

Application Containers

Introdução Engenharia Informática

Mário Antunes

October 20, 2025

Universidade de Aveiro

Application Containers & Sandboxing on Linux

A Deep Dive into AppImage, Snap, and Flatpak

The Core Problem: “Linux Dependency Hell”



Traditional Linux apps rely on **shared system libraries** (.so files).

- **The Conflict:**

- App A needs `libXYZ v1.0`
- App B needs `libXYZ v2.0`

- **The Result:**

- Your package manager (apt, dnf) can often only install one version.
- Installing App B breaks App A (or vice-versa).

The Need for Isolation & Portability

- **Portability:** An app packaged with its dependencies will “run anywhere” on any Linux distro, regardless of its system libraries.
- **Stability:** Apps can’t conflict with each other’s dependencies.
- **Security:** If an app is isolated (sandboxed), it can’t read your SSH keys, browser history, or other sensitive data.

How Other OSes Handle This

This isn't just a Linux problem.

- **Windows:** Apps bundle almost *all* their .dll files in their installation folder (e.g., C:\Program Files\App).
 - **Pro:** Prevents conflicts.
 - **Con:** Lots of duplication; inefficient.
- **macOS:** .app “bundles” are just folders that contain the app's binary and all its libraries.
 - **Pro:** Self-contained and portable.
 - **Con:** Also duplicates libraries.

“Natural” Isolation: VMs & Runtimes

Some technologies provide isolation by their very nature.

- **Java Virtual Machine (JVM):**

- The OS runs the java process, not your app directly.
- The JVM runs the Java bytecode in a managed, sandboxed environment.

- **Python Virtual Environments (venv):**

- This is **dependency isolation**, not security sandboxing.
- Creates a local folder (`.venv`) with its own Python interpreter and packages (like `pygame`).
- A `requirements.txt` file lists all dependencies, allowing `pip install -r requirements.txt` to create a reproducible environment, just as we did in our exercise.
- This solves the “App A vs. App B” problem on our local machine but doesn’t stop the app from reading our files.

The Modern Linux Solutions

Three major technologies emerged to solve this for *any* application, aiming to bundle the app *and* its dependencies.

1. AppImage

- **Philosophy:** “One app = one file.” No installation needed.

2. Snap

- **Philosophy:** “A secure, universal package.” Backed by Canonical (Ubuntu).

3. Flatpak

- **Philosophy:** “The future of desktop apps.” Backed by Red Hat & the GNOME community.

Deep Dive: AppImage




- **Isolation: None by default.** It's about portability, not security. The app runs as a normal user process.
 - *(Can be sandboxed by optional, external tools like firejail).*
- **Dependencies: "Bundle Everything."** The app bundles all libraries it needs, assuming only a minimal base system.
- **Host Access: Full User Access.** The app can see and modify anything the user who ran it can.

- **Isolation: Strong Sandbox.** Uses Linux kernel features like **cgroups**, **namespaces**, and **AppArmor** to strictly confine the app.
- **Dependencies: Bundled + Core Snaps.** Apps bundle their specific libraries but also depend on a shared core snap (e.g., **core22**) that provides a base Ubuntu runtime.
- **Host Access: "Interfaces."** Denied by default. The app must declare what it needs (e.g., **network**, **home**, **camera**).




Deep Dive: Flatpak 📁

- **Isolation: Strong Sandbox.** Uses kernel **namespaces** and a tool called **Bubblewrap (bwrap)** to create a private environment for the app.
- **Dependencies: Shared Runtimes.** An app requests a “Runtime” (e.g., `org.gnome.Platform`). This is downloaded *once* and shared by all apps that need it. Very efficient.
- **Host Access: “Portals.”** Denied by default. When an app needs a file, it asks a Portal, which opens a file-picker *outside* the sandbox. The user picks a file, and *only* that file is given to the app.

Comparison: Sandboxing & Dependencies

Feature	AppImage	Snap	Flatpak
Sandboxing	 None (by default)	 Strong (AppArmor)	 Strong (Bubblewrap)
Permissions	Full user access	Interfaces (Declarative)	Portals (Interactive)
Dependency Model	All bundled in file	Bundled + Core snaps	Shared Runtimes

Comparison: Distribution & Backing

Feature	AppImage	Snap	Flatpak
Distribution	Decentralized (any URL)	Centralized (Snap Store)	Decentralized (Repos)
Central Backer	Community	Canonical (Ubuntu)	Red Hat / GNOME
Needs a Daemon?	 No	 Yes (snapd)	 Yes (flatpak-daemon)
Desktop Integration	Optional (appimaged)	Automatic	Automatic

Limitations: The Trade-Offs

- **Disk Space:**
 - **AppImage/Snap:** Bundling can be inefficient. A 10MB app might become a 150MB package.
 - **Flatpak:** Runtimes are large (often 500MB+), but this is a **one-time** download.
- **Startup Time:**
 - **AppImage:** Must mount the compressed file system on every launch (can be slow).
 - **Snap:** Notoriously slow *first launch* as it sets up the sandbox.

Limitations: The “Jail” Problem

- **Security vs. Usability:**

- The sandbox is a “jail.” This is great for security but can be frustrating.
- “Why can’t my app see my home folder?” This is a **feature**, not a bug, but it requires apps to be written to use Portals correctly.

- **Not for Everything:**

- Poorly suited for command-line tools that need deep system integration (e.g., docker, htop, system drivers).

Practical: The AppImage AppDir Structure

An AppImage is just a compressed directory. This directory is called the **AppDir**.

MyGame.AppDir/ (The root folder)

- **AppRun (Required):** The entrypoint script. This is what runs when you double-click the AppImage. It's our job to write this script to set up the environment (like `PYTHONPATH` for Pygame) and launch the main binary.

- **my-game.desktop (Required):** The desktop integration file. It tells the system's app menu:
 - Name=My Game
 - Exec=AppRun (Always AppRun)
 - Icon=my-game (The name of the icon, without extension)
- **my-game.png (Required):** The icon file named in the .desktop file.
- **usr/...:** A standard Linux structure containing your binaries, libraries, and the portable Python interpreter.

Practical: AppImage “Hello World”

Here, we create the *minimal* AppDir structure.

1. Create the directory, script, and metadata:

```
mkdir -p HelloWorld.AppDir
cd HelloWorld.AppDir

# Create the AppRun entrypoint
echo '#!/bin/bash' > AppRun
echo 'echo "Hello from an AppImage!"' >> AppRun
chmod +x AppRun

# Create the desktop file
echo '[Desktop Entry]' > hello.desktop
echo 'Name=Hello' >> hello.desktop
echo 'Exec=AppRun' >> hello.desktop
echo 'Icon=hello' >> hello.desktop
echo 'Type=Application' >> hello.desktop

# Add a dummy icon
touch hello.png
```

Practical: Bundling the AppImage

1. Bundle it!

```
# Go back to parent dir  
cd ..
```

```
# Download appimagetool (only need to do this once)  
wget https://github.com/AppImage/AppImageKit/releases/download/continuous/appimagetool  
chmod +x appimagetool-x86_64.AppImage
```

```
# Run the tool on your directory  
# We must set ARCH for script-based apps  
ARCH=x86_64 ./appimagetool-x86_64.AppImage HelloWorld.AppDir
```

Result: You now have `Hello-x86_64.AppImage`. Run it: `./Hello-x86_64.AppImage`

Practical: The Flatpak Manifest (.yml)

A Flatpak is built from a “manifest” file that acts as a “recipe.”

- `app-id`: The unique name (e.g., `com.example.HelloWorld`).
- `runtime / sdk`: The base system to build upon (e.g., `org.gnome.Platform`). We don't bundle Python; we use the one from the runtime.
- `command`: The executable to run.
- `modules`: The list of “parts” to build. This is where we list our app's code and its dependencies (like `pygame` from PyPI or our game from a `git` URL).

Practical: Flatpak “Hello World”

1. Create the script:

```
# Create a file named hello.sh
echo '#!/bin/sh' > hello.sh
echo 'echo "Hello from a Flatpak Sandbox!"' >> hello.sh
```

2. Create the manifest (com.example.HelloWorld.yml):

```
app-id: com.example.HelloWorld
runtime: org.freedesktop.Platform
runtime-version: '25.08'
sdk: org.freedesktop.Sdk
command: hello.sh
modules:
  - name: hello-module
    buildsystem: simple
    build-commands:
      # Install the script into the sandbox
      - install -Dm755 hello.sh /app/bin/hello.sh
sources:
  - type: file
    path: hello.sh
```

Practical: The flatpak-builder Tool

The flatpak-builder command reads your .yaml manifest and performs the build inside a clean, sandboxed environment.

1. Build and install the app

```
flatpak-builder --user --install --force-clean \
  build-dir com.example.HelloWorld.yaml
```

- **--user**: Installs for the current user (no sudo).
- **--install**: Installs the app as soon as it's built.
- **--force-clean**: Deletes the old build directory for a fresh start.
- **build-dir**: A temporary folder for the build process.

2. Run your new app!

```
flatpak run com.example.HelloWorld
```

Practical: Flatpak Repositories

Flatpak is decentralized, like git. There is no single “store.”

- **What is a Repository?**

- A server (or local folder) that hosts apps, managed by `ostree`.
- You can have multiple “remotes” (repositories) configured.

- **Flathub: The “Main” Repo**

- `flathub.org` is the *de facto* central repository for most desktop apps (Spotify, VS Code, GIMP, Steam).
- `flatpak remote-add --if-not-exists flathub https://flathub.org/repo/flathub.flatpakrepo`

- **How to Publish:**

- To get your app on Flathub, you submit your `.yaml` manifest file to their GitHub repository as a pull request.
- Their build system automatically builds, signs, and publishes your app for you.

Conclusion

- **Isolation** solves “Dependency Hell” and adds **security**.
- **AppImage**: Best for simple **portability**.
 - *Focus*: Manually creating a file structure (AppDir) and an AppRun script.
- **Snap**: Strong in **IoT/Server** and on Ubuntu.
 - *Focus*: Central store, strong security.
- **Flatpak**: The leader in the **desktop** space.
 - *Focus*: Writing a declarative “recipe” (.yml manifest) and letting flatpak-builder and shared runtimes do the heavy lifting.