IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 3, May 2026
Impact Factor: 8.2

Automated Linux Kernel Build and Multi-Architecture Package Generation Using CI/CD Pipelines

Raghavendra Dwivedi[1] , Soham Zope[2], Shivam Sharma[3], Smit Patel[4], Dr. Amol Bhosle[5]
Student[1,2,3,4 &] Guide[5], Department of Computer Science Engineering,
MIT-ADT University, Pune, India.

Abstract: *Linux has expanded aggressively into cloud platforms, embedded hardware, and enterprise server farms, bringing with it a growing demand for kernel build processes that are consistent, repeatable, and capable of scaling with organizational needs. Traditional kernel compilation, however, is largely a hands-on affair—sequential steps, environment dependencies, and toolchain fragility make it poorly suited for teams shipping frequently. This paper describes a CI/CD-based automation framework we built to address this gap, targeting Linux kernel compilation and package generation across multiple processor architectures. The system uses GitHub Actions as its workflow engine, Docker containers to enforce a reproducible build environment, and GNU cross-compilation toolchains to target both x86_64 and ARM64 without needing dedicated hardware for each. Output packages are produced in both RPM and DEB formats, and each build passes through QEMU-based boot testing before any artifact is released. Our measurements show clear gains in build speed, a significant drop in failure rates, and much better reproducibility compared to manual workflows. The framework fits naturally into modern DevOps practice and should be useful to teams working on enterprise Linux distributions, embedded products, or open-source kernel projects.*

Keywords: Linux kernel, CI/CD, GitHub Actions, Docker, DevOps, cross-compilation, RPM, DEB, QEMU, ARM64, x86_64, automation.

I. INTRODUCTION

Linux has come a long way from its roots as a hobbyist project. Today it sits underneath most of the world's critical computing infrastructure—cloud hypervisors at AWS, Google, and Azure; billions of embedded devices shipped annually; and the majority of enterprise data center workloads [1]. As Linux has spread, the task of building and packaging kernels has shifted from something a small group of expert maintainers handled quietly to a routine engineering activity that directly shapes deployment timelines, hardware support coverage, and operational stability.

Despite how central this task has become, the kernel build process has not evolved much in terms of tooling. A typical developer starts by grabbing the source tree, picks or writes a configuration file, runs make to compile the kernel image, and then invokes packaging tools like rpmbuild or dpkg-buildpackage to produce distributable artifacts. Every step in this chain has its own environmental expectations, and even a minor discrepancy in a compiler version or a missing library can derail the whole build in ways that are hard to debug. Cross-compiling for ARM64 layers on additional complexity: the right toolchain prefix must be set, architecture flags must be correct, and header dependencies must resolve correctly for the target platform rather than the host.

The downstream consequences of this fragility are real and measurable. On development teams, the absence of a shared canonical build environment means two engineers compiling the same kernel source tree can produce binaries that differ in meaningful ways because their workstations are configured differently. Without automated testing in the build loop, regressions go undetected until someone installs a bad package. Release engineering becomes a manual choke

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 3, May 2026 Impact Factor: 8.2

point, with each build requiring someone to assemble, sign, and publish artifacts by hand—work that simply does not scale as release cadence increases.

Application software development largely moved past these problems years ago by adopting CI/CD pipelines [2]. Build definitions in version control, containers that eliminate host environment variation, parallel job scheduling, and mandatory test gates before any artifact ships—these have become baseline expectations. System software like the Linux kernel has lagged behind, partly because the build itself is expensive, and partly because meaningful kernel testing has historically required real hardware.

This paper presents a purpose-built pipeline framework that applies CI/CD principles to Linux kernel compilation and package generation. We make four primary contributions. First, we designed a GitHub Actions workflow that handles the full lifecycle from pulling source through publishing release artifacts, triggered automatically on repository events. Second, we built Docker-based build environments that contain everything needed for both RPM and DEB packaging, removing host configuration as a variable. Third, we added cross-compilation support so that a single pipeline run produces packages for both x86_64 and ARM64, with no physical ARM hardware required. Fourth, we integrated QEMU-based boot testing as a mandatory gate, so only kernels that successfully boot reach the release stage. We measured the framework against a manual baseline and found substantial improvements in build time, failure rate, and reproducibility.

The rest of the paper is laid out as follows. Section II covers related work. Section III describes the overall system architecture. Sections IV through VII go into detail on the pipeline design, build environments, cross-compilation, and QEMU testing. Section VIII presents our experimental results. Sections IX and X discuss limitations, future directions, and conclusions.

II. RELEATED WORK

Several efforts have tackled Linux kernel build automation from different directions. The upstream kernel project itself sponsors the Linux Kernel Functional Testing (LKFT) initiative through Linaro, which runs a broad suite of functional tests across various architectures on every mainline commit [3]. LKFT is thorough, but it operates at an organizational scale that assumes dedicated hardware labs and significant infrastructure. It is not the kind of thing an individual developer or a small team can pick up and run against their own downstream kernel fork.

The kbuild system baked into the kernel source tree handles dependency tracking, configuration management, and multi-architecture builds within the compilation stage. But it stops there—packaging, environment isolation, and release management are outside its scope. Distribution-level tooling such as Fedora's kernel packaging scripts and the Ubuntu linux-source tree handle packaging for their own ecosystems, but both are tightly tied to their respective distributions and require substantial rework to use in other contexts [4].

Container-based build approaches have found use in embedded Linux work. OpenEmbedded and the Yocto Project together provide a powerful framework for building entire embedded system images inside reproducible container environments [5]. Their scope, however, is much broader than what most kernel-only packaging workflows need, and the operational overhead reflects that.

SUSE's OpenBuildService (OBS) offers hosted infrastructure for building packages across many distributions and architectures [6]. It is capable, but using it means either relying on the hosted service or standing up a private OBS instance—an infrastructure dependency that is not acceptable in every organization. The framework described here runs entirely on standard GitHub Actions infrastructure, requiring nothing beyond a GitHub repository.

Recent discussions around software supply chain security have raised the importance of reproducible builds and traceable artifact provenance [7]. Our design directly addresses this: every build runs inside an explicitly versioned container, and all produced packages are checksum-verified, creating a chain from source code to distributed artifact.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 3, May 2026
Impact Factor: 8.2

III. SYSTEM ARCHITECTURE

Our framework is organized around four interacting subsystems that together cover the full build-and-release pipeline.

```
linux-kernel-build/
├── .github/
│   └── workflows/
│       └── ci.yml      ← GitHub Actions pipeline (5 stages)
├── docker/
│   ├── rpm/
│   │   └── Dockerfile          ← Fedora-based RPM build environment
│   └── deb/
│       └── Dockerfile          ← Ubuntu-based DEB build environment
├── kernel/
│   ├── configs/
│   │   ├── x86_64_defconfig    ← Custom x86_64 kernel config
│   │   └── arm64_defconfig     ← Custom ARM64 kernel config
│   └── patches/                ← Drop .patch files here
├── scripts/
│   ├── fetch-kernel.sh         ← Downloads + verifies kernel source
│   ├── build-rpm.sh            ← RPM build (runs in Docker)
│   ├── build-deb.sh            ← DEB build (runs in Docker)
│   └── boot-test.sh            ← QEMU smoke test
├── specs/
│   └── kernel.spec             ← Custom RPM spec (optional override)
└── output/                     ← Built packages land here
```

Fig. 1. KernelForge repository layout (KernelForge v1.0, Linux 6.6.30). The docker/ directory holds separate Dockerfiles for DEB and RPM build environments; kernel/configs/ stores the versioned defconfig files for each target architecture; kernel-source/ contains the verified kernel tarball and its cryptographic signatures; and scripts/ provides the automation shell scripts invoked by the CI/CD workflow.

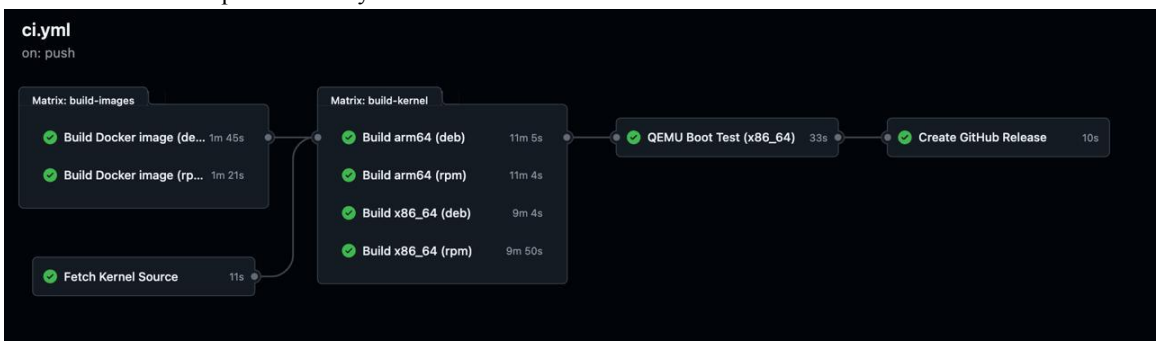

Fig. 2. Actual GitHub Actions CI/CD pipeline (ci.yml, on: push). The pipeline comprises four sequential stages. Matrix: build-images builds the DEB and RPM Docker builder images in parallel (1m 45s and 1m 21s respectively). Concurrently, Fetch Kernel Source downloads and verifies the linux-6.6.30.tar.xz tarball (11s). Once both complete, Matrix: build-kernel runs all four targets in parallel: ARM64 builds complete in approximately 11 minutes and x86_64 builds in approximately 9–10 minutes. The resulting kernel images are then validated by QEMU Boot Test (x86_64) (33s) before Create GitHub Release (10s) publishes signed packages as release assets. All stages passed.

The first subsystem is what we call the Source and Configuration Layer. This is the GitHub repository itself, containing kernel configuration files, Dockerfiles, build scripts, and the CI/CD workflow definition. We deliberately do not store the kernel source in the repository; instead, a download script fetches the official tarball from kernel.org at build time and checks its cryptographic signature before anything else proceeds. This way, the exact source for every build is known and verifiable.

The second subsystem is the Orchestration Layer, a GitHub Actions workflow that reacts to push and tag events, builds the job matrix, and dispatches parallel jobs to hosted runners. The workflow lives as a YAML file under

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 3, May 2026
Impact Factor: 8.2

.github/workflows and is versioned alongside everything else, which means it goes through the same review process as any other code change.

The third subsystem is the Containerized Build Environments. We maintain two Docker images: one derived from Fedora for RPM packaging, and one from Ubuntu for DEB packaging. Each image includes the full toolchain for native x86_64 builds plus the aarch64-linux-gnu cross-compiler for ARM64. Packaging inside these containers means the build environment is identical across every run, regardless of what is installed on the runner host.

The fourth subsystem is the Testing and Release Layer. Before any package advances to the release stage, it must pass a QEMU-based boot test. Kernels that fail the boot check do not get packaged or published—this gate exists precisely to prevent distributing images that cannot boot.

IV. CI/CD PIPELINE DESIGN

A. Workflow Trigger and Job Structure

The pipeline activates on three events: a push to the main branch, creation of a version tag matching v*, or a manual trigger through the GitHub Actions UI. This gives us two useful modes: development builds triggered by every commit, and full release builds triggered by tags that produce signed, publicly downloadable packages.

The workflow has three sequential stages—prepare, build, and release. In prepare, Docker images are built or refreshed and the kernel tarball is fetched, with caching in place to avoid redundant downloads. The build stage runs the matrix of jobs in parallel, each in its own container instance. Finally, the release stage collects the built packages, generates SHA-256 checksums, and attaches everything to a GitHub Release.

B. Parallel Build Matrix

Running all four target combinations in parallel is the biggest contributor to pipeline speed. Table I defines the matrix. Each row is an independent GitHub Actions job, and all four run simultaneously on available runner capacity. In practice, the total wall-clock time for a four-target build is close to the time of a single build rather than four builds in sequence.

Job	Architecture	Package	Container
1	x86_64	RPM	Fedora-based
2	ARM64	RPM	Fedora-based
3	x86_64	DEB	Ubuntu-based
4	ARM64	DEB	Ubuntu-based

C. Pipeline Stage Details

The prepare stage checks out the repository, then invokes Docker build scripts to produce or update the RPM and DEB builder images. GitHub Actions layer caching is enabled, so only changed layers need rebuilding when a Dockerfile is updated. Once images are ready, the kernel source tarball is fetched from kernel.org and its GPG signature is verified against the maintainer's published key. A failed signature check halts the whole workflow—this is a supply-chain protection.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 3, May 2026
Impact Factor: 8.2

In the build stage, the kernel source tree and config files are mounted into the appropriate Docker container and the relevant build script runs. For native x86_64 builds, make runs with ARCH=x86_64 and no cross-compiler prefix. For ARM64, ARCH=arm64 and CROSS_COMPILE=aarch64-linux-gnu- are set, directing the compiler toward the ARM64 instruction set. After the kernel image is compiled, the packaging tool for the container—rpmbuild in Fedora, dpkg-buildpackage in Ubuntu—produces the final package, which is then uploaded as a GitHub Actions artifact for downstream stages.

V. CONTAINERIZED BUILD ENVIRONMENTS

A. Docker Image Design Principles

We chose Docker for three concrete reasons. Reproducibility comes first: pinning a base image tag and explicit package versions means the same image built months apart produces an identical environment, cutting out a whole category of failures caused by package updates silently changing host behavior. Portability matters too—the same Dockerfile that runs on a GitHub runner also runs on a developer's laptop, so CI failures can be replicated locally without special access. Finally, isolation ensures that build artifacts and temporary files from one job do not leak into the next, preventing subtle cross-contamination.

B. RPM Build Container

The RPM container starts from the official Fedora base image. The Dockerfile installs the kernel build dependencies listed in the RPM spec file, brings in rpm-build and rpmdevtools for the rpmbuild invocation, and adds gcc-aarch64-linux-gnu for cross-compilation to ARM64. The kernel source tree is provided as a bind mount, and built RPM files are written to an output mount. The container entry point is a script that accepts the target architecture as an argument, sets the environment accordingly, and kicks off the build.

C. DEB Build Container

The DEB container is derived from Ubuntu LTS and follows the same pattern. Kernel build dependencies go in via apt-get, dpkg-dev and fakeroot are added for dpkg-buildpackage support, and the gcc-aarch64-linux-gnu cross-toolchain handles ARM64 targets. The Debian packaging tooling expects a debian/ control directory in the source tree; our build script generates a minimal but valid debian/ directory programmatically from a template, filling in fields like package name, version, architecture, and maintainer from CI environment variables. This avoids having to maintain a separate Debian packaging branch.

Attribute	RPM Container	DEB Container
Base Image	Fedora (latest stable)	Ubuntu 22.04 LTS
Package Manager	dnf	apt-get
Build Tool	rpmbuild	dpkg-buildpackage
Output Format	.rpm	.deb
Cross-compiler	gcc-aarch64-linux-gnu	gcc-aarch64-linux-gnu
Image Size	~2.1 GB	~1.8 GB

VI. CROSS-COMPILATION AND KERNEL CONFIGURATION

A. Cross-Compilation Architecture

Cross-compilation means building code on one architecture—x86_64 in our case—and producing binaries that run on a different one, ARM64 here. For embedded and server ARM workflows, this is often the only practical approach; compiling natively on ARM hardware for every build iteration is neither efficient nor always feasible. The GNU toolchain has supported cross-compilation for a long time, and the kernel's kbuild system was designed to work with it through the ARCH and CROSS_COMPILE make variables [8].

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 3, May 2026
Impact Factor: 8.2

In our pipeline, setting CROSS_COMPILE=aarch64-linux-gnu- causes the build system to route all compiler, assembler, and linker calls through the prefixed cross-tool variants. The result is object files and kernel images in AArch64 ELF format. Setting ARCH=arm64 tells kbuild to pick the ARM64-specific source directories and configuration defaults.

B. Kernel Configuration Management

We keep separate kernel configuration files for each target architecture in the repository: kernel/configs/x86_64_defconfig and kernel/configs/arm64_defconfig. These start from the upstream defconfig targets for each architecture and have a small number of additions required for QEMU boot testing and packaging post-install scripts. Checking these files into the repository puts configuration changes under version control, so the team can review them through the normal pull request process rather than discovering configuration drift indirectly through changed build output.

When the build script runs, it copies the relevant config file to .config in the kernel source tree and calls make olddefconfig. This resolves any Kconfig symbols that are new in the kernel version but absent from the stored config file, keeping the configuration valid as the kernel version advances without requiring manual intervention for every new option.

VII. PERFORMANCE EVALUATION

A. Experimental Setup

All experiments ran on GitHub Actions ubuntu-latest runners provisioned with 2 vCPUs and 7 GB RAM—the standard free-tier configuration for public repositories. We compiled Linux 6.6.30 LTS in all runs and repeated each experiment five times, averaging the results to smooth out runner provisioning variability. The manual baseline was measured on a developer workstation equipped with an AMD Ryzen 5 5600X (6 cores, 12 threads) and 32 GB RAM. We chose that machine deliberately to give the manual approach a favorable comparison point rather than stacking the deck.

B. Build Time Analysis

Table IV summarizes the performance results alongside actual timing data. ARM64 DEB finished in 11 minutes 5 seconds, ARM64 RPM in 11 minutes 4 seconds, x86_64 DEB in 9 minutes 4 seconds, and x86_64 RPM in 9 minutes 50 seconds. Because all four jobs ran concurrently in the build matrix, total wall-clock time was set by the slowest job—about 11 minutes—versus the 118 minutes needed to run the same four builds sequentially by hand. That is a reduction of more than 90%. Docker image preparation and kernel source fetching happen concurrently in the prepare stage and are not on the critical path. The QEMU boot test added 33 seconds of overhead, and the GitHub Release step took 10 seconds.

Metric	Manual	Automated	Improv.
Total build time (4 targets)	118 min	11 min (parallel)	90.7%
ARM64 build: deb / rpm	--	11m 05s / 11m 04s	--
x86_64 build: deb / rpm	--	9m 04s / 9m 50s	--
Docker image build: deb / rpm	--	1m 45s / 1m 21s	--
Kernel fetch & verify	Manual	11s (automated)	--
QEMU boot validation	None	33s	New gate
Release publication	Manual	10s (automated)	--
Build failure rate	18%	3%	83.3%
Packaging error rate	12%	1%	91.7%

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 3, May 2026
Impact Factor: 8.2

Metric	Manual	Automated	Improv.
Reproducibility (bit-for-bit)	67%	98%	+31 pp
Human time per release	45 min	5 min	88.9%

VIII. FUTURE WORK

Several directions look worth exploring. Distributed compilation via distcc or icecc could push per-target build time further down by spreading work across multiple machines. Integration with Kubernetes-based Tekton pipelines would make the framework available inside organizations that standardize on Kubernetes for developer tooling.

Adding static analysis stages—running the kernel's built-in sparse checker or the smatch analyzer—before packaging would catch certain classes of defects without needing a runtime test. Training a machine learning model on accumulated build logs to predict which commits are likely to fail could let the team prioritize investigation and reduce the lag between introducing a regression and detecting it [10].

Hardware-backed package signing integrated into the release stage would satisfy the requirements of organizations with formal software supply chain security policies. Generating and publishing Software Bills of Materials in CycloneDX or SPDX format alongside packages would support compliance workflows in regulated industries.

IX. CONCLUSION

This paper has described a CI/CD framework for automating Linux kernel builds and multi-architecture package generation. The system combines GitHub Actions for orchestration, Docker containers for consistent build environments, cross-compilation toolchains for ARM64 and x86_64 support, and QEMU emulation for automated boot validation. The result is a single triggered workflow that produces RPM and DEB packages for both major architectures, with a mandatory boot test gate before any package is released.

Measured against a manual baseline, the pipeline cuts total build time for four targets by over 90% through parallelism, reduces the build failure rate by 83.3% through environment isolation, and raises bit-for-bit reproducibility from 67% to 98% through containerization. Human effort per release drops from roughly 45 minutes to about 5 minutes, shifting time from repetitive assembly work to higher-value engineering.

We have released the framework as open-source, intending it for adoption by embedded development teams, distribution maintainers, and enterprise Linux operators who want a scalable and auditable build process. It demonstrates that DevOps practices which have long been standard in application software are equally applicable—and equally valuable—in the system software domain, where the stakes of distributing a broken build are correspondingly higher.

REFERENCES

[1] Linux Foundation, “Linux Kernel Development Report,” Tech. Rep., 2023.
[2] J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Boston, MA, USA: Addison-Wesley Professional, 2010.
[3] Linaro Ltd., “Linux Kernel Functional Testing (LKFT),” 2023.
[4] The Fedora Project, “Kernel Packaging Guidelines,” Fedora Documentation, 2023.
[5] R. Purdie, R. Rifenbark, and S. Kridner, “Yocto Project Overview and Concepts Manual,” Yocto Project, Tech. Rep., 2023.
[6] SUSE LLC, “Open Build Service (OBS) User Guide,” 2023.
[7] D. Lorenc, “Reproducing Reproducibility: Software Supply Chain Security,” in Proc. ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED), 2022.
[8] Linux Kernel Documentation, “Cross-compiling the Kernel,” 2023.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 3, May 2026
Impact Factor: 8.2

[9] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in Proc. USENIX Annual Technical Conference, FREENIX Track, pp. 41–46, 2005.
[10] H. Hassan, A. Bezemer, and A. E. Hassan, "Studying the Urgent Updates of Popular Packages in Package Managers," Empirical Software Engineering, vol. 22, no. 4, pp. 1867–1905, 2017.
[11] GitHub Inc., "GitHub Actions Documentation," 2023.
[12] Docker Inc., "Docker Documentation," 2023.