Containers in HPC

Maciej Marchwiany, PhD

https://www.linkedin.com/pulse/ai-powered-physics-how-ai-revolutionizing-scientific-discovery-cain-ddbwc

Plan

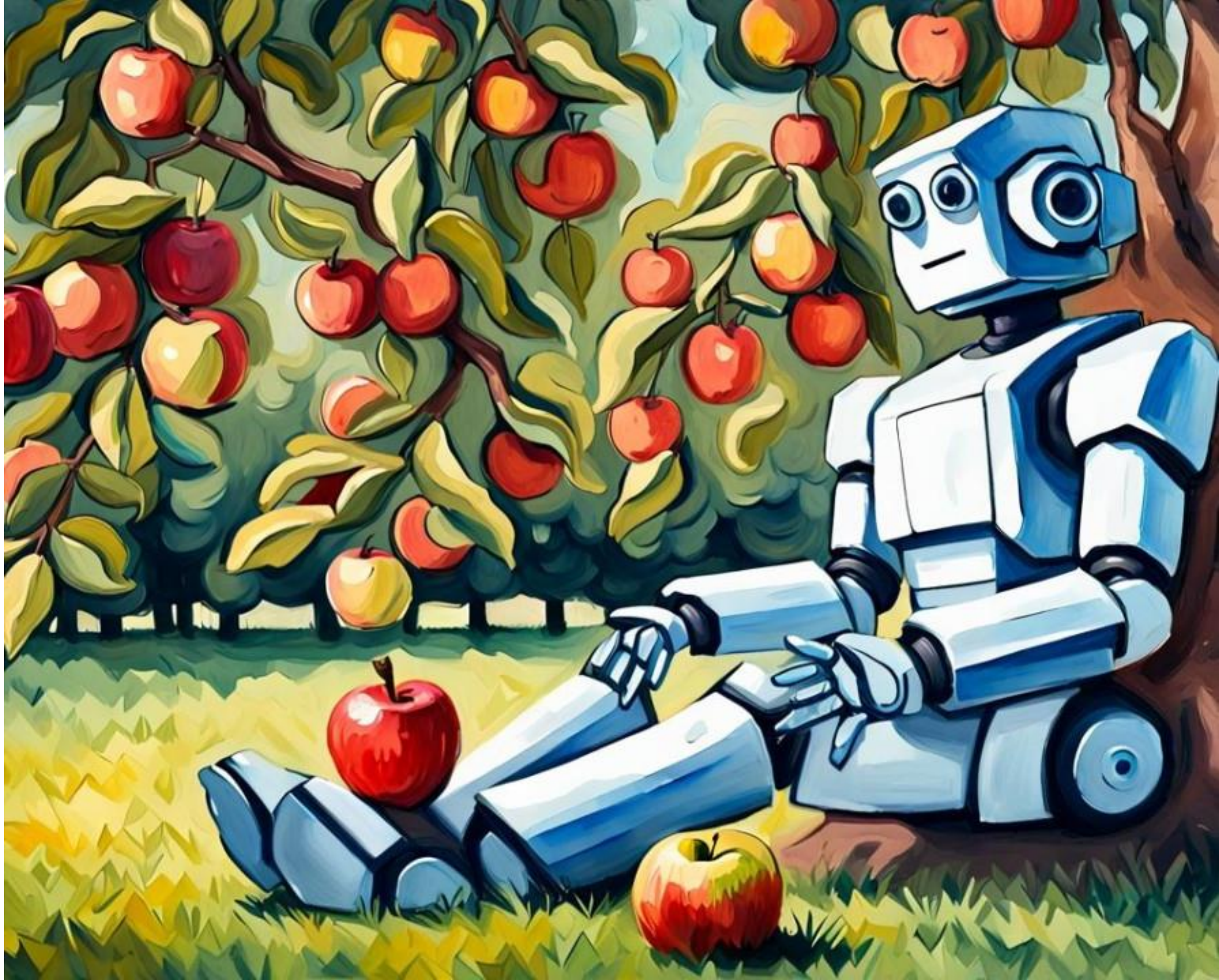
containers

tools

creation

MPI, CUDA,..

Containers



Containers

A container is a lightweight, portable unit that encapsulates an application along with all its **dependencies**, **libraries**, and **configuration** files. This allows the application to run consistently across **different** computing **environments**, ensuring isolation from other applications and systems. Containers facilitate **reproducibility**, **simplify deployment**, and enhance resource utilization by sharing the host operating system's kernel while maintaining separate user spaces.A

When to Use Containers in HPC

- **Reproducibility**:
  Ensures consistent application performance across different environments.
- **Simplified Deployment**:
  Packages applications with all necessary dependencies, reducing complexity.
- **Cross-Platform Compatibility**:
  Facilitates easy movement and execution of applications across various HPC systems.
- **Resource Efficiency**:
  Shares the host system's kernel, leading to lower overhead compared to traditional virtual machines.
- **Faster Startup Times**:
  Allows for nearly instant application startup, enhancing responsiveness and scalability.
- **Improved Dependency Management**:
  Minimizes conflicts by encapsulating all necessary dependencies within the container.
- **Testing and Development**:
  Enables developers to create isolated environments that mimic production settings for effective testing.
- **Support for Complex Workflows**:
  Streamlines execution of multi-step processes by encapsulating each step within its own environment.
- **Enhanced Security**:
  Provides isolation between applications and the host system, improving security in multi-user environments.

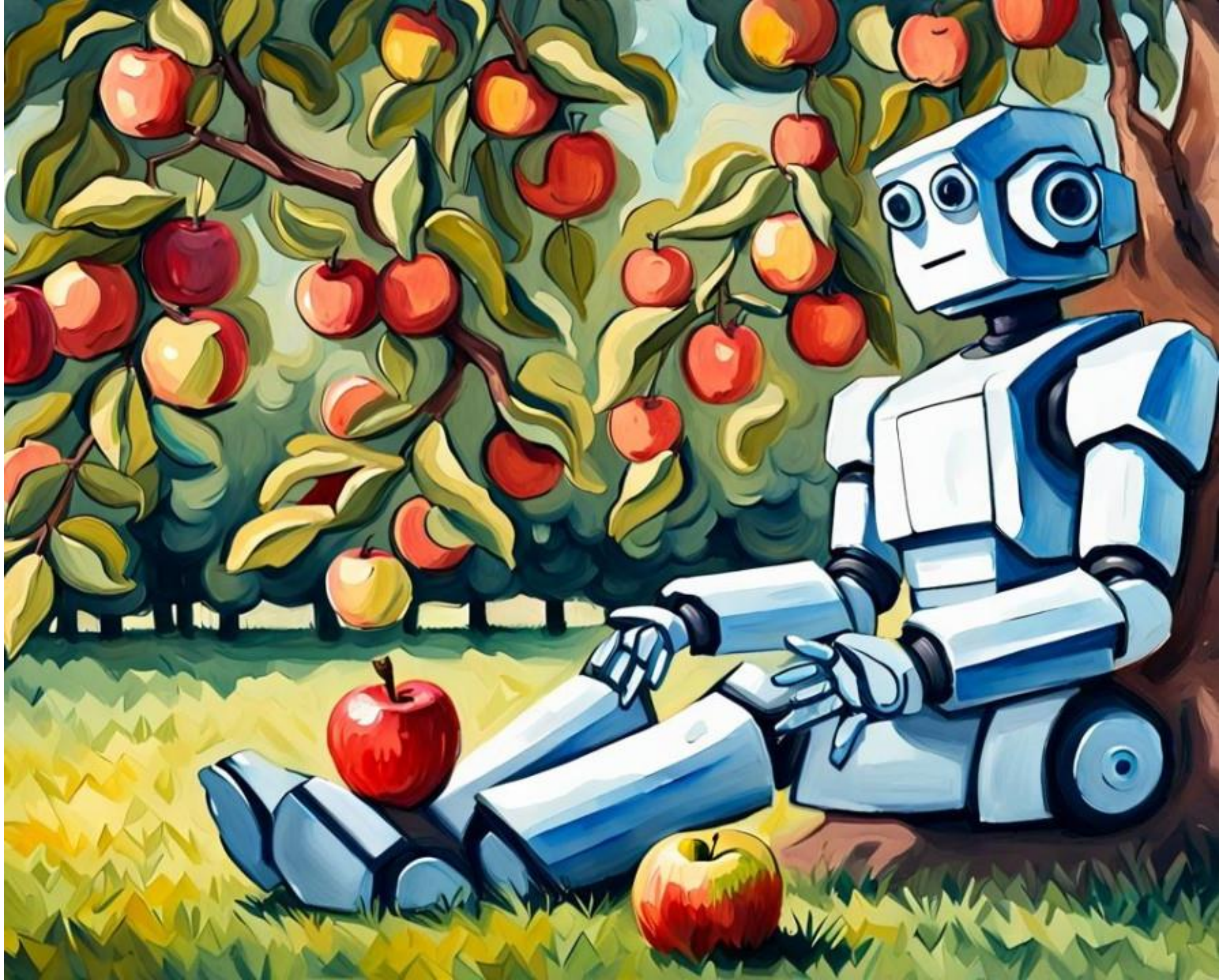
Use Cases

1. **Bring Your Own Environment** - Engineering work-flows for research computing can be a complicated and iterative process, and even more so on a shared and somewhat inflexible production environment. Apptainer solves this problem by making the environment flexible.
2. **Reproducible science** - Apptainer containers can be built to include all of the programs, libraries, data and scripts such that an entire demonstration can be contained and either archived or distributed for others to replicate no matter what version of Linux they are presently running.

Use Cases

3. **Static environments (software appliances)** - Fund once, update never software development model. While this is not ideal, it is a common scenario for research funding. A certain amount of money is granted for initial development, and once that has been done the interns, grad students, post-docs, or developers are reassigned to other projects. This leaves the software stack un-maintained, and even rebuilds for updated compilers or Linux distributions can not be done without unfunded effort.
4. **Legacy code on old operating systems** - Similar to the above example, while this is less than ideal it is a fact of the research ecosystem. As an example, I know of one Linux distribution which has been end of life for 15 years which is still in production due to the software stack which is custom built for this environment. Apptainer has no problem running that operating system and application stack on a current operating system and hardware.

Tools



Tools

- **Docker**:
The most popular containerization platform, Docker provides a robust ecosystem for building and deploying applications in containers. However, its default security model may not be compatible with HPC requirements.

- **Shifter**:
Designed for HPC, Shifter allows users to run Docker containers in an HPC environment without the need to build them directly on the cluster.

- **Enroot**:
Developed by Nvidia, Enroot is a simple tool that converts traditional container images into unprivileged sandboxes suitable for HPC applications.

- **Singularity**:
Specifically designed for HPC environments, Singularity offers strong reproducibility features and seamless integration with workload managers. It allows unprivileged users to run containers and is optimized for scientific workloads.

- **Apptainer**:
An evolution of Singularity, Apptainer aims to enhance security and performance for HPC workloads while maintaining compatibility with existing Singularity features.

Apptainer

Apptainer is a ***container*** **platform**. It allows you to **create** and **run** containers that package up pieces of software in a way that is **portable** and **reproducible**. You can build a container using Apptainer on your laptop, and then run it on many of the largest HPC clusters in the world, local university or company clusters, a single server, in the cloud, or on a workstation down the hall. Your container is a single file, and you don’t have to worry about how to install all the software you need on each different operating system.

Use containers

Available Commands:

build	Build an Apptainer image
config	Manage various apptainer configuration
delete	Deletes requested image from the library
exec	Run a command within a container
inspect	Show metadata for an image
pull	Pull an image from a URI
push	Upload image to the provided URI
run	Run default command within a container
search	Search a Container Library for images
shell	Run a shell within a container
test	Run the user-defined tests within a container

Running a container

Once you have your container ready, you can run applications inside it.

Executing commands inside the container

```
apptainer exec my_container.sif <command>
```

Interactive Shell

```
apptainer shell my_container.sif
```

Run the Container

```
apptainer run my_container.sif
```

exec vs run

The **run** command is used to **execute the default action** defined in the container. This action is specified by the `%runscript` section in the container's definition file.

The **exec** command allows you to **run** a specific **command** or **script** inside the container without executing the default action defined by the `%runscript`.

Using Apptainer run will trigger any `%runscript` defined in the container, whereas apptainer exec bypasses this and runs only the specified command.

arguments to exec

You can pass arguments to the runscript of a container, if it accepts them.

```
apptainer exec my_container.sif echo
“Hello”
apptainer exec my_container.sif echo
“\$HOSTNAME”
```

Working with Files

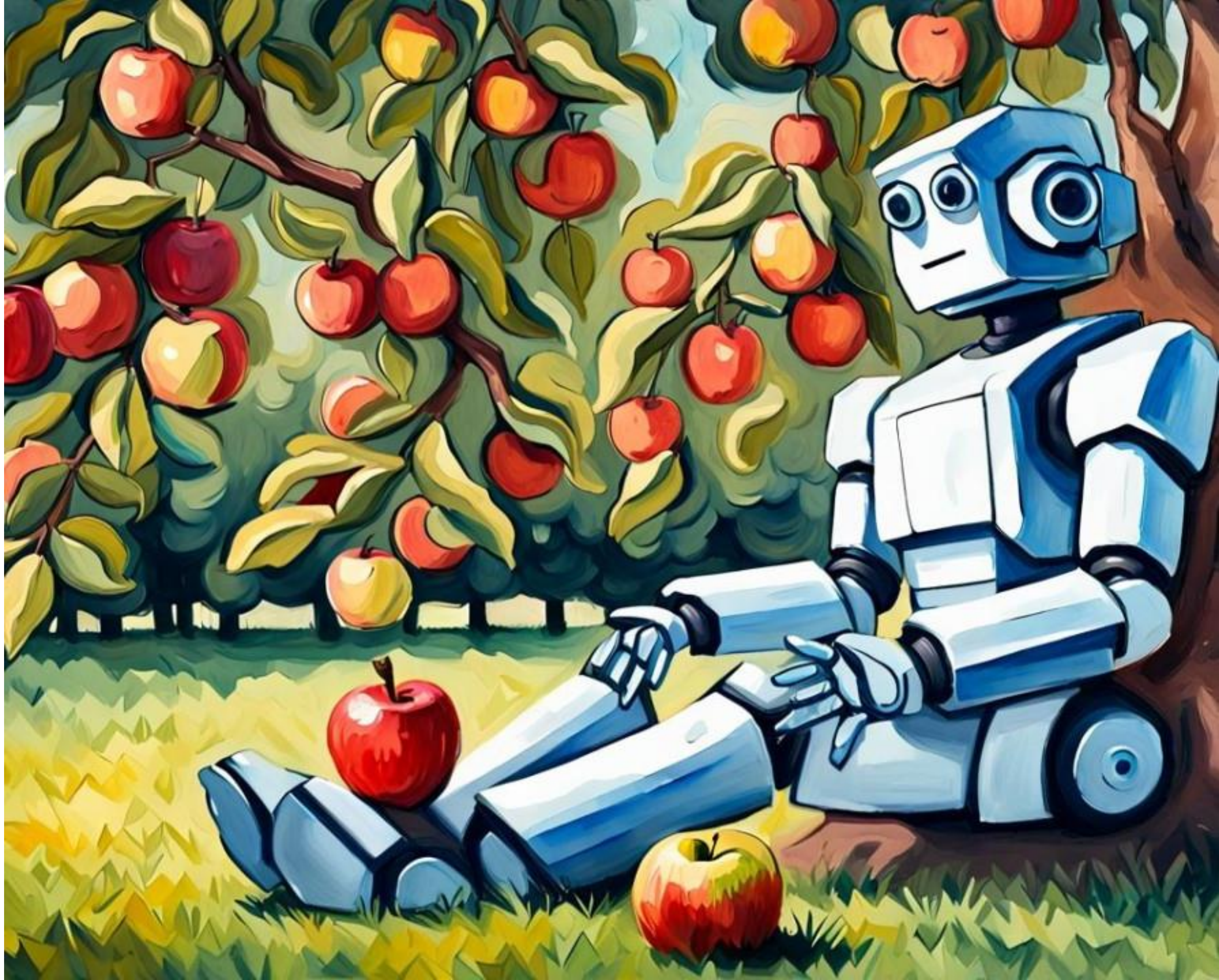
By default, Apptainer bind mounts:

- `$HOME`,
- the current working directory, and
- additional system locations from the host into the container.

You can specify additional directories to bind mount into your container with the `--bind` option.

```
apptainer exec --bind /data:/mnt
my_container.sif du -sh /mnt/
```

Creation



Downloading image

You can use the **pull** and **build** commands to **download** images from an external resource like an OCI registry.

```
apptainer pull docker://alpine

apptainer build alpine.sif
docker://alpine
```

Images repositories

1. **Docker Hub**: You can pull Docker images directly. For example:
   ```
   apptainer pull
   docker://tensorflow/tensorflow:latest
   ```
2. **Singularity Hub**: Apptainer supports pulling images from Singularity Hub. For instance:
   ```
   apptainer pull shub://vsoch/apptainer-
   images
   ```
3. **Apptainer Library**: You can also pull images from the Apptainer Library. An example command would be:
   ```
   apptainer pull
   library://gmk/default/centos7-devel
   ```
4. **NVIDIA GPU Cloud (NGC)**: Apptainer allows pulling images from NVIDIA's registry, which is useful for GPU-accelerated applications. An example command would be:
   ```
   apptainer pull
   docker://nvcr.io/nvidia/pytorch:23.05-py3
   ```
5. **OCI Registries**: Images can also be pulled from other OCI-compatible registries, such as Quay.io or Azure Container Registry.

pull vs build

The **pull** command is used to **download** a container image from a specified URI, such as Docker Hub or Singularity Hub.

The **build** command is more versatile and is used to **create** a **new** container image. This can be done from an existing image, a definition file, or even from scratch.

Pull focuses on **downloading** existing images, while **build** is used for **creating** new images or modifying existing ones. Build converts images into the latest Apptainer format, whereas pull simply retrieves them.

Building images from scratch - definition file

For a **reproducible**, **verifiable** and **production-quality** container, it is recommended that you build a SIF file using an Apptainer **definition file**. This also makes it **easy** to add files, environment variables, and install custom software. You can start with base images from Docker Hub and use images directly from official repositories such as Ubuntu, Debian, Fedora, Arch, and BusyBox.

```
apptainer build new.sif definition_file.def
```

Building images from scratch - definition file

A definition file has a **header** and a **body**. The header determines the base **container** to begin with, and the body is further divided into sections that perform tasks such as software installation, environment setup, and copying files into the container from host system.

Building images from scratch - definition file

The most important sections:

- **`%post`** - Contains commands run during container build, like installing software or configuring the environment.
- **`%environment`** Sets environment variables that will be active inside the container.
- **`%files`** Copies files from the host system into the container.
- **`%runscript`** Specifies the default command to execute when the container runs.
- **%startscript** Defines commands for starting the container in "instance" mode.

Building images from scratch - definition file

```
BootStrap: docker
From: ubuntu:22.04

%files
  code.tar.gz /opt/mycodes/

%post
   apt-get -y update
   apt-get -y install cowsay lolcat

%environment
   export LC_ALL=C
   export PATH=/usr/games:$PATH

%runscript
   date | cowsay | lolcat

%labels
   Author Alice
```

Building images from scratch - sandbox

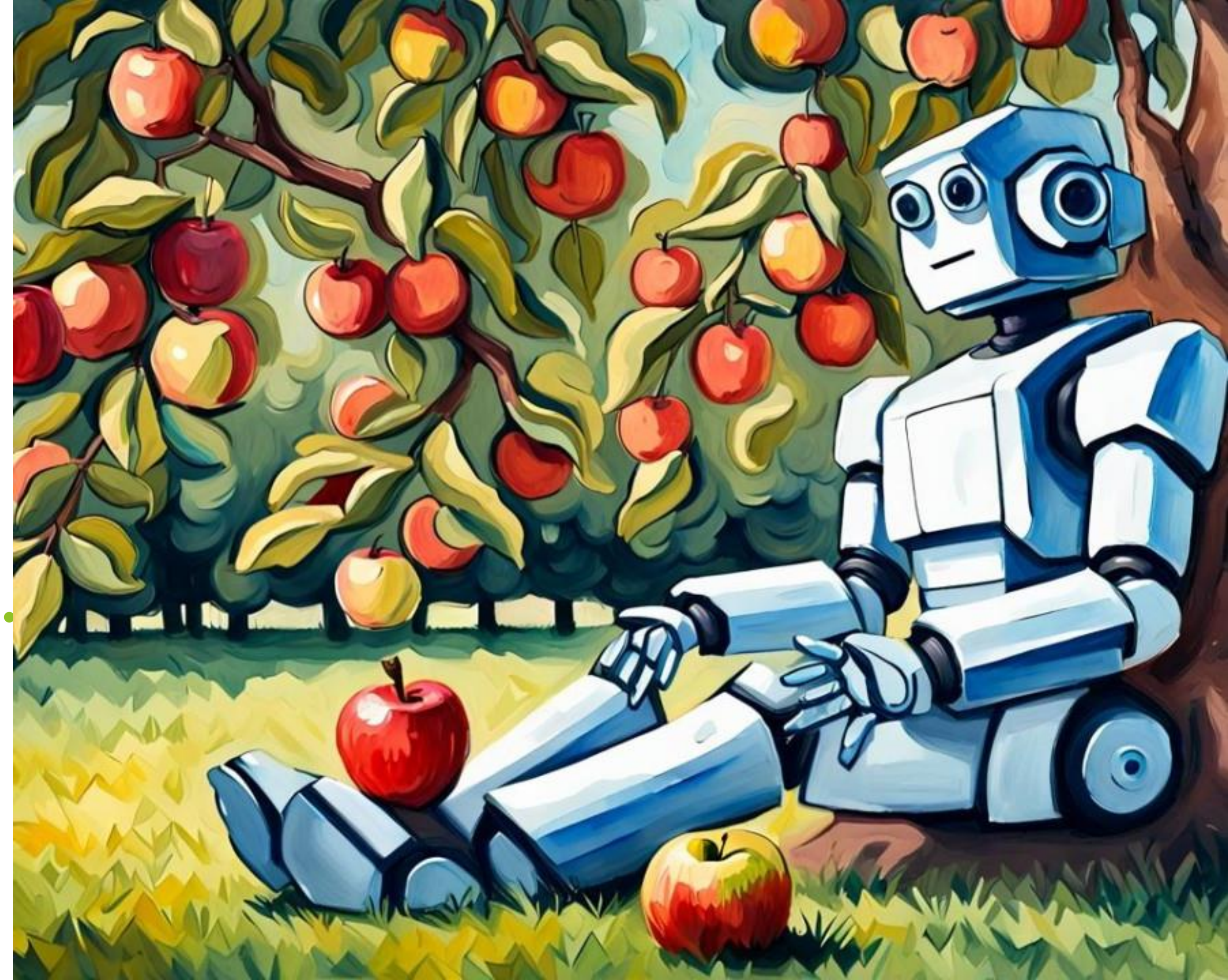
In Apptainer, a **sandbox** refers to a **writable** directory structure that represents the file system of a container. This allows users to **build** and **interact** with containers in a more flexible and iterative manner compared to traditional container images. It alloves:

1. Create a writable environment
2. Interactive development
3. Building from base images
4. Interactive testing

Building images from scratch - sandbox

1. Create a container in a directory (use the build `--sandbox`)
```
apptainer build --sandbox ubuntu/ docker://ubunt
```
2. Modify the container in the directory (`shell --writable`)
```
apptainer shell --writable ubuntu/
install all extras
```
or execute installation commands
```
apptainer exec --writable ubuntu/
pip install numpy
```
3. Converting images from the directory to sif
```
apptainer build new.sif ubuntu/
```

MPI, CUDA,…



Running in SLURM

```
#!/bin/bash -l
#SBATCH -J __JOBNAME__
#SBATCH -N __NUMBER_OF_NODES__
#SBATCH -n __ NUMBER_OF_PROCESORS
#SBATCH -A __GRANT_NAME_GXXX__
#SBATCH -p __PARTITION__

module load common/go/1.13.3  #okeanos
module load common/apptainer/3.4.2 #okeanos


apptainer exec --bind ~/data/:/train/ \
       new.sif python train.py
```

MPI applications

Running MPI applications:

1. load MPI module
   ```
   module load MPI
   ```
2. Run MPI application:
   - the hybrid model:
     ```
     mpirun -n <NUMBER_OF_RANKS> apptainer exec
     <IMAGE> <BINARY>
     ```
   - the bind model:
     ```
     mpirun -n <NUMBER_OF_RANKS> apptainer exec
     --bind <PATH/TO/HOST/MPI/DIRECTORY>:
     <PATH/IN/CONTAINER>
     <IMAGE> <BINARY>
     ```

MPI modes

Hybrid Model

The Hybrid Model involves using both the MPI implementation available on the host system and an MPI implementation within the Apptainer container. This model allows for the integration of MPI processes running outside the container with those inside.

When executing an MPI application, you call an MPI launcher (such as mpirun or mpiexec) from the host. This launcher communicates with the MPI daemon (ORTED) to manage processes. The host's MPI handles process management and communication between nodes, while the containerized application uses its own MPI libraries to perform message passing.

MPI modes

Bind Model

The Bind Model relies solely on the MPI implementation available on the host system, without including any MPI libraries in the Apptainer container. Instead, it binds the host's MPI libraries into the container at runtime.

Similar to the Hybrid Model, you initiate an MPI application using an MPI launcher from the host. However, in this model, Apptainer mounts/binds the host's MPI libraries into the container so that the application can use them directly.

GPU

`--nv` flag allows you to mount the Nvidia CUDA libraries from your host environment into your build environment.

```
apptainer exec --nv new.sif python train.py
```

Multiple GPUs

The `--nv` option will ensure that all nvidia devices on the host are present in the container.

To control which GPUs are used in an Apptainer container that is run with `--nv` you can set `APPTAINERENV_CUDA_VISIBLE_DEVICES` before running the container, or `CUDA_VISIBLE_DEVICES` inside the container. This variable will limit the GPU devices that CUDA programs see.