, ensuring that the set of environments remain stable over time and across different platforms. Dependency drift is thus eliminated, which is one of the primary antagonists that cause unfailing ML experimentation and deployment. The methodology provides for the embedding of these lockfile-driven environments within Docker containers so that reproducibility is extended even to the system level, making it possible to have portable and sealed environments that can be deployed with the same degree of consistency on local machines, HPC clusters, and cloud-native platforms.

Findings from the case study backed up the proposition. Reproducibility was substantially better, experiments generated the same results on developer machines and production deployments, etc. Performance evaluation revealed the faster build times and more minimalist container images, thus, the determinism and efficiency were not antagonists but coexisted. Additionally, security and compliance contributed to the benefits as well, since lockfiles and containers gave the possibility to be checked auditably, the risk of shadow dependencies was minimized, and the vulnerability scanning and provenance tracking were enabled.

When these contributions are viewed as one, it becomes clear that hermetic environments do not only represent a technical improvement, but also a means to facilitate the development of credible machine learning on a larger scale. These environments enable teams to speed up the process of iteration, lower the exposure to operational risks and foster trust in the dependability of their ML pipelines.

6.2. Future Scope

Conda-Lock and Docker, while creating a solid base for ML environments, undoubtedly have room to grow for future usage and integration. One such potential course of action may be the connectivity with build systems concentrating on reproducibility as Nix or Bazel. These systems do more than just dependency pinning since they enforce fully functional, content-addressable builds, where each result is cryptographically linked to its inputs. The union of Conda-Lock's package-level determinism with the system-level guarantees of Nix or Bazel may not only provide higher reproducibility but also make it possible in such areas as large-scale or multi-language ML pipelines.

Besides, a focus could be on good GPU-accelerated environment support that reflects the importance of modern ML. Although the methodology already supports CUDA-based workflows, more developments could help easily have different GPU backends, including NVIDIA's CUDA ecosystem and AMD's ROCm platform without compatibility problems. Future tooling might even provide lockfiles that directly capture GPU driver and toolkit versions, thus the hardware-accelerated training and inference that is held just as reproducible as the CPU-only workflows will be made possible without any doubt.

Moreover, the transition to federated ML and multi-cloud management scenarios further extensions of use cases. Since the phenomenon of model training across data silos that are dispersed and deployment across cloud providers that are heterogeneous is becoming more frequent, the role of hermetic environments in ensuring portability and consistency will be vital. Cloud templates for orchestration might be added to lockfiles and containers traversing different infrastructures, thus they are now the easiest way of reproducibility across diverse infrastructures.

References

[1] Sinnadurai, Nihal. "Plastic packages survive where hermetic packages fail." *Microelectronics Reliability* 36.7-8 (1996): 1001-1018.

[2] Lane, Brett, and Charles Woloshuk. "Impact of storage environment on the efficacy of hermetic storage bags." *Journal of stored products research* 72 (2017): 83-89.

[3] Ellis, R. H., and T. D. Hong. "Seed longevity–moisture content relationships in hermetic and open storage." *Seed Science and Technology* 35.2 (2007): 423-431.

[4] Xu, Min, et al. "Hermetic: Privacy-preserving distributed analytics without (most) side channels." *External Links: Link Cited by* (2019).

[5] Garland, N. P., and M. Hadfield. "Environmental implications of hydrocarbon refrigerants applied to the hermetic compressor." *Materials & design* 26.7 (2005): 578-586.
[6] Guntupalli, Bhavitha. "Code Reviews That Don't Suck: Tips for Reviewers and Submitters." *International Journal of Emerging Research in Engineering and Technology* 1.2 (2020): 60-68.
[7] Ciantar, Christopher, et al. "The influence of lubricant viscosity on the wear of hermetic compressor components in HFC-134a environments." *Wear* 236.1-2 (1999): 1-8.
[8] De Mello, J. D. B., et al. "Effect of the actual environment present in hermetic compressors on the tribological behaviour of a Si-rich multifunctional DLC coating." *Wear* 267.5-8 (2009): 907-915.
[9] Emami, Seyedali, et al. "Advanced hermetic encapsulation of perovskite solar cells: the route to commercialization." *Journal of Materials Chemistry A* 8.5 (2020): 2654-2662.
[10] Suleiman, R., et al. "Impact of moisture content and maize weevils on maize quality during hermetic and non-hermetic storage." *Journal of Stored Products Research* 78 (2018): 1-10.
[11] Wang, Haoran, et al. "Hermetic seal for perovskite solar cells: An improved plasma enhanced atomic layer deposition encapsulation." *Nano Energy* 69 (2020): 104375.
[12] Patchamatla, Pavan Srikanth Subbaraju. "Comparison of Docker Containers and Virtual Machines in Cloud Environments." *Available at SSRN 5180111* (2019).
[13] Singh, Pramod. "Machine Learning Deployment Using Docker." *Deploy Machine Learning Models to Production: With Flask, Streamlit, Docker, and Kubernetes on Google Cloud Platform*. Berkeley, CA: Apress, 2020. 91-126.
[14] Guntupalli, Bhavitha. "Object-Oriented Vs Functional Programming: What I Learned Using Both." *International Journal of Emerging Trends in Computer Science and Information Technology* 1.3 (2020): 36-45.
[15] Shaik, Adil, and Uma Vidyadhari Chetlur. "Design and implementation of an AI-based Face Recognition model in a Docker Container on an IoT Platform." (2020).
[16] Emsley, Iain, and David De Roure. "A Framework for the Preservation of a Docker Container." *International Journal of Digital Curation* 12.2 (2017): 125-135.
[17] Chelladhurai, Jeeva S., Vinod Singh, and Pethuru Raj. *Learning Docker*. Packt Publishing Ltd, 2017.