



Kworkflow

A Linux kernel Developer Automation Workflow System

David Tadokoro
Free Software Competence Center
Institute of Mathematics and Statistics
University of São Paulo
São Paulo, SP, Brazil
davidbtadokoro@usp.br

Rodrigo Siqueira
Free Software Competence Center
Institute of Mathematics and Statistics
University of São Paulo
São Paulo, SP, Brazil
siqueirajordao@riseup.net

Paulo Meirelles
Free Software Competence Center
Institute of Mathematics and Statistics
University of São Paulo
São Paulo, SP, Brazil
paulormm@ime.usp.br

ABSTRACT

The Linux kernel is a central project to the Free/Libre and Open Source Software (FLOSS) ecosystem. Developers who interact with the kernel source code routinely face a variety of repetitive and error-prone tasks, including setting up testing environments, managing kernel configuration files, compiling with various toolchains, deploying to varied targets, and correctly submitting patches. These tasks, though essential, often require handcrafted scripts that become fragile, difficult to maintain, and disconnected from the core development goals. Kworkflow (kw) is a Developer Automation Workflow System (DAWS) that aims to provide high-quality tools that mitigate known bottlenecks experienced by kernel developers in their daily workflows. It provides a unified Command-Line Interface (CLI) that automates and streamlines many tasks, allowing practitioners to focus more on developing code and less on setup overhead and details. The hub-like design of the project reflects its philosophy of incorporating existing tools to avoid duplication of efforts and wasted resources from the community, a by-product of the many *ad-hoc* scripts developers commonly create. The kw tool also offers a data collection infrastructure, which makes it an excellent platform for further scientific research in the Linux kernel development model. In this sense, kw serves as a robust tool for real kernel developers and an opportunity for academic research work.

Demo video: Software Heritage [permalink](#) [19].

KEYWORDS

Linux, Free Software, Linux kernel development model, Free Software development model, software engineering, kernel workflows.

1 Introduction

As a software solution, the Linux kernel is highly customizable, being on the cutting edge of diverse fields, including Internet infrastructure, embedded systems, cloud computing, and machine learning, making it critical for solving real-world problems of society at large.

When looking at the Linux kernel community from its data points, it is safe to conclude that this project is one of the most thriving Free/Libre and Open Source Software (FLOSS) projects by many metrics, such as the volume of volunteer developers worldwide or the number of companies that invest in the project. The Linux community has brought many people worldwide from different cultures, beliefs, and time zones together to improve the project.

Even though the project has been successful for over thirty years, it is still evolving from the software engineering perspective. For

example, due to its size, the kernel requires developers to follow multiple processes and practices that compose many workflows, some of which have associated bottlenecks that slow down development; in this sense, newcomers need some time to get used to the basic workflows, but even more experienced developers forget some aspects of the development model and make mistakes. This hardship is natural and, in some way, expected when considering the vastness of the Linux kernel.

To mitigate workflow bottlenecks of large FLOSS projects, communities create tools to streamline and automate their tasks. In the case of Linux, tools like `get_maintainers.pl`, `checkpatch.pl`, and `git-send-email` help in the tasks of finding the correct recipients to send contributions, checking for code style violations, and distributing patches (code changes) via email, respectively. It is worth noting that configuring these tools is often a complex and time-consuming chore by itself.

The main point is that multiple tools maintained inside and outside the Linux project support the many kernel workflows. Beyond more consolidated and widely adopted tools, for instance, the ones maintained inside the project codebase in the `scripts/` directory, it is commonplace for practitioners to develop *ad-hoc* scripts highly coupled with their development contexts as solutions to gaps in the workflows not yet covered by existing tools or to configuration overheads. As the primary goal of Linux developers is not to develop and maintain supporting tools, these local solutions (although helpful) duplicate community efforts and resources are spread thin, which could be spent on fostering robust, high-quality collaborative tools.

In other words, although many solutions extensively cover parts of the kernel workflows, they are not unified, sometimes involve considerable effort to configure, and are often unknown outside specific development contexts. All Linux developers (contributors or maintainers, newcomers or veterans) would benefit from a project that robustly integrates these solutions and implements novel ones, presenting a hub-like interface that comprehensively abstracts the kernel workflows.

With all those ideas in mind, and considering that the described tools are invaluable to maintaining a large-scale project like the Linux kernel, to the point that they are crucial for its long-term sustainability [5], this paper presents a set of tools in a unified interface called Kworkflow (kw), that streamlines and automates the many kernel workflows while enforcing the project rules.

2 Background

The FLOSS dynamics are incredibly diverse, and each project, including the Linux kernel, has unique complexities. This section

unravels the intricate concepts of kernel development and provides a solid foundation for the ideas presented in this paper.

Linux is an operating system kernel that runs on multiple hardware configurations and has an extensive domain scope. To give the development model scalability, the project is broken down into sub-projects called subsystems, each with its particular focus, community, rules, and (generally) a dedicated Git repository (called a *kernel tree*). Even inside the subsystems, more specific development contexts can recursively partition the subsystem into smaller sub-projects, although the recursion usually does not go deep [4].

This dynamic forms a hierarchy in the shape of an umbrella, where maintainers of a given kernel tree receive patches from contributors and, upon approval, send them to the upper levels of the hierarchy: either the *mainline*, which is the tree at the top of the hierarchy representing the top-level Linux project, or a mid-level tree. This structure is reminiscent of a FLOSS equivalent to a city divided into neighborhoods [21]. The hierarchy is held together as a *web of trust* [3], where upper levels trust that lower levels will send correct contributions in the project standard, avoiding duplication of code-reviewing efforts, and allowing for highly specialized personnel in all parts of the project.

For instance, the *Direct Rendering Management* (DRM) subsystem encapsulates all device drivers associated with GPUs. In DRM, multiple vendors maintain their drivers in their trees. In this scenario, a given vendor specialized in its respective drivers will have a dedicated kernel tree that accepts contributions and feeds them to DRM. In turn, DRM will integrate and stabilize all contributions from all its sources and feed them to the mainline.

The Linux development model is much more complex, involving strict release cycles, *Continuous Integration* (CI) workflows, stable releases, and more. Nevertheless, the previous description gives us the necessary information to understand the overarching project dynamics.

2.1 Fundamental tasks of Linux developers

Linux kernel workflows are composed of fundamental tasks, with some being more technical than others. One of the most common tasks that a kernel developer has to deal with daily is compiling the kernel from the source code, which is called *building* the kernel. Unlike other projects that need compilation, this is not trivial in the Linux context. Linux is widespread and runs on virtually any type of commercial processor; x86, PowerPC, ARM, MIPS, and RISC-V are a few of the architectures supported. Other details deepen the complexity of the build task. For instance, processors can support 16-, 32-, and 64-bit architectures; there are different compilation toolchains like GCC and Clang, and some cases require cross-compilation techniques. To top it all, some developers need to compile the same source code to different targets, resulting in different environments that are hard to manage. Noteworthy is the fact that compiling a kernel is a CPU-bound task that can take minutes to hours, even when allocating all CPU cores of a machine with considerable computational power.

Another fundamental task is installing the necessary artifacts from a kernel build into a target machine and making the custom kernel bootable for validation, also called *deploying* a kernel. The target device can be many things, such as a virtual machine running

in the developer's host, another machine connected via network or serial cable, or even the host machine where the developer built the kernel. To deploy a kernel, the developer must consider these details and deal with bootloader configuration, distro-specific behavior, and other minor details, like the precise placement of artifacts.

To manage all the possible build configurations, Linux uses a file named `.config` that, essentially, describes which features and modules to compile and how. Producing the correct `.config` for a given purpose relies on many details, just like deploying a kernel, and developers usually have a set of `.config` files that they need to manage manually. This file is precious for developers since it can (among other things) be optimized for specific target machines, reducing compilation time.

Finally, an important task is to send a patch to the maintainers/reviewers and mailing lists, which involves more than resolving the recipients, such as ensuring the message follows rigid formatting, has a standardized preamble in the subject, and follows other obligatory practices.

Linux developers have to be proficient in interacting with their computer systems through the *Command-Line Interface* (CLI), commonly using an OS shell compatible with *Bourne-Again Shell* (Bash). Not only are some shell scripts part of the Linux codebase, but a significant portion of a Linux developer's work, besides writing code, involves issuing commands in a terminal. Some of them, like *GNU Make* commands, demand many options and/or arguments in a specific order, which become error-prone and inefficient with repetition, even for veteran developers.

2.2 Tools that support Linux development

Several tools have been created in the Linux context to mitigate workflow bottlenecks, sometimes gaining so much traction that they result in broader public adoption. A prime example of a now project-agnostic tool initially devised in the Linux context by its creator, Linus Torvalds, is `git` [20]¹. We can split the tools supporting kernel developers into two categories: officially supported and community-maintained. The former reside inside kernel trees (usually in the `scripts/` directory) and have numerous people caring for them. The latter set comprises any other tool not officially supported, ranging from personal-use shell scripts that are never publicly published to projects diligently maintained by developers (voluntarily or sponsored) that have considerable adoption by the community.

Regarding officially supported tools, we can highlight the script `get_maintainers.pl` [11], which matches the current changes in the patch against the target kernel tree and, based on that, lets the developer know which mailing lists and developers should become recipients. A tool like that improves the contributor's chance of getting their code accepted by the maintainer. Another example is `checkpatch.pl` [9], which helps developers evaluate whether their changes violate code style rules described in the documentation and avoid wasting review cycles on basic things. In other words, `checkpatch.pl` helps developers comply with the code style rules and optimizes the review cycle.

¹All software cited in this paper includes a permalink to its archive on *Software Heritage* (SWH), a platform for collecting, preserving, and sharing source code with precise references down to individual file fragments.

As mentioned before, git can also be considered a Linux tool, despite its pervasiveness in any project, as it is paramount for most of the Linux development model. The `git-send-email` command integrates kernel trees into the project email-based communication.

Moving toward community-maintained projects, the `b4` [12] tool helps maintainers work with patches available in the <https://lore.kernel.org> public-inbox, an archive of most Linux mailing lists that represents an on-demand approach to consuming the flow of messages without needing to subscribe to lists actively.

To name a few other examples that are not necessarily exclusive to the Linux context: `QEMU` [2], `virtme-ng` [1], the *DRM Maintainer Tools* [7], and the *LLVM Project Tools* [10]. Regardless, most Linux tools are *ad-hoc* scripts created by developers to support their specific variations of the kernel workflows. Usually, developers maintain this type of script in some Git forge or, more often than not, only locally; many companies also develop some of these tools and maintain them only for internal use. This fragmentation leads to many duplicated efforts and wasted opportunities to scale the productivity of the Linux community.

2.3 Patchset development and reviewing workflows

Two core and overarching kernel workflows are the *patchset development workflow* (Figure 1) and the *patchset reviewing workflow* (Figure 2), which are more related to the work of contributors and maintainers, respectively. A *patchset* is a set of patches related by an overarching context that is submitted together and represents a contribution.

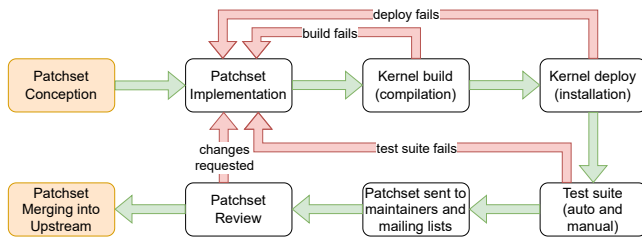


Figure 1: Concept map of the patchset development workflow.

3 Kworkflow

Kworkflow [13] (kw) is a *Developer Automation Workflow System* (DAWS) that targets Linux kernel developers. kw is a FLOSS project licensed under the *GNU General Public License version 2 (GPLv2)*.

Any developer working with the Linux source code can benefit from using kw. For example, in the GNU/Linux distribution (*distro*) layer, developers build on official releases (*upstream*) and maintain their kernel trees (*downstream*) with specific modifications, occasionally contributing back to upstream. Regardless of their involvement in Linux development, these developers must configure, build, and deploy the kernel, making kw a compelling tool. kw streamlines and automates kernel workflows through a unified CLI, offering diverse commands to manage `.config` files, build and deploy kernels, handle virtual and network-accessible machines, send

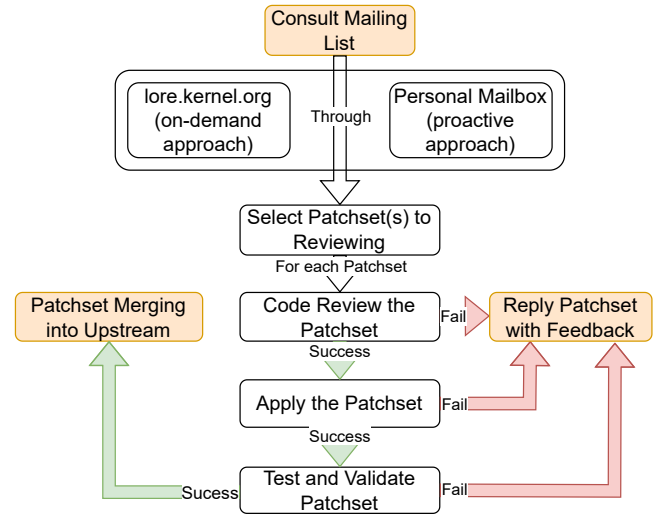


Figure 2: Concept map of the patchset reviewing workflow.

patches per Linux guidelines, and more. Implemented in Bash, kw uses *GNU coreutils* and other CLI tools to provide sensible defaults and a *plug-n-play* experience without limiting customization.

3.1 Installation

Currently, the best way to install kw is by cloning the project locally with git and running the `setup.sh` script [18] with the `-install` option. After running the script, the user will be prompted for superuser credentials to install the necessary dependencies using the appropriate package manager (`apt`, `pacman`, `dnf`, etc.); then the script installs kw files in the user's system. Listing 1 illustrates the mentioned steps.

Listing 1: Installing kw via the `setup.sh` script.

```
$ git clone https://github.com/kworkflow/kworkflow
$ cd kworkflow
$ ./setup.sh --install
```

3.2 Features

Once installed, kw offers a significant number of commands (features) that can be invoked as described in Listing 2. Features of kw can be categorized depending on their purpose, and Table 1 lists all features with their categories and descriptions.

Listing 2: kw commands invocation.

```
$ kw <command> <options/arguments>
```

In the remainder of this section, we will deep dive into some of the kw features, their use cases, and highlight interesting aspects of each. Nevertheless, **keep in mind** that kw does a lot more than what is showcased next.

3.2.1 kw build. The `build` command, along with `deploy`, is one of the flagship features of kw and probably one of the most elaborate ones. As mentioned in Section 2.1, building a kernel is not an easy task, even when you have a `.config` ready. *GNU make* (the make

Table 1: kw commands

Command	Category	Description
build	kernel build/deploy	Build kernel and modules
deploy	kernel build/deploy	Deploy kernel and modules
kernel-config-manager	kernel build/deploy	Manage .config files
env	kernel build/deploy	Manage different environments for same kernel tree
bd	kernel build/deploy	Build and Deploy kernel and modules
send-patch	patch submission	Send patches via email
maintainers	patch submission	get_maintainers.pl wrapper
codestyle	patch submission	checkpatch.pl wrapper
remote	target machine	Manage machines in the network
vm	target machine	QEMU wrapper
ssh	target machine	ssh wrapper
device	target machine	Show hardware information
debug	code inspection	Linux debug utilities
explore	code inspection	Explore string patterns
diff	code inspection	Diff files
init	kw management	Initialize kw kernel tree
config	kw management	Set kw configs
self-update	kw management	Self-update mechanism
backup	kw management	Save and restore kw data
clear-cache	kw management	Clear kw cache
patch-hub	misc	TUI for patches from lore.kernel.org
drm	misc	DRM specific utilities
pomodoro	misc	Pomodoro technique
report	misc	Show usage statistics

utility) immensely helps in this task; nevertheless, one would still have to take into account the target architecture, cross-compilation (if needed), the number of CPU cores allocated, which KCFLAGS (short for *kernel compiler flags*) to use, and so on.

Listing 3: Example of kernel build make command.

```
$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu-
-j8 'KCFLAGS=-DNICE_MACRO'
```

Listing 4: kw build use case.

```
$ kw build
```

For instance, say the user wishes to build a kernel from an x86_64 host to an ARM64 target machine using all his CPU cores (eight cores available) while defining a macro called NICE_MACRO. Assuming that the necessary .config is in place, **Listing 3** shows a make command for this scenario. In comparison, after a one-time simple setting of the aforementioned information using the kw command

config (as it will be described), the user only needs to issue the command in **Listing 4**. The build command offers many options that go beyond simply automating the equivalent make commands, e.g., -info, which displays information about the current compilation with kernel name and number of modules to be compiled, -llvm, which enables use of the LLVM toolchain, and -from-sha, which builds every commit version of the kernel from a git reference passed as an argument.

3.2.2 kw deploy. Although much less demanding regarding resources than building, deploying a kernel can be more complicated and detail-filled than compiling. Simply put, the task involves copying the necessary compiled artifacts to the correct place with the correct name in the target machine filesystem, defining a temporary filesystem needed for the kernel to boot the system, and updating the bootloader. Apart from these being non-trivial steps, they can also mutate quite a lot depending on the target machine CPU architecture, Linux distro, and many other factors. The deploy command absorbs most of these complexities in its code, so users do not need to configure most of the steps, with the feature detecting information about the target machine and resolving the details. Unlike the pair of **Listing 3** and **Listing 4**, an equivalent series of commands that would represent a single kw deploy call would involve many commands (in the range of dozens) that are extensive, so presenting a comparison here would only produce visual pollution. This speaks in favor of how well deploy not only automates but streamlines the task of installing kernels.

Like with build, deploy has options that offer other enhancements, such as: -list, to list installed kernels in a target machine; -uninstall, to uninstall kernels and related artifacts; and -create-package and -from-package, to generate a compressed file that can be transported offline (in a USB device, for instance) and be used to deploy the kernel on another machine, which only has to have kw installed.

3.2.3 bd. Because deploying a kernel often comes right after building it, kw offers the bd command (from *build and deploy*) to chain the two previously mentioned features, optimizing even more the task of booting a kernel built from the source code.

3.2.4 kernel-config-manager. Automating and streamlining the generation of a .config file that is the perfect fit for a developer use case is an open problem, with some attempts to solve this in the context of Linux CI [22]. As such, kw still does not cover the generation part, but offers a very comprehensive feature, kernel-config-manager, for users to easily keep records of a great variety of .config files. The feature works differently depending on the subcommand used: -save saves the current .config with a name and an optional description; -get restores a saved .config with the name passed as an argument; and -remove removes the entry related to the name passed as an argument.

Listing 5: kw kernel-config-manager use case.

```
$ kw k --save foo --description 'bar'
$ kw k --get foo
$ kw k --remove foo
```

Listing 5 exemplifies the usage of these three subcommands (note that we use the short name k of the feature for readability).

Figure 3 is a screen capture that showcases another subcommand, `-list`, which lists all `.config` files managed by the feature.

```

) kw kernel-config-manager --list
List of .config files managed by kw
-----
Name                Description                                Last updated
-----
arm64-mainline-dev   Mainline-friendly config for arm64 testing    2025-03-09 17:30:05
arm64-perf-tuned     Tuned for benchmarking on ARM server cores    2025-03-09 17:49:38
legacy-x86-debug     Old x86 config with full debug symbols        2025-03-09 18:19:44
modern-x86-minimal   Lean x86 config for fast build testing        2025-03-09 18:21:02
arm64-modular-dev    Modular config for out-of-tree module dev     2025-03-19 07:48:16
debian-12-amd64-staging Staging branch builds for Debian 12 VM (amd64) 2025-03-26 10:00:46
debian-12-amd64-production Stable config for Debian 12 VM deployment    2025-03-26 10:57:19

```

Figure 3: kw kernel-config-manager -list example.

3.2.5 env. The result of building the kernel, i.e., the object files and related artifacts, by default, resides in the same kernel tree used for the source code of the compilation. However, some developers need to build kernels for different targets from the same kernel tree, which becomes unmanageable as the number of targets rises, due to the need for multiple `.config` files, build configurations, and more.

Listing 6: kw env use case.

```

$ kw env --create F00
$ kw env --use F00
$ kw env --destroy F00

```

To solve this problem, the `env` command allows users to create different environments that isolate development contexts using a single kernel tree. The corresponding files are stored inside a cache directory managed by kw. With the use of the subcommands `-create`, `-use`, and `-destroy`, users can manage their environments (**Listing 6**), and with `-list` consult their environments (**Figure 4**).

```

) kw env --list
Current env:
-> ARM_64: /home/davidbtadokoro/.cache/kw/envs/ARM_64

Other kw environments:
* ARM_32: /home/davidbtadokoro/.cache/kw/envs/ARM_32
* ARM_64: /home/davidbtadokoro/.cache/kw/envs/ARM_64

```

Figure 4: kw env -list example.

3.2.6 send-patch. Converting a commit or set of commits into patches and sending them through email is covered mainly by the `git-send-email` tool. However, it can be considered raw because it requires a lot of configuration, verbose commands, and, mainly, lacks integration with the `get_maintainers.pl` tool. This last problem forces users to resolve and manually add the correct recipients. The `send-patch` command solves all three issues by providing default base configurations, shorter commands, and automatically resolving recipients. **Listings 7** and **8** show, respectively, a `git-send-email` command to send the current commit explicitly stating the recipients with additional options, and the respective `send-patch` command.

Listing 7: Example of git-send-email command.

```

$ git send-email -1 --annotate --to=[foo@bar.com](mailto:foo@bar.com) --cc=[bar@foo.com](mailto:bar@foo.com)

```

Listing 8: kw send-patch use case.

```

$ kw send-patch --send

```

3.2.7 config. With many features, kw has a system to manage each configuration and a dedicated user interface feature. The syntax of the `config` feature is composed of the pair `feature.configuration` followed by the value to be bound to the configuration. For example, **Listing 9** illustrates a case of configuring the target architecture of the build command to be `x86_64`.

Listing 9: kw config use case.

```

$ kw config build.arch 'x86_64'

```

3.2.8 patch-hub. A singular feature of kw is `patch-hub`, as it is a sub-project with a dedicated git repository, is written in Rust, and is a *Terminal User Interface* (TUI), a text-based equivalent of a *Graphical User Interface* (GUI), which significantly diverges from the fully CLI approach of the rest of kw. Nonetheless, it still provides interesting services to kernel developers, specifically maintainers, as the feature is a TUI that allows users to consult the flow of patches in `lore.kernel.org`, apply actions on them using other kw features, and more. The feature aims to cover the patchset reviewing workflow described in **Figure 2**. `patch-hub` has many more capabilities and is a work in progress, but **Figure 5** gives an overview of the feature potential.

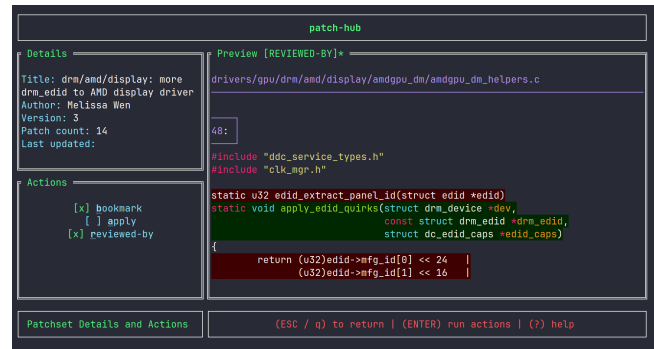


Figure 5: kw patch-hub use case.

3.3 Project architecture

Describing kw as hub-like fits not only from the user's perspective, as it is a collection of diverse features in a unified interface, but also from the project developer's perspective. The project code base is structured as follows:

- (1) Entry point file named `kw` located at the root of the repository [14], which is our *hub* component;
- (2) Feature-specific files named after their respective feature (e.g., the build feature has the `build.sh` file) located in the `src/` directory [15]. These are our *features* components;
- (3) Library files named after their scope (e.g., `kw_string.sh` deals with string manipulations) located in the `src/lib/` directory [16], shared across feature-specific files. These are our *libraries* components;

- (4) Plugin files with code heavily coupled with specific distros, CPU architectures, subsystems, and the like, that can change externally and are very mutable, located in the `src/plugins/` directory [17]. These are our *plugins* components.

As an illustration of kw inner workings, let us simulate the execution of a `kw deploy` call. First, the entry point `kw` file is loaded and executed. At the beginning, global variables are dynamically defined; then, using a switch-case, we resolve the feature as `deploy`, load the `deploy.sh` file, and call the standard main function of feature-specific files (in this case, `deploy_main()`). This function handles the rest of the deploy execution, using other functions defined inside `deploy.sh` and leveraging functions from the libraries and plugins components.

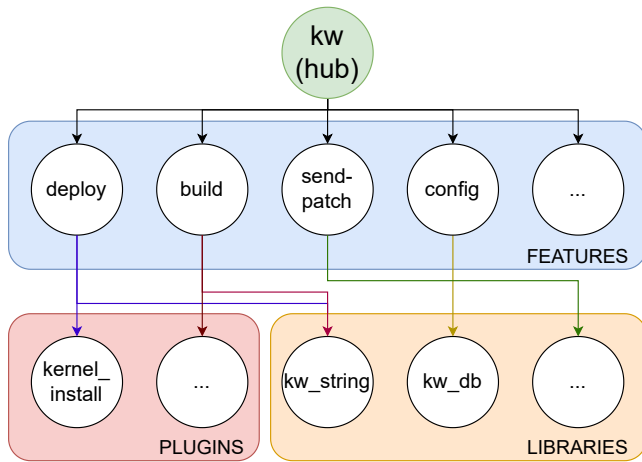


Figure 6: kw conceptual architecture.

The diagram in **Figure 6** models kw architecture, showing how extensible and modularized the project is. Note that this diagram is for illustration purposes and does not reflect the precise load and execution relationship between features and libraries/plugins.

3.4 Data collection for scientific research

Inside the kw project, we have implemented an infrastructure to collect statistics and other data, along with a lightweight, single-file database using *SQLite3* [8]. For instance, every time a developer builds a kernel, the time it took for the compilation is captured (using the *statistics* library) and registered in kw database (using the *kw_db* library). This type of data collection occurs in other parts of kw. However, additional data point opportunities can be leveraged for further scientific research on kernel workflows.

To name a few, kw captures: build and deploy times, logging whether they were successful; the list of kernels installed on the target machine, and uninstall events. We can also identify data points of interest for research on kernel workflows: during builds, capturing hardware specifications and cross-compilation strategies can enable analysis of efficient hardware setups and everyday compiler use; during deployment, recording the target distro, bootloader, and deployment method can uncover the landscape of testing environments in use; during patch submission, logging maintainer/mailling

list targeting can provide insights into overloaded personnel or subsystems in the Linux project hierarchy.

All data is stored locally in a single file within the user's file system, never exfiltrated, and users can easily opt out by turning off data collection. We also plan to implement a telemetry system for users to voluntarily submit data for research, inspired by the *Debian Popularity Contest* [6], a standard in telemetry and anonymization.

3.5 Comparison with other tools

kw works as a hub-like tool that continuously incorporates existing robust tools to avoid duplication, while also implementing features from scratch when no existing tools (satisfactorily) solve a given pain point. Incorporating other tools involves adding our biases, which we believe are closely tied to the Linux kernel development model, its rules, and its workflows. In this respect, no other tool exists that proposes a similar all-in-one approach to mitigate bottlenecks in kernel workflows, making kw a novel solution.

3.6 Future plans

For the future of kw, we aim to: (1) push for a ready-to-use patch-hub version, as the current implementation is closer to a proof of concept and does not yet cover the patchset reviewing workflow; (2) enhance data collection and implement data telemetry to enable the potential for scientific research; and (3) make integration tests more robust and extensive, encompassing a broad range of user workflows.

4 Conclusion

Kworkflow (kw) automates and streamlines kernel workflows by leveraging existing solutions fragmented across the Linux ecosystem and presenting a cohesive and unified CLI to users. The tool abstracts complexity in tasks such as build, deploy, and patch submission, mitigating bottlenecks in development, and supporting the long-term sustainability of Linux, while reducing duplication of efforts and wasted resources by the community. As a hub-like solution, kw is built to be extensible and modularized, allowing continuous incorporation of other tools to cover more kernel developers' pain points. Beyond its immediate utility, kw presents an incredible and robust opportunity for data collection that can further scientific research in the Linux kernel development model.

ARTIFACT AVAILABILITY

The specific version of the kworkflow project described in this work can be found in this **Software Heritage** [permalink](#). Other complementary links are the **kw upstream repository** and the **kw website**.

ACKNOWLEDGMENTS

This study is financed, in part, by CAPES (Finance Code 001), the University of São Paulo — USP (Proc. 22.1.9345.1.2), the São Paulo Research Foundation — FAPESP (Proc. 19/26702-8) and the São Paulo State Data Analysis System Foundation — SEADE (FAPESP Proc. 2023/18026-8), Brazil. We also thank the community for collaboratively building and maintaining kw, members of the Linux ecosystem for the feedback provided, and the students from IME-USP who have voluntarily fostered the project throughout the years.

REFERENCES

- [1] Andrea Arighi. 2025. *virtme-ng*. virtme-ng Project. <https://archive.softwareheritage.org/swlh:1.dir:d489c05da2b1b87a75d61bdf88a7309b7e351f6;origin=https://github.com/arighi/virtme-ng;visit=swlh:1.snp:f1ea6cb726992578ca5fe9c819f0fd09016123ed;anchor=swlh:1.rev:842007e7f3af88d50ed51341dfb42ceea0c3b517> Archived in Software Heritage: swlh:1.dir:d489c05da2b1b87a75d61bdf88a7309b7e351f6.
- [2] Fabrice Bellard and QEMU team. 2025. *QEMU*. QEMU Project. <https://archive.softwareheritage.org/swlh:1.dir:6a9686165e35116cc97e6501a7742646d2f9f479;origin=https://gitlab.com/qemu-project/qemu;visit=swlh:1.snp:f6695b73ad77f3371d011b846173b23e53db7359;anchor=swlh:1.rev:1dae461a913f9da88df05de6e2020d3134356f2e> Archived in Software Heritage: swlh:1.dir:6a9686165e35116cc97e6501a7742646d2f9f479.
- [3] Jonathan Corbet. 2014. *How 4.4's patches got to the mainline*. Linux Weekly News. <https://lwn.net/Articles/670209/>
- [4] Jonathan Corbet. 2017. *Patch flow into the mainline for 4.14*. Linux Weekly News. <https://lwn.net/Articles/737093/>
- [5] Jonathan Corbet and Greg Kroah-Hartman. 2017. *Linux Kernel Development Report*. Linux Foundation. <https://www.linuxfoundation.org/resources/publications/linux-kernel-report-2017?hsLang=en>
- [6] Debian Project. 2025. Debian Popularity Contest. <https://popcon.debian.org/>. Accessed: 2025-05-20.
- [7] DRM Maintainers. 2025. *DRM Maintainer Tools*. Intel Corporation. <https://archive.softwareheritage.org/swlh:1.dir:358b189f89b01e7d193a0630027cd59e528258b1;origin=https://gitlab.freedesktop.org/drm/maintainer-tools.git;visit=swlh:1.snp:24a80c7455a6110e9df33787f975f3e436325843;anchor=swlh:1.rev:202708c00696422fd217223bb679a353a5936e23> Archived in Software Heritage: swlh:1.dir:358b189f89b01e7d193a0630027cd59e528258b1.
- [8] Richard Hipp. 2025. *SQLite3*. SQLite Project. <https://archive.softwareheritage.org/swlh:1.dir:4621e6010913950ce033e02a1395432b68d1ad3b;origin=https://sqlite.org/2025/sqlite-src-3490100.zip;visit=swlh:1.snp:274ec34463a96122e714cf7b77f0eb21dc8cdd97> Archived in Software Heritage: swlh:1.dir:4621e6010913950ce033e02a1395432b68d1ad3b.
- [9] Dave Jones, Joel Schopp, Andy Whitcroft, and Joe Perches. 2025. *checkpatch.pl*. Linux Kernel Community. <https://archive.softwareheritage.org/swlh:1.cnt:3d22bf863e9edf0c2fe7ad7a8986d7e8ad892a;origin=https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git;visit=swlh:1.snp:1f0c2964234351f43ccac6b787c4da7f0aa8103c;anchor=swlh:1.rev:4d0be1aa26b7dba4960c37d9f8d695eb513bb04d;path=/scripts/checkpatch.pl> Archived in Software Heritage: swlh:1.cnt:3d22bf863e9edf0c2fe7ad7a8986d7e8ad892a.
- [10] LLVM Developer Group. 2025. *The LLVM Compiler Infrastructure*. LLVM Project. <https://archive.softwareheritage.org/swlh:1.dir:b3b208939883e92f2778e6d55cf181b65891e277;origin=https://github.com/llvm/llvm-project;visit=swlh:1.snp:351d2fcb5d3e55c13766a875e4e479389cee4155;anchor=swlh:1.rev:680b3b742da02972bc0b55298b6f472d2b95ca90a> Archived in Software Heritage: swlh:1.dir:b3b208939883e92f2778e6d55cf181b65891e277.
- [11] Joe Perches. 2025. *get_maintainer.pl*. Linux Kernel Community. https://archive.softwareheritage.org/swlh:1.cnt:4414194bedcfd747bd24199b5de9ccf04bf6d227;origin=https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git;visit=swlh:1.snp:1f0c2964234351f43ccac6b787c4da7f0aa8103c;anchor=swlh:1.rev:4d0be1aa26b7dba4960c37d9f8d695eb513bb04d;path=/scripts/get_maintainer.pl Archived in Software Heritage: swlh:1.cnt:4414194bedcfd747bd24199b5de9ccf04bf6d227.
- [12] Konstantin Ryabitsev. 2025. *B4 tools*. B4 Project. <https://archive.softwareheritage.org/swlh:1.dir:6146dd4655a3423b13a8ce77a1ceb9669251f133;origin=https://github.com/mricon/b4;visit=swlh:1.snp:8d816b83932d9e48c286481c898e3d3e05e6429d;anchor=swlh:1.rev:6f78e874e96b0b3bac1767a1743b20af20cb0e2f> Archived in Software Heritage with SWHID: swlh:1.dir:6146dd4655a3423b13a8ce77a1ceb9669251f133.
- [13] Rodrigo Siqueira, David Tadokoro, and Matheus Tavares. 2025. *Kworkflow*. University of São Paulo. <https://archive.softwareheritage.org/swlh:1.dir:90dc41328e09271597eb1f4f47d8a4c2e972a5bb;origin=https://github.com/kworkflow/kworkflow;visit=swlh:1.snp:4e47e956d8c04a55836d674aedf3c7be842b2641;anchor=swlh:1.rev:e1ac9a1965c366e200f9ab242a89a6545adcc7cc> Archived in Software Heritage: swlh:1.dir:90dc41328e09271597eb1f4f47d8a4c2e972a5bb.
- [14] Rodrigo Siqueira, David Tadokoro, and Matheus Tavares. 2025. *Kworkflow entry point*. University of São Paulo. <https://archive.softwareheritage.org/swlh:1.cnt:ea47bc419ea4eb2479f201cb4d27c2b6fec14282;origin=https://github.com/kworkflow/kworkflow;visit=swlh:1.snp:4e47e956d8c04a55836d674aedf3c7be842b2641;anchor=swlh:1.rev:e1ac9a1965c366e200f9ab242a89a6545adcc7cc;path=/kw> Archived in Software Heritage: swlh:1.cnt:ea47bc419ea4eb2479f201cb4d27c2b6fec14282.
- [15] Rodrigo Siqueira, David Tadokoro, and Matheus Tavares. 2025. *Kworkflow features*. University of São Paulo. <https://archive.softwareheritage.org/swlh:1.dir:d696593d47c764d8a32279581bbd52fe76061815;origin=https://github.com/kworkflow/kworkflow;visit=swlh:1.snp:4e47e956d8c04a55836d674aedf3c7be842b2641;anchor=swlh:1.rev:e1ac9a1965c366e200f9ab242a89a6545adcc7cc;path=/src/lib/> Archived in Software Heritage: swlh:1.dir:d696593d47c764d8a32279581bbd52fe76061815.
- [16] Rodrigo Siqueira, David Tadokoro, and Matheus Tavares. 2025. *Kworkflow libraries*. University of São Paulo. <https://archive.softwareheritage.org/swlh:1.dir:6fa251dc395901dfead332adde7964cd7a18cb5d;origin=https://github.com/kworkflow/kworkflow;visit=swlh:1.snp:4e47e956d8c04a55836d674aedf3c7be842b2641;anchor=swlh:1.rev:e1ac9a1965c366e200f9ab242a89a6545adcc7cc;path=/src/lib/> Archived in Software Heritage: swlh:1.dir:6fa251dc395901dfead332adde7964cd7a18cb5d.
- [17] Rodrigo Siqueira, David Tadokoro, and Matheus Tavares. 2025. *Kworkflow plugins*. University of São Paulo. <https://archive.softwareheritage.org/swlh:1.dir:53e76813cec821f79856d1f822a96f5a19ce5a28;origin=https://github.com/kworkflow/kworkflow;visit=swlh:1.snp:4e47e956d8c04a55836d674aedf3c7be842b2641;anchor=swlh:1.rev:e1ac9a1965c366e200f9ab242a89a6545adcc7cc;path=/src/plugins/> Archived in Software Heritage: swlh:1.dir:53e76813cec821f79856d1f822a96f5a19ce5a28.
- [18] Rodrigo Siqueira, David Tadokoro, and Matheus Tavares. 2025. *Kworkflow setup.sh script*. University of São Paulo. <https://archive.softwareheritage.org/swlh:1.cnt:b724ddb1e8c4fd7d45f2b0f212d7770c4a3715b9;origin=https://github.com/kworkflow/kworkflow;visit=swlh:1.snp:4e47e956d8c04a55836d674aedf3c7be842b2641;anchor=swlh:1.rev:e1ac9a1965c366e200f9ab242a89a6545adcc7cc;path=/src/plugins/> Archived in Software Heritage: swlh:1.cnt:b724ddb1e8c4fd7d45f2b0f212d7770c4a3715b9.
- [19] David Tadokoro. 2025. Kworkflow demo video. <https://archive.softwareheritage.org/swlh:1.cnt:3695bf04c525dc03d850baa96485d7bcdca7f9b3;origin=https://github.com/davidbtadokoro/sbes-tool-2025-kw-demo;visit=swlh:1.snp:21a2e8110b4a8c659c6ba451723f836d9fbb8db;anchor=swlh:1.rev:1cef3fedffb5a4848bfe8f337cf1acd1380a7793;path=/sbes-tool-2025-kw-demo.mp4> YouTube link: <https://youtu.be/SX0cApjhiHM>.
- [20] Linus Torvalds and Junio C. Hamano. 2025. *Git - fast, scalable, distributed revision control system*. Git Project. <https://archive.softwareheritage.org/swlh:1.dir:630e361bc0ae5b9def1e1f303ccad8b0a4ab5f76;origin=https://git.kernel.org/pub/scm/git/git.git;visit=swlh:1.snp:69ed932516cf4fa91546d7274f1c69ab68d8b53d;anchor=swlh:1.rev:6c0bd1fc70efaf053abe4e57c976afdc72d15377> Archived in Software Heritage: swlh:1.dir:630e361bc0ae5b9def1e1f303ccad8b0a4ab5f76.
- [21] Melissa Shihfan Ribeiro Wen. 2021. *What Happens When the Bazaar Grows: A comprehensive study on the contemporary Linux kernel development model*. Master's thesis. Universidade de São Paulo.
- [22] Yildiran, Necip Fazil and Oh, Jeho and Lawall, Julia and Gazzillo, Paul. 2024. Maximizing Patch Coverage for Testing of Highly-Configurable Software without Exploding Build Times. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 427–449.