




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.xxxxxx/draft](https://doi.org/10.xxxxxx/draft)

Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Editor: [Daniel S. Katz](#) 

Reviewers:

- [@samfrm](#)
- [@akshaymittal143](#)
- [@rcannood](#)

Submitted: 16 September 2025

Published: unpublished

License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC BY 4.0](#)).

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 widely 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—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.

pythainer addresses these pain points by adding a programmable front-end 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. The target audience includes research groups and labs (robotics, vision, ML systems, 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. 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 but remains Docker-native: it renders deterministic Dockerfiles, builds them with the Docker engine, and executes containers with reproducible runtime settings. pythainer provides:

- **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. The tool remains Docker-native: it emits standard Dockerfiles and uses the Docker engine to build images ([Docker, Inc., 2013](#)).
- **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.
- **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.

pythainer is designed to be naturally extensible: users can easily define their own builders or runners, or build on top of those provided in the library.

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.

83 Compared with editor-centric templates such as VS Code devcontainers (Dev Containers Spec,
84 2022) or domain-specific generators such as repo2docker (Project Jupyter, 2017), pythainner
85 treats environment recipes as code with tests and deterministic rendering. Functional package
86 managers such as Nix and Guix offer deep system-level reproducibility but require adopting a
87 different stack (Courtès, 2013; Dolstra et al., 2004); pythainner stays Docker-native for easier
88 adoption in labs and CI. Pragmatically, many third-party packages (e.g., CUDA and ROS 2)
89 are primarily supported on Ubuntu, so staying Docker-native with Ubuntu-based images eases
90 reproduction without changing the base distribution.

91 Projects such as Caliban (Ritchie et al., 2020) and x11docker (Viereck, 2019) address re-
92 lated pain points in research containerization. Caliban streamlines packaging and running
93 ML experiments across local and cloud environments, while x11docker provides secure and
94 convenient ways to run GUI applications inside Docker. However, neither of these works
95 addresses general-purpose composition of images and runtime policy. In contrast, pythainner
96 focuses on composable image construction and reusable execution policy—including mounting
97 what is required to access the GUI server—while remaining domain-agnostic and Docker-native.

98 Acknowledgements

99 We thank contributors for feedback and patches that improved early designs and examples,
100 including Attilio Discepoli, Yuwen Shen, Aaron Bogaert, and Samuel Beeson.

101 References

- 102 Courtès, L. (2013). *Functional Package Management with Guix*. <https://arxiv.org/abs/1305.4584>
103
- 104 *CUDA C++ Programming Guide*. (2025). [Computer software]. NVIDIA Corporation;
105 <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>, accessed 2025-09-15.
- 106 De Greef, R., Discepoli, A., Aguililla Klein, E., Engels, T., Hasselmann, K., & Paolillo, A.
107 (2025). Towards Macro-Aware C-to-Rust Transpilation (WIP). *Proceedings of the 26th*
108 *ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for*
109 *Embedded Systems*, 57–61. <https://doi.org/10.1145/3735452.3735535>
- 110 Dev Containers Spec. (2022). *Development Containers Specification*. <https://containers.dev>
111 accessed 2025-09-15.
- 112 Discepoli, A., Huygen, M. L., & Paolillo, A. (2025). Compute Kernels as Moldable Tasks:
113 Towards Real-Time Gang Scheduling in GPUs. *Proceedings of the 19th Workshop on*
114 *Operating Systems Platforms for Embedded Real-time Applications (OSPRT 2025)*,
115 29–33.
- 116 Docker, Inc. (2013). *Docker: Accelerated Container Application Development*. <https://www.docker.com/>
117 accessed 2025-09-15.
- 118 Docker, Inc. (2014a). *Docker Compose*. <https://docs.docker.com/compose/> accessed 2025-
119 09-15.
- 120 Docker, Inc. (2014b). *Docker SDK for Python*. <https://docker-py.readthedocs.io/> accessed
121 2025-09-15.
- 122 Dolstra, E., Jonge, M. de, & Visser, E. (2004). Nix: A Safe and Policy-Free System for Software
123 Deployment. *Proceedings of the 18th USENIX Conference on System Administration*,
124 79–92.
- 125 imec ITF World 2024 SAFEBOT demo. (2024). https://www.youtube.com/watch?v=F7m5_kQ_mRQ
126 accessed 2025-09-15.

- 127 Lattner, C., & Adve, V. (2004). LLVM: a compilation framework for lifelong program analysis
128 & transformation. *International Symposium on Code Generation and Optimization, 2004.*
129 *CGO 2004.*, 75–86. <https://doi.org/10.1109/CGO.2004.1281665>
- 130 Lattner, Chris, Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R.,
131 Shpeisman, T., Vasilache, N., & Zinenko, O. (2021). MLIR: Scaling Compiler Infrastructure
132 for Domain Specific Computation. *2021 IEEE/ACM International Symposium on Code*
133 *Generation and Optimization (CGO)*, 2–14. [https://doi.org/10.1109/CGO51591.2021.](https://doi.org/10.1109/CGO51591.2021.9370308)
134 [9370308](https://doi.org/10.1109/CGO51591.2021.9370308)
- 135 Macenski, S., Foote, T., Gerkey, B., Lalancette, C., & Woodall, W. (2022). Robot Operating
136 System 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66), eabm6074.
137 <https://doi.org/10.1126/scirobotics.abm6074>
- 138 Mayoral-Vilches, V., Neuman, S. M., Plancher, B., & Reddi, V. J. (2022). RobotCore: An
139 Open Architecture for Hardware Acceleration in ROS 2. *2022 IEEE/RSJ International*
140 *Conference on Intelligent Robots and Systems (IROS)*, 9692–9699. [https://doi.org/10.](https://doi.org/10.1109/IROS47612.2022.9982082)
141 [1109/IROS47612.2022.9982082](https://doi.org/10.1109/IROS47612.2022.9982082)
- 142 Millane, A., Oleynikova, H., Wirbel, E., Steiner, R., Ramasamy, V., Tingdahl, D., & Siegwart,
143 R. (2024). nvblox: GPU-Accelerated Incremental Signed Distance Field Mapping. *2024*
144 *IEEE International Conference on Robotics and Automation (ICRA)*, 2698–2705. [https:](https://doi.org/10.1109/ICRA57147.2024.10611532)
145 [//doi.org/10.1109/ICRA57147.2024.10611532](https://doi.org/10.1109/ICRA57147.2024.10611532)
- 146 Project Jupyter. (2017). *repo2docker: Turn repositories into Jupyter-enabled Docker images.*
147 <https://repo2docker.readthedocs.io/> accessed 2025-09-15.
- 148 QEMU Project. (2003). *QEMU: A generic and open source machine emulator and virtualizer.*
149 <https://www.qemu.org/> accessed 2025-09-15.
- 150 Ritchie, S., Slone, A., & Ramasesh, V. (2020). Caliban: Docker-based job manager for
151 reproducible workflows. *Journal of Open Source Software*, 5(53), 2403. [https://doi.org/](https://doi.org/10.21105/joss.02403)
152 [10.21105/joss.02403](https://doi.org/10.21105/joss.02403)
- 153 Shen, Y., Mynsbrugge, J. V., Roshandel, N., Bouchez, R., FirouziPouyaei, H., Scholz, C., Cao,
154 H., Vanderborcht, B., Joosen, W., & Paolillo, A. (2025). SentryRT-1: A Case Study in
155 Evaluating Real-Time Linux for Safety-Critical Robotic Perception. *Proceedings of the*
156 *19th Workshop on Operating Systems Platforms for Embedded Real-time Applications*
157 *(OSPERT 2025)*, 35–41.
- 158 Tani, J., Daniele, A. F., Bernasconi, G., Camus, A., Petrov, A., Courchesne, A., Mehta,
159 B., Suri, R., Zaluska, T., Walter, M. R., Frazzoli, E., Paull, L., & Censi, A. (2020).
160 Integrated Benchmarking and Design for Reproducible and Accessible Evaluation of Robotic
161 Agents. *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems*
162 *(IROS)*, 6229–6236. <https://doi.org/10.1109/IROS45743.2020.9341677>
- 163 Viereck, M. (2019). x11docker: Run GUI applications in Docker containers. *Journal of Open*
164 *Source Software*, 4(37), 1349. <https://doi.org/10.21105/joss.01349>