Canonical

Ubuntu Kernel

© 2026 Canonical Ltd.
All rights reserved.

Canonical

Contents

1 In this documentation **3**

2 How this documentation is organized **4**

3 Project and community **5**

3.1 How-to guides 5

3.2 Reference 36

3.3 Explanation 77

The Ubuntu Linux kernel is the core software enabling applications on Ubuntu to interact with system resources.

The Ubuntu kernel handles communication between system hardware and user-space applications, managing tasks like memory, processing, and security. Regular stable release updates (SRU) ensure the kernel stays secure, stable, and optimized.

Ubuntu kernels provide a reliable foundation for applications and system processes, meeting the need for secure, high-performance, Ubuntu environments. Kernels are also tested consistently for regressions to provide users with a reliable and smooth experience. Kernels are tailor made for Ubuntu Desktop, Ubuntu Server, a wide range of architectures, IoT devices, cloud providers, and more.

This documentation serves developers, partners, and others working with Ubuntu kernels, offering guidance on kernel workflows, tools, SRU timelines, and processes for customization and maintenance.

1. In this documentation

Contributing to Ubuntu kernels	*Patch acceptance criteria* (page 44) • *Stable patch format* (page 36) • *How to send patches to the Ubuntu mailing-list* (page 8) • *How to send patches to the Linux kernel mailing-list* (page 10)
Kernel development	*How to enable kernel source package repositories* (page 5) • *How to obtain and manage kernel source for an Ubuntu release using Git* (page 6) • *How to build an Ubuntu Linux kernel* (page 12) • *How to build an Ubuntu Linux kernel snap* (page 16) • *How to develop kernel modules for Ubuntu* (page 30) • *How to test pre-release Ubuntu kernels* (page 24) • *About Ubuntu Linux kernel sources* (page 86)
Kernel release and maintenance	*Ubuntu kernel SRU lifecycle* (page 78) • *Releasing an SRU kernel* (page 68) • *Kernel rollback* (page 70)
Kernel variants	*About kernel stable release updates (SRU)* (page 81) • *Kernel security and update policy for post-release trees* (page 79) • *HWE kernels* (page 56) • *OEM kernels* (page 61) • *Ubuntu kernel variants and branches* (page 50)
Upload rights	*Kernel upload rights* (page 73) • *DKMS upload rights* (page 75)

2. How this documentation is organized

This documentation uses the Diátaxis documentation structure[1].

- *How-to guides* (page 5) assumes you have basic familiarity with kernel development and provide generic instructions for common tasks involved in kernel development.
- *Reference* (page 36) provides detailed information about submitting patches and their criteria, and other processes related to Ubuntu kernels.
- *Explanation* (page 77) discusses the different aspects of the Ubuntu kernel and kernel development process at Canonical.

[1] https://diataxis.fr/

3. Project and community

Kernel documentation is a member of the Ubuntu family. It's an open source documentation project that warmly welcomes community contributions, suggestions, fixes and constructive feedback.

- Code of conduct[2]
- *Contribute to kernel docs* (page 32)

3.1. How-to guides

These guides accompany through the various stages of building and publishing kernel packages and components.

3.1.1. How to enable kernel source package repositories

If you want to build or modify an Ubuntu kernel package from source, you will first need the kernel source code. This is provided via `deb-src` - a line in the `sources.list` or `ubuntu.sources` file that points to repositories containing source packages instead of pre-built binaries. `deb-src` will need to be enabled on your build machine.

Enable `deb-src`

Noble Numbat 24.04 (and newer)

Add "deb-src" to the `Types:` line in the `/etc/apt/sources.list.d/ubuntu.sources` file.

```
Types: deb deb-src
URIs: http://archive.ubuntu.com/ubuntu
Suites: noble noble-updates noble-backports
Components: main universe restricted multiverse
Signed-By: /usr/share/keyrings/ubuntu-archive-keyring.gpg
```

Mantic Minotaur 23.10 (and older)

Check that you have the following entries in the `/etc/apt/sources.list` file. If not, add or uncomment these lines for your Ubuntu release.

```
deb-src http://archive.ubuntu.com/ubuntu jammy main
deb-src http://archive.ubuntu.com/ubuntu jammy-updates main
```

[2] https://ubuntu.com/community/docs/ethos/code-of-conduct

Update package list

Once you have updated `sources.list` or `ubuntu.sources`, update the package list for the changes to take effect:

```
sudo apt update
```

3.1.2. How to obtain and manage kernel source for an Ubuntu release using Git

The kernel source code for each Ubuntu release is maintained in its own repository in Launchpad. Downloading the kernel source may be needed for customization, development, or troubleshooting the kernel.

This document shows how you can obtain and manage the kernel source for an Ubuntu release using Git.

Prerequisites

You must have the git package[3] installed on your system.

```
sudo apt-get install git
```

Get local copy of kernel source for single release

Each Ubuntu release (e.g. "Noble", "Resolute") has its own codename-based kernel series and corresponding Git repository. Use `git clone` with the selected protocol to obtain a local copy of the kernel source for the Ubuntu kernel series you are interested in.

For example, to obtain a local copy of the Jammy kernel tree, run any of the following `git clone` commands:

```
git clone git+ssh://git.launchpad.net/~ubuntu-kernel/ubuntu/+source/linux/+git/jammy
git clone https://git.launchpad.net/~ubuntu-kernel/ubuntu/+source/linux/+git/jammy
```

See *Protocols for accessing kernel sources* (page 87) for more information.

Clone multiple releases using a shared reference repository

Cloning a single kernel tree downloads several hundred megabytes of data. If you plan to work with more than one Ubuntu kernel series, clone the upstream Linux kernel repository first and use it as a reference repository for the subsequent Ubuntu clones. This saves time and disk usage by reducing duplicate downloads.

```
git clone https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
git clone --reference linux https://git.launchpad.net/~ubuntu-kernel/ubuntu/+source/linux/
+git/jammy
git clone --reference linux https://git.launchpad.net/~ubuntu-kernel/ubuntu/+source/linux/
+git/noble
```

[3] https://packages.ubuntu.com/search?keywords=git

Each Ubuntu `git clone` creates a separate directory for a given series and has the full source and history available locally.

Caution:

Once two trees are linked this way, you cannot delete or move the upstream `linux` reference tree without manually updating `.git/objects/info/alternates` in each Ubuntu kernel tree that references it.

Add multiple series as remotes

If you want to compare or switch between several Ubuntu kernel series in one working repository, add each series as a remote and switch between them using branches. This keeps the series in one repository, but makes its remote and branch structure more complex than using separate clones:

```
git remote add jammy https://git.launchpad.net/~ubuntu-kernel/ubuntu/+source/linux/+git/
jammy
git fetch jammy
git checkout -b jammy --track jammy/master
git checkout -b jammy-next --track jammy/master-next
```

For example, if you want to check out the current Ubuntu kernel source for Jammy, use the `jammy` branch, which tracks `master`. If you instead want to check out the commits staged for the next Stable Release Update (SRU), you can create the `jammy-next` branch, which tracks `master-next`.

Work with multiple series in separate subdirectories

If you need the source for each Ubuntu kernel series in its own subdirectory within one repository, use `git subtree add`. This layout is useful when the series should be visible together in one working tree, but it puts the large kernel histories into that repository and requires explicit update commands.

```
git subtree add --prefix=jammy https://git.launchpad.net/~ubuntu-kernel/ubuntu/+source/
linux/+git/jammy master
git subtree add --prefix=noble https://git.launchpad.net/~ubuntu-kernel/ubuntu/+source/
linux/+git/noble master
```

This creates a `jammy/` subdirectory containing the Jammy kernel source and a `noble/` subdirectory containing the Noble kernel source, all within the same repository.

To pull future updates into a subtree, specify the remote URL and ref explicitly - there is no automatic upstream tracking for subtrees:

```
git subtree pull --prefix=jammy https://git.launchpad.net/~ubuntu-kernel/ubuntu/+source/
linux/+git/jammy master
```

Work with a specific kernel version using tags

By default, cloning checks out the repository's default branch, which is `master` for the Ubuntu kernel repositories described here. To work with a specific, previously released kernel version, use an Ubuntu release tag. An Ubuntu release tag starts with `Ubuntu-` and identifies a released kernel version.

To list matching tags in the cloned repository:

```
git tag -l Ubuntu-*
```

Example output:

```
Ubuntu-5.4.0-47.51
Ubuntu-5.4.0-48.52
Ubuntu-5.4.0-49.53
Ubuntu-5.4.0-51.56
Ubuntu-5.4.0-52.57
...
```

To check out a specific version while keeping a branch for additional local commits, create a branch pointing to that tag:

```
git checkout -b temp Ubuntu-5.4.0-52.57
```

The new branch points to the tagged snapshot, so you can add commits without leaving the repository in a detached `HEAD` state.

Related topics

- *About Ubuntu Linux kernel sources* (page 86)

3.1.3. How to send patches to the Ubuntu mailing-list

The Ubuntu mailing-list is the place to submit Ubuntu-specific changes, such as Ubuntu kernel configs.

Note:

If your patch otherwise applies more generally to the Linux kernel, you should *submit your patches upstream* (page 10) instead.

Patches must be sent to the Ubuntu kernel-team mailing list[4] (`kernel-team@lists.ubuntu.com`). If you have not done so already, subscribe to the list before submitting your first patch, as messages from non-subscribers are held for moderation.

To send kernel patches to the mailing list, you should use the `git send-email` command.

Note:

Most of the options explained here do not appear in the man page of `git-send-email` but `git-format-patch` instead. They however work the same.

[4] https://lists.ubuntu.com/mailman/listinfo/kernel-team

You may want to create a specific identity for keeping common settings when sending kernel patches. The commonly used settings are:

```
git config set sendemail.ubuntu-kernel.chainReplyTo false
git config set sendemail.ubuntu-kernel.suppresscc true
git config set sendemail.ubuntu-kernel.thread true
git config set sendemail.ubuntu-kernel.to kernel-team@lists.ubuntu.com

# And then include these settings with `--identity=ubuntu-kernel`
git send-email --identity=ubuntu-kernel ...
```

Specify series

All patches must be targeted at some series (unstable, noble, …). Specify the targeted series with the `--subject-prefix` option:

```
git send-email --subject-prefix="SRU][O/N/J:linux-azure][PATCH" ...
```

The tags used in this example show that this patchset is targeting the following kernels for an SRU update: *oracular*, *noble*, and *jammy:linux-azure*.

Send a new version of a patchset

Mistakes happen; we are all human. If you want to send a new version of your patchset that fixes some issues, you can use the `-v, --reroll-count` option:

```
git send-email --subject-prefix=... -v 2
```

This will generate `[PATCH v2]` instead of just `[PATCH]` to indicate that this is a new revision of a patchset.

You should first make sure that your original patchset was rejected by having a NAK/NACK in its thread. You can reply to the email saying that you will send a new version of the patchset.

If you found a mistake you made, you can NAK and say that you will send a new version in the same email.

In the cover letter of the new patchset, describe what was changed compared to the previous submitted version.

See also

- (Reference) *Patch acceptance criteria* (page 44)

3.1.4. How to send patches to the Linux kernel mailing-list

This guide shows you how to submit a change to the upstream Linux kernel. It assumes you have already made a code change, in a separate branch, that would benefit the wider kernel community.

Note:

Ubuntu kernel maintainers prefer to apply changes to the Ubuntu kernel by cherry picking an upstream patch, which is why upstreaming matters. Ubuntu-specific changes (for example, Ubuntu kernel config changes) do not need to be submitted upstream.

1. **Read the submission process.** Read the upstream patch submission documentation[5] and follow it accurately. If you need help, try asking other developers in the Ubuntu Kernel room[6] on the Ubuntu Matrix server.
2. **Match the coding style.** Make sure the change follows the upstream kernel coding style[7].
3. **Test your kernel.** If you have a test kernel for an Ubuntu release, post a link to it on the relevant Launchpad bug and get it tested by the community.
4. **Choose the correct git tree.** If your change is in a subsystem, target that subsystem maintainer's tree (these are later merged into Linus's tree). Otherwise, clone Linus's tree:

   ```
   git clone https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
   ```

5. **Commit and sign off.** Commit each logical change with a sign-off using `git commit -s`. Signing off adds a tag to the bottom of the commit message that marks you as being responsible for the code.

   If a patch fixes a bug in a released kernel and should be backported, add a `Cc: stable@vger.kernel.org` line to the sign-off area of its commit message (NOT as an email recipient). Pair it with a `Fixes:` tag identifying the commit that introduced the bug where possible.

   Note:

   The upstream patch submission documentation describes other tags as well, such as `Closes:`, `Reported-by:`, `Suggested-by:`, `Tested-by:`, `Link:`, etc.. Add them as applicable.

6. **Generate patches.** To generate patches from the commits, run:

   ```
   git format-patch --base=<base> -o outgoing/ <base>..
   ```

   Replace `<base>` with the branch or tag you based your work on top of (for example `master`). `--base=<base>` records the base commit the patches are built on top of. `<base>..` generates one patch per commit on top of `<base>` (e.g. all commits in your branch).

[5] https://docs.kernel.org/process/submitting-patches.html
[6] https://matrix.to/#/#kernel:ubuntu.com
[7] https://docs.kernel.org/process/coding-style.html

If you are sending more than one patch, add `--cover-letter` to also generate `outgoing/0000-cover-letter.patch`, which is used to describe the patchset overall. A single patch does not need a cover letter.

> **Note:**
>
> Both the cover letter and commit messages can be used to leave comments, so keep them distinct:
>
> - **The commit messages** explain *this specific change*: the problem it solves, why it is needed, and any implementation details a reader needs to understand. Make it self-contained — do not rely on external links or earlier patch versions. It is copied verbatim into the permanent git changelog, so write it for someone reading `git log` years later.
> - **The cover letter** describes *the series as a whole*: The overall goal, how the patches fit together, and the testing performed. It is not recorded in git history — it exists only to give reviewers context on the mailing list.

7. **Find the recipients.** Run `get_maintainer.pl` against your patch to list the maintainers and developers to mail it to:

```
./scripts/get_maintainer.pl ./0001-your-patch.patch
```

For example, a wireless driver patch produces a list such as:

```
Ivo van Doorn <IvDoorn@gmail.com>
Gertjan van Wingerde <gwingerde@gmail.com>
"John W. Linville" <linville@tuxdriver.com>
linux-wireless@vger.kernel.org
users@rt2x00.serialmonkey.com
netdev@vger.kernel.org
linux-kernel@vger.kernel.org
```

8. **Send the patch.** Submit the patch as inline text in a plain text email with no attachments or hyperlinks. Many email applications do not send plain text by default, so the simplest way is to use `git send-email`, which formats each patch correctly and lets you derive the recipient list from `get_maintainer.pl`. An example command may look like:

```
git send-email \
--to linux-kselftest@vger.kernel.org \
--to netdev@vger.kernel.org \
--cc-cmd='./scripts/get_maintainer.pl --norolestats outgoing/0001-selftests-net-fix-test_1.patch' \
--cc kernel.engineer@canonical.com \
--no-chain-reply \
outgoing/0000-cover-letter.patch \
outgoing/0001-selftests-net-fix-test_1.patch \
outgoing/0002-selftests-net-fix-test_2.patch
```

In this example, `--cc-cmd` runs `get_maintainer.pl` to add the subsystem maintainers automatically, `--cc` adds your own address, and `--no-chain-reply` keeps every patch threaded as a reply to the cover letter rather than to the previous patch.

9. **Respond to feedback.** It's not uncommon for patches to be revised before being accepted. Typically this involves discussing potential changes with maintainers, then resubmitting a revised patchset.

When you send a new revision of a patchset make it clear this is a revised submission (e.g. add 'PATCH v2' in the cover letter subject), add a changelog of what changed between versions (for example `v2: fixed refcount edge case in patch 3`) and a link to the previous version(s) on the mailing list. Put these in the cover letter, or below the `---` separator line for a single patch submission. Content below `---` is stripped when the patch is applied, so it never reaches the permanent changelog.

3.1.5. How to build an Ubuntu Linux kernel

If you have patches you need to apply to the Ubuntu Linux kernel, or you want to change some kernel configs, you may need to build your kernel from source. Follow these steps to customize and build the Ubuntu Linux kernel.

Important:

Kernels built using this method are not intended for use in production.

Prerequisites

- This guide supports Xenial Xerus and newer.
- It is recommended to have at least 8GB of RAM and 40GB free disk space on the build machine.

If this is the first time you are building a kernel on your system, you will need to *Set up build environment* (page 12) and *Install required packages* (page 12).

Otherwise, skip ahead to *Obtain the source for an Ubuntu release* (page 13).

Set up build environment

To build an Ubuntu kernel, you will need to enable the necessary source repositories in the `sources.list` or `ubuntu.sources` file.

See *How to enable kernel source package repositories* (page 5) for details.

Install required packages

To install the required packages and build dependencies, run:

```
sudo apt update && \
    sudo apt build-dep -y linux linux-image-unsigned-$(uname -r) && \
    sudo apt install -y fakeroot llvm libncurses-dev dwarves
```

Obtain the source for an Ubuntu release

There are different ways to get the kernel sources, depending on the kernel version you want to make changes to.

Get the latest kernel source for your Ubuntu release

Use the `apt source` command to get the latest kernel source for your Ubuntu release. This is the version in the archive, which may be newer than the kernel you are running.

In this guide we will assume you are using the generic kernel:

```
apt source linux-image-unsigned-$(uname -r)
```

Otherwise, append your flavour name onto the end (like `-realtime` for the real-time kernel). Keep in mind some folder names will be different in this case.

This will download and unpack the kernel source files to your current working directory.

```
<working_directory>
├── linux-X.Y.Z/
│   └── *
├── linux_X.Y.Z-*.diff.gz
├── linux_X.Y.Z-*.dsc
└── linux_X.Y.Z.orig.tar.gz
```

Get kernel source for a specific version

To build a particular kernel version, including the one you are currently running, use Git. Released versions are available as `Ubuntu-*` tags. See *How to obtain kernel source for an Ubuntu release using Git* (page 6) for detailed instructions.

Prepare the kernel source

Once you have the kernel source, go to the kernel source working directory (e.g. “linux-6.8.0”) and run the following commands to ensure you have a clean build environment and the necessary scripts have execute permissions:

```
cd <kernel_source_working_directory>

chmod a+x debian/scripts/* && \
    chmod a+x debian/scripts/misc/* && \
    fakeroot debian/rules clean
```

Modify ABI number

You should modify the kernel version number to avoid conflicts and to differentiate the development kernel from the kernel released by Canonical.

To do so, modify the ABI number (the number after the dash following the kernel version) to "999" in the first line of the changelog file. For the generic kernel this file is at `debian.master/changelog`.

For example, modify the ABI number to "999" for Noble Numbat:

```
linux (6.8.0-999.48) noble; urgency=medium
```

If you are building something other than the generic Ubuntu Linux kernel, modify the ABI number in `debian.<flavour>/changelog` instead.

Modify kernel configuration

(Optional) To enable or disable any features using the kernel configuration, run:

```
fakeroot debian/rules editconfigs
```

This will invoke the `menuconfig` interface for you to edit specific configuration files related to the Ubuntu kernel package. You will need to explicitly respond with `Y` or `N` when making any config changes to avoid getting errors later in the build process.

Note:

If you do not have the compiler toolchain installed for each architecture supported by the kernel being configured, you'll see errors that the configs for these uninstalled architectures are missing. These can be ignored as long as you don't intend to build binaries for those architectures.

Customize the kernel

(Optional) Add any firmware, binary blobs, or patches as needed.

Build the kernel

Resolute Raccoon 26.04 (and newer)

To simplify the build process for development, export this config option:

```
export do_linux_main_modules_depends=false
```

Important:

zfs and v4l2loopback modules are not included in this mode. If you need these modules during development we recommend installing their dkms packages.

Questing Quokka 25.10 (and older)

No action needed.

You are now ready to build the kernel.

```
fakeroot debian/rules clean && \
    fakeroot debian/rules binary
```

Note:

Run `fakeroot debian/rules clean` to clean the build environment each time before you recompile the kernel after making any changes to the kernel source or configuration.

If the build is successful, several .deb binary package files will be produced in the directory one level above the kernel source working directory.

For example, building a kernel with version "6.8.0-999.48" on an x86-64 system will produce the following .deb packages (and more):

- `linux-headers-6.8.0-999_6.8.0-999.48_all.deb`
- `linux-headers-6.8.0-999-generic_6.8.0-999.48_amd64.deb`
- `linux-image-unsigned-6.8.0-999-generic_6.8.0-999.48_amd64.deb`
- `linux-modules-6.8.0-999-generic_6.8.0-999.48_amd64.deb`

Install the new kernel

Important:

The packages produced by this build are unsigned. On a system with Secure Boot enabled, the new kernel will not boot. Either test on a system with Secure Boot disabled, such as a virtual machine, or sign the kernel image with an enrolled Machine Owner Key (MOK) first before installing.

Install all the debian packages generated from the previous step (on your build system or a different target system with the same architecture) with dpkg -i and reboot:

```
cd ..
sudo dpkg -i linux-headers-<kernel version>*_all.deb \
    linux-headers-<kernel version>-<flavour>*.deb \
    linux-image-unsigned-<kernel version>-<flavour>*.deb \
    linux-modules-<kernel version>-<flavour>*.deb
sudo reboot
```

Test the new kernel

Run any necessary testing to confirm that your changes and customizations have taken effect. You should also confirm that the newly installed kernel version matches the value in the `<kernel_source_working_directory>/debian.master/changelog` file by running:

```
uname -r
```

3.1.6. How to build an Ubuntu Linux kernel snap

If you are running an Ubuntu Core system and want to use boot into a custom kernel, you will need a kernel snap.

This guide shows how to build a kernel snap for local development and testing.

Important:

Kernel snaps built using this method are not intended for use in production.

Prerequisites

Before you begin, you will need:

- A Launchpad account
- To be part of the Launchpad team that owns the project (for private repositories)
- A build machine running Ubuntu
- A device running an Ubuntu Core image with “dangerous” model assertion grade to install the custom kernel snap

Note:

The Ubuntu version of the build host must match the version of the device where the kernel snap will be installed. For example, use an Ubuntu 22.04 (Jammy) host to build the kernel snap for an Ubuntu Core 22 device.
See Snap - Build environment options[8] for more information.

[8] https://documentation.ubuntu.com/snapcraft/stable/reference/build-environment-options/

Set up build environment

Set up the host machine which will be used to build the kernel snap.

Install snapcraft

Snapcraft is used to create a managed environment to build the kernel snap. You are recommended to use the latest/stable version of the snapcraft snap from the Snap Store.

On the build machine, remove any existing snapcraft debian package and install snapcraft by running:

```
sudo apt-get update
sudo apt-get -y upgrade
sudo apt purge -y snapcraft
sudo snap install snapcraft --classic
```

Configure source repositories

Configure the package source repositories for the host architecture by specifying the architecture (e.g. "[arch=amd64]" for x86-64 hosts) for each deb source list in the data sources file.

Ubuntu 24.04 (Noble) and newer

Update the `/etc/apt/sources.list.d/ubuntu.sources` file. For example, on a x86-64 host running Ubuntu 24.04 (Noble):

```
[...]
Types: deb deb-src
URIs: http://archive.ubuntu.com/ubuntu
Suites: noble noble-updates noble-backports
Components: main universe restricted multiverse
Architectures: amd64
[...]
```

Ubuntu 23.10 (Mantic) and older

Update the `/etc/apt/sources.list` file. For example, on a x86-64 host running Ubuntu 22.04 (Jammy):

```
deb [arch=amd64] http://archive.ubuntu.com/ubuntu focal main restricted
```

Alternatively, if you are running a default installation of Ubuntu, you can do a global update of all sources in the `/etc/apt/sources.list` file.

```
sudo sed -ie 's/deb http/deb [arch=amd64] http/g' /etc/apt/sources.list
```

Add support for cross-compilation

Add the target architecture (e.g. “arm64”) to the list of supported architectures. This step is only required if the build machine is running on a different architecture than the target device for the kernel snap.

For example, if you want to build a kernel snap for an ARM64 device on a x86-64 host, run:

```
sudo dpkg --add-architecture arm64
sudo apt update
```

Confirm that support for the target architecture has been added successfully by running dpkg --print-foreign-architectures:

```
user@host:~$ dpkg --print-foreign-architectures

arm64
```

Configure SSH settings for Launchpad access

Enable SSH access to `git.launchpad.net` for your Launchpad account. This step is only required if you are building a snap from a private repository in Launchpad.

Add the following in the `~/.ssh/config` file:

```
Host git.launchpad.net
  User <your Launchpad username>
```

Clone the kernel snap recipe

Once you have set up your host machine, clone the Ubuntu Linux kernel snap recipe.

```
git clone <kernel-source-repository>
```

Customize the kernel

Add any firmware or binary blobs, or customize `initrd` as needed. This step is only required if you want to make your own changes to the kernel.

Build the kernel snap

You are now ready to build the kernel snap.

1. Go to the directory with the cloned kernel repository.

   ```
   cd <kernel-source-repository>
   ```

2. Create an alias for the `snapcraft.yaml` file. This is only required if there are multiple YAML configuration files in the `snap/local/` tree.

```
ln -s snap/local/<project>.yaml snapcraft.yaml
```

3. (Optional) Add the `sed` command in the `snapcraft.yaml` file to set the Kconfig value `CONFIG_MODULE_SIG_ALL` to `n` for your target architecture. This allows unverified modules to be loaded into the kernel and should only be set to `n` for local testing and development.

   For example, if the kernel snap is for an ARM64 device, set `'arm64': 'n'`:

```
[...]
parts:
  kernel:
    override-build: |
      [...]
      # override configs
      sed -i "s/^\(CONFIG_MODULE_SIG_FORCE\).*/\\1 policy\<{'arm64': 'n', 'armhf': 'n
'}\>/" ${DEBIAN}/config/annotations
      sed -i "s/^\(CONFIG_MODULE_SIG_ALL.*\)'arm64': 'y'\(.*\)/\\1'arm64': 'n'\\2/" $
{DEBIAN}/config/annotations
      [...]
```

4. Build the kernel snap package.

UC24 and UC22

```
sudo snapcraft --build-for=arm64 --destructive-mode
```

UC20

```
sudo snapcraft --target-arch=arm64 --destructive-mode --enable-experimental-target-
arch
```

5. You should get a `<name>_<version>_<arch>.snap` file in the kernel repository root, where:

   - `<name>` is the identified set in `snapcraft.yaml`
   - `<version>` is the kernel version
   - `<arch>` is the target architecture for the kernel snap

6. Copy the kernel snap to your target device and reboot into latest kernel to verify your changes.

```
snap install --dangerous --devmode <name>_<version>_<arch>.snap
```

Note:

Local snaps can only be installed if the Ubuntu Core image on the target device was created with a model assertion that specifies the “dangerous” grade.

3.1.7. How to test a bug with a mainline kernel

Warning:

Mainline build kernels are **not supported** and **never receive security updates**. Mainline build kernels are **not suitable for production** use. **Remove** them once you've finished testing. **Support is not provided** to users who are running mainline build kernels outside of diagnostic uses.

When you encounter an Ubuntu kernel bug, testing an unmodified upstream kernel can help you determine whether an issue is specific to Ubuntu's kernel patches or has already been fixed upstream.

The Ubuntu Kernel Team publishes mainline builds on a best effort basis using upstream kernel source and Ubuntu kernel configuration files; these are packaged as `.deb` files which are ready to be installed.

This guide shows you how to select, verify, install, and remove one of these mainline builds.

See also

- For background, see *About mainline kernel builds* (page 77).
- For file names and archive layout, see *Mainline kernel archive* (page 63).
- To test kernels that Ubuntu has built but not yet released, see *How to test pre-release Ubuntu kernels* (page 24).

Before you start

- **Use an installed system.** You are recommended to install the mainline kernel in an installed environment, restart, and choose the new kernel at boot.
- **Remove out-of-tree modules.** If you use proprietary or out-of-tree modules, such as the NVIDIA proprietary driver or VirtualBox, uninstall them first.
- **Note your current kernel.** Run *uname(1)*[9] with the `-r` option, so you can tell your normal kernel from the test kernel later.
- **Check your architecture.** Run `dpkg --print-architecture`.

[9] https://manpages.ubuntu.com/manpages/resolute/en/man1/uname.1.html

Choose a version

Choose a tagged build from the mainline archive[10]. Start with the latest stable version when checking whether upstream has fixed an issue. To locate when a regression began, test tagged versions on either side of the known working and broken versions.

Optionally, use the Ubuntu to mainline version mapping[11] to find the upstream release that your current Ubuntu kernel is based on. This helps you choose versions on either side of it.

Download the files

Open the directory for the version you chose and check the build status for your architecture. If the build failed, choose another version.

Open the directory for your architecture and download all `.deb` files for **one** available flavor. Available flavors vary by architecture and kernel version. Don't mix files from different flavors or builds.

For example, a complete `generic` package set for `amd64` normally contains:

- common headers, with a file name ending in `_all.deb`
- architecture- and flavor-specific headers
- the kernel image
- kernel modules

Download `CHECKSUMS` and `CHECKSUMS.gpg` from the same architecture directory. Keep the packages and checksum files in a dedicated directory.

See also

For details about the package names, see *Package files* (page 64).

Verify the downloaded files

Warning:

A compromised kernel can compromise the security of the entire system. Verify every kernel package before you install it.

1. Import the mainline builder's signing key from the Ubuntu keyserver:

```
gpg --keyserver hkps://keyserver.ubuntu.com --recv-key
"60AA7B6F30434AE68E569963E50C6A0917C622B0"
```

2. Display the imported key's fingerprint:

[10] https://kernel.ubuntu.com/mainline/?C=N;O=D
[11] https://people.canonical.com/~kernel/info/kernel-version-map.html

```
gpg --fingerprint "60AA7B6F30434AE68E569963E50C6A0917C622B0"
```

The output should match the value below:

```
pub   rsa2048 2008-05-01 [SC]
      60AA 7B6F 3043 4AE6 8E56  9963 E50C 6A09 17C6 22B0
uid           [ unknown] Kernel PPA <kernel-ppa@canonical.com>
```

3. Check that `CHECKSUMS` was signed with that key, using *gpg(1)*[12]:

```
gpg --verify CHECKSUMS.gpg CHECKSUMS
```

You should see `Good signature from "Kernel PPA <kernel-ppa@canonical.com>"`. A warning that the key isn't certified with a trusted signature is expected, because you haven't marked the key as trusted. Compare the key's fingerprint with the one in the reference instead.

```
gpg: Signature made Wed 19 Aug 2026 08:18:38 PM +08
gpg:                using RSA key 60AA7B6F30434AE68E569963E50C6A0917C622B0
gpg: Good signature from "Kernel PPA <kernel-ppa@canonical.com>" [unknown]
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: 60AA 7B6F 3043 4AE6 8E56  9963 E50C 6A09 17C6 22B0
```

4. Check the downloaded files against the checksums:

```
shasum -c CHECKSUMS 2>&1 | grep 'OK$'
```

You should get a line ending in `OK` for each downloaded `.deb` file and for each type of checksum listed in `CHECKSUMS`. If a file is missing from the output, don't install it.

Install the kernel

Change to the dedicated directory that contains the downloaded files, then install all the packages with *dpkg(1)*[13]:

```
sudo dpkg -i ./*.deb
```

Resolve installation failures

If `dpkg` reports an error, match it against the following causes.

- A module fails to build

  A Dynamic Kernel Module Support (DKMS) error mentions VirtualBox or another out-of-tree module. Uninstall the module, as described in *Before you start* (page 20), then repeat the kernel installation.

- Unsatisfied dependencies

  The error is similar to `depends on <package> (>= <version>); however: Package <package> is not installed.`

[12] https://manpages.ubuntu.com/manpages/resolute/en/man1/gpg.1.html
[13] https://manpages.ubuntu.com/manpages/resolute/en/man1/dpkg.1.html

  - If the package is installed but the version is older than required, you need a newer Ubuntu release before you can use this kernel.
  - If the package isn't installed, and the version in your release is new enough, install it, then repeat the installation.

- Any other failure

  Check the build status and log in the archive. If the build failed or its packages are incomplete, try the next most recent suitable version.

Boot and test

1. Reboot your system. To reach the boot menu on a BIOS system, hold `Shift` during startup. On a UEFI system the key varies, often `Esc`.
2. Select **Advanced options for Ubuntu**.
3. Select the new entry, which looks like `Ubuntu with Linux <version>-generic`.
4. Confirm the running kernel with `uname -r` and record its version, then try to reproduce your issue.
5. If a Launchpad bug report exists for your issue, report your test results and findings in that bug.

Return to your normal kernel

Warning:

Mainline build kernels are **not supported** and **never receive security updates**. Mainline build kernels are **not suitable for production** use. **Remove** them once you've finished testing. **Support is not provided** to users who are running mainline build kernels outside of diagnostic uses.

Once you've completed the necessary testing, make sure to remove the mainline build and return to your original kernel.

1. Reboot and select your normal Ubuntu kernel from the same menu.

   Mainline kernels use a separate application binary interface (ABI) namespace, so they install alongside Ubuntu kernels. Remove the mainline kernel anyway, because it never receives security updates.

2. List the installed kernel packages:

   ```
   dpkg -l | grep "linux\-[a-z]*\-"
   ```

3. Purge the packages that belong to the mainline kernel with *apt(8)*[14]:

   ```
   sudo apt purge <KERNEL_PACKAGES_TO_REMOVE>
   ```

[14] https://manpages.ubuntu.com/manpages/resolute/en/man8/apt.8.html

Match the version in each package name to the mainline kernel version you recorded while testing. **Do not** remove the packages for your normal Ubuntu kernel. One kernel version typically includes common headers, architecture-specific headers, an image, and modules.

Related topics

- *About mainline kernel builds* (page 77)
- *Mainline kernel archive* (page 63)
- *How to test pre-release Ubuntu kernels* (page 24)

3.1.8. How to test pre-release Ubuntu kernels

Pre-release Ubuntu kernels are available in the Canonical Kernel Team (CKT) build PPAs and `-proposed` pocket before they are published to `-updates`. To install these pre-release kernels you need to opt in, as neither source is enabled by default.

This guide explains how to enable and test pre-release Ubuntu kernels, and how to report any regressions you find.

Caution:

Kernels published to the build PPAs are not suitable for production systems - enabling them could cause boot failures or regressions and is not recommended.

For details on the kernel lifecycle, see the *Ubuntu kernel SRU lifecycle* (page 78).

Enable the -proposed pocket to software sources

To install packages from -proposed, you need to enable the relevant source repositories on your Ubuntu machine.

Enable the -proposed pocket via GUI

1. Open “Software & Updates”.
2. Go to the “Developer Options” tab.
3. Enable the *Pre-released updates (<series>-proposed)* option.

Enable the -proposed pocket via CLI

Noble Numbat 24.04 (and newer)

Add "<series>-proposed" (e.g. "noble-proposed") to the `Suites:` line in the `/etc/apt/sources.list.d/ubuntu.sources` file.

```
Types: deb
URIs: http://archive.ubuntu.com/ubuntu
Suites: noble noble-updates noble-backports <series>-proposed
Components: main universe restricted multiverse
Signed-By: /usr/share/keyrings/ubuntu-archive-keyring.gpg
```

Jammy Jellyfish 22.04 (and older)

Add "<series>-proposed" (e.g. "jammy-proposed") to the following line in:

- `/etc/apt/sources.list`:

  ```
  deb http://archive.ubuntu.com/ubuntu/ <series>-proposed restricted main multiverse
  universe
  ```

- `/etc/apt/sources.list` (for non-x86 architectures):

  ```
  deb http://ports.ubuntu.com/ubuntu-ports <series>-proposed restricted main multiverse
  universe
  ```

Enable the CKT build PPA

You can add the CKT build PPAs to your system as follows:

```
sudo add-apt-repository ppa:canonical-kernel-team/ppa
sudo add-apt-repository ppa:canonical-kernel-team/ppa2
```

Warning:

Do remember that these kernels may be entirely untested at time of download and thus potentially fundamentally broken. Only test from build PPAs if you have a reason to do so. Do not test from build PPAs if you have Secure Boot enabled.

Install the pre-release kernel

First, update the sources cache:

```
sudo apt update
```

Then proceed to install the kernel using either a metapackage or a specific ABI-named image.

Install via kernel metapackage

Use this approach if you want to receive automatic updates for the latest version of the kernel in that series.

If the kernel version in -proposed is the highest in any pocket, run:

```
sudo apt install linux-<flavor>
```

If you want a specific (earlier) version of a metapackage, include the version in the command:

```
sudo apt install linux-<flavor>=<version>
```

Install via ABI-named kernel image

Use this method to install a specific kernel version without being tied to the kernel series metapackage.

```
sudo apt install linux-image-<abi>-<flavor>
```

Boot into the new kernel

After installing the kernel, reboot your machine. After booting up again, verify that the correct kernel is loaded with:

```
uname -r
```

This should print the correct kernel version and flavor.

Test the kernel

Once you have the new kernel installed, testing can begin.

If you do not have your own test suite and need an example workload, you can start with the built-in Linux selftests[15]. To run these selftests, download the kernel source and compile the tests.

```
apt source linux-image-unsigned-$(uname -r)
cd <kernel_source_working_directory>
sudo make -C tools/testing/selftests run_tests
```

For other examples of kernel testing projects, see:

- Linux Test Project[16]
-

[15] https://docs.kernel.org/dev-tools/kselftest.html
[16] https://linux-test-project.readthedocs.io/en/latest/

Report regression bugs

If you encounter a regression or bug while testing the kernel, please file a bug report on Launchpad. You can submit your report using any of the following methods:

1. Run the `ubuntu-bug` tool on the system with the newly installed kernel.

   ```
   ubuntu-bug
   ```

2. Manually file a bug online at https://bugs.launchpad.net/ubuntu/+filebug. Make sure to target the correct kernel source package and Ubuntu series.

For more information on Ubuntu bug reporting, see How to report a bug[17].

Related topics

- Ubuntu Wiki - Enable Proposed[18]

3.1.9. How to rebuild a single kernel module

If you have a patch for a specific kernel driver and want to test it quickly, you can rebuild just that module out-of-tree instead of rebuilding the entire kernel. This approach is significantly faster and is well suited for iterating on a driver fix.

Important:

Modules built using this method are not intended for use in production. For managing kernel modules across kernel upgrades, consider using *dkms(8)*[19] instead.

[19] https://manpages.ubuntu.com/manpages/resolute/en/man8/dkms.8.html

Prerequisites

Use this method when your patch is confined to the driver's own sources.

If your patch changes shared kernel headers, Kconfig options, or the signature of an exported symbol, the rebuilt module no longer matches the running kernel. `CONFIG_MODVERSIONS` refuses to load a module whose exported symbol signatures changed, but it does not detect structure layout changes inside headers. For those changes, build and boot a complete kernel instead. See *How to build an Ubuntu Linux kernel* (page 12).

The module is built against the running kernel version (`uname -r`) and works only on that version. Installing a different kernel requires rebuilding the module.

This guide supports Trusty Tahr and newer.

[17] https://ubuntu.com/project/docs/contributors/qa-and-testing/report-a-bug/
[18] https://wiki.ubuntu.com/Testing/EnableProposed

Install required packages

```
sudo apt update && \
    sudo apt install -y linux-source build-essential linux-headers-$(uname -r)
```

Note:

If you are developing against a custom kernel you will need to manually install it's headers using `dpkg`, rather than getting Ubuntu kernel headers from `apt`.

Obtain and patch the kernel source

Install the kernel source package and extract it to your working directory:

```
sudo apt install -y linux-source
tar xjf /usr/src/linux-source-$(uname -r | cut -d- -f1).tar.bz2
```

Note:

The tarball name uses the base kernel version (e.g. `linux-source-6.8.0.tar.bz2`), not the full ABI version string reported by `uname -r`.

Apply your patch to the extracted source tree:

```
cd linux-source-$(uname -r | cut -d- -f1)
patch -p1 < /path/to/your.patch
```

Build the module

Build only the driver subdirectory containing your change. Replace `drivers/<path/to/driver>` with the actual path relative to the kernel source root (for example, `drivers/net/ethernet/intel/e1000e`):

```
make -C /lib/modules/$(uname -r)/build \
    M=$PWD/drivers/<path/to/driver> modules
```

The compiled module file (`<driver>.ko`) will appear inside `drivers/<path/to/driver>/`.

Load and test the module

Unload the old module if it is currently loaded, then load the new one:

```
sudo modprobe -r <driver>
sudo insmod drivers/<path/to/driver>/<driver>.ko
```

Important:

On a system with Secure Boot enabled, loading an unsigned module will fail. Either test on a system with Secure Boot disabled, such as a virtual machine, or sign the module with an enrolled Machine Owner Key (MOK).

Confirm the module loaded successfully:

```
lsmod | grep <driver>
sudo dmesg | tail -20
```

Test and verify that your patch is working as intended.

To restore the module shipped with the kernel, run `sudo modprobe -r <driver>` followed by `sudo modprobe <driver>`.

Note:

Loading a module with `insmod` affects only the running system and will not persist through reboots.

Make the module persist across reboots

To test the behaviour of a module at boot, install it into the `updates` directory. `depmod` searches this directory ahead of the one holding the module shipped with the kernel, so your build takes precedence without modifying any file owned by a package:

```
sudo mkdir -p /lib/modules/$(uname -r)/updates
sudo cp drivers/<path/to/driver>/<driver>.ko /lib/modules/$(uname -r)/updates/
sudo depmod -a
modinfo -F filename <driver>
```

The path reported by `modinfo` must be the one under `updates`.

If the driver is in the initramfs, you need to rebuild that too:

```
lsinitramfs /boot/initrd.img-$(uname -r) | grep <driver>
sudo update-initramfs -u -k $(uname -r)
```

To remove the override, delete the file from `updates`, then rerun `depmod`, and `update-initramfs` if you rebuilt the initramfs.

Note:

The override applies only to the kernel version you installed it under. Installing a new kernel provides a new module tree, and your build is no longer used.

3.1.10. How to develop kernel modules for Ubuntu

This guide is for developers who write kernel modules intended to run on Ubuntu Classic, or to eventually land in the Ubuntu kernel.

It assumes you are familiar with Linux kernel development basics, and focuses on Ubuntu-specific steps: Which source tree to use and where to get it, how to build it, and what rules apply if you want your work accepted into Ubuntu.

Select a kernel source tree

Ubuntu offers many kernels, so it is important to understand which kernel you plan to target. Ubuntu kernel sources are organized by release codename and package name. Each combination is a kernel series, for example `noble:linux` or `jammy:linux-azure`. See *Ubuntu kernel variants and branches* (page 50) for the full list of kernel variants, their target releases, and branching strategy.

Pick the series that matches your target:

- The Ubuntu release you want to support, such as Resolute Raccoon 26.04.
- The kernel package for that release. Use the generic `linux` package unless you target a specific variant such as `linux-aws` or `linux-oem`.

Obtain the source

You have two options, depending on your goal:

- Git clone from Launchpad. Best for development and patch submission.
  - Within a series repository, the `master` branch contains the current released kernel, `master-next` has the changes staged for the next Stable Release Update (SRU). Previous releases are tagged with an `Ubuntu-*` tag. See *How to obtain and manage kernel source for an Ubuntu release using Git* (page 6) for the full workflow, including tags, remotes, and reference clones.
- Source package via `apt`. Best for a quick build of the kernel you are running. Enable `deb-src` first as described in *How to enable kernel source package repositories* (page 5), then run `apt source linux-image-unsigned-$(uname -r)`.

Build the Ubuntu way

Build with the Ubuntu kernel packaging so your module matches the production kernel configuration, signing setup, and ABI.

- To build a full kernel, follow *How to build an Ubuntu Linux kernel* (page 12).
- To iterate on one driver, rebuild only that module against the running kernel. Follow *How to rebuild a single kernel module* (page 27).
- To ship a module that survives kernel upgrades, package it with *dkms(8)*[20].

[20] https://manpages.ubuntu.com/manpages/resolute/en/man8/dkms.8.html

Important:

Locally built kernels and modules are for testing only. They are not signed for Secure Boot and are not intended for production.

Constraints for landing work in Ubuntu

If you intend your module to be accepted into the Ubuntu kernel, work under these constraints from the start:

- **Upstream first.** Get your driver into the upstream Linux kernel before requesting it in Ubuntu. Ubuntu regularly follows upstream, so your module will also land in Ubuntu shortly after. Patches that are not upstreamed (i.e. 'SAUCE' patches) are avoided as much as possible. See *Patch acceptance criteria* (page 44).
- **Prefer dynamically loading your module.** Ubuntu prefers to dynamically load kernel modules, as opposed to building them in. Ensure that your module behaves well when dynamically loaded.
- **Test thoroughly.** Many kernel module bugs originate from race conditions with dependencies, implicit dependencies on firmware or packages providing them, or not gracefully handling cases where a device is hotplugged, not connected, disconnected, etc.
- **Do not modify core kernel code.** Changes outside your own driver subsystem are unlikely to be accepted.
- **Do not change existing in-tree drivers** beyond what your patch requires. Unrelated changes will be rejected.
- **Do not rely on out-of-tree only interfaces.** Your module must build against the unmodified Ubuntu kernel headers for the kernel you target.
- **Follow upstream patch rules.** Run `scripts/checkpatch.pl` before submitting. Patches that fail upstream checks are rejected.
- **Follow the Ubuntu patch format.** Use the correct subject prefix, `BugLink`, and sign-off tags. See *Stable patch format* (page 36).
- **One Launchpad bug per submission.** SRU patchsets require a dedicated bug with the SRU justification template. See *Patch acceptance criteria* (page 44).
- **No new features directly to LTS releases.** New functionality must go through the development release or an HWE kernel first.

When your patch is ready, submit it to the Ubuntu kernel team mailing list as described in *How to send patches to the mailing list* (page 8).

Development and debugging suggestions

These are optional aids, not requirements:

- Use *How to rebuild a single kernel module* (page 27) for fast build-test cycles on a single driver.
- Use *dkms(8)*[21] during development to rebuild your module automatically when the kernel changes.
- Test against pre-release kernels to catch breakage early. See *How to test pre-release Ubuntu kernels* (page 24).

Related topics

- *About Ubuntu Linux kernel sources* (page 86)
- *About kernel stable release updates (SRU)* (page 81)
- *DKMS upload rights* (page 75)

3.1.11. How to contribute to Kernel documentation

We believe that everyone has something valuable to contribute, whether you're a coder, a writer, or a tester. Here's how and why you can get involved:

- **Why join us?** Work with like-minded people, develop your skills, connect with diverse professionals, and make a difference.
- **What do you get?** Personal growth, recognition for your contributions, early access to new features, and the joy of seeing your work appreciated.
- **Start early, start easy**: Dive into code contributions, improve documentation, or be among the first testers. Your presence matters, regardless of experience or the size of your contribution.

The guidelines below will help keep your contributions effective and meaningful.

Code of conduct

When contributing, you must abide by the Ubuntu Code of Conduct[22].

License and copyright

By default, all contributions to Kernel documentation are licensed under the Creative Commons Attribution-Share Alike 3.0 Unported License. To view a copy of this license, visit https://creativecommons.org/licenses/by-sa/3.0/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

All contributors must sign the Canonical contributor license agreement[23], which grants Canonical permission to use the contributions. The author of a change remains the copyright owner of their code (no copyright assignment occurs).

[21] https://manpages.ubuntu.com/manpages/resolute/en/man8/dkms.8.html
[22] https://ubuntu.com/community/docs/ethos/code-of-conduct
[23] https://canonical.com/legal/contributors

Environment setup

Kernel documentation is built on top of the Canonical Sphinx starter pack[24] and hosted on Read the Docs[25].

To work on the project, you will need to have Python, `python3.12-venv`, and `make` packages installed.

```
sudo apt install make
sudo apt install python3
sudo apt install python3.12-venv
```

Documentation

The documentation source files are stored in the `docs` directory of the repository.

For general guidance, refer to the starter pack guide[26].

For syntax help and guidelines, refer to the Canonical documentation style guides[27].

In structuring, the documentation employs the Diátaxis[28] approach.

To run the documentation locally before submitting your changes:

```
make run
```

Automatic checks

GitHub runs automatic checks on the documentation to verify spelling, validate links, and suggest inclusive language.

You can (and should) run the same checks locally before committing and pushing a change:

```
make spelling
make linkcheck
make woke
```

Submissions

If you want to address and issue or bug in this project, leave a comment in the issue indicating your intent to work on it. Also, reference the issue when you submit the changes.

- (Kernel docs members) Create a branch off the `main` branch of the Kernel documentation GitHub repository[29] and add your changes to it.
- (External contributors) Fork the Kernel documentation GitHub repository[30] and add the changes to your fork.

[24] https://github.com/canonical/sphinx-docs-starter-pack
[25] https://about.readthedocs.com/
[26] https://canonical-starter-pack.readthedocs-hosted.com/
[27] https://docs.ubuntu.com/styleguide/en/
[28] https://diataxis.fr/
[29] https://github.com/canonical/kernel-docs
[30] https://github.com/canonical/kernel-docs

- Properly structure your commits, provide detailed commit messages, and *sign off your commits* (page 35).
- Make sure the updated project builds and runs without warnings or errors; this includes linting, documentation, code (where applicable), and tests.
- Submit the changes as a pull request (PR)[31].

Your changes will be reviewed in due time; if approved, they will eventually be merged.

Describing pull requests

To be properly considered, reviewed, and merged, your pull request must provide the following details:

- **Title**: Summarize the change in a short, descriptive title.
- **Description**: Explain the problem that your pull request solves. Mention any new features, bug fixes, or refactoring.
- **Relevant issues**: Reference any related issues, pull requests, and repositories[32].
- **Testing**: Explain whether new or updated tests are included.
- **Reversibility**: If you propose decisions that may be costly to reverse, list the reasons and suggest steps to reverse the changes if necessary.

Commit structure and messages

Use separate commits for each logical change, and for changes to different sections in the Kernel documentation. Prefix your commit messages with the names of sections or pages that they affect, using the code tree structure. For example, start a commit that updates the explanation page about SRU cycles with `explanation/about-sru:`.

Use conventional commits[33] to ensure consistency across the project:

```
docs(how-to/get-sources): Update generic format for source package repository

* Added new URL structure details for Noble: https://github.com/canonical/kernel-docs/
issues/12345
* Separate content for pre- and post-24.04 release
```

Such structure makes it easier to review contributions and simplifies porting fixes to other branches.

[31] https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requ
creating-a-pull-request-from-a-fork
[32] https://docs.github.com/en/get-started/writing-on-github/working-with-advanced-formatting/
autolinked-references-and-urls
[33] https://www.conventionalcommits.org/

Sign off on commits

All changes that go into the Kernel documentation repository need to be signed off (using the `-s` or `--signoff` option) by the contributor.

```
git commit -s -m "docs(explanation/about-sru): updated life cycle diagram"
```

This sign off confirms that you made the changes or have the right to commit it as an open-source contribution.

If you made a commit without signing off, you can run the following to amend the most recent commit, append the "Signed-off-by" line without changing the commit message, and push again:

```
git commit --amend --no-edit -n -s
git push --force
```

Start contributing

If you are ready to contribute but unsure where to start, here are some suggested starting points.

1. Pick up an existing GitHub Issue[34].

   Whether you're a seasoned pro, or just beginning your journey in kernel development and/or open source, there's always a variety of tasks for your unique skills. Find an open issue that sparks your interest, assign it to yourself, and start collaborating.

2. Update and remove old documentation.

   If you browse through the project and find information or whole pages that are either outdated or obsolete, submit a PR with changes to update or delete them.

3. Migrate content from Ubuntu Wiki.

   In an effort to make collaboration efforts more effective, and keep content accurate and up-to-date, we aim to migrate as much content to our Read the Docs instance. If you come across an article that is useful and relevant, migrate the content from Wiki by creating a new file and/or section in this repository.

4. Work with what's in front of you.

   If none of the earlier suggestions appeal to you, then just browse through the existing Kernel documentation with an open mind and keen eye. If you see a paragraph that can be written in a more concise manner, or a set of instructions that can be made clearer, send along your suggestions for these improvements. Big or small, an improvement is always a step in the right direction.

Thank you, and looking forward to your contributions!

[34] https://github.com/canonical/kernel-docs/issues

3.1.12. Source code access and management

You can get access to kernel source code via `apt` or directly from the kernel Git repositories. You can also check the formatting requirements, review process, and best practices for submitting kernel patches to the Ubuntu kernel team mailing list.

- *Enable kernel source package repositories* (page 5)
- *Obtain kernel source for an Ubuntu release using Git* (page 6)
- *Send patches to the Ubuntu mailing-list* (page 8)
- *Send patches to the Linux kernel mailing-list* (page 10)

3.1.13. Development and customization

The steps to build a kernel is similar but may have slightly difference configuration requirements on depending on the build package (snap, debs), platform and/or architectures.

- *Build an Ubuntu Linux kernel* (page 12)
- *Build an Ubuntu Linux kernel snap* (page 16)
- *Rebuild a single kernel module* (page 27)
- *Develop kernel modules for Ubuntu* (page 30)

3.1.14. Testing and verification

These guides relate to testing the kernel to ensure its stability and functionality before you push or release a patch.

- *Test pre-release Ubuntu kernels* (page 24)

3.2. Reference

Reference material about Ubuntu kernel development processes, terminology, and more.

3.2.1. Kernel patch & contribution guidelines

It's important to follow the Ubuntu kernel patch guidelines so your contributions are accepted into the Ubuntu kernel tree without delays or rejection. These pages show how to properly format and submit your patches.

 patches mailing list

Stable patch format

Every Ubuntu LTS release during standard security maintenance period welcomes contributions from anyone. However, patches must comply with the required format.

See The Ubuntu lifecycle and release cadence[35] for more information.

[35] https://ubuntu.com/about/release-cycle

Prerequisites

Subscribe here[36] to join kernel-team@lists.ubuntu.com before submitting your first patch. Messages from non-subscribers will be held in a queue pending admin approval.

Preparing commits

Every patch **must** adhere to the following guidelines.

Subject line

Every patch submitted to a stable kernel **must** have have its subject line starting with "[SRU]" followed by the release name against which the patch is targeted.

The release name **must** be enclosed in "[ ]" brackets and **should** be abbreviated using the first letter of the release name (e.g. "N" for "Noble") in upper case. For example:

```
[SRU][N][PATCH 0/1] Fix error of resume on rtl8168fp
```

- If a patch is to be applied to multiple releases, a list of release names must be provided, with "/" separating the uppercase letter representing each release. For example, when it targets Bionic and Focal:

  ```
  [SRU][B/F][PATCH 1/1] KVM: fix overflow of zero page refcount with ksm running
  ```

  Note:

  Historically, the rule has been somewhat flexible, and various styles have been permitted. You may find examples of various styles (such as the ones below) in the mailing list archive[37]:

  ```
  [SRU][B,F][PATCH 1/1] KVM: fix overflow of zero page refcount with ksm running
  [SRU][B][F][PATCH 1/1] KVM: fix overflow of zero page refcount with ksm running
  [SRU][Bionic,Focal][PATCH 1/1] KVM: fix overflow of zero page refcount with ksm
  running
  [SRU][Bionic/Focal][PATCH 1/1] KVM: fix overflow of zero page refcount with ksm
  running
  [SRU][Bionic][Focal][PATCH 1/1] KVM: fix overflow of zero page refcount with ksm
  running
  ```

  Please adhere to the `[B/F]` style.
  - Use initial letter(s) in upper case
  - Separate each series with "/"

  [37] https://lists.ubuntu.com/archives/kernel-team/

- If the patch has to be applied to a specific derivative for multiple releases, indicate the derivative after the release. For example:

[36] https://lists.ubuntu.com/mailman/listinfo/kernel-team

```
[SRU][B:linux-kvm/F:linux-kvm][PATCH 0/1] UBUNTU: [Config] kvm: Add support for
modifying LDT
```

Subject line for non-upstream patches

Note:

Upstream patches refer to patches that only include commits that already reside in Linus's mainline tree.

If the patch requested doesn't come from upstream, it must contain one of the following on the subject line after the release name and patch number.

Descriptor	Meaning
UBUNTU: SAUCE:	This is a patch to the kernel code that has not been applied on mainline (Linus' tree). This category covers the following cases: 1. The submitter has either authored the patch or obtained the patch from a non-upstream source. 2. The patch has been applied to an upstream tree but not yet merged on mainline. 3. The patch is never expected to be submitted upstream but is of enough value for Ubuntu to carry it in our tree. 4. The patch has been submitted to upstream but is of enough value for Ubuntu to carry it in our tree regardless of upstream acceptance.
UBUNTU: [Packaging]	This is an update relevant to Ubuntu Packaging, including the contents of the various `debian*/` directories.
UBUNTU: [Config]	This is an update to the kernel configuration as recorded in the `debian.<branch>/config` directory. See the `debian.master/config/README.rst` or Discourse - Kernel configuration in Ubuntu[38] for more information about the config format.
UBUNTU: ubuntu	This is an update to an Ubuntu specific driver in the `ubuntu/` directory. This category is rarely used anymore except in special cases.
UBUNTU:	This subject line is internally used by some automation scripts. Avoid using it unless none of the other categories are appropriate for your patch.

For example, for a patch that falls under the "UBUNTU: SAUCE:" category:

```
[SRU][F][PATCH 2/2] UBUNTU: SAUCE: shiftfs: prevent ESTALE for LOOKUP_JUMP lookups
```

[38] https://discourse.ubuntu.com/t/kernel-configuration-in-ubuntu/35857

Comment body

1. Every patch associated with a Launchpad bug must have a link to the bug in the comment section of the commit, in the form of a "BugLink" block.

   A "BugLink" block must immediately follow the subject line and be the first text in the body of the commit comment. A "BugLink" block consists of:

   1. A blank line.
   2. One or more lines containing "BugLink:" and a URL to the Launchpad bug. The URL must be of the format: "https://bugs.launchpad.net/bugs/<bug-id>", where <bug-id> is the bug number of the associated Launchpad bug tracker.
   3. Another blank line.

   Every stable patch **must** have an associated Launchpad bug for tracking by the kernel stable and SRU teams. Exceptions are patches for CVE fixes (*see below* (page 41)).

   Example:

   ```
   Subject: [SRU][F][PATCH 1/1] UBUNTU: SAUCE: netfilter: nf_tables: Fix EBUSY on
   deleting unreferenced chain

   BugLink: https://bugs.launchpad.net/bugs/2089699

   [...]
   ```

2. Every patch **must** have a "Signed-off-by" line for the person submitting the patch. The "Signed-off-by" line **must** follow all other provenance lines and should be the last line in the commit comment.

   Example:

   ```
   Signed-off-by: Jesse Barnes <jbarnes@virtuousgeek.org>
   Signed-off-by: Linus Torvalds <torvalds@linux-foundation.org>
   (backported from commit 5620ae29f1eabe655f44335231b580a78c8364ea)
   Signed-off-by: Manoj Iyer <manoj.iyer@canonical.com>
   ```

3. Every patch **must** display the provenance of the patch. We want to preserve where the patch came from, who signed off on it, who ack'd it, whether it was cherry-picked from upstream and applied cleanly or not and who finally applied it to an official Ubuntu source tree.

   Backported patches:

   - If the patch required changes (e.g. it did not apply cleanly), use "backported from commit <sha1>" between brackets "()". For example:

     ```
     (backported from commit <sha1> <upstream repo name>)
     ```

     There must be a brief explanation immediately after the "(backported from …)" block, between square brackets, with the name of the person who introduced the change.

     ```
     (backported from commit <sha1> <upstream repo name>)
     [roxanan: Had to adjust the context due to missing commit <sha1>]
     ```

Cherry-picked patches:

- If the patch is a simple cherry-pick from an upstream repo and it applies cleanly, that **must** also be spelled out in the provenance section in the format “backported from commit <sha1>” between brackets “()”. For example:

```
(cherry picked from commit <sha1> <upstream repo name>)
```

Note:

Omit the “<upstream repo name>” if the patch comes from the mainline tree.

Example:

```
Signed-off-by: Adam Jackson <ajax@redhat.com>
Signed-off-by: Eric Anholt <eric@anholt.net>
Signed-off-by: Greg Kroah-Hartman <gregkh@suse.de>
(cherry picked from commit d4e0018e3e4dd685af25d300fd26a0d5a984482e linux-2.6.34.y)
Signed-off-by: Manoj Iyer <manoj.iyer@canonical.com>
```

4. Every **CVE** patch **must** contain a line just before your sign-off that specifies the CVE number(s) related to the patch.

   A “BugLink” is optional for CVE patches.

   Example:

```
[... commit message body ...]

Signed-off-by: Lion Ackermann <nnamrec@gmail.com>
Acked-by: Toke Høiland-Jørgensen <toke@toke.dk>
Signed-off-by: David S. Miller <davem@davemloft.net>
(cherry picked from commit 5eb7de8cd58e73851cd37ff8d0666517d9926948)
CVE-2024-53164
Signed-off-by: Ian Whitfield <ian.whitfield@canonical.com>
```

Preparing to submit patches

In most cases, patches should be submitted as a patch series accompanied by a cover letter. However, if the patch series is relatively large (e.g. more than 20 commits), consider sending a git pull request instead.

Sending as a patch series

1. Every patch submitted to a stable kernel **must** be sent in a patch series with a cover letter, even if the patch series contains a single patch.
2. The cover letter **must** contain the same “BugLink” line as in the patches themselves, when one is present.
3. CVE cover letters **must** have the CVE number as the subject.

4. The cover letter **must** contain the SRU justification from the launchpad bug or the CVE fix. See Kernel security and update policy for post-release trees[39] for more information about the SRU justification format to be added to a bug.
5. All the emails in the patch series **must** be numbered (e.g. "[PATCH 0/3]", "[PATCH 1/3]", etc.) and all the patches sent in reply to the cover letter (PATCH 0/N).

> **Tip:**
>
> When sending patches with git-send-email, use the option "--suppress-cc=all" in order to prevent adding the original author of the patch and other people from the provenance block as CC.

Sending as a pull request

1. Include the git pull request information in the cover letter email.
2. The cover letter **must** contain the same "BugLink" line as in the patches themselves, when one is present.
3. CVE cover letters should have the CVE number as the subject.
4. The cover letter **must** contain the SRU justification from the launchpad bug or the CVE fix. See Kernel security and update policy for post-release trees[40] for more information about the SRU justification format to be added to a bug.
5. The subject line of the cover letter **must** contain the "[PULL]" tag, instead of "[PATCH X/N]".
6. The git repository **must** be publicly accessible.
7. The body of the commits should follow the same rules as for a patch series.
8. The format of the title of the commits contained in the pull request should be the same as for the patch series, except for the tags at the beginning of the subject enclosed in "[]" brackets which would be removed by `git am` on application.

Submitting the patch

Stable patches must be sent to kernel-team@lists.ubuntu.com.

Once the patch receives two "Acked-by" replies from members of the Ubuntu Kernel Team, it will be merged.

[39] https://ubuntu.com/project/docs/SRU/reference/exception-Kernel-Updates/
[40] https://ubuntu.com/project/docs/SRU/reference/exception-Kernel-Updates/

Patch series example

Here is an excerpt from an example patch series that adheres to the guidelines.

Cover letter (PATCH 0/1)

```
Subject: [SRU][F][PATCH 0/1] s390/cpum_cf: Add new extended counters for IBM z15 (LP:
1881096)
From: frank.heimes@canonical.com
Date: 24.06.20, 22:11
To: kernel-team@lists.ubuntu.com

BugLink: https://bugs.launchpad.net/bugs/1881096

SRU Justification:

[Impact]

With perf from Ubuntu 20.04 on IBM z15 hardware, some counters
reported with lscpumf are not usable with 'perf stat -e'.
[...]

[Fix]

Cherry-pick upstream commit:
d68d5d51dc89 ("s390/cpum_cf: Add new extended counters for IBM z15")

[Test Plan]

Requires the fix/patch of the perf tool, as mentioned in the bug, too.
[...]

[Where problems could occur]

The regression can be considered as low, since:
[...]

[Other Info]

This requires a patch to be included into the perf itself, too - please
see bug description for more details.
[...]
```

Patch 1/1

```
Subject: [SRU][F][PATCH 1/1] s390/cpum_cf: Add new extended counters for IBM z15
From: frank.heimes@canonical.com
Date: 24.06.20, 22:11
To: kernel-team@lists.ubuntu.com

From: Thomas Richter <tmricht@linux.ibm.com>
```

(continues on next page)

(continued from previous page)

```
BugLink: https://bugs.launchpad.net/bugs/1881096

Add CPU measurement counter facility event description for IBM z15.

Signed-off-by: Thomas Richter <tmricht@linux.ibm.com>
Reviewed-by: Sumanth Korikkar <sumanthk@linux.ibm.com>
Signed-off-by: Vasily Gorbik <gor@linux.ibm.com>
(cherry picked from commit d68d5d51dc898895b4e15bea52e5668ca9e76180)
Signed-off-by: Frank Heimes <frank.heimes@canonical.com>

[...]
```

Related topics

- Kernel security and update policy for post-release trees[41]: shows the SRU Justification format to be added to a bug.
- ubuntu-check-commit[42]: script to check commits against Ubuntu submission rules.

Patch acceptance criteria

Generally, any patch is eligible for inclusion in the Ubuntu kernel, though some criteria apply.

Patch category

A patch falls into 3 categories:

cherry-pick
: The patch is part of the stable upstream for the desired kernel version, but picking it with the stable upstream patches may be delayed and there's a request to have the specific patch as soon as possible.

backport
: The patch is upstream (either mainline[43] or stable[44]), but part of a newer kernel version, and the submitter asks for a backport to an older Ubuntu kernel version.

SAUCE
: The patch is not upstream and/or never will.

Usually, cherry-picks have the biggest rate of approval if it's done correctly. Backports and SAUCE patches are a bit tricky. In general, we avoid merging those as much as possible.

For a detailed description on how to format patches before submission, see *Stable patch format* (page 36), while here is a quick (and far from complete) introduction.

[41] https://ubuntu.com/project/docs/SRU/reference/exception-Kernel-Updates/
[42] https://kernel.ubuntu.com/forgejo/actions/ubuntu-check-commit/src/branch/main/ubuntu-check-commit
[43] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/
[44] https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/

All patches

Reasons why the Ubuntu Kernel Team won't approve a patch:

Patch does not apply

The patch should always be based on a recent kernel. Expect to resubmit again if the tip changes and the patch has conflicts.

Patch fails upstream checks

Patches are expected to pass the upstream kernel patch tools provided with the kernel source tree. A patch that fails these checks will be rejected.

Before you submit, do the following:

- Read `Documentation/process/submitting-patches.rst` for the upstream patch submission guidelines.
- Run `scripts/checkpatch.pl` against a generated patch to catch style and formatting problems.
- Run `scripts/cleanpatch` if the patch needs to be cleaned before submission.
- Run `scripts/cleanfile` on a new file before you create the patch.
- Run `scripts/Lindent` if a file needs to follow the kernel indentation style.

SRU patches

This section describes additional reasons why the Ubuntu Kernel Team won't approve a SRU patch.

Launchpad bug

Each patch must be related to a dedicated Launchpad bug. The bug should be targeted to the kernels and series that the patch is aiming to lend.

The bug description must follow the SRU template[45].

See this example Launchpad bug[46].

[45] https://documentation.ubuntu.com/sru/en/latest/reference/bug-template/
[46] https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1995957

BugLink

Patches and cover letter should have a link to a Launchpad bug as the first line of the description.

The link must be in the short form `https://bugs.launchpad.net/bugs/XXXXXX`.

Example:

```
Subject: [SRU][O/N][PATCH v2 0/1] ALSA: hda/realtek: fix mute/micmute LEDs for a HP
EliteBook 645 G10

BugLink: https://bugs.launchpad.net/bugs/2087983

...
```

The link must not be in its long form `https://bugs.launchpad.net/ubuntu/+source/linux/+bug/XXXXXX`.

Bad example:

```
Subject: [SRU][O/N][PATCH v2 0/1] ALSA: hda/realtek: fix mute/micmute LEDs for a HP
EliteBook 645 G10

BugLink: https://bugs.launchpad.net/ubuntu/+source/linux/+bug/2087983

...
```

SRU cover letter

The patch should come with a *cover letter* that has both a short link to the SRU bug and a copy of the *SRU Justification* from the bug. It can be generated using the `--cover-letter` option of the *`git-send-email(1)`*[47] command.

Example cover letter:

```
Subject: [SRU][O/N][PATCH v2 0/1] ALSA: hda/realtek: fix mute/micmute LEDs for a HP
EliteBook 645 G10

BugLink: https://bugs.launchpad.net/bugs/2087983

SRU Justification:

[ Impact ]

Mute/mic LEDs don't function on HP EliteBook 645 G10.

[ Test Plan ]

Test mute and mic LEDs with proposed kernel once patched.

[ Where problems could occur ]
```

(continues on next page)

[47] https://manpages.ubuntu.com/manpages/resolute/en/man1/git-send-email.1.html

(continued from previous page)

```
Unknown regressions in the sound subsystem.

Kernel Engineer (1):
  ALSA: hda/realtek: fix mute/micmute LEDs for a HP EliteBook 645 G10

 sound/pci/hda/patch_realtek.c | 1 +
 1 file changed, 1 insertion(+)
```

If the patchset is a new version of a previous patchset posted on the mailing-list, the cover letter should explain what has changed for this new submission.

If the patchset involved some decisions that were not obvious, it should be explained in the cover letter to ease the review of the patchset. If you choose to send a SAUCE patch instead of the other options, the rationale should be explained in the cover letter.

Cherry-pick or backport

This section describes additional reasons why the Ubuntu Kernel Team won't approve a cherry-pick or backport patch.

Upstream

The patch should be in the mainline[48] or the stable[49] tree. Having the patch in a maintainer subtree is not enough, because the subtree might change. Having the patch in linux-next[50] is bare minimum.

Source

The patches should have a *cherry picked from* or *backported from* line with the appropriate from the upstream. It can be generated using the `-x` option of the *`git-cherry-pick(1)`*[51] command. This line should appear just before your *Signed-off-by* line.

```
(cherry picked from commit 622f21994506e1dac7b8e4e362c8951426e032c5)
```

```
(backported from commit 622f21994506e1dac7b8e4e362c8951426e032c5)
```

In case the upstream source is linux-next, you should explicit state it.

```
(cherry picked from commit 622f21994506e1dac7b8e4e362c8951426e032c5 linux-next)
```

In case the upstream source is one of the stable trees, you should indicate which one the commit belongs to:

```
(cherry picked from commit e0aab7b07a9375337847c9d74a5ec044071e01c8 linux-4.19.y)
```

[48] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/
[49] https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/
[50] https://www.kernel.org/doc/man-pages/linux-next.html
[51] https://manpages.ubuntu.com/manpages/resolute/en/man1/git-cherry-pick.1.html

In case the upstream source is another Ubuntu kernel (even a SAUCE patch), you can explicit it with the name of the source kernel:

```
(cherry picked from commit 622f21994506e1dac7b8e4e362c8951426e032c5 plucky:linux)
```

In case the provenance is anything else, you should explicit the source git tree in full:

```
(cherry picked from commit 622f21994506e1dac7b8e4e362c8951426e032c5 git://git.kernel.org/
pub/scm/linux/kernel/git/broonie/sound.git)
```

Signed-off-by

The patches must have your Signed-off-by (SoB) as the last line, after the upstream cherry-picked line. It can be generated using the `-s` option of the *`git-cherry-pick(1)`*[52] command.

If the patch is from yourself and already has your SoB, a new SoB must be added.

Example:

```
Subject: [PATCH] ufs: ufs_sb_private_info: remove unused s_{2,3}apb fields

BugLink: https://bugs.launchpad.net/ubuntu/oracular/+source/linux/+bug/2087853

These two fields are populated and stored as a "frequently used value"
in ufs_fill_super, but are not used afterwards in the driver.

Moreover, one of the shifts triggers UBSAN: shift-out-of-bounds when
apbshift is 12 because 12 * 3 = 36 and 1 << 36 does not fit in the 32
bit integer used to store the value.

Closes: https://bugs.launchpad.net/ubuntu/+source/linux/+bug/2087853
Signed-off-by: Agathe Porte <agathe.porte@canonical.com>
Signed-off-by: Al Viro <viro@zeniv.linux.org.uk>
(cherry picked from commit 6cfe56fbad32c8c5b50e82d9109413566d691569 linux-next)
Signed-off-by: Agathe Porte <agathe.porte@canonical.com>
```

SAUCE

This section describes additional reasons why the Ubuntu Kernel Team won't approve a SAUCE patch.

SAUCE prefix

The patches must have the `UBUNTU: SAUCE:` prefix.

Example:

```
Subject: UBUNTU: SAUCE: wifi: ath11k: avoid deadlock during regulatory update in ath11k_
regd_update()
```

(continues on next page)

[52] https://manpages.ubuntu.com/manpages/resolute/en/man1/git-cherry-pick.1.html

(continued from previous page)

```
BugLink: https://bugs.launchpad.net/bugs/1995041

...

Signed-off-by: Aaron Ma <aaron.ma@canonical.com>
```

Backport or SAUCE

This section describes additional reasons why the Ubuntu Kernel Team won't approve a SAUCE or backport patch.

Testing

It is very important for patches to have the upstream maintainer(s) review and do wider testing on different types of hardware for various types of scenarios. Even though the patch was tested by the submitter, the tests may be limited to a specific use case and prone to breaking other parts of the kernel affected by this change. In the case of backports, it was not tested upstream for the specific kernel version, therefore it may cause issues.

Maintenance

Maintaining a patch in our tree is not easy. Let's say we include v0.54 of some patch. Later, we want to sync up to the latest version of this patch. It's not easy to simply revert v0.54, because merges could have changed some of the code. Not to mention, there are very few patches like this that provide incremental changes between versions.

Core code impact

If our kernel contains multiple SAUCE patches or backports, it will diverge from the upstream kernel. In case we need help from upstream to solve bugs, we will have to first test if one of these patches does not cause the bug and then ask the community for help.

Merge conflict

It may cause merge conflicts later when someone from upstream changes the same piece of code. If the component is prone to frequent changes upstream, we will have to deal with this a lot and it will require extra effort on our side.

Security concerns

It may open up unforeseen security issues. Not that this does not happen with upstream code, but having the code there reaches a wider audience, and more people are involved in mitigating the issue.

Bug Prone

It may introduce new bugs that have a wider impact due to limited testing, especially if the change affects a component used in many places.

Quality

Not a very common reason, but the patch may not fit into our standards of code quality or may not serve any real purpose.

Lack of time

Maintaining these patches, with all the arguments from above, will be time-consuming on our side, and we don't have the resources to both do this and deliver a stable Linux OS

3.2.2. Kernel variants and snaps

Different kernel variants and branches serve different purposes - long-term support, new hardware enablement, or experimental features. Understanding them helps you choose the right kernel for testing, development, or deployment.

Ubuntu kernel variants and branches

Ubuntu maintains several kernel variants to balance the needs of stability, hardware enablement, and rapid development. This document outlines the primary kernel trees, their roles in the development lifecycle, and the git branching strategies used by the Canonical Kernel Team.

See also

- See the list of *reference topics* (page 36) for more technical details.
- See the Ubuntu kernels from Canonical[53] for general information about Ubuntu Linux kernels.

[53] https://ubuntu.com/kernel

Ubuntu development kernels

Ubuntu releases every 6 months; therefore, the Canonical Kernel Team constantly works on the active Ubuntu development series to integrate the latest Linux kernel features for the release. The development kernels serve as the testing ground for new features, upstream updates, and partner integration before they reach the stable releases.

The following table summarizes the main Ubuntu development kernels:

Kernel tree	Purpose	Stability	Upstream tracking	Notes
linux-unstable (page 51)	Early integration and testing	Highly volatile	Tracks upstream RCs (frequent rebases)	Lacks full Ubuntu integration (AppArmor, NVIDIA, ZFS, etc.)
Ubuntu Kernel Next (UKN) (page 52)	Stable snapshot for integration work	Moderately stable	Snapshot of `linux-unstable`	Read-only; used for partner / integration testing
linux (page 52)	Ubuntu kernel	Relatively stable	Tracks upstream stable releases	Becomes the GA kernel for the Ubuntu release

The `linux-unstable` tree

The linux-unstable tree[54] represents the bleeding edge of Ubuntu kernel development. It is a fast-moving target designed to track the latest upstream Linux developments.

- **Upstream alignment:** It is usually rebased weekly against the latest upstream Release Candidate (`-rcX`) during the most active development period. New patches are also constantly applied to the upstream branch.
- **Volatility:** Patchsets can be removed if problems are found.
- **Purpose:** It serves as the initial landing ground for upstream code and feature integration. It may still lack support for core components (e.g., AppArmor, NVIDIA drivers, ZFS, etc.) and is not guaranteed to have all features necessary to fully support an Ubuntu system.
- **Availability:** As it usually does not have all core components necessary to run a full Ubuntu user-space, it is available only via development PPAs and not in the Ubuntu archive.

[54] https://code.launchpad.net/~ubuntu-kernel/ubuntu/+source/linux/+git/unstable

Ubuntu Kernel Next (UKN)

Because the `linux-unstable` kernel moves too fast for larger integration work, the Canonical Kernel Team maintains the Ubuntu Kernel Next[55] tree. UKN integration tree designed to bridge the gap between the volatility of `linux-unstable` and the stability required for partner development.

- **Workflow:** It is a periodic, read-only snapshot of `linux-unstable`.
- **Integration:** It provides a stable code base for integration work into the next development kernel. Integration issues (e.g., conflicts) can be spotted and fixed earlier.

The `linux` tree

Once a `linux-unstable` kernel version includes support for all core components and is deemed relatively stable, it is moved to the `linux` tree.

- **Upstream alignment:** Similar to the `linux-unstable` tree, it usually follows the latest upstream release. However, rebases are less frequent as some level of stability needs to be maintained. During a development cycle, it is expected to be updated to all `major.minor` upstream releases until the last version available before the Ubuntu release GA.
- **Purpose:** The `linux` kernel is the Ubuntu generic kernel which will be available as the default choice for most installations, and will be the base for all derivatives and custom kernels based on the same upstream `major.minor` version.

Kernel flavours

A **kernel flavour** is a distinct binary produced from a single kernel source package, sharing the same source code and patches as all other flavours from that package. The only differences between flavours are the configuration options applied at build time.

For example, the Ubuntu `linux` source package produces both `linux-generic` and `linux-lowlatency` binaries from the same tree, each with a different `.config` file tailored to its target workload.

Note:

This config-only distinction applies within a single source package. Kernels with separate source packages (such as `linux-aws` or `linux-raspi`) are *optimized kernels* (page 54) and may carry additional patches beyond config changes.

For the generic kernel, the current Ubuntu flavours and their target workloads are:

[55] https://canonical-kteam-docs.readthedocs-hosted.com/public/reference/kernels/uknext/uknext.html

Flavour	Architectures	Target workload
`generic`	amd64, arm64, armhf, ppc64el, riscv64, s390x	Default for desktop and server systems
`generic-64k`	arm64	Desktop and server systems requiring 64K memory pages
`lowlatency`	amd64, arm64	Latency-sensitive workloads (audio production, real-time data)
`lowlatency-64k`	arm64	Combines lowlatency tuning with 64K memory pages
`realtime`	amd64, arm64	Deterministic latency via the PREEMPT_RT patchset

Derivative kernels might support only a subset of these flavours, and might also add a portion of the derivative name to the flavour identifier. For example, the raspi kernel has flavours `arm64-raspi` and `arm64-raspi-realtime`.

Migration of kernel flavours

At times the functionality of one kernel flavour may end up being entirely superseded by the functionality of another, making the original kernel flavour no longer necessary. In this scenario, users of the original kernel flavour can be automatically migrated to the superseding one.

The following table provides a summary of these now-obsolete kernel flavours, and their replacements:

Obsoleted in release	Obsolete flavour	Replacement flavour	Notes
23.10	`kvm`	`virtual`	The `virtual` kernel flavour received support for `virtio_scsi` and was configured to boot without an initramfs, allowing it to replace `linux-kvm`.
24.04	`generic lpae`	`generic`	The `generic` flavour enabled the armhf Large Physical Address Extension by default. (LP: #2025265)[56]
24.04	`laptop-23.10`	`generic`	The `laptop-23.10` flavour was intended to support select ARM-based laptops, and in 24.04 the `generic` flavour received support for these systems as well.
24.04	`intel-iotg`	`generic-hwe-24.04`	All unique functionality of the `intel-iotg` flavour is included in the Noble `generic` kernel flavour. (LP: #2128721)[57]

Optimized kernels

In addition to the generic kernel, Canonical also provides optimized kernels which are derived from the generic kernel.

See also

- Ubuntu kernel variants from Canonical[58]

- **Purpose:** Kernels that have their configuration, hardware support, or additional features optimized for a variety of hardware platforms and workloads. Examples include kernels optimized for the Raspberry Pi ARM board (`linux-raspi`), for running as guests on the major public cloud providers, for IoT devices, etc.
- **Release and update cadence:** Optimized kernels are released and receive security updates with the same cadence as the Ubuntu generic kernel upon which they are based.

OEM kernels

OEM kernels are optimized derivatives designed specifically for Original Equipment Manufacturer (OEM) projects to support hardware pre-installed with Ubuntu.

See also

- *OEM kernels* (page 61)

- **Purpose:** They address unique timelines and hardware needs that may not align with the generic kernel release cadence and scope.
- **Staging strategy:** OEM kernels often serve as a staging area. Modifications made here are intended to be merged into the generic Ubuntu kernel in subsequent releases.

Production and enablement kernels

Once an Ubuntu version is released to General Availability (GA), all kernels available in that release become production-ready and stable. They are maintained for the duration of the Ubuntu release support period and receive security updates and bug fixes via *Stable Release Updates (SRU)* (page 81).

[56] https://bugs.launchpad.net/bugs/2025265
[57] https://bugs.launchpad.net/bugs/2128721
[58] https://ubuntu.com/kernel/variants

HWE (Hardware Enablement) kernels

Hardware Enablement (HWE) kernels are available for Ubuntu Long Term Support (LTS) releases. They provide a way for LTS users to consume newer kernel versions and receive support for newer hardware (such as the graphics stack).

See also

- *HWE kernels* (page 56)

- **Rolling release:** HWE kernels are "rolling" in nature; they are typically backported from subsequent interim or LTS Ubuntu releases to the previous LTS.
- **Lifecycle:** HWE kernels are supported until the next HWE stack is released, at which point users are encouraged to upgrade to the newer version.

Git branching strategy

When working with Ubuntu kernel repositories, you will primarily interact with two key branches.

`master-next`/`main-next` branches

The `master-next` or `main-next` branches serve as the staging area for the **next** Stable Release Update (SRU).

- **Purpose:** Commits intended for the next update are applied here first after being reviewed and acknowledged by the Canonical Kernel Team. This branch will also contain commits for the kernel builds that are published for testing in the `-proposed` pocket of the Ubuntu archive.
- **Flow:** Patches land in `master-next` for integration and testing before being promoted to `master` after the kernel SRU update is published to `-updates`.

`master`/`main` branches

The `master` or `main` branches represent the current state of the kernel source as it exists in the `-updates` pocket of the Ubuntu archive. It contains the linear history of all the stable releases published for that kernel.

HWE kernels

This document provides some reference information about Hardware Enablement (HWE) kernels: the support life cycle, current kernel in development, the next planned Ubuntu base kernel version, kernel source code, and how to install the HWE kernels for use on your machine.

Support life cycle for HWE kernels

HWE kernels are only enabled on Ubuntu long-term support (LTS) releases, and have similar life cycles as their newer Ubuntu kernel counterparts. They will typically get rolled off to the next HWE kernel once a new Ubuntu series is released (until the next LTS).

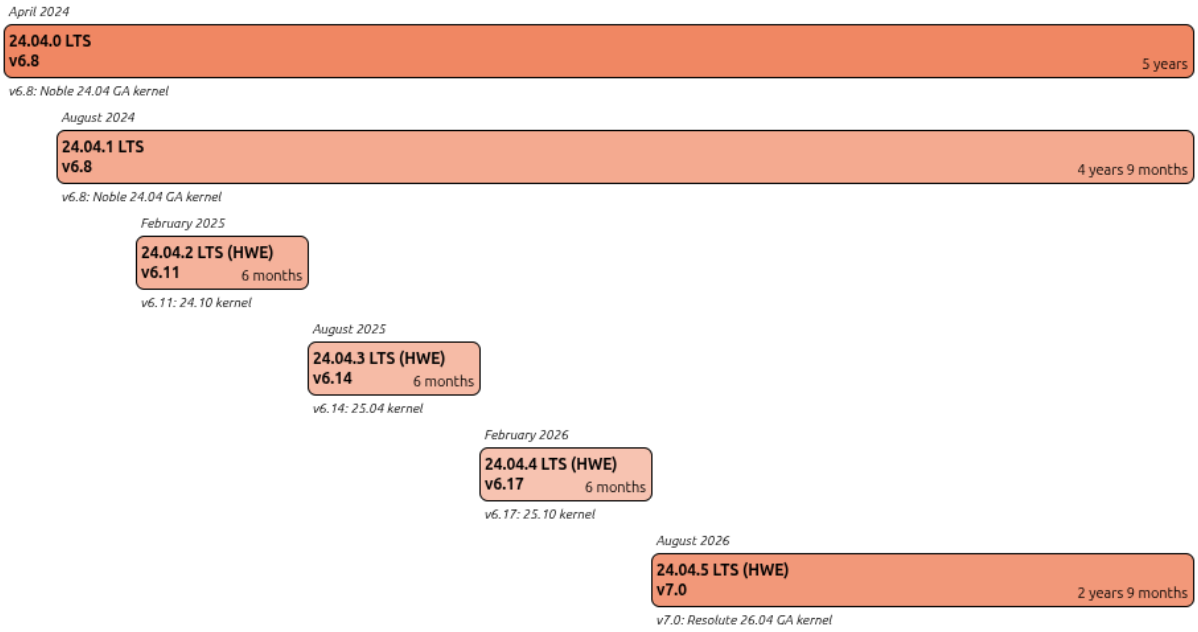

Fig. 1: Example of HWE kernel release cycle for Noble Numbat

The table below summarizes the support life cycle, development and release schedule, end-of-life (EOL) and Extended Security Maintenance (ESM) dates for supported and upcoming HWE kernels.

Table 1: HWE kernel life cycle and package details

Ubuntu series	Ubuntu version	Kernel version	Key dates	
Resolute Raccoon	*26.04.0 LTS*	*7.0*	*Release*	*April 20*
			EOL	*April 20*
			ESM	*March 2*
Noble Numbat	24.04.5 LTS (HWE)	*7.0*	*Edge* (page 60)	*June 20*
			Release	*August*

continues on nex

Table 1 – continued from previous page

Ubuntu series	Ubuntu version	Kernel version	Key dates	
			EOL	*April 20*
			ESM	*March 2*
	24.04.4 LTS (HWE)	6.17	Release	Februar
			EOL	July 202
	24.04.0 LTS	6.8	Release	April 20
			EOL	April 20
			ESM	March 2
Jammy Jellyfish	22.04.5 LTS (HWE)	6.8	Release	August
			EOL	April 20
			ESM	March 2
	22.04.0 LTS	5.15	Release	April 20
			EOL	April 20
			ESM	March 2
Focal Fossa	20.04.5 LTS (HWE)	5.15	Release	August
			EOL	April 20
			ESM	April 20
	20.04.0 LTS	5.4	Release	April 20
			EOL	April 20
			ESM	April 20
Bionic Beaver	18.04.5 LTS (HWE)	5.4	Release	August
			EOL	April 20
			ESM	April 20
	18.04.0 LTS	4.15	Release	April 20
			EOL	April 20
			ESM	April 20
Xenial Xerus	16.04.5 LTS (HWE)	4.15	Release	August
			EOL	April 20
			ESM	April 20
	16.04.0 LTS	4.4	Release	April 20
			EOL	April 20
			ESM	April 20

continues on nex

Table 1 – continued from previous page

Ubuntu series	Ubuntu version	Kernel version	Key dates	
Trusty Tahr	14.04.5 LTS (HWE)	4.15	Release	August
			EOL	April 20
			ESM	April 20

Note:

HWE kernels that have reached EOL and are no longer under the ESM or Legacy add-on support phase are excluded from the table above. See the Ubuntu kernel release cycle[59] for information.

[59] https://ubuntu.com/about/release-cycle#ubuntu-kernel-release-cycle

Installing a HWE kernel

24.04 LTS

22.04 LTS

20.04 LTS

18.04 LTS

16.04 LTS

14.04 LTS

Ubuntu 24.04 LTS (Noble Numbat)

By default, Ubuntu Desktop installations of 24.04 default to tracking the HWE stack. Server installations will default to the general availability (GA) kernel and provide the HWE kernel as an option.

Desktop:

```
sudo apt-get install --install-recommends linux-generic-hwe-24.04
```

Server:

```
sudo apt-get install --install-recommends linux-generic-hwe-24.04
```

Ubuntu 22.04 LTS (Jammy Jellyfish)

By default, Ubuntu Desktop installations of 22.04 default to tracking the HWE stack. Server installations will default to the GA kernel and provide the HWE kernel as an option.

Desktop:

```
sudo apt-get install --install-recommends linux-generic-hwe-22.04
```

Server:

```
sudo apt-get install --install-recommends linux-generic-hwe-22.04
```

Ubuntu 20.04 LTS (Focal Fossa)

By default, Ubuntu Desktop installations of 20.04 default to tracking the HWE stack. Server installations will default to the GA kernel and provide the HWE kernel as an option.

Desktop:

```
sudo apt-get install --install-recommends linux-generic-hwe-20.04
```

Server:

```
sudo apt-get install --install-recommends linux-generic-hwe-20.04
```

Ubuntu 18.04 LTS (Bionic Beaver)

By default, Ubuntu Desktop installations of 18.04.2 and newer point releases will ship with an updated kernel and X stack. Server installations will default to the GA kernel and provide the HWE kernel as an option.

Desktop:

```
sudo apt-get install --install-recommends linux-generic-hwe-18.04 xserver-xorg-hwe-18.04
```

Server:

```
sudo apt-get install --install-recommends linux-generic-hwe-18.04
```

Ubuntu 16.04 LTS (Xenial Xerus)

By default, Ubuntu Desktop installations of 16.04.2 and newer point releases will ship with an updated kernel and X stack. Server installations will default to the GA kernel and provide the HWE kernel as an option.

Desktop:

```
sudo apt-get install --install-recommends linux-generic-hwe-16.04 xserver-xorg-hwe-16.04
```

Server:

```
sudo apt-get install --install-recommends linux-generic-hwe-16.04
```

Ubuntu 14.04 LTS (Trusty Tahr)

By default, Ubuntu Desktop installations of 14.04.2 and newer point releases will ship with an updated kernel and X stack. Server installations will default to the GA kernel and provide the HWE kernel as an option.

Desktop:

```
sudo apt-get install --install-recommends linux-generic-lts-xenial xserver-xorg-core-lts-
xenial xserver-xorg-lts-xenial xserver-xorg-video-all-lts-xenial xserver-xorg-input-all-
lts-xenial libwayland-egl1-mesa-lts-xenial
```

Server:

```
sudo apt-get install --install-recommends linux-generic-lts-xenial
```

Installing an edge HWE kernel

Note:

Edge variants of HWE kernels are considered to be in development mode and are not supported. These edge variants may have missing components, missing *DKMS*, or contain bugs. Edge HWE kernels are not stable releases and should not be used in a production environment.

You can get early access to the next HWE kernel - that will be shipped with a newer kernel version - by installing the `-edge` variant.

To install the latest edge kernel variant for Ubuntu 24.04 LTS, run:

```
sudo apt-get install --install-recommends linux-generic-hwe-24.04-edge
```

For more information, see *edge kernel* for more information.

Reporting bugs on HWE kernels

There are two recommended approaches to report a bug against a HWE kernel package.

1. Using the `apport-bug` command.

   ```
   apport-bug linux
   ```

2. Through the "Report a bug" form for the `linux` package in Launchpad: https://bugs.launchpad.net/ubuntu/+source/linux/+filebug.

Related topics

- See the Stable Updates Cycles[60] for the dates of the last day for kernel patches (for HWE kernels) for each stable update cycle.
- See the Ubuntu kernel release cycle[61] for more details about the kernel support life cycle, including the ESM support phase.
- See the Ubuntu kernel life cycle and enablement stack[62] for more details about HWE kernels and their support status.

[60] https://kernel.ubuntu.com/
[61] https://ubuntu.com/about/release-cycle#ubuntu-kernel-release-cycle
[62] https://ubuntu.com/kernel/lifecycle

OEM kernels

The OEM kernel is an optimized derivative Ubuntu kernel, designed specifically for use in Original Equipment Manufacturer (OEM) projects. OEM kernel variants are typically developed and customized for hardware that will be pre-installed with Ubuntu.

This document provides some reference information about OEM kernels: the support life cycle for rolling releases, current kernel in development, the next planned generic Ubuntu kernel version, kernel source code, and how to install the OEM kernels for use on your machine.

Support life cycle for OEM kernels

OEM kernels have shorter life cycles than their generic Ubuntu kernel counterparts. They will typically get rolled off to the next HWE kernel once all the fixes have been forward-ported.

The table below summarizes the support life cycle, development and stable release schedules, EOL dates, and kernel migration target for supported and upcoming OEM kernels.

Table 2: OEM kernel life cycle and package details

Kernel and Ubuntu version	Source code and meta package	Key dates		Migration target
6.11 24.04 LTS (Noble)	s: linux-oem-6.11[63] m: linux-oem-24.04b	Devel	August 2024	linux-oem-6.14[64]
		Stable	November 2024	
		EOL	July 2025	
6.14 24.04 LTS (Noble)	s: linux-oem-6.14[65] m: linux-oem-24.04c	Devel	March 2025	linux-oem-6.17[66]
		Stable	April 2025	
		EOL	February 2026	
6.17 24.04 LTS (Noble)	s: linux-oem-6.17[67] m: linux-oem-24.04d	Devel	September 2025	linux-hwe-7.0
		Stable	October 2025	
		EOL	July 2026	

[63] https://kernel.ubuntu.com/forgejo/kernel/noble-linux-oem/src/branch/oem-6.11-next
[64] https://kernel.ubuntu.com/forgejo/kernel/noble-linux-oem/src/branch/oem-6.14-next
[65] https://kernel.ubuntu.com/forgejo/kernel/noble-linux-oem/src/branch/oem-6.14-next
[66] https://kernel.ubuntu.com/forgejo/kernel/noble-linux-oem/src/branch/oem-6.17-next
[67] https://kernel.ubuntu.com/forgejo/kernel/noble-linux-oem/src/branch/oem-6.17-next

Note:

- OEM kernels are *supported in LTS releases only*. Interim releases are *unsupported*.
- OEM kernels that have reached end-of-life (EOL) are excluded from the table above.

Selection guidelines for OEM kernels

In general, we need at least three OEM kernels for each Ubuntu LTS release to support our OEM projects.

- First OEM kernel

  Released early in the Ubuntu LTS cycle to meet the needs of OEM projects that require the latest Ubuntu LTS release. This OEM kernel is based on the Ubuntu LTS kernel, with the same kernel version. Normally, this will be migrated to the *.2 HWE (Hardware Enablement) kernel.

- Second OEM kernel

  The second OEM kernel is typically released in the second half of the same year as the Ubuntu LTS release, and it is for supporting the latest Intel and AMD hardware platforms. It could be based on either the xx.10 Ubuntu kernel or the upstream LTS kernel, and may later migrate to the *.3 or *.4 HWE kernel.

- Third OEM kernel

  The final OEM kernel introduced in an LTS cycle to support the latest hardware near the end of the release timeline. This will be migrated to the *.5 HWE kernel.

These guidelines serve as a reference only and may be adjusted as necessary to accommodate hardware schedules. Additional OEM kernels may be introduced to support cutting-edge hardware designs and to meet the time-to-market requirements of OEM partners.

Downloading and installing OEM kernels

To view and/or download the source code for OEM kernels, go to the kernel repository (e.g. “s: linux-oem-6.5”) listed in the “Source code and meta package” column in the *OEM kernel life cycle and package details* (page 61) table.

To install an OEM kernel, use the meta-package name (e.g. “m: linux-oem-22.04d”) for the kernel version listed in the “Source code and meta package” column in the *OEM kernel life cycle and package details* (page 61) table. For example, to install OEM kernel version 6.8, run:

```
apt install linux-oem-24.04a
```

Tip:

Use the meta-package name when installing the OEM kernel to ensure that you continue receiving automated updates even after the OEM kernel is rolled off to the target migration kernel.

Reporting bugs on OEM kernels

There are two recommended approaches to report a bug against an OEM kernel package.

1. Using the `apport-bug` command with the OEM kernel package name. For example, to report a bug for the "linux-oem-6.8" kernel, run:

   ```
   apport-bug linux-oem-6.8
   ```

2. Through the "Report a bug" form in Launchpad. For example, to report a bug for the "linux-oem-6.8" kernel, go to https://bugs.launchpad.net/ubuntu/+source/linux-oem-6.8/+filebug.

Related topics

- See the Stable Updates Cycles[68] for the dates of the last day for kernel patches (for OEM kernels) for each stable update cycle.
- See the Forgejo repositories for noble-linux-oem[69] for pending pull requests and details on the patches that are merged and released for each OEM kernel (currently only open to partners).

Mainline kernel archive

Warning:

Mainline build kernels are **not supported** and **never receive security updates**. Mainline build kernels are **not suitable for production** use. **Remove** them once you've finished testing. **Support is not provided** to users who are running mainline build kernels outside of diagnostic uses.

The Ubuntu mainline kernel archive[70] contains mainline builds published on a best effort basis by the Kernel Team at Canonical.

To install and test a build, see *How to test a bug with a mainline kernel* (page 20).

Directory layout

Location	Contents
`v<version>/`	A build for a tagged Linux release or release candidate
`v<version>/<architecture>/`	Packages, checksums, status, and logs for one architecture

[68] https://kernel.ubuntu.com/
[69] https://kernel.ubuntu.com/forgejo/kernel/noble-linux-oem/pulls
[70] https://kernel.ubuntu.com/mainline/?C=N;O=D

A version directory identifies the source commit and reports the build status for each architecture. The architectures and kernel flavors available depend on the build.

Files in an architecture directory

File	Purpose
`*.deb`	Installable kernel packages
`CHECKSUMS`	Checksums for the published files
`CHECKSUMS.gpg`	OpenPGP signature for `CHECKSUMS`
`log`	Complete build log
`status`	Numeric build status
`summary.yaml`	Build result and package summary
`REBUILD`	Information about a rebuilt package set, when present
`self-tests/`	Build self-test results, when present

Package files

Package names identify the kernel version, flavor, build, and architecture. A typical `generic` package set for `amd64` contains:

```
linux-headers-<version>_<version>.<build>_all.deb
linux-headers-<version>-generic_<version>.<build>_amd64.deb
linux-image-unsigned-<version>-generic_<version>.<build>_amd64.deb
linux-modules-<version>-generic_<version>.<build>_amd64.deb
```

Package name	Contents
`linux-headers-<version>_<version>.<build>_all.deb`	Headers shared by all architectures and flavors
`linux-headers-<version>-<flavor>_<version>.<build>_<architecture>.deb`	Headers for one architecture and flavor
`linux-image[-unsigned]-<version>-<flavor>_<version>.<build>_<architecture>.deb`	Bootable kernel image
`linux-modules-<version>-<flavor>_<version>.<build>_<architecture>.deb`	Loadable kernel modules

The exact package set varies by architecture and build. Some architectures use `linux-image` instead of `linux-image-unsigned`, and some provide more than one flavor.

Signing key

The mainline builder signs `CHECKSUMS` with the following key:

```
Fingerprint: 60AA 7B6F 3043 4AE6 8E56 9963 E50C 6A09 17C6 22B0
User ID: Kernel PPA <kernel-ppa@canonical.com>
```

Validate the key against its record on the Ubuntu keyserver before relying on the signature.

Ubuntu to mainline version mapping

The Ubuntu kernel version mappings[71] table shows which upstream release an Ubuntu kernel is based on. An Ubuntu kernel isn't identical to the corresponding mainline kernel because Ubuntu applies its own patches and configuration.

Related topics

- *How to test a bug with a mainline kernel* (page 20)
- *About mainline kernel builds* (page 77)
- *Ubuntu kernel variants and branches* (page 50)

 kernel snap packaging

Snap lifecycle

This document aims to document the lifecycle of the various kernel snap forms.

Snap builds

The majority of kernels with snaps are consumed both as Debian packages and snaps. To reduce testing requirements and streamline production the same binaries are used for both forms. Due to the Ubuntu requirement for source to be included with the binaries, it is simplest to generate the binaries as part of generating the Debian binary packages and repackage those into snaps where needed. Where a kernel is to be signed this is performed during the packaging process in those Debian package builds.

Workflow support

Kernel snaps are represented as a separate phase in the kernel workflow. There is a Workflow Tracker for each Debian kernel build, and a separate subordinate tracker for the snap kernel build. The snap tracker will have the Debian trackers as its parent and will proceed once that Debian tracker is complete.

[71] https://people.canonical.com/~kernel/info/kernel-version-map.html

Debian pocket usage

The Debian package builds flow through their own lifecycle proceeding from the `build` location, to `proposed`, and onward to `updates` and `security` as testing allows.

Kernels in the `build` location are unsigned and intended for simple boot testing or for testing for signing compliance. Kernels in `proposed` are signed (if applicable) and formal candidates for regression and certification testing.

Snap risk usage

Risk (channel)	Purpose	Build source
`edge`	Snaps are unsigned and intended for simple boot testing.	Kernel snaps are built from Debian binaries in the `build` location.
`beta`	Snaps are signed and intended for certification testing.	Kernel snaps are built from Debian binaries in the `proposed` location.

Track usage

We make heavy use of store tracks to separate series-specific snaps from each other.

For Ubuntu LTS releases which align with Ubuntu Core releases those tracks are typically the Ubuntu Core release years (e.g. `24`). For interim Ubuntu releases these are the full release name (e.g. `24.10`).

Where a series has a *hardware enablement kernel* (page 56), those are placed on the HWE specific tracks (e.g. `24-hwe`).

Unsigned kernels

Unsigned kernels such as the `pi-kernel` will be directly generated in the Debian `main` package. The `linux-image` packages are consumed and `ubuntu-core-initramfs` used to generate an initramfs to accompany it. These are packaged up along with any required firmware.

Signed kernels

Signed kernel such as the `pc-kernel` will be generated in the Debian `main` package, and passed through the signing pipeline as part of the Debian `signed` package. The `linux-image` packages (now generated by the `signed` package) are consumed and `ubuntu-core-initramfs` used to generate an initramfs to accompany it. These are packaged up along with any required firmware. `ubuntu-core-initramfs` is installed and invoked as part of the kernel handling to convert the existing `vmlinux-<verflav>` image into a `kernel.efi-<verflav>` image.

Kernel UKIs

For kernels use cases which require measurement we also produce Unified Kernel Images (UKIs). That is a bootable PE executable which contains the kernel binary, an initramfs, and the kernel command line. This UKI is generated in the `linux-signed` package through use of an additional mode of the `ubuntu-core-initramfs` tooling. This process produces a single binary and is signed after it is combined via the signing pipeline.

Stubble kernels

On `arm64` we have an additional problem. For a number of platforms, the `dtb` is not correctly supplied by the firmware. To handle these cases a `stubble` wrapper is used to detect those platforms and to inject the appropriate `dtb` as appropriate, then handing off control to the wrapped kernel image. The kernel image is taken from the Debian `linux-image` package in the normal way. The `stubble-kernel` package is installed and invoked as part of the kernel handling to convert the existing `vmlinuz-<verflav>` image into a `stubble.efi-<verflav>` image.

Snap workflow lifecycle

The snap workflow lifecycle runs in parallel to and interlocked with the Debian Workflow lifecycle. This ensures that the snap workflow waits for the prerequisite binaries. It also ensures that testing of both Debian packages and snap must be complete before they can progress further, finally ensuring that the Debian packages and Snaps release together.

Unsigned build

When a Debian kernel build completes in the `build` location the `edge` build of the Snap is triggered. This causes an auto-crank of the snap which the `snapcraft.yaml` configuration, and kicks off builds against the appropriate snap build recipe. This causes the kernel to be processed into a snap and uploaded to the snap-store. The store will automatically publish this onto the `edge` channel for testing.

Early testing

Once published we trigger any available early testing. This includes `boot-testing`, `abi-testing` and `signing-signoff`. Once each of these is successfully completed the Debian package may progress into the signing pipeline and on into its `proposed` location.

Proposed build

Once the Debian kernel is in its `proposed` location a second auto-crank is triggered to process the kernel into a snap via a second snap recipe. This causes the kernel to be uploaded to the `beta` channel ready for wider formal testing.

For signed kernels this ensures the snap on the `beta` channel has a signed payload. We also regenerate the snap for unsigned kernel, while this may seem redundant it allows us to perform experimental builds only to `edge` without disrupting the workflow once the build has progressed to `beta`.

Testing

Once we have a snap on the `beta` channel formal testing is triggered. This includes `certification-testing` for the snap. Once this testing is complete the snap will be promoted to the `candidate` channel.

QA testing

Once we have a formal candidate snap this may be sent for further acceptance testing in QA. Testing for the Debian package and snap are combined and gate the further promotion of both. Promotion is further gated by any applicable signoff tasks.

Release

Once all gating factors, testing, signoff, and cycle boundaries are satisfied the snap will be promoted to the `stable` channel; this occurs in lock-step with the promotion of the Debian package to `updates`. The Debian package may then promote further to `security` but there is no equivalent channel for snaps.

3.2.3. Kernel release and maintenance

 sru tracking bug release pockets

Releasing an SRU kernel

If you need to expedite the release of a kernel build as part of the SRU cycle process but you are unable to get hold of a Kernel Archive Admin (AA), you can use the following recipe.

Prepare (Kernel)

To release a kernel it must be in calling to be released via the `promote-to-updates` task. Liaise with the Kernel Stable team to get the testing and into an appropriate state to cause the tracker, a Launchpad bug against kernel-sru-workflow project, to ask to release.

We can then form a release command for the Archive Admins to execute.

```
./copy-package-kernel --from-route proposed --to-route updates --tracker <tracker>
```

Execute (Archive Admin)

Kernels are promoted using the `copy-package-kernel` command from ubuntu-archive-tools[72]. This command makes use of the kernel-team databases to identify the source and destination for the copies. It also has internal validation to confirm that the package collection in the destination pocket will be internally consistent by versions after the copies.

The kernel team will bring the bones of a `copy-package-kernel` command for the required promotion for execution.

First, check that the tracker provided with the command is requesting to be released. There should be a task against `promote-to-updates` which should be in "Confirmed" state.

- If this is *not* the case, then this should be handed back to the kernel-team for resolution.
- If it is, assign that task to yourself, and move it to "In Progress".

The supplied command can be safely run with the `-n` argument to see what it would do; you can also add the `–verbose` option to dump out the equivalent `copy-package` kernel commands for validation.

```
user@host:~$ /copy-package-kernel --from-route proposed --to-route updates
--tracker 2127318 -n --verbose
```

[72] https://code.launchpad.net/ubuntu-archive-tools

```
copy-tracker: 2127318 (focal:linux-iot) proposed updates
  Versions:   -final-        -was-
    main      5.4.0-1056.59  5.4.0-1055.58
    meta      5.4.0.1056.54  5.4.0.1055.53
    signed    5.4.0-1056.59  5.4.0-1055.58
  Copies:
    linux-iot                                   5.4.0-1056.59
ppa:canonical-kernel-esm/ubuntu/proposed:Release -> ppa:ubuntu-esm/ubuntu/esm-infra-
security:Release ... dry-run
      copy-package -n --include-binaries --auto-approve \
        --from ppa:canonical-kernel-esm/ubuntu/proposed         --from-suite focal \
        --to ppa:ubuntu-esm/ubuntu/esm-infra-security --to-suite focal \
        --version 5.4.0-1056.59 linux-iot
    linux-meta-iot                              5.4.0.1056.54
ppa:canonical-kernel-esm/ubuntu/proposed:Release -> ppa:ubuntu-esm/ubuntu/esm-infra-
security:Release ... dry-run
      copy-package -n --include-binaries --auto-approve \
        --from ppa:canonical-kernel-esm/ubuntu/proposed --from-suite focal \
        --to ppa:ubuntu-esm/ubuntu/esm-infra-security --to-suite focal \
        --version 5.4.0.1056.54 linux-meta-iot
    linux-signed-iot                            5.4.0-1056.59
ppa:canonical-kernel-esm/ubuntu/proposed:Release -> ppa:ubuntu-esm/ubuntu/esm-infra-
security:Release ... dry-run
      copy-package -n --include-binaries --auto-approve \
        --from ppa:canonical-kernel-esm/ubuntu/proposed --from-suite focal \
        --to ppa:ubuntu-esm/ubuntu/esm-infra-security --to-suite focal \
        --version 5.4.0-1056.59 linux-signed-iot
```

If you are happy with the output, rerun it with `-y` to execute it. It is safe to run the command more than once as it is idempotent. Running it a second time will confirm the copies have been accepted by Launchpad.

When the copy completes external tooling should manage the state of `promote-to-updates` through to “Fix Released”.

 sru respin release pockets tracking bug

Kernel rollback

When a kernel is found to be so bad that the only option is to withdraw it from the archive, the typical approach is to replace it with the previous kernel. This will not fix anything for those who have already upgraded their kernel, but can further prevent other becoming affected.

There have also been cases where upgrades are no longer possible with an update and reverting the update can restore the ability to upgrade.

This recipe will guide you through identifying the kernel versions to rollback to, and how to produce a recipe for a member of the Archive Admins (AA) to follow to perform the required rollback.

Prerequisites

Install the tooling needed for the various kernel workflow playbook items:

```
pipx install ckt_workflow@git+https://git.launchpad.net/~apw/+git/ckt_workflow@latest
```

Note:

The repository location for this project is subject to change.

Prepare (Kernel)

In order to revert the kernel in a pocket we need to identify an earlier version of a good kernel. We typically identify this via a previous cycle or spin number, and handle.

We want to remove any existing kernel package publications for this handle, and then copy back earlier publications. Use the `revert-kernels-to-spin` command to generate AA commands to effectuate these:

```
/revert-kernels-to-spin --spin s2025.09.15 --handle noble:linux \
        --pocket updates --reason "Causing upgrade issues" --yes
```

Validation (Kernel)

The kernel team should review the versions that the revert is settled on, as shown in the revert output. It is vital to confirm that any LRM or signed respins have been included. Where there is a later version the tooling will emit a warning as below. The versions should be updated manually in this case.

```
# jammy:linux-azure: spin=s2025.10.13-2 full_versions={'lrm': '5.15.0-1102.111+1', 'main':
'5.15.0-1102.111', 'meta': '5.15.0.1102.100', 'signed': '5.15.0-1102.111'}
[...]
# WARNING: linux-restricted-modules-azure looks to have a repin not in the spin (5.15.0-
1102.111+1)
```

Execute (Archive Admins)

In order to revert a kernel, all of the packages which make up a kernel (e.g. `linux`, `linux-signed`, `linux-meta`, `linux-restricted-modules` etc) must be reverted together. The kernel team will identify these packages and the versions of which are faulty, and the older package versions which should be reinstated. They will use *kernel tooling* (page 71) to generate `remove-package` and `copy-package` commands to roll-back the published versions of these packages.

These will consist of two groups of commands: an initial set of removals, one per package, plus a second set of copies for the same packages. While it is possible for the two sets to differ, additional consideration is necessary in this case. For example:

```
remove-package linux                       --version 6.8.0-88.89   --archive ubuntu --
suite      noble-updates --removal-comment='Causing upgrade issues' -y
remove-package linux-meta                  --version 6.8.0-88.89   --archive ubuntu --
suite      noble-updates --removal-comment='Causing upgrade issues' -y
remove-package linux-restricted-modules    --version 6.8.0-88.89+1 --archive ubuntu --
suite      noble-updates --removal-comment='Causing upgrade issues' -y
remove-package linux-restricted-signatures --version 6.8.0-88.89+1 --archive ubuntu --
suite      noble-updates --removal-comment='Causing upgrade issues' -y
remove-package linux-signed                --version 6.8.0-88.89   --archive ubuntu --
suite      noble-updates --removal-comment='Causing upgrade issues' -y
copy-package   linux                       --version 6.8.0-87.88   --from    ubuntu --
from-suite noble-updates --include-binaries --force-same-destination --auto-accept -y
copy-package   linux-meta                  --version 6.8.0-87.88   --from    ubuntu --
from-suite noble-updates --include-binaries --force-same-destination --auto-accept -y
copy-package   linux-restricted-modules    --version 6.8.0-87.88+1 --from    ubuntu --
from-suite noble-updates --include-binaries --force-same-destination --auto-accept -y
copy-package   linux-restricted-signatures --version 6.8.0-87.88+1 --from    ubuntu --
from-suite noble-updates --include-binaries --force-same-destination --auto-accept -y
copy-package   linux-signed                --version 6.8.0-87.88   --from    ubuntu --
from-suite noble-updates --include-binaries --force-same-destination --auto-accept -y
```

Note:

If the `security` pocket is later than the newly rolled-back kernel version in `updates`, the same procedure should be applied to the `security` pocket.

Execute (IS)

Where the affected series include those in ESM, removals from the primary PPAs the packages will also need removing from the repository. Take a list of the removed packages in the ESM series to mattermost ~IS channel, and request for them to be removed.

A sample removal command is shown below:

```
reprepro --basedir /srv/esm-archive/fips-updates/reprepro/ removesrc \
      focal-infra-security openssh '1:9.6p1-3ubuntu13.7+Fips1'
```

Tip:

If you are unable to contact the Canonical IS team directly, liaise with a member of the Canonical Kernel team to request the removal of said packages.

3.2.4. Privileges

Understand the criteria and process to apply for Ubuntu kernel and DKMS package upload rights.

Kernel upload rights

Those who are allowed to upload the kernel to the Ubuntu archive have a serious responsibility.

To obtain per package upload rights for the Ubuntu kernel, you need to apply to become a member of the ubuntu-kernel-uploaders[73] team in Launchpad. People can join this team only after going through a thorough application and review process.

The sections below describe the general member profile of the ubuntu-kernel-uploaders[74] team, as well as the application process.

Member profile

Below is the general profile for those having per package upload rights for the Ubuntu kernel.

- Generally have commit access to the Ubuntu kernel git repositories[75].
- Actively follow and participate in discussions and patch reviews on the Ubuntu kernel-team mailing list[76].
- Are collectively responsible for the maintenance of packages in the Ubuntu kernel package set for all supported releases as well as the development release.
- Have a strong working knowledge of kernel packaging concepts and techniques, refined through experience.
- Have a strong working knowledge of Ubuntu project procedures, especially those related to the release process and support commitments, and an understanding of the reasons why they exist.
- Have a history of substantial and direct contribution to the distribution, particularly to kernel-related packages.
- Feel a sense of personal responsibility for the quality of Ubuntu releases and for the satisfaction of Ubuntu users.
- Exercise great care in their work, with the understanding that their efforts have a direct impact on others, including:
  - every Ubuntu user;
  - the Ubuntu release team;
  - corporate partners who provide support for Ubuntu.

[73] https://launchpad.net/~ubuntu-kernel-uploaders
[74] https://launchpad.net/~ubuntu-kernel-uploaders
[75] https://kernel.ubuntu.com/git
[76] https://lists.ubuntu.com/mailman/listinfo/kernel-team

Application process

As alluded to in the member profile above, membership consideration for the ubuntu-kernel-uploaders[77] team adheres to a strict policy. Anyone considering applying should align with the general profile outlined in the previous section and meet the criteria listed below:

1. A thorough understanding of the Ubuntu kernel patch submission process:
    a. Demonstrates an understanding of this process by having submitted multiple patches which were accepted over a six-month development cycle.
    b. Demonstrates an understanding of this process by having reviewed and multiple patches over a six-month development cycle.
2. A thorough understanding of the Ubuntu release cycle and associated milestone and freeze dates.
3. A thorough understanding of the Ubuntu Kernel *SRU cycle cadence* (page 81).
4. A thorough understanding of the upstream kernel development cycle and how it relates to the Ubuntu kernel development cycle.
5. Demonstrate a chain of trust by having multiple sponsored kernel uploads over a six-month development cycle by various existing members of the ubuntu-kernel-uploaders team.

If you are not an official ubuntu-kernel-uploaders member yet, but fulfill all of the criteria above, you are likely a promising candidate for joining the team.

Application template

If you are interested in joining, start by preparing your application using the following template:

https://discourse.ubuntu.com/t/kernel-ppu-application-template/88551

At least three existing ubuntu-kernel-uploaders members must confirm that they have worked with you sufficiently to assess your skills and verify that you meet the criteria above. These three individuals are typically your sponsors.

Screening process

Once your application has been prepared and you are ready to be screened, send an email to the Ubuntu kernel-team mailing list[78] requesting your application be reviewed.

You will then get a notification from the team about the scheduled Matrix meeting – in the Ubuntu Kernel room (at matrix:ubuntu.com – where you will be interviewed and a vote regarding your membership will be taken.

As part of the interview you will be asked to briefly introduce yourself, so prepare a 2-3 line introduction beforehand to speed up the process. Only existing members of the ubuntu-kernel-uploaders team are allowed to vote. An applicant must receive a minimum of 3 in order to be added to the team.

[77] https://launchpad.net/~ubuntu-kernel-uploaders
[78] https://lists.ubuntu.com/mailman/listinfo/kernel-team

Once an applicant has successfully passed the application process, an announcement will be made to both the Ubuntu kernel-team and devel-permissions mailing lists[79]. The applicant will then be added to the ubuntu-kernel-uploaders team by an administrator.

DKMS upload rights

Sometimes you want to be able to upload the DKMS package without having get full MOTU[80] or Ubuntu Core Developer[81] rights. The *kernel-dkms packageset* is a list of packages that can be uploaded by members of the ubuntu-kernel-dkms-uploaders[82] Launchpad team.

There is one packageset per release. See for example the Noble kernel-dkms packageset[83].

Adding packages to the packageset

If you need to add DKMS packages to the packageset, send a mail to the `devel-permissions@lists.u.c` mailing list. List the source packages and releases you need in your request.

See for example this request for the mofed-modules-24.10 package[84].

Applying for packageset upload rights

Like all applications, you first need to create a wiki page with your application details. Use the DeveloperApplicationTemplate[85] to create your `DKMSUploadApplication` page. See for example PaoloPisati/DKMSUploadApplication[86].

Then, you will need to reserve a meeting with the Developer Membership Board (DMB)[87] for applying[89]. To do so, edit the DMB agenda[88] to add yourself to a free slot.

[79] https://lists.ubuntu.com/mailman/listinfo/devel-permissions
[80] https://wiki.ubuntu.com/UbuntuDevelopers#MOTU
[81] https://wiki.ubuntu.com/UbuntuDevelopers#CoreDev
[82] https://launchpad.net/~ubuntu-kernel-dkms-uploaders
[83] https://ubuntu-archive-team.ubuntu.com/packagesets/noble/kernel-dkms
[84] https://lists.ubuntu.com/archives/devel-permissions/2025-January/002679.html
[85] https://wiki.ubuntu.com/UbuntuDevelopment/DeveloperApplicationTemplate
[86] https://wiki.ubuntu.com/PaoloPisati/DKMSUploadApplication
[87] https://wiki.ubuntu.com/DeveloperMembershipBoard
[89] Unlike the ubuntu-kernel-uploaders[90] Launchpad group for *Kernel upload rights* (page 73), the kernel team has no admin[91] for the ubuntu-kernel-dkms-uploaders[92] Launchpad group. This means we have no team process to review applications and must delegate to the DMB.
[90] https://launchpad.net/~ubuntu-kernel-uploaders
[91] https://launchpad.net/~ubuntu-kernel-dkms-uploaders/+contactuser
[92] https://launchpad.net/~ubuntu-kernel-dkms-uploaders
[88] https://wiki.ubuntu.com/DeveloperMembershipBoard/Agenda

3.2.5. General

Kernel glossary

This page is a running list of terminology that is frequently used when talking about kernels.

ABI
: Application Binary Interface, or ABI defines a stable interface between user space applications and the kernel. It ensures that the binaries of applications compiled for one version of the kernel remain compatible with subsequent versions, as long as the ABI remains unchanged.

DKMS
: Dynamic Kernel Module Support, or DKMS is a framework that provides support for installing supplementary versions of kernel modules in a simplified manner. See the dkms manpages[93] for more information.

edge kernel
: An edge kernel is the next HWE kernel still in development with features and/or updates that will be backported from the latest Ubuntu release (until the next LTS).

HWE
: Hardware enablement, or HWE kernels are Ubuntu kernels based on newer upstream kernel versions (compared to the Ubuntu LTS GA release) that typically contain newer features, improved performance and security, and support for newer classes of hardware. Newer kernels are usually shipped with interim and LTS releases, and will then be enabled on the latest Ubuntu LTS release as the HWE kernel. This provides an easier upgrade path for existing LTS users, and enables new deployments to immediately benefit from the newer kernel version.

  See *HWE kernels* (page 56) for more information.

linux-meta
: Refers to a set of meta-packages in Linux distributions like Ubuntu. These meta-packages do not contain the kernel binaries or source code themselves but instead define dependencies that point
to the latest kernel packages. By installing a linux-meta package (e.g. linux-generic), users can ensure they always receive the latest version of a specific kernel series through updates. In the kernel development and *SRU* life cycle, linux-meta acts as a bridge between the release of new kernel versions and the package manager. When a new kernel version is released and marked stable, the linux-meta package is updated to reference the new version, allowing automatic upgrades.

linux-signed
: Refers to kernel packages that are cryptographically signed to ensure their integrity and authenticity. These signatures are crucial for secure boot environments, as they enable the
system firmware to verify that the kernel has not been tampered with and is from a trusted source. In the kernel *SRU* life cycle, linux-signed is created after the corresponding unsigned kernel (e.g. linux-image-unsigned-6.8.0-50-generic) has been built. The signing process is part of the release pipeline, ensuring compliance with secure boot requirements and enhancing security in the kernel deployment process. This package works in tandem with the linux-meta package to deliver signed kernel updates.

[93] https://manpages.ubuntu.com/manpages/noble/en/man8/dkms.8.html

OEM kernel
: The OEM kernel is an optimized derivative Ubuntu kernel, designed specifically for use in Original Equipment Manufacturer (OEM) projects. OEM kernel variants are typically developed and customized for hardware that will be pre-installed with Ubuntu.

  See *OEM kernels* (page 61) for more information.

respin
: A kernel respin is a rebuild of a kernel package in the same kernel SRU cycle to incorporate fixes or important updates.

SAUCE
: A SAUCE patch is a patch not included in Linus Torvalds' tree or linux-next, either because it hasn't been pulled in, or because it is obtained from other non-upstream sources and is unlikely to be upstreamed.

  See *Patch acceptance criteria* (page 44) for more information.

SRU
: Stands for Stable Release Update, a process in distributions like Ubuntu used to provide important updates to packages, including kernel packages, after the release of a stable version. SRUs deliver fixes for critical bugs, security vulnerabilities, and hardware enablement while ensuring the stability of the system.

unstable kernel
: The linux-unstable kernel is used for the latest Ubuntu kernel developments. The unstable tree is primarily utilized by the development team and closely tracks the latest mainline kernel releases and *SAUCE* patches.

  Development is conducted in the unstable Git repository[94] on Launchpad.

3.3. Explanation

The explanatory guides in this section talk about different aspects of the kernel and kernel development process at Canonical.

3.3.1. About mainline kernel builds

Warning:

Mainline build kernels are **not supported** and **never receive security updates**. Mainline build kernels are **not suitable for production** use. **Remove** them once you've finished testing. **Support is not provided** to users who are running mainline build kernels outside of diagnostic uses.

The Ubuntu Kernel Team publishes builds from upstream kernel source on a best effort basis, using Ubuntu kernel configuration files and packaging the result as `.deb` files. These mainline builds let you compare an issue with upstream kernel code.

Ubuntu kernels include Ubuntu-specific patches and configuration, so they differ from mainline builds. Comparing the two can help determine whether an issue is specific to Ubuntu's changes or is also present upstream. A mainline build is a diagnostic comparison, not a replacement for an Ubuntu kernel.

[94] https://code.launchpad.net/~ubuntu-kernel/ubuntu/+source/linux/+git/unstable

Module compatibility

Differences in the kernel interface and used to build kernels mean that modules built for an Ubuntu kernel might not work with a mainline build.

To compare kernels in a bug investigation, see *How to test a bug with a mainline kernel* (page 20). For the mainline archive structure, see *Mainline kernel archive* (page 63).

 sru release pockets signoff

3.3.2. Ubuntu kernel SRU lifecycle

Each Stable Release Update (SRU) cycle, kernels move through several stages from initial preparation and build testing through to final publication in the `-updates` or `-security` pockets. Each stage adds more testing and confidence before the kernel reaches a broader audience.

Preparation

The lifecycle begins when the kernel sources are prepared and cranked, uploading packages to the build PPA. The goal is to confirm that package generation is proceeding correctly. The kernel is not yet ready for testing at this point.

Build PPA and early testing

Once built, the kernel undergoes early validation in the build PPA:

- **boot-testing**: confirms the kernel boots
- **abi-testing**: checks ABI compatibility
- **sru-review**: review by the kernel SRU team
- **new-review**: review by the Archive Admin team

A kernel at this stage is a candidate only. It has not been signed with official Canonical keys and has not yet undergone broad testing.

Promote to -proposed

Once early testing and reviews pass, the kernel is copied to `-proposed`, passing through a signing PPA first if it contains artifacts that require signing. At this point the kernel is fully formed and all signed elements carry official Canonical keys.

Testing in -proposed

With the kernel in `-proposed`, formal testing begins:

- **automated-testing**: Ubuntu Auto Package Testing (ADT)
- **certification-testing**: testing on certified platforms in the certification team's lab
- **regression-testing**: CKCT testing in the kernel team lab
- **verification-testing**: validation of individual bugs that have an original report in Launchpad

are also collected here — from the security team, the kernel owner, and where needed, the kernel team internally.

This is the main confidence-building phase. The kernel stays in `-proposed` until all required tests and are complete.

Promote to -updates and -security

Once testing is complete and the cycle is ready to release, the kernel is promoted to `-updates` and is considered ready for general use. If the security team's signoff indicates it is required, the kernel is also released to `-security`.

3.3.3. Kernel security and update policy for post-release trees

This document describes the process and criteria for post-release kernel updates.

The kernel is a very complex source package, and it is fundamentally different than other packages in the archive. The described process and criteria are built on the normal *Stable release updates* (page 81) document, and where these documents conflict, this document takes precedence.

What sort of updates are allowed for post-release kernels?

In addition to the generic *SRU* requirements, the Ubuntu Kernel team will accept patches that fall into any of the following categories:

1. It fixes a critical issue (e.g. data-loss, , crashes) or is security related. Security related issues might be covered by security releases which are special in handling and publication.
2. Simple, obvious and short fixes or hardware enablement patches. If there is a related upstream stable tree open, this class of patches is required to come through the upstream process. Patches sent upstream for that reason must include their BugLink reference.
3. The patch is included in a corresponding upstream stable or extended stable release. For the lifetime of both LTS and non-LTS release, the Ubuntu Kernel team will be pulling upstream stable updates from the corresponding series. There will be one tracking bug report for each stable update but additional references to existing bugs will be added to the contained patches (on a best-effort basis).
4. Fixes to drivers which are not upstream are accepted directly if they fall into the first two categories.

How does the process work?

- First step for every SRU is to have a bug associated with the patch.
- The patch or patchset must contain the link to the Launchpad bug and contain a "Signed-off-by" line from the submitter. See *Stable patch format* (page 36) for detailed requirements on the Ubuntu Kernel SRU patch format.
- The beginning of the description area of the bug needs to have a SRU justification which should look like this example:

```
SRU Justification:
Impact: <a short description about the symptoms and the impact of the bug>
Fix: <how was this fixed, where did the fix come from>
Testcase: <how can the fix be tested>
```

- If the fix for a problem meets the requirements for SRU and has been tested to successfully solve the bug, then the next step depends on whether the fix is serious enough to be directly applied to an Ubuntu kernel series and/or whether it should go in via upstream stable (as long as that is appropriate for upstream stable).
  - For fixes for serious issues, the patch should be sent to the kernel-team mailing list[95] in parallel to being submitted upstream. SRU patches submitted for inclusion into an Ubuntu kernel require ACKs from at least two senior Ubuntu Kernel team members before being applied to an Ubuntu kernel tree. Again, even when going into an Ubuntu kernel tree on an accelerated path, the patch should also be submitted upstream. See the *Stable patch format example* (page 43) for more information.
  - For all other patches that do not need an accelerated path into an Ubuntu kernel, it is advised to push the fix upstream when appropriate (i.e. the problem also exists upstream) and CC stable@kernel.org during the process. As soon as the patch is accepted into upstream/upstream-stable, it will find it's way back down into our Ubuntu kernel in a subsequent release. This ensures patches are getting vetted and applied upstream, which reduces overall maintenance costs for the Ubuntu Kernel team.

How will updates be provided in the archive?

- Security updates will be uploaded directly into -security without other changes. The next full release will include these security changes in addition to the normal changes.
- Normal updates will be provided as pre-releases through the corresponding kernel build PPA. At certain points those get made into proposed releases which are uploaded to the -proposed pocket. Before proposed releases can migrate to other pockets, it must be verified that the changes fix the targeted issues without causing regressions.

 sru respin mailing lists kernel dashboard

[95] kernel-team@lists.ubuntu.com

3.3.4. About kernel stable release updates (SRU)

The Ubuntu Kernel Stable Release Updates (*SRU*) is a structured procedure to ensure that kernel updates in Ubuntu's stable releases are both reliable and non-disruptive to users.

Supported Ubuntu kernels proceed through each SRU cycle, though individual kernels may be held or dropped from a particular cycle based on quality gates, testing results, and specific release needs.

This document aims to provide an overview about the various aspects of the Ubuntu kernel SRU process.

SRU purpose

Kernel SRU focuses on delivering necessary updates without changing core functionalities with low potential of introducing regressions in stable Ubuntu releases. This is especially critical for addressing the growing volume of and security vulnerabilities. Typical updates covered include:

- Upstream stable updates
- Bug fixes that address relevant issues or improve system stability
- Common Vulnerabilities and Exposures (CVE) security updates
- Hardware enablement (HWE) patches

SRU cycle cadence

The Ubuntu Kernel Team has transitioned to a unified 2-week Stable Release Update cycle[96] published weekly. This rapid cycle enables faster delivery of critical CVE fixes and security updates while maintaining rigorous testing and certification standards.

The cycles cascade – each cycle begins the week after the previous one starts, resulting in kernel releases happening weekly.

- **Week 1 - Preparation (builds, smoke tests, and initial integration and regression testing)**

  Kernel package preparation phase where updates and patches are selected for each kernel based on specific needs. Builds are created, and smoke tests, along with initial integration and regression testing, confirm kernels boot and run correctly. By the end of this week, kernels are published in the `-proposed` pocket for early testing by users.

- **Week 2 - Testing (complete integration and regression testing, and certification testing)**

  By the end of this week, test artifacts are completed. The full suite of integration and regression testing continues throughout Week 2. In addition, extensive certification testing is carried out on kernels intended for Ubuntu Certified[97] hardware, covering diverse hardware types to ensure reliability and stability. By the end of Week 2, all testing is complete and results are ready for release decision.

[96] https://canonical.com/blog/accelerating-delivery-of-cve-fixes-with-a-new-kernel-release-strategy
[97] https://ubuntu.com/certified

- **Release**

  Kernels that pass our tests and quality standards are released to the `-updates` and `-security` pockets and are ready for production.

Individual kernels may be held, deferred, or excluded from a particular cycle depending on testing results, quality gates, or specific release needs.

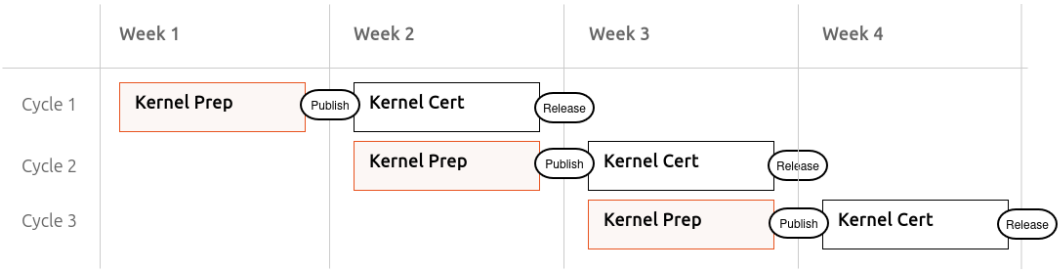

Fig. 2: Ubuntu kernel SRU cycle timeline

See the Ubuntu Kernel Team[98] home page for details on SRU cycle dates.

SRU patch submission and review process

All updates that are applied to stable kernels go through the following patch submission and review process. More details can be found at *Stable patch format* (page 36).

Kernel access

The Ubuntu Kernel Team maintains kernel repositories for collaborative development[99].

If you are a Canonical partner or customer, reach out to a member of the Ubuntu Kernel Team for other means of collaboration.

Patch creation

The first step for every SRU is to create a patch containing all the necessary information, including a link to the associated public Launchpad bug report that contains the SRU justification. The only exception to this are CVE fixes, where only the CVE number is required.

See the Kernel security and update policy for post-release trees[100] for more information on the Kernel SRU requirements and justification.

[98] https://kernel.ubuntu.com/
[99] https://kernel.ubuntu.com/git/
[100] https://ubuntu.com/project/docs/SRU/reference/exception-Kernel-Updates/

Patch submission

Next, contributors send the stable patches to the Kernel Team Email Patch Reviews mailing list (kernel-reviews@canonical.com). Where appropriate, the patch should also be submitted to upstream stable in parallel.

Mailing list review

Stable patch sets on the mailing list (ML) are then carefully reviewed by the Ubuntu Kernel Team. This review process involves validating that the patch fixes the intended issues, ensuring no regressions are introduced to the kernel, evaluating the risk and relevance of including the patch into a stable release, and reconciling mainline and Ubuntu-specific changes.

Patch acceptance

Once a mailing list patch has been vetted and has at least two ACKs from senior members of the Ubuntu Kernel Team, the commit will then be applied to the associated stable Ubuntu kernel tree. The patch will then be considered for release in an upcoming SRU cycle if all the patch acceptance criteria are met.

See the *Patch acceptance criteria* (page 44) for more information.

Held kernels

Given the overlapping 2-week SRU cadence, there is limited flexibility to respin kernels without impacting subsequent release cycles. As a result, kernels may need to be held or deferred from a release cycle for various reasons, including:

- A regression was introduced in the current cycle.
- Additional critical fixes need to be added before release.
- Quality assurance or certification testing failed or has not yet completed.
- Specific release needs require excluding a kernel from a particular cycle.

Once held kernels are resolved and meet all quality criteria, they proceed to release in a subsequent cycle.

Kernel streams

Kernels progress through a testing and release pipeline as part of the SRU cycle. The -proposed pocket is key to this process.

- **-proposed pocket:**

  During Week 1 of the SRU cycle, kernel release candidates are published in the -proposed pocket for early testing by users and systems that prioritize faster turnaround over waiting for certification or extensive tests to complete. This allows users to begin their own acceptance tests and catch issues before full release. Kernels in -proposed are updated weekly as part of the overlapping SRU cycles.

- **Certified and released kernels:**

  After passing Week 2 certification and extensive testing, kernels are released to the `-updates` and `-security` pockets.

  Individual kernels may be held, deferred, or excluded from a particular cycle based on testing results and quality gates, so not every kernel proceeds through every cycle.

Related topics

- Kernel team stable dashboard[101]

3.3.5. Ubuntu kernel patch life cycle

A patch does not move straight from a developer's tree into the Ubuntu kernel. It follows a path designed to keep Ubuntu aligned with upstream, maintainable over the long term, and safe for the wide range of hardware that Ubuntu supports.

Typical kernel patch life cycle

The typical life cycle of a patch looks like this:

1. A bug or missing feature is identified.
2. A patch is developed and thoroughly tested.
3. A Launchpad bug is filed to track the issue.
4. The patch is submitted to the upstream Linux kernel.
5. Once accepted upstream, the patch flows down into the Ubuntu kernel.
6. If a patch needs to land in Ubuntu before the next upstream sync, it can be cherry-picked or backported through the *Ubuntu stable release update process* (page 81).

This flow reflects a core principle of Ubuntu kernel development: patches that benefit the whole kernel community should be accepted upstream first. Ubuntu then consumes those changes through its regular rebasing and update process.

Why upstream first

The Ubuntu kernel is not a separate fork. It is an Ubuntu packaging and configuration layer on top of the upstream Linux kernel, plus a small set of Ubuntu-specific changes. Because of this, the easiest and safest way for a patch to get into Ubuntu is for it to be accepted into the upstream kernel first.

Submitting upstream has several advantages:

- The patch receives review from the maintainers who know the affected subsystem best.
- Wider testing across different hardware and configurations reduces the risk of regressions.
- Once the patch is in upstream, it automatically reaches Ubuntu through the normal update cycle.

[101] https://kernel.ubuntu.com/reports/kernel-stable-board/

Patches that are not accepted upstream are harder for the Ubuntu kernel team to carry. They must be maintained across rebases and may conflict with future upstream changes.

Tracking work in Launchpad

Every change that lands in an Ubuntu stable kernel is tied to a Launchpad bug. The bug serves as the record of the problem, the proposed fix, and the justification for the update.

Before a patch is sent anywhere, the bug description should explain:

- The observable problem or the missing capability.
- The impact on users.
- How the patch resolves the issue.
- How the fix was tested.

The patch can be attached to the bug so reviewers can inspect it, and the bug number is later referenced in the patch itself through the `BugLink:` line.

From upstream into Ubuntu

Once a patch is accepted into the upstream Linux kernel or the stable kernel tree, it will eventually be picked up by Ubuntu through the team's regular rebase process. The timing depends on which Ubuntu series the patch is needed for and where it sits in the upstream release cycle.

For patches that need to reach Ubuntu more quickly, the Ubuntu kernel team can cherry-pick or backport the upstream commit into the relevant Ubuntu kernel series. This is done through the stable release update process and requires the patch to meet SRU criteria.

When a patch is not accepted

A patch may be rejected upstream for many reasons: it may need more testing, require a different approach, or not fit the upstream direction. When this happens, the path into Ubuntu becomes much harder.

The Ubuntu kernel team generally does not carry patches that have been rejected upstream unless there is a strong Ubuntu-specific reason to do so. In most cases, the correct response is to address the upstream feedback and resubmit.

Related topics

- *How to send patches to the Ubuntu mailing-list* (page 8) explains how to send patches to the Ubuntu kernel team mailing list.
- *Patch acceptance criteria* (page 44) describes the criteria for accepting patches into the Ubuntu kernel.
- *Stable patch format* (page 36) covers the required patch format.
- *About kernel stable release updates (SRU)* (page 81) explains how stable release updates work.

3.3.6. About Ubuntu Linux kernel sources

Ubuntu Linux kernel source packages are essential for users and developers who want to build, modify, or understand the kernel that powers Ubuntu systems. These packages are stored in Launchpad and organized by series (or release), making it easy to find and work with the appropriate kernel version for any given Ubuntu release.

Launchpad Git URL structure for Ubuntu kernel sources

The Launchpad Git repository URL for Ubuntu Linux kernel sources follow one of the general formats below:

```
https://git.launchpad.net/~ubuntu-kernel/ubuntu/+source/<source>/+git/<series>
https://git.launchpad.net/~canonical-kernel/ubuntu/+source/<source>/+git/<series>
```

For example, the source for the generic Jammy Jellyfish (Ubuntu 22.04 LTS) can be found at:

```
https://git.launchpad.net/~ubuntu-kernel/ubuntu/+source/linux/+git/jammy
```

While the URL for the AWS kernel variant for Noble Numbat (Ubuntu 24.04 LTS) is:

```
https://git.launchpad.net/~canonical-kernel/ubuntu/+source/linux-aws/+git/noble
```

You can get the correct URL by checking the list of Git repositories for the Ubuntu Kernel Repositories team[102] or Canonical Kernel team[103], or in the automatically updated list of currently supported Ubuntu kernel Repositories[104].

Kernel source repository branches

You will find the following branches in each Ubuntu kernel source repository.

- `master`: The source for the Ubuntu kernel.
- `master-next`: Contains the commits that will be merged into the `master` branch for the next stable release update (SRU) for the series.

[102] https://code.launchpad.net/~ubuntu-kernel/+git
[103] https://code.launchpad.net/~canonical-kernel/+git
[104] https://kernel.ubuntu.com/git/

Protocols for accessing kernel sources

Protocol	Authentication needed?	Use case	Command sample
Git protocol	No	Public repositories, require read-only access	`git clone git://<kernel source URL>`
SSH protocol	Yes (SSH key)	Private repositories, require write access	`git clone git+ssh://<kernel source URL>`
HTTPS protocol	Yes (if private)	Public and private repositories,for easy access	`git clone https://<kernel source URL>`