The Journal of Open Source Software

pythainer: composable and reusable Docker builders and runners for reproducible research

Antonio Paolillo [1]

1 Software Languages Lab, Vrije Universiteit Brussel (VUB), Belgium

DOI: 10.21105/joss.09059

Software

- Review
- Repository
- Archive

Editor: Daniel S. Katz

Reviewers:

- @samfrm
- @akshaymittal143
- @rcannood

Submitted: 16 September 2025
Published: 09 January 2026

Summary

Software experiments today often depend on complex Linux environments that combine several toolchains, devices, and graphical interfaces. Many research projects (Mayoral-Vilches et al., 2022; Millane et al., 2024), for instance, need to compose ROS 2 (Macenski et al., 2022) with CUDA (*CUDA C++ Programming Guide*, 2025), require non-root users, provide GPU and GUI access, and must be reproducible across time and machines. Docker (Docker, Inc., 2013) is a widely adopted substrate for packaging and running such environments, and is commonly used to improve reproducibility in research software (Tani et al., 2020). However, writing and maintaining Dockerfiles and project-specific `docker run` scripts becomes a burden as requirements grow.

`pythainer` raises the level of abstraction while remaining Docker-native. It lets users describe images as small, testable Python *builders* that can be composed (e.g., ROS 2 + CUDA) and executed with reusable *runners* that capture runtime policy (GPU, GUI, users, mounts). `pythainer` renders deterministic Dockerfiles, builds standard images, and centralizes run configuration, hence improving reuse and reducing duplication across repositories.

Statement of need

Plain Dockerfiles are intentionally minimal: they offer sequential shell steps but no first-class functions, loops, or composition. This is adequate for simple images, yet it complicates reuse in research settings where environments must be combined and parameterized. In particular, merging two existing images (e.g., community ROS 2 and NVIDIA CUDA) is not first-class: multi-stage builds help trim artifacts but require intimate knowledge of which files, environment variables, and paths must be copied and preserved. On the runtime side, real projects often need non-root users, persistent volumes, access to GPUs and GUIs (X11/Wayland), and device mappings. These concerns are typically maintained as long shell scripts that are copy-pasted and diverge across projects.

The primary target audience of `pythainer` is anyone who needs to write and maintain multiple Dockerfiles or complex containerized environments that share interchangeable build steps and runtime requirements. This includes, but is not limited to, research groups and labs (e.g., robotics, vision, ML, compilers, systems), instructors who need reliable student environments, and continuous integration (CI) maintainers who prefer deterministic builds and centralized run policy over ad-hoc scripts.

Functionality

`pythainer` is a lightweight Python package and CLI that provides a programmable front-end to Docker. It addresses the above pain points by adding a programmable abstraction for image

construction and a reusable abstraction for execution policy. Builders are Python objects and functions that support ordinary programming constructs (conditionals, loops, parameters) and can be composed with a simple operator. Runners encapsulate repeatable `docker run` policy, so launching a container is a matter of selecting presets rather than rewriting long commands.

Instead of writing raw Dockerfiles and shell scripts, users compose images with builders and control runtime behavior with runners. The library integrates naturally into Python workflows while emitting standard, human-readable Dockerfiles that are built and executed using the Docker engine with reproducible runtime settings. `pythainer` is centered around two core abstractions, builders and runners:

- **Builders (image construction).** A small API exposes common steps (e.g., FROM/RUN/ENV/WORKDIR, package installs). Builders can be composed via an in-place operator to form larger images (e.g., ROS 2 + CUDA). Output rendering is deterministic, which simplifies testing and review.
- **Runners (execution policy).** A runner object assembles `docker run` flags for typical research needs: non-root user mapping, volumes, devices, GPUs, and GUI/X11 forwarding. Presets capture best practices (e.g., mapping the X socket and `DISPLAY`, requesting `--gpus all` with the expected environment variables), reducing duplication across repositories.

`pythainer` is designed around composable building blocks: users can define their own builders or runners and combine them across projects. The library also ships a small set of representative builders and runners for common research needs (e.g., language toolchains, emulation, GPU and GUI support), which can be reused directly or extended in project-specific workflows. This enables reuse that is difficult to achieve with monolithic Dockerfiles.

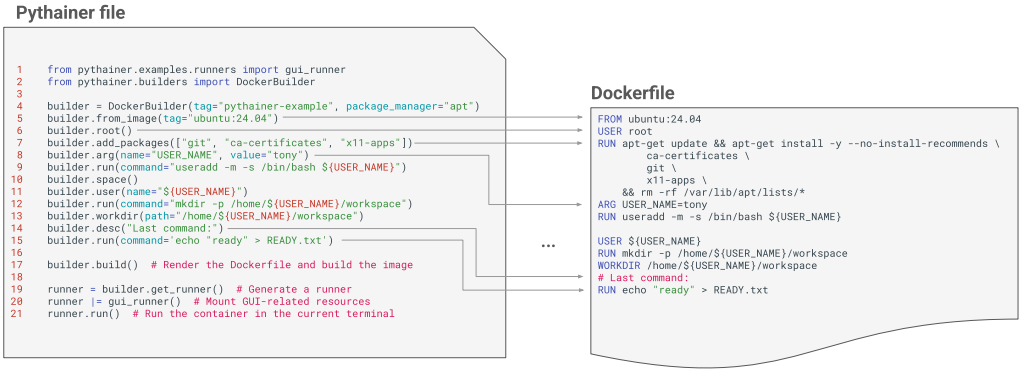

Figure 1: Mapping between a pythainer builder recipe (left) and the resulting Dockerfile (right). Each builder method contributes a deterministic Dockerfile block. Execution policy is defined separately via runners (e.g., GUI/X11 support), which assemble the docker run invocation without modifying the image.

Figure 1 illustrates the core workflow of pythainer. Users specify image construction declaratively in Python using builders, which are rendered into a standard Dockerfile and built with the Docker engine. Execution policy is handled separately via runners, which assemble the required docker run flags before launching the container.

`pythainer` is accompanied by supporting tooling:

- **CLI.** A command-line interface provides two convenience commands: `scaffold` generates a starter Python script (builders + runners) and `run` composes and executes directly for one-offs.
- **Examples and tests.** The package ships small composition recipes (e.g., LLVM/MLIR,

QEMU, Rust) (Chris Lattner et al., 2021; C. Lattner & Adve, 2004; QEMU Project, 2003). Unit tests lock down Dockerfile rendering and CLI behavior; an opt-in integration test builds a tiny image to validate the end-to-end flow. Continuous integration runs tests and linters.

Research applications

We have used `pythainer` to assemble environments for (i) robotics experiments combining ROS 2 with CUDA toolchains (*Imec ITF World 2024 SAFEBOT Demo*, 2024; Shen et al., 2025); (ii) compiler research that requires pinned LLVM toolchains (De Greef et al., 2025); (iii) systems evaluations using QEMU built from source; and (iv) GPU scheduling experiments where deterministic containerized environments are required (Discepoli et al., 2025).

In each case, the same small recipes are reused and composed across projects, which shortens setup time and reduces configuration drift. Because `pythainer` emits human-readable Dockerfiles, the resulting images remain transparent and easy to audit, and the approach integrates well with existing Docker-centric CI.

Related work

`pythainer` complements the Docker ecosystem by adding a programmable composition model on top of Dockerfiles. Unlike Docker Compose or the Docker SDK for Python, which focus on orchestrating multi-service deployments or driving the daemon (Docker, Inc., 2014a, 2014b), `pythainer` focuses on single-image construction and single-container execution policy. This makes it especially suited for research projects where the goal is to provide a single reproducible environment for experiments rather than a full service-oriented stack.

Compared with editor-centric templates such as VS Code devcontainers (Dev Containers Spec, 2022) or domain-specific generators such as repo2docker (Project Jupyter, 2017), `pythainer` treats environment recipes as code with tests and deterministic rendering. Functional package managers such as Nix and Guix offer deep system-level reproducibility but require adopting a different stack (Courtès, 2013; Dolstra et al., 2004); `pythainer` stays Docker-native for easier adoption in labs and CI. Pragmatically, many third-party packages (e.g., CUDA and ROS 2) are primarily supported on Ubuntu, so staying Docker-native with Ubuntu-based images eases reproduction without changing the base distribution.

Projects such as Caliban (Ritchie et al., 2020) and x11docker (Viereck, 2019) address related pain points in research containerization. Caliban streamlines packaging and running ML experiments across local and cloud environments, while x11docker provides secure and convenient ways to run GUI applications inside Docker. However, neither of these works addresses general-purpose composition of images and runtime policy. In contrast, `pythainer` focuses on composable image construction and reusable execution policy while remaining domain-agnostic and Docker-native.

Acknowledgements

We thank contributors for feedback and patches that improved early designs and examples, including Attilio Discepoli, Yuwen Shen, Aaron Bogaert, Samuel Beesoon, Robbe De Greef, and Esteban Aguililla Klein.

References

Courtès, L. (2013). *Functional package management with Guix*. https://arxiv.org/abs/1305.4584

CUDA C++ programming guide. (2025). [Computer software]. NVIDIA Corporation; https://docs.nvidia.com/cuda/cuda-c-programming-guide/, accessed 2025-09-15.

De Greef, R., Discepoli, A., Aguililla Klein, E., Engels, T., Hasselmann, K., & Paolillo, A. (2025). Towards macro-aware C-to-Rust transpilation (WIP). *Proceedings of the 26th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 57–61. https://doi.org/10.1145/3735452.3735535

Dev Containers Spec. (2022). *Development containers specification*. https://containers.dev accessed 2025-09-15.

Discepoli, A., Huygen, M. L., & Paolillo, A. (2025). Compute kernels as moldable tasks: Towards real-time gang scheduling in GPUs. *Proceedings of the 19th Workshop on Operating Systems Platforms for Embedded Real-time Applications (OSPERT 2025)*, 29–33.

Docker, Inc. (2013). *Docker: Accelerated container application development*. https://www.docker.com/ accessed 2025-09-15.

Docker, Inc. (2014a). *Docker compose*. https://docs.docker.com/compose/ accessed 2025-09-15.

Docker, Inc. (2014b). *Docker SDK for python*. https://docker-py.readthedocs.io/ accessed 2025-09-15.

Dolstra, E., Jonge, M. de, & Visser, E. (2004). Nix: A safe and policy-free system for software deployment. *Proceedings of the 18th USENIX Conference on System Administration*, 79–92.

Imec ITF world 2024 SAFEBOT demo. (2024). https://www.youtube.com/watch?v=F7m5_kQ_mRQ accessed 2025-09-15.

Lattner, C., & Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 75–86. https://doi.org/10.1109/CGO.2004.1281665

Lattner, Chris, Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., & Zinenko, O. (2021). MLIR: Scaling compiler infrastructure for domain specific computation. *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

Macenski, S., Foote, T., Gerkey, B., Lalancette, C., & Woodall, W. (2022). Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, *7*(66), eabm6074. https://doi.org/10.1126/scirobotics.abm6074

Mayoral-Vilches, V., Neuman, S. M., Plancher, B., & Reddi, V. J. (2022). RobotCore: An open architecture for hardware acceleration in ROS 2. *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 9692–9699. https://doi.org/10.1109/IROS47612.2022.9982082

Millane, A., Oleynikova, H., Wirbel, E., Steiner, R., Ramasamy, V., Tingdahl, D., & Siegwart, R. (2024). Nvblox: GPU-accelerated incremental signed distance field mapping. *2024 IEEE International Conference on Robotics and Automation (ICRA)*, 2698–2705. https://doi.org/10.1109/ICRA57147.2024.10611532

Project Jupyter. (2017). *repo2docker: Turn repositories into Jupyter-enabled Docker images*. https://repo2docker.readthedocs.io/ accessed 2025-09-15.

QEMU Project. (2003). *QEMU: A generic and open source machine emulator and virtualizer*. https://www.qemu.org/ accessed 2025-09-15.

Ritchie, S., Slone, A., & Ramasesh, V. (2020). Caliban: Docker-based job manager for

reproducible workflows. *Journal of Open Source Software*, *5*(53), 2403. https://doi.org/10.21105/joss.02403

Shen, Y., Mynsbrugge, J. V., Roshandel, N., Bouchez, R., FirouziPouyaei, H., Scholz, C., Cao, H., Vanderborght, B., Joosen, W., & Paolillo, A. (2025). SentryRT-1: A case study in evaluating real-time linux for safety-critical robotic perception. *Proceedings of the 19th Workshop on Operating Systems Platforms for Embedded Real-time Applications (OSPERT 2025)*, 35–41.

Tani, J., Daniele, A. F., Bernasconi, G., Camus, A., Petrov, A., Courchesne, A., Mehta, B., Suri, R., Zaluska, T., Walter, M. R., Frazzoli, E., Paull, L., & Censi, A. (2020). Integrated benchmarking and design for reproducible and accessible evaluation of robotic agents. *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 6229–6236. https://doi.org/10.1109/IROS45743.2020.9341677

Viereck, M. (2019). x11docker: Run GUI applications in Docker containers. *Journal of Open Source Software*, *4*(37), 1349. https://doi.org/10.21105/joss.01349