

# Contents

- [AerynOS](#)
- [Overview](#)
- [Philosophy](#)
  - [Stateless \(aka hermetic /usr\)](#)
  - [Atomic updates](#)
  - [Self healing](#)
- [Contributing](#)
  - [Financial contributions to AerynOS](#)
  - [Contributing to our codebases](#)
  - [Other contributions](#)
- [FAQ](#)
- [Installation FAQ](#)
- [Filesystems](#)
- [AerynOS Features](#)
- [General FAQ](#)
- [Social engagement](#)
- [Lacking Features](#)
- [Users](#)
- [Getting Started](#)
- [Requirements](#)
  - [Minimum System Requirements](#)
  - [Installer Requirements](#)
- [Downloading AerynOS](#)
  - [Downloading the ISO](#)
  - [Verifying the Checksums](#)
- [Creating the Live Environment](#)
  - [Prerequisites](#)
  - [Option 1: Using Ventoy on a USB drive](#)
  - [Option 2: Preparing an install medium using Etcher](#)
- [Bootting the Live Environment](#)
  - [Bootting from a USB Drive](#)
  - [Testing the Live Environment](#)

- [Installing AerynOS](#)
  - [Create your partition layout](#)
  - [Install AerynOS](#)
- [System Management](#)
- [Configuration Locations](#)
  - [System defaults](#)
  - [System overrides](#)
  - [User-level configuration](#)
  - [Where to look next](#)
- [Manage Moss States and Packages](#)
  - [Update the system](#)
  - [Search for packages](#)
  - [Search for installed files](#)
  - [Install new software packages](#)
  - [Remove software packages](#)
  - [List currently installed software packages](#)
  - [Atomic and independent states](#)
  - [Fetch package .stone files for backup purposes](#)
- [Desktops](#)
- [COSMIC](#)
- [GNOME](#)
- [Plasma](#)
- [Window Managers](#)
- [Sway](#)
- [Packaging](#)
- [Workflow](#)
- [Prerequisites](#)
  - [Installing the build-essential package](#)
  - [Activating the AerynOS helper scripts](#)
  - [Adding /etc/subuid and /etc/subgid entries](#)
- [Basic packaging workflow](#)
  - [Understanding moss-format repositories](#)
  - [Creating a local repository](#)

- [Building recipes using the local-x86\\_64 profile](#)
- [Updating the installed system state](#)
- [Cleaning the local repository](#)
- [Ending notes](#)
- [Preparing for packaging](#)
  - [Update your clone of the recipes repository](#)
  - [Switch to a new git branch](#)
- [Creating a new package recipe](#)
  - [Prepare your workspace](#)
  - [Scaffold the recipe directory](#)
  - [Fill the recipe step by step](#)
  - [Update/correct the monitoring.yaml file](#)
  - [Build and test the package](#)
- [Updating an existing package recipe](#)
  - [Prepare your workspace](#)
  - [Simple updates to a package](#)
  - [Wider updates to a package](#)
  - [Build and test the package](#)
- [Building and testing packages](#)
  - [Build the package](#)
- [Submitting a PR](#)
  - [Naming Pull Requests](#)
  - [Content of Pull Request descriptions](#)
- [Checking for package updates](#)
  - [Use ent to check for package updates](#)
- [Recipes](#)
- [Overview](#)
  - [A basic recipe](#)
- [Upstreams](#)
  - [Plain sources](#)
  - [Git sources](#)
- [Metadata](#)
  - [Mandatory keys](#)
- [Monitoring](#)

- [File layout](#)
- [Release tracking](#)
- [Security metadata](#)
- [Where to find the data](#)
- [Example templates](#)
- [Build dependencies](#)
  - [\\$name - standard deps](#)
  - [binary\(\) - Standard binaries](#)
  - [sysbinary\(\) - System binaries](#)
  - [pkgconfig\(\) - PkgConfig / pkgconf](#)
  - [pkgconfig32\(\) - 32-bit PkgConf](#)
  - [cmake\(\) - CMake modules](#)
- [Package definitions](#)
  - [Package metadata](#)
  - [Defining a subpackage](#)
  - [Overriding defaults](#)
- [Triggers](#)
- [Overview](#)
  - [Basic mechanism](#)
  - [Capturing globs](#)
- [Transaction triggers](#)
  - [Sample trigger](#)
- [System Accounts](#)
- [Groups](#)
  - [Example](#)
- [Overview](#)
- [Users](#)
  - [Example](#)
- [Macros](#)
- [autotools](#)
  - [%configure](#)
  - [%make](#)
  - [%make\\_install](#)

- %reconfigure
- %autogen
- cargo
  - %cargo\_set\_environment
  - %cargo\_fetch
  - %cargo\_build
  - %cargo\_install
  - %cargo\_test
- cmake
  - %cmake
  - %cmake\_unity
  - %cmake\_build
  - %cmake\_install
  - %cmake\_test
- meson
  - %meson
  - %meson\_unity
  - %meson\_build
  - %meson\_install
  - %meson\_test
- Miscellaneous
  - %install\_bin
  - %install\_dir
  - %install\_exe
  - %install\_file
  - %patch
  - %tmpfiles
  - %sysusers
- perl
  - %perl\_setup
- python
  - %python\_setup
  - %python\_install

- [%pyproject\\_build](#)
- [%pyproject\\_install](#)
- [%python\\_compile](#)
- [Developers](#)
- [Stone Format](#)
- [Prelude](#)
  - [Fields](#)
- [Overview](#)
- [Header](#)
  - [Fields](#)
- [Payload's sub-header](#)
- [Attribute](#)
- [Content](#)
- [Index](#)
- [Layout](#)
- [Meta](#)

# AerynOS

AerynOS is an independent Linux-based operating system that diverges significantly from traditional distributions whilst still aiming to provide a familiar and comfortable environment. In this section of the documentation, you can find high level information about the project itself and what sets it apart from other distributions.

## **Overview**



Overview of the AerynOS project and its technologies

## **Philosophy**



The philosophy of AerynOS

## **Contributing**



Contributing to AerynOS

# Overview

AerynOS is a Linux-based operating system designed to eliminate years of technical baggage. It is an engineering led effort in that the distribution is produced entirely by the tooling we have developed. Every new feature, technology, or enabling is carefully considered, drawing on our own experiences and by studying the impact in similar decision spaces in other projects.

Despite being heavily engineering led, we are not averse to design. We aim to provide the best in class user experience atop a solid, innovative foundation, whilst ensuring we have the scope and scalability to meet the needs of the future.

In essence, we're producing a distribution based on sound technical principles, in order to deliver a "daily driver" that truly looks after itself. Its aim is to get out of the way when you need it to, and provide the tools you need when you need them.

If anything, AerynOS is "operating-system-as-infrastructure", providing a solid foundation for your daily computing needs. We're not just a distribution, we're a platform for the future.

## **Caution**

Remember, AerynOS is still in development. Despite our goals, we must be clear that we've deemed ourselves to be alpha quality software.

# Philosophy

## Stateless (aka hermetic /usr)

Most Linux distributions follow the [Filesystem Hierarchy Standard](#) which sets the structure for all files and directories on a Unix-like system. In traditional FHS based Linux distributions, package files can be installed to multiple directories, these can be directories or files that users may interact with (such as config files).

In AerynOS, packages are forbidden from containing any files outside of /usr directory. The /usr directory exclusively belongs to the system with the user not intended to make any changes in this directory what-so-ever. Files written under the /usr directory by a user will get removed (or reverted) the next time the system is updated.

In order to enable this, some packages and/or configurations are altered in AerynOS to ensure they can operate in the absence of a user provided configuration. This forces AerynOS to have sane defaults baked in at all levels, and eliminates 3-way merge conflicts on package updates. There are no conflicts, because everything in /etc and /var belongs to the user.

The stateless Linux concept was originally proposed by Red Hat in 2004 and the idea has continued to evolve from there. AerynOS leans towards the approach developed by Clear Linux, and we are refining it further.

However, it might still be necessary to create or update system configuration files in lockstep with package installation. In AerynOS, the only way for files to get created or updated under /etc or /var during package installation is via package “triggers”. Triggers are small scripts that are run at the tail end of package installation. AerynOS supports two forms of package triggers: Transaction triggers and System triggers.

## Transaction Triggers

Transaction triggers are run at the end of a transaction in an ephemeral container (Linux namespace) and may affect the contents of the transaction-specific /usr tree. This is useful for interdependent packages that need to dynamically produce plugin registries, for example.

## System Triggers

System triggers do not run in an isolated container, but instead are run in the context of the host system after the transaction has been successfully built and applied. It is these (minimally used) triggers that invoke `systemd-tmpfiles`, `systemd-sysusers` etc. For these cases we take special care to ensure that our default configs are sane and that a rebuild is always possible.

## Atomic updates

An atomic update is a series of changes to a system that are treated as a single, indivisible operation. If any part of this update fails, then the entire update is canceled with all prior parts of the incomplete update being rolled back. This means that either an update completes fully as intended, or the system is left in the state it was in before the update was attempted. This is important because partial updates often cause significant issues such as bricked installs.

AerynOS' approach to atomic updates is fairly different to the approach taken by other Linux distributions, which mostly use an A/B switch model using specific read-only filesystems to swap the whole system upon reboot. Atomic updates in AerynOS are managed by its package manager `moss` (which we also refer to as a system state manager). As such, AerynOS is not tied to using read-only filesystems and this allows for the use of XFS, ext4 and F2FS.

As mentioned above, AerynOS utilizes a stateless design where packages can only be installed to the `/usr` directory. The knowledge that packages can only be installed to this directory allows AerynOS to innovate in its approach to atomic updates.

AerynOS packages are packaged up as bespoke `.stone` `moss`-format files. Hence, AerynOS does not use or rely on e.g. Debian `.deb` format package files or Fedora/RHEL `.rpm` format package files. These `.stone` files contain a deduplicated set of hashed files compressed using `zstd`. When a `.stone` file is installed via `moss`, the files are decompressed and stored into a global, deduplicated content addressable store under `/.moss/`. Relevant metadata about these files is also stored in a database under `/.moss/`.

As part of the final stages of an atomic transaction, `moss` creates (or “blits”) a new `/usr` directory based on [hardlinks](#) to the global content addressable store. It then swaps this new `/usr` directory into place using the `renameat2` Linux kernel syscall with the

RENAME\_EXCHANGE flag, which allows for atomically exchanging an old path for a new path.

As hardlinks do not take up any significant additional space on disk, and since the global content addressable store is always deduplicated as part of every transaction, moss stores every /usr directory from every transaction. This allows for retaining system snapshots with minimal overhead and provides the ability to perform atomic rollbacks to earlier states so long as the user does not prune those.

## Self healing

As part of our boot management solution, every moss transaction ID is encoded into the kernel command line and is picked up during early boot into our initramfs, before /sysroot is pivoted to. Every kernel is correctly synchronized with the right rootfs based on the moss transaction it was associated to. Given that every transaction creates a new bootloader entry, AerynOS prunes all but the last 5 transactions from the bootloader list to keep it manageable.

## What are the implications of this?

On a Gnome based system, if you were to delete gtk3, GDM, and gnome-shell you would not be able to log back into the gnome session (as you've just deleted some really important part of the gnome session!). In this case, on boot you would be greeted by a Linux console login prompt, which would only let you log into your user's command line shell, which is less than ideal.

In AerynOS, instead of this scenario, you can enter the bootloader (by mashing your spacebar) on reboot, and in the bootloader. Select the second to last entry and this will automatically switch to the /usr filesystem transaction where gtk3, GDM and gnome-shell had not yet been deleted. On activating this entry with the Enter key, you will boot back into a working GDM for a graphical user experience.

Taking this a step further, if you were to remove glibc, given how integral it is to the functioning of AerynOS and how it specifically includes the renameat2 function used by moss to complete transactions, the system would be left in a state where the atomic update did not complete and the whole system would be broken. In a traditional Linux distribution, this will be very difficult, or impossible to resolve without resorting to a fresh reinstall.

In AerynOS, however, upon trying to boot into this last transaction, the system will discover that there is an issue with the transaction and will atomically roll back to the prior bootloader entry with the associated correct /usr directory that works. This rollback process only takes around a second (or a couple of seconds, depending on your hardware) and you will automatically be dropped back into a live working AerynOS system.

## Could this happen?

Whilst it is unlikely that a user would ever knowingly delete these very important packages (though it could happen). The more likely scenario on traditional Linux distributions is that there is a partial update that may have deleted very important aspects for a functioning system with the newer versions not having been yet installed before the update stopped. By the design features mentioned above, this is impossible on AerynOS.

# Contributing

## Financial contributions to AerynOS

AerynOS is an independent effort run by a handful of volunteers. The team is currently targeting an income of 500€ per month to cover:

1. Infrastructure and project costs
2. Repayment to project stakeholders for initial project seed funding
3. Build up a buffer for unexpected costs and future initiatives.

We currently accept donations via Stripe and Ko-Fi. For the latest information on how to donate to the project, check out our [main site](#).

## Contributing to our codebases

AerynOS utilizes GitHub to manage code changes, including updates to our websites. Each repository will have its own Readme that will include instructions on how to make updates to it. They can be found [here](#). To specifically make contributions to our websites, you can visit the following repositories:

- AerynOS.com site [repo](#)
- AerynOS.dev site [repo](#)

## Other contributions

The team is open to all forms of contribution, including any wallpapers or artwork that you may wish to submit. The only requirement is that e.g. wallpapers or artwork are licensed under an open license.

# FAQ

Our FAQ section provides answers to frequently asked questions about AerynOS. If you have any questions that are not answered here, please feel free to reach out to us via [Zulip](#).

The FAQ section, by it's nature, is a work in progress and subject to change.

## **Installation FAQ**



Installation FAQ for AerynOS

## **Filesystems**



Information about filesystem support on AerynOS

## **AerynOS Features**



How to use the key features of AerynOS

## **General FAQ**



General FAQ

## **Social engagement**



FAQ around social engagement for AerynOS

## **Lacking Features**



What are the features known to be missing in AerynOS?

# Installation FAQ

## Installation Questions

### Which CPUs does AerynOS support?

AerynOS is currently only compiled for the x86-64-v2 target architecture, which means that it will run on CPUs supporting x86-64-v2 or greater psABI feature levels.

Checking the currently supported x86-64 psABI feature level of a system can be done by typing the following command in a terminal as a normal user:

```
/usr/lib64/ld-linux-x86-64.so.2 --help | grep "x86-64-"
```

On an x86-64-v2 based system, you will see the following output:

```
x86-64-v4  
x86-64-v3  
x86-64-v2 (supported, searched)
```

#### **Note**

If the x86-64-v3 and x86-64-v4 psABI feature levels were supported, they would also show (supported, searched) next to them. AerynOS will still work on these systems.

### Does AerynOS offer NVIDIA GPU support?

Due to the way NVIDIA distributes its drivers, maintaining them in a distro is labor-intensive and frustrating when they do not work as advertised.

Given AerynOS is in the Alpha development stage, only limited, best effort NVIDIA enablement related to cards supported by the so-called nvidia-open-gpu-kernel-modules is currently offered.

You can check the status of NVIDIA support in [AerynOS/recipes#435](#)

## Does AerynOS support being installed alongside another OS? [🔗](#)

Officially? Not yet.

You can try, but there is no guarantee that AerynOS won't eat your other OS.

You have been warned.

## What is the recommended partition layout for AerynOS? [🔗](#)

In practice, we recommend that you install AerynOS to a separate drive with:

- A  $\geq 256$ MB ESP FAT32 partition (type 1 in fdisk).
  - This must be manually formatted for the installer to recognize it.
- A 4GB XBOOTLDR FAT32 partition (type 142 in fdisk, bls\_boot in gparted).
  - This must be manually formatted for the installer to recognize it.
  - This partition is large, because it is where the AerynOS kernel+initramfs and (in the future) rescue image files will be saved.
- A  $> 20$  GB system xfs partition
  - This must be manually formatted for the installer to recognize it.
  - The larger the xfs system (/ or root) partition is, the more OS /usr directory rollback states it can support in /.moss/.

```

ermo@virgil:~
> sudo fdisk -l /dev/nvme1n1
Disk /dev/nvme1n1: 931,51 GiB, 1000204886016 bytes, 1953525168 sectors
Disk model: Samsung SSD 980 PRO 1TB
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: ED391D3B-7BCC-4407-911F-FF7B2CECB45A

Device                Start          End          Sectors      Size Type
/dev/nvme1n1p1        2048          526335      524288      256M EFI System
/dev/nvme1n1p2       526336        8914943     8388608       4G Linux extended boot
/dev/nvme1n1p3     8914944     1953523711  1944608768  927,3G Linux root (x86-64)
ermo@virgil:~
> sudo lsblk -f /dev/nvme1n1
NAME                FSTYPE FSVER LABEL UUID                                FSAVAIL
FSUSE% MOUNTPOINTS
nvme1n1
├─nvme1n1p1 vfat   FAT32      CA93-B86A
├─nvme1n1p2 vfat   FAT32      C837-2227
└─nvme1n1p3 xfs                    569404f0-74ce-4c9e-936a-96aca25c7cd0  845,6G
9% /
ermo@virgil:~

```

NB: Remember, there is nothing stopping you from creating an extra partition, formatting it with a filesystem of your choice, and then configuring `/etc/fstab` to mount it as `/home` after AerynOS has been installed. You can do this if you want to use a different filesystem than `xfs` for your `/home` folders for whatever reason.

# Filesystems

## Filesystems

### Why do you recommend the xfs filesystem for the root partition?

Testing has shown that, due to how moss saves rollback states, xfs is by far the quickest filesystem in practice for AerynOS root partition usage.

We currently do not recommend either ext4 or f2fs root partitions, because testing has shown that they offer very poor performance on the first update (`sudo moss sync -u`) after a cold start of your computer compared to xfs.

### Do you support installing AerynOS on ext4?

We currently **strongly recommend** that you use **xfs** on your root partition for the best experience with moss and AerynOS.

### Do you support installing AerynOS on f2fs?

We currently **strongly recommend** that you use **xfs** on your root partition for the best experience with moss and AerynOS.

### Do you support installing AerynOS on btrfs?

Not yet

### Do you support installing AerynOS on bcachefs?

Not yet

### Do you support installing AerynOS on ZFS?

Not at this stage. We may or may not decide to support it at some unspecified point in the future, provided we can guarantee that we are legally in the clear to do so.

# AerynOS Features

## Usage Questions

### How do I make my system check for updates and install them?

```
sudo moss sync -u
```

This is a short hand form of:

```
# update the local systems "view" of which packages are available
sudo moss repo update
# synchronize the installed state against the list of available packages
sudo moss sync
```

### How do I access the rollback feature at boot?

Hold down or mash your Space key repeatedly after your computer starts up.

### How do I verify the integrity of my install states in AerynOS?

```
sudo moss state verify
```

Tack on --help to see the options for verify.

### How do I clean out older install states in AerynOS?

```
sudo moss state prune
```

Tack on --help to see the options for prune.

# General FAQ

## Project identity

### What does AerynOS mean and how do I pronounce it?

AerynOS is a stylized spelling of “Erin”, alluding to the project’s Irish roots. It is pronounced exactly the same as “Erin” - “AIR-in” OS. It’s also a play on “aer” and the phonetic “air” sound, indicative of our desire to produce an open, trusted and high-performance operating system.

It’s pronounced as “AIR-in” OS.

### What was Serpent OS?

Serpent OS is the former name of AerynOS. We [announced our rebrand](#) back in February 2025, which culminated with the inaugural release of AerynOS 2025.03 on March 25th, 2025. Per the announcement, our desire to rebrand was chiefly driven due to effectively being lumbered with a hastily chosen name, that poorly reflected the project’s goals and aspirations.

The project itself remains the same, with the same goals and aspirations, but with a new name and a fresh coat of paint.

## Usage Questions

### Why don’t application icons for newly installed apps show up in my current session after a sync or install?

It is a known issue and we are working on a solution.

For now, log out and back in again and they will show up.

### When do I need to reboot after updates?

- Kernel updates require a reboot.

- Some updates require you to log out of your desktop session and back in (see above).
- Most updates only require you to close the apps that were updated and start them again.

## How do I configure custom kernel command line parameters applied at each boot?

See the [blsforme repo readme](#) for the expected format.

Typically, it is necessary to change the installed system state with `moss` for command-line snippets to take effect.

One way of doing that is to `sudo moss boot sync`. This will also provide an output to show the new cmdline snippet is now active.

## Package Questions

### How come your package repository is so small?

We are still in heavy development (“Alpha”) and are developing our backend and associated automated rebuild processes.

If we discover that it is necessary for us to rebuild our entire repository, we would like the ability to do so in the span of an afternoon (using multiple builders in parallel).

Once our backend story and our automated rebuild process story are both further advanced, we will begin scaling out the repository to contain more packages.

### Could you package (...) please?

See above.

For now, we encourage users to use flatpaks for the applications we do not yet carry in our repository.

Currently, we are focusing on adding must-have packages for platform bring-up, for things that give us a development edge, or for things that help us showcase AerynOS capabilities.

If the use-case for the package you are proposing is in line with the ethos above, you can make a package request [here](#)

## Where can I learn how to package for AerynOS?

Consult the packaging documentation [here](#).

In addition, consult the AerynOS [recipes/ repository](#).

Finally, join the [AerynOS Zulip space](#) and make sure to join the #Onboarding channel in the General - Public space.

## Project Questions

### Which distribution is AerynOS derived from?

AerynOS has been bootstrapped and built from scratch and is not based on any other distro.

This implies that AerynOS has its own:

- package manager (moss)
- package build tool (boulder)
- build pipeline consisting of:
  - the package build dashboard and controller (summit)
  - the builder-as-a-service middleware (avalanche)
  - the package repository manager (vessel)

This also implies that AerynOS does NOT build upon or use either:

- .rpm related tooling from Red Hat
- .deb related tooling from the Debian Project
- Arch-related tooling

### When will AerynOS be considered stable?

AerynOS is taking on the ambitious task of creating a distribution from scratch, whilst building its own tooling and solutions for this.

As such, there is no official ETA.

Now that the project has hit alpha status, you will see frequent updates and progress reports.

## **What is AerynOS' position on the use of LLMs**

Contributions must not include content generated by large language models or other probabilistic tools like ChatGPT, Claude, and Copilot.

This policy exists due to:

- ethical concerns about the data gathering for training these models
- the disproportionate use of electricity and water of building / running them
- the potential negative influence of LLM-generated content on quality
- potential copyright violations

This ban of LLM-generated content applies to all parts of the projects, including, but not limited to, code, documentation, issues, and artworks. An exception applies for purely translating texts for issues and comments to English. We may make more exceptions for other accessibility-related uses.

## **Project-related use of LLMs**

We heavily discourage the use of LLM chat bots as a replacement for reading AerynOS's documentation.

Support requests referencing misleading or false LLM output relating to the project may be ignored, since it is a waste of time for us to “debug” where things went wrong based on this output before human support was sought. Once you're ready to submit your code, create a pull request, and one of our maintainers will review it.

# Social engagement

## Social engagement

### Is it ok to share links to video content of AerynOS in action in the Zulip rooms?

Yes! We absolutely love seeing people using AerynOS in the wild!

- Please first share them as a post in the [Show and Tell category](#),
- Then, share a link to your post in the [AerynOS Zulip space](#) in the #Show-and-Tell channel in the General - Public space so the link to the video doesn't get lost in the Zulip chat.

### Why don't you have a Discord server?

AerynOS is an open-source project that values transparency, collaboration, and community engagement. When looking at Discord, it is a closed-source platform that we believe **has different priorities**, specifically around monetization and data collection.

To this end, AerynOS would much rather promote the use of an open-source community platform such as Zulip. In the time we have been using Zulip, we have been very happy with the platform and how it has enabled us to build a strong and engaged community.

# Lacking Features

## What features are missing?

### Secure Boot

Secure Boot is a UEFI firmware security feature that ensures only digitally signed, trusted software, such as operating system loaders and drivers, can run during the startup process. This prevents malicious rootkits or an unauthorized OS from loading. It is a mandatory requirement for Windows 11 and is looked for as an additional security feature on Linux distributions.

The AerynOS project is committed to providing Secure Boot support for its users in the future. However, this will come further down the line.

### Dual booting from the same drive

Officially, this is not yet supported when running AerynOS. We know of some more experienced users who have successfully installed AerynOS alongside other operating systems however this is done at their own risk.

Our tooling is currently developed purely for making sure AerynOS is installed correctly and makes no consideration for other operating systems. As such, we highly recommend users avoid trying to dual boot with another OS from the same drive.

If you have multiple drives in your system, you could dedicate one drive to AerynOS and another to your other OS.

### Disk encryption

Disk encryption is a security feature that encrypts the data on a hard drive or solid-state drive (SSD) to protect it from unauthorized access. This is done by using a key to encrypt the data before it is written to the drive and decrypting it when it is read.

The AerynOS project is committed to providing disk encryption support for its users in the future. However, this will come further down the line.

# Users

## **Caution**

Currently we are in an alpha stage of development so please expect breakages and bugs. We are working hard to get to a stable release.

### **Getting Started**



Getting started with AerynOS

### **System Management**



Keep an AerynOS system healthy with stateless configuration guidance and moss lifecycle tasks.

### **Desktops**



Desktop environments

# Getting Started

## Requirements



Requirements for AerynOS

## Downloading AerynOS



Downloading the AerynOS ISO file and verifying the checksums

## Creating the Live Environment



Creating a live environment to boot into and run the AerynOS installer

## Booting the Live Environment



Booting into the AerynOS Live Environment

## Installing AerynOS



Installing AerynOS on your system

# Requirements

## Minimum System Requirements

- **Architecture:** x86\_64-v2
- **Firmware:** UEFI (CSM Support must be disabled)
- **Processor (CPU):** Quad-core processor with a minimum clock speed of 2GHz
- **System Memory (RAM):** 4GB or more
- **Storage:** Minimum of 40GB available space; SSD highly recommended

## Installer Requirements

To successfully create a bootable USB drive for installing AerynOS, the following requirements must be met:

- **Network:** An active internet connection is required for installation
- **USB Flash Drive:** Ensure you have a USB flash drive with at least 4GB of free space.
- **Image Flashing Software:** Utilize one of the following recommended tools to flash the AerynOS ISO image onto the USB drive:
  - **Ventoy:** A versatile tool that allows you to create a multi-boot USB drive.
  - **Balena Etcher:** A simple and user-friendly tool for creating bootable USB drives.
  - **dd:** A Linux command-line utility available on most Linux distributions for creating bootable USB drives.
  - **Rufus:** A widely-used utility that provides advanced options for creating bootable USB drives.
  - **Fedora Media Writer:** A reliable and user-friendly tool for creating bootable USB drives.
- **Additional Hardware:** A physical keyboard, mouse, and monitor (or screen) are required to interact with the installation process. Ensure that all these peripherals are properly connected to the system before starting the installation.

# Downloading AerynOS

## Downloading the ISO

1. Visit the [AerynOS download page](#).
2. Use the direct download or bittorrent option to get the latest release.
3. Download the checksum file

## Verifying the Checksums

Before creating a bootable USB drive, it's important to verify the checksums to ensure the integrity of the downloaded ISO file.

### Tip

AerynOS utilizes a GNOME based live environment for installation. Our installer lichen is a netinstaller that can be used to install GNOME, KDE Plasma, Cosmic or a terminal-only environment based on your requirements.

## Linux

1. Open a terminal window and navigate to the directory where the ISO file is located along with the checksums.

```
cd ~/Downloads
```

2. Run the following command to verify the checksums:

```
sha256sum -c <checksum_file>
```

You should see a message indicating that the checksums match if the ISO file is valid.

```
AerynOS-2026.01-GNOME-live-x86_64.iso: OK
```

If the checksums do not match, download the ISO file again and repeat the verification process.

## Windows

1. Open a Command Prompt window and navigate to the directory where the ISO file is located along with the checksums.

```
cd C:\Users\<<username>\Downloads
```

2. Run the following command to verify the checksums:

```
certutil -hashfile aerynos-<version>.iso SHA256
```

This will give you the checksum of the file, compare this to the checksum found inside the checksum file.

# Creating the Live Environment Creating a Bootable USB Drive

## Prerequisites

- Instructions for [Downloading AerynOS](#).
- You will need a spare USB drive that you don't mind wiping.

## Option 1: Using Ventoy on a USB drive

Ventoy is an open source tool to create bootable USB drives that can boot multiple operating systems by placing their respective ISO files on the USB drive. This flexibility is particularly useful as it allows users to have a single USB drive for installation of multiple operating systems. Support for AerynOS was added in version [1.1.10](#).

Ventoy has several options for creating a bootable USB drive. These instructions will vary depending on your current OS. You can find the instructions for creating a bootable Ventoy USB drive on the [Ventoy website](#).

1. Follow the instructions for creating a bootable Ventoy USB drive.
2. Copy the latest AerynOS ISO to the root directory of the "Ventoy" drive

## Option 2: Preparing an install medium using Etcher

1. Download the latest version of [Balena Etcher](#) for your OS.
2. Launch the application
3. Select the latest AerynOS ISO you have already downloaded
4. Select the inserted USB stick
5. Flash!
6. Ensure the USB drive is properly ejected after flashing the ISO to avoid data corruption.

## **Danger**

Creating a bootable USB drive using Etcher will erase all data on the USB drive. Make sure to back up any important data before proceeding.

## **Alternative options**

There are several alternative options that can be used for creating a bootable USB drive. These include, but are not limited to:

- **Rufus**: A free and open-source tool for Windows that can create bootable USB drives.
- **dd**: A Linux command-line utility that can be used to create bootable USB drives from the command line.
- **GNOME Disks**: A graphical tool that can be used to create bootable USB drives on Linux.

You can find instructions for these various options online via your favorite search engine.

# Booting the Live Environment

## Booting from a USB Drive

### **Caution**

Currently NVIDIA Drivers are not implemented by the live environment and will fallback to nouveau drivers.

1. Insert the bootable USB drive into your system and boot from the USB drive.
2. Adjust or override your BIOS settings to boot from the USB drive.
3. You should see the AerynOS boot process, and you will be presented with the live environment.
4. **If utilizing Ventoy** select the AerynOS ISO file to boot from.

## Testing the Live Environment

### **Tip**

For a representative experience, you will need to install AerynOS. The live environment will run slower than an installed system due to the limitations of running from a USB drive.

Once you have booted into the live environment, you can test AerynOS without installing it on your system.

- Explore the desktop environment.
- Test the pre-installed applications.
- Check the system performance.
- Verify the hardware compatibility.
- Connect to the internet and browse the web.

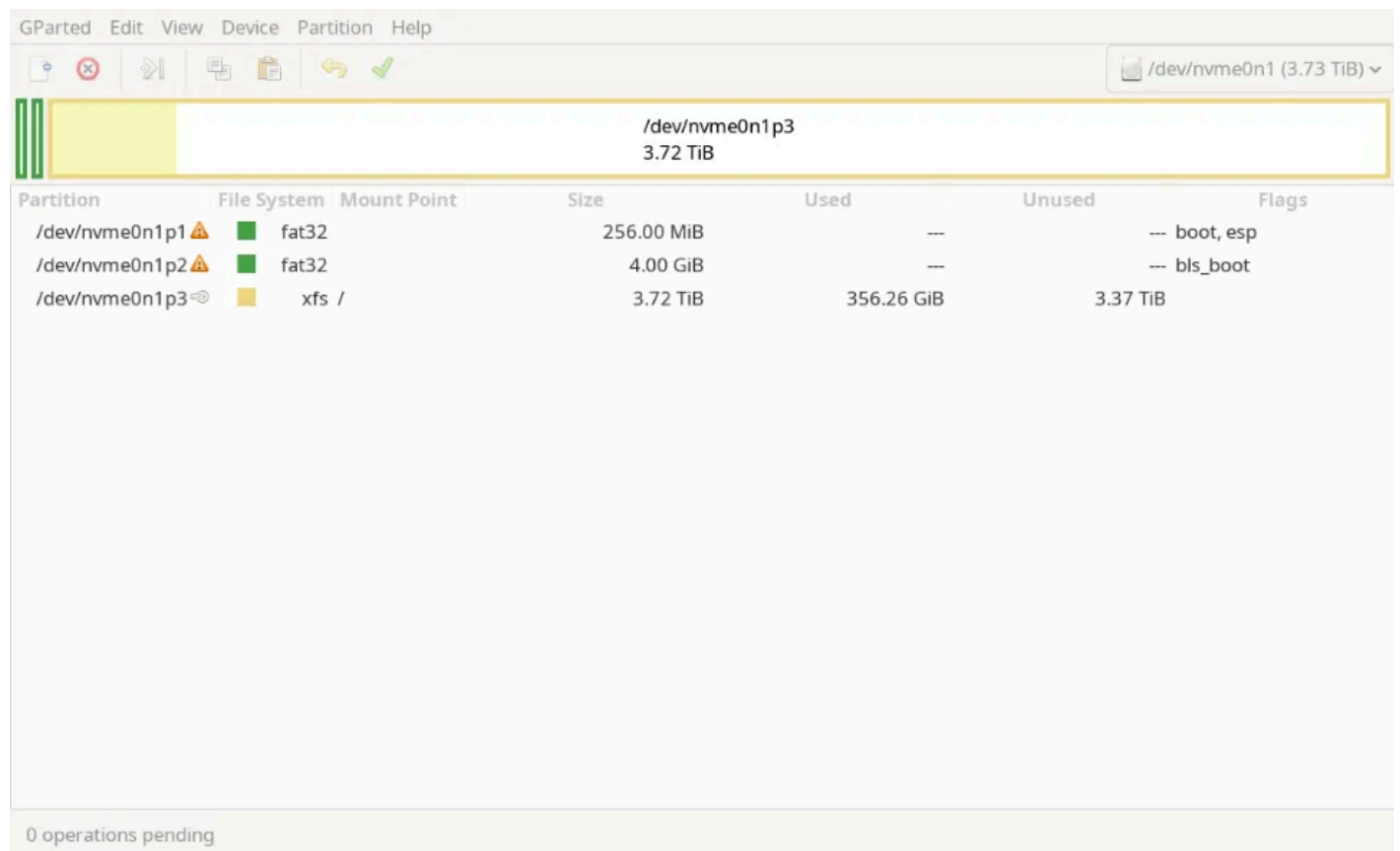
# Installing AerynOS

## Create your partition layout [🔗](#)

For the easiest experience, we recommend using gparted which is included in the live environment.

1. Load up gparted from the live environment.
2. Create a gpt partition table on the drive you want to install AerynOS on.
3. Create a  $\geq 256$ mb FAT32 partition for ESP with the boot and esp flags
4. Create a 4gb FAT32 partition for XBOOTLDR with the bls\_boot flag
5. Utilize the rest of the space for a xfs root partition

Once you have finished, your partition layout should look like this:



## Install AerynOS [🔗](#)

From the app menu, select the “Install AerynOS” option. This is a shortcut to opening a terminal and running the following command:

```
sudo lichen
```

Follow the instructions on the screen to complete your installation. You will be prompted to select your:

1. Location
2. Timezone
3. Partition for ESP (and optionally XBOOTLDR)
4. Partition for root
5. Admin password
6. User name
7. User password
8. Filesystem type for root partition
9. Desktop environment

If you follow all the steps above, lichen will download the packages from our repository and install the latest version of AerynOS onto your system. Once completed, you can reboot your system and enjoy your new AerynOS installation!

# System Management

Use this section to manage an installed system, from understanding where configuration lives to operating moss states safely.

## Configuration Locations [→](#)

Understand where packages ship their default configuration and how to override it on a stateless system.

## Manage Moss States and Packages [→](#)

Learn how to update, install, remove, and search for packages, and how to inspect, switch and clean up states with moss.

# Configuration Locations

AerynOS ships configuration in a stateless layout. Packages deliver defaults in `/usr/share/defaults`, while administrator and user changes live elsewhere so updates can proceed without overwriting your work.

## System defaults [🔗](#)

Default files mirror the traditional `/etc` hierarchy under `/usr/share/defaults`.

Purpose	Default location	Example contents
Base system settings	<code>/usr/share/defaults/etc</code>	<code>ld.so.conf</code> , <code>libn1</code> , <code>tpm2-tss</code>
PAM policies	<code>/usr/share/defaults/pam.d</code>	<code>sudo</code> , <code>system-login</code> , <code>polkit-1</code>
Shell profiles	<code>/usr/share/defaults/profile</code> and <code>/usr/share/defaults/profile.d</code>	<code>00-aeryn.sh</code> , interactive shell tweaks
Service defaults	<code>/usr/share/defaults/environment.d</code>	Session-wide environment snippets
Sudo configuration	<code>/usr/share/defaults/sudo</code>	<code>sudoers</code> , drop-in files
SSH defaults	<code>/usr/share/defaults/ssh</code>	<code>ssh_config</code> , <code>sshd_config</code>

Packages may add more directories under `/usr/share/defaults` as required. The layout always mirrors where the file would appear under `/etc` on a traditional filesystem.

## System overrides [🔗](#)

Place administrator overrides in `/etc`. Files in `/etc` shadow anything under `/usr/share/defaults` and survive package updates. Use drop-in directories such as `/etc/pam.d` or `/etc/sudoers.d` to keep customisations scoped and easy to audit.

When you need to revert to the shipped defaults, remove the override from `/etc` and Moss will fall back to the matching file in `/usr/share/defaults`.

## User-level configuration

Desktop and application settings follow the XDG Base Directory specification. Store per-user changes in:

- `~/.config` for configuration files
- `~/.local/share` for data files

These paths override both `/etc` and `/usr/share/defaults` for the owning user.

## Where to look next

Run the following command to explore the current defaults on your system:

```
ls /usr/share/defaults
```

Combine this with `moss search-file` to identify which package owns a specific default file when you need to adjust or report an issue.

# Manage Moss States and Packages

Moss keeps track of packaging-related operations that change the state of the /usr directory by creating a new filesystem transaction (fstx) for each associated moss operation, be it package installation, removal, or upgrades.

Use the commands below to manage your installed software packages, and keep your system current.

## Update the system [🔗](#)

Keep the entire system current with a sync operation.

```
sudo moss sync --update
```

--update (-u) pulls fresh repository metadata before applying upgrades. Moss records the result as a new state, so you can roll back if something goes wrong.

## Search for packages [🔗](#)

Use keyword searches to discover software by name or summary.

```
sudo moss search something
```

Add --installed (-i) if you only want to search software that is already present on the system.

## Search for installed files [🔗](#)

Look up which package delivered a specific file when you troubleshoot or audit an installation.

```
sudo moss search-file filename
```

`moss search-file` scans files from installed packages only.

## Install new software packages

1. Refresh repository metadata when needed.

```
sudo moss repo update
```

2. Install one or more packages.

```
sudo moss install somepackage
```

Moss creates a new state automatically. Confirm success with `moss state active`.

## Remove software packages

Uninstall packages you no longer need.

```
sudo moss remove somepackage
```

Moss snapshots the removal in a new state. Use `moss state list` to find the previous state if you have to recover.

## List currently installed software packages

```
moss list installed
```

## Atomic and independent states

In the preceding examples, it was briefly mentioned that moss creates new “states”.

When using moss, state management is an important concept. This section details the fundamentals of moss state management.

Moss is based on the concept of atomic updates. Atomic in this context means “independent units”. Each time the user instructs moss to change the state of the system, moss creates a new, independent state. Because each state is independent, it allows moss to roll back to an earlier state, and roll forward to a later state.

## Check the current active state [🔗](#)

1. List the active state to confirm what is running right now.

```
██████████  
moss state active
```

2. Review the state history when you need context for a rollback.

```
██████████  
moss state list
```

Use the state ID (the number after State #) when you need to query, activate, or remove a specific state.

## Activate a different state [🔗](#)

Follow these steps to roll back or advance to another state safely.

1. Identify the target state ID with `moss state list`.
2. Activate that state.

```
██████████  
sudo moss state activate 128
```

3. Verify the change.

```
moss state active
```

Activating a state atomically swaps the currently active state's /usr directory with the new state's /usr directory, using the Linux kernel [renameat2 syscall](#).

On successful activation of the new state, it is recommended to reboot the system, so that long-running services start with the expected binaries, libraries, and configurations.

## Clean up retained states [🔗](#)

Over time, a system managed by moss will accumulate more and more states. Each state takes up space that is equivalent to the difference in packages between one state to another. If two states share a lot of package versions, the storage needed to keep the difference between states will be small.

There are two ways to clean up retained states:

- Prune, which removes all states except the N latest states
- Remove, which is capable of removing individual states and ranges of states.

## Prune states [🔗](#)

This will prune state history to keep only the 10 latest states, including the current active state:

```
sudo moss state prune
```

If you want to keep a different number of states, the `-k / --keep` parameter will let you specify the number of states you wish to keep:

```
# Keep the five latest states, including the active state
sudo moss state prune --keep 5
# or
sudo moss state prune -k 5
```

## Remove specific states [🔗](#)

Sometimes, it is useful to be able to remove one or more specific states. This can be accomplished with the `state remove` operation.

```
moss state remove 2 8-12 15 17-21
```

As you can see, `state remove` enables you to remove both individual states and ranges of states. Ranges are inclusive, meaning the start and end states of the specified range are also removed.

## Fetch package .stone files for backup purposes [🔗](#)

If you would like to keep an archive of stones, `moss` supports a `fetch` operation.

Use `moss fetch` to download one or more package `.stone` files by name without installing them on the current system.

```
moss fetch somepackage
```

Fetch multiple packages into a custom output directory:

```
moss fetch package1 package2 --output-dir ~/stones
```

`moss fetch` writes downloaded files to the current directory by default. Add `--verbose` when you need more detailed progress output. If a package cannot be resolved, refresh repository metadata and try again.

Note also that `moss` fetches `.stones` against currently active repository `stone.index` files.

You can list your currently active repository index with `moss repo list`.

# Desktops

## **COSMIC**

COSMIC Desktop



## **GNOME**

GNOME Desktop



## **Plasma**

KDE Plasma Desktop



## **Window Managers**

Lightweight window manager environments



# COSMIC

The [COSMIC Desktop](#) from [System76](#) is a highly popular choice with AerynOS users. COSMIC is notable for being written in Rust and using a modern multiprocess architecture, while being Wayland-only. For many, this makes AerynOS and COSMIC an ideal partnership.

## Installing COSMIC on AerynOS [🔗](#)

AerynOS currently only offers one iso with a GNOME live environment. However, [Lichen](#) is a net based installer that allows users to select their Desktop Environment at install time. As such, you can install AerynOS COSMIC edition directly from the GNOME based AerynOS installer ISO.

If you are already using GNOME, you are able to install Cosmic Desktop side by side and select which Desktop Environment to use in GDM at login. You do this by installing one of three package sets:

```
sudo moss install pkgset-aeryn-cosmic-minimal
sudo moss install pkgset-aeryn-cosmic-recommended
sudo moss install pkgset-aeryn-cosmic-full
```

The names are fairly self explanatory:

- Minimal: The minimum number of packages required for a Cosmic Desktop session
- Recommended: The minimal Cosmic Desktop session plus additional recommended applications
- Full: The recommended Cosmic Desktop session plus additional optional applications

## Controlling the display manager [🔗](#)

If you've installed COSMIC over the top of a GNOME install, you can still log into your COSMIC session from gdm. You can also safely remove gdm and have cosmic-greeter take over. Note: GNOME Shell still expects gdm for full functionality.

## Installing cosmic-greeter [🔗](#)

```
sudo moss install cosmic-greeter
```

## Removing gdm [↗](#)

If you wish to remove gdm, you would use the following command:

```
sudo moss remove gdm
```

# GNOME

The default desktop environment for the AerynOS live environment and for installs using `lichen` is [GNOME](#). We utilize Wayland display server protocol and do not offer X11 (or any fork of X11).

GNOME has been chosen as the default environment due to our familiarity with the GNOME software stack and therefore our ability to maintain it whilst we work on fleshing out the AerynOS tooling and infrastructure.

It is recommended to install GNOME using `lichen`, rather than adding to an existing install.

## Gnome defaults [🔗](#)

- Terminal: [ptyxis](#)
- Media Player: [Celluloid](#)
- Software: [Gnome Software](#)
- Document Viewer: [Gnome Papers](#)
- System Monitor: [Gnome Resources](#)
- Code editor: [Zed](#)

# Plasma

AerynOS now offers [KDE Plasma](#) as a Desktop Environment though it is currently considered beta status. You can track the progress of identifying and resolving Plasma related issues on [Github](#).

## Installing Plasma on an existing AerynOS install [🔗](#)

If you are already using GNOME, you are able to install KDE Plasma side by side and select which Desktop Environment to use in GDM at login. You do this by installing one of three package sets:

```
sudo moss install pkgset-aeryn-plasma-minimal
sudo moss install pkgset-aeryn-plasma-recommended
sudo moss install pkgset-aeryn-plasma-full
```

The names are fairly self explanatory:

- Minimal: The minimum number of packages required for a Plasma desktop session
- Recommended: The minimal Plasma desktop session plus additional recommended applications
- Full: The recommended Plasma desktop session plus all available KDE applications

## Controlling the display manager [🔗](#)

If you've installed Plasma over the top of a GNOME install, you can still log into your Plasma session from gdm. You can also safely remove gdm and have either sddm or plasma-login-manager take over. Note: GNOME Shell still expects gdm for full functionality.

## Installing either sddm or plasma-login-manager [🔗](#)

```
sudo moss install sddm
```

or

```
sudo moss install plasma-login-manager
```

## Removing gdm [↗](#)

If you wish to remove gdm, you would use the following command:

```
sudo moss remove gdm
```

# Window Managers

## **Sway**



Sway Wayland window manager

# Sway

[Sway](#) is a dynamic tiling window manager designed as a drop-in replacement for i3, but built for Wayland. It offers a lightweight workflow that is well suited to machines where users prefer keyboard driven navigation over graphical shell integrations.

## Installing Sway on AerynOS [🔗](#)

Sway is currently packaged as a single minimal session that you can add to any existing AerynOS installation. Install the package set with moss:

```
sudo moss install pkgset-aeryn-sway-minimal
sudo moss install branding-aeryn-sway
```

After installation you can select the Sway session from your display manager, or start it directly from a TTY with `exec sway`.

# Packaging

Here you can find all the packaging documentation.

## **Workflow**



Understanding how moss and boulder make use of repositories in AerynOS

## **Recipes**



The `stone.yaml` recipe format and `boulder` form the core of all package builds in AerynOS

## **Macros**



Tools to simplify life - macros

# Workflow

## **Prerequisites**



Prerequisites for building packages on Aeryn OS

## **Basic packaging workflow**



Building packages locally and testing them

## **Preparing for packaging**



Preparing for packaging on AerynOS

## **Creating a new package recipe**



Creating a new package recipe from scratch

## **Updating an existing package recipe**



How to update an existing package recipe

## **Building and testing packages**



How to build and test packages locally on your system

## **Submitting a PR**



How to submit a PR into the AerynOS repository

## **Checking for package updates**



How to check for package updates

# Prerequisites

To set up a AerynOS system to be able to build package recipes, a few prerequisites need to be installed, and a new directory for storing local build artifacts needs to be set up.

## Installing the build-essential package

We maintain a `build-essential` metapackage that should contain the basics for getting started with packaging on AerynOS.

```
sudo moss sync -u
sudo moss it build-essential
```

## Activating the AerynOS helper scripts

The easiest way to create a local repository is to use the helper script distributed with the AerynOS recipe repository in the `tools/` directory.

Start by cloning the `recipes/` git repository:

```
mkdir -pv repos/aerynos/
pushd repos/aerynos
git clone https://github.com/AerynOS/recipes
```

After the `recipes/` git repository has been cloned, symlink `helpers.bash` into `~/.bashrc.d/`:

```
popd
mkdir -pv ~/.bashrc.d/
ln -sv ~/repos/aerynos/recipes/tools/helpers.bash ~/.bashrc.d/90-aerynos-
helpers.bash
```

Finally, execute the following in a new terminal tab:

```
cd ~  
gotoaosrepo
```

If the helpers script has been correctly loaded, the `gotoaosrepo` command should switch to the directory containing the recipes/ git repository clone.

## Setting up git hooks and linters [🔗](#)

The `just` command runner should have been installed as part of `build-essential`.

Run the following:

```
gotoaosrepo  
just init
```

This will setup git hooks that will lint for the most common packaging errors upon git commit, as well as fill out commit message templates for you to edit as appropriate.

## Setting up git diff auto-conversion of manifest.\*.bin files [🔗](#)

This will make it so you can view `git diff` output for binary `manifest.*.bin` files in both `git diff` and `git log -p .` invocations.

Edit the recipe repo `.git/config` file to contain the following below the `[core]` section:

```
[diff "moss"]  
  textconv = moss inspect  
  binary = true
```

The recipe repo already contains the `.gitattributes` file that sets up the `moss diff` filter referenced here.

## Setting up the git gone alias [🔗](#)

This will make it so that executing `git gone` will remove any local branches that no longer exist upstream.

Edit your `~/.gitconfig` file to contain the following:

```
[alias]
  gone = "!f() { git fetch --all --prune; git branch -vv | awk '/:
gone]/{print $1}' | xargs git branch -D; }; f"
```

## Adding `/etc/subuid` and `/etc/subgid` entries [🔗](#)

Since `boulder` uses user-namespaces to set up isolated build roots, it is necessary to set up a `subuid` and a `subgid` file for the relevant users first:

```
sudo touch /etc/sub{uid,gid}
sudo usermod --add-subuids 1000000-1065535 --add-subgids 1000000-1065535 root
sudo usermod --add-subuids 1065536-1131071 --add-subgids 1065536-1131071 "$USER"
```

If `/etc/subuid` and `/etc/subgid` already exist, adapt the above as appropriate.

# Basic packaging workflow

Once the [prerequisites](#) have been handled, it is time to learn how to install newly built local moss-format `.stone` packages.

## Understanding moss-format repositories

When building and testing packages locally, boulder (and moss) can be configured to consult a local moss-format repository containing moss-format `.stone` packages and a `stone.index` local repository index.

### `stone.index` files

The `stone.index` file is what both moss and boulder consult when they check which packages are available to be installed into moss-maintained system roots.

Adding a moss-format repository is as simple as registering a new location from where to fetch `stone.index` files, which will be shown in detail later on this page.

### moss build roots

Every time a package is built, boulder calls out to moss to have it construct a pristine build root directory (called a ‘buildroot’) with the necessary package build prerequisites installed.

The packages in this buildroot are resolved from one or more moss `stone.index` files, sorted in descending priority, such that the highest priority repository “wins” when package providers are resolved.

The lowest priority repository will typically be the official AerynOS upstream package repository.

If higher priority repositories are added, packages from these will in turn override the packages available in the official AerynOS upstream package repository.

The next section deals with how to create and register a higher priority local moss-format repository, which is colloquially referred to as a “local repo”.

## Creating a local repository [🔗](#)

After the helper script has been activated in bash, open a new tab or a new terminal, and execute the following commands:

```
# create a new tab or open a new terminal
gotoaosrepo
just create-local
just index-local
```

The `just create-local` invocation will set up an empty `~/.cache/local_repo/x86_64/` directory, and the `just index-local` invocation will create a `stone.index` file for the directory.

## Making boulder use the local repository [🔗](#)

Boulder will need to have its list of “build profiles” be updated before it will consult the `~/.cache/local_repo/x86_64/stone.index` moss-format repository index created above:

```
boulder profile list
# output
default-x86_64:
  - volatile = https://cdn.aerynos.dev/stream/volatile/x86_64/stone.index [0]

# add new local-x86_64 build profile
# note: ${HOME} will be replaced by the actual home directory of the user
#       invoking the command. In the example below, ${HOME} = /home/ermo
boulder profile add \
  --repo
name=volatile,uri=https://cdn.aerynos.dev/stream/volatile/x86_64/stone.index,prior
\
  --repo
name=local,uri=file://${HOME}/.cache/local_repo/x86_64/stone.index,priority=100 \
  local-x86_64
boulder profile list
# output
default-x86_64:
  - volatile = https://cdn.aerynos.dev/stream/volatile/x86_64/stone.index [0]
local-x86_64:
  - volatile = https://cdn.aerynos.dev/stream/volatile/x86_64/stone.index [0]
  - local = file:///home/ermo/.cache/local_repo/x86_64/stone.index [100]
```

Behind the scenes, boulder builds and saves an appropriately named build profile to `~/.config/boulder/profile.d/`.

This is what `local-x86_64.yaml` should look like after the above commands have been run successfully:

```
local-x86_64:
  repositories:
    local:
      description: ''
      uri: file:///home/ermo/.cache/local_repo/x86_64/stone.index
      priority: 100
      active: true
    volatile:
      description: ''
      uri: https://cdn.aerynos.dev/stream/volatile/x86_64/stone.index
      priority: 0
      active: true
```

## Making moss use the local repository [🔗](#)

Listing and adding moss-format repositories containing stone.index files is done as follows:

```
moss repo list
# output
- unstable = https://cdn.aerynos.dev/stream/unstable/x86_64/stone.index [0]
# add repositories
# note: ${HOME} will be replaced by the actual home directory of the user
#       invoking the command. In the example below, ${HOME} = /home/ermo"
sudo moss repo add volatile
https://cdn.aerynos.dev/stream/volatile/x86_64/stone.index -p 10
sudo moss repo add local file://${HOME}/.cache/local_repo/x86_64/stone.index -p
100
moss repo list
# output
- unstable = https://cdn.aerynos.dev/stream/unstable/x86_64/stone.index [0]
- volatile = https://cdn.aerynos.dev/stream/volatile/x86_64/stone.index [10]
- local = file:///home/ermo/.cache/local_repo/x86_64/stone.index [100]
```

## Package resolution order [🔗](#)

In the above priority tower, each moss-format package would first get resolved via the local repository (priority 100), then from the volatile repository (priority 10), and

finally from the unstable repository (priority 0). The latter of which is the official upstream AerynOS moss-format .stone package repository.

## Disabling moss-format repositories [↗](#)

Users of AerynOS should generally *not* have the `volatile` repository be enabled, because this repository is where new .stone packages land right after being built. This means the repository can potentially be in an undefined and volatile state when building large build queues (hence the name).

Therefore, it can be useful to disable moss-format repositories without deleting their definitions from the local system:

```
sudo moss repo disable volatile
sudo moss repo disable local
moss repo list
# output
- unstable = https://cdn.aerynos.dev/stream/unstable/x86_64/stone.index [0]
- volatile = https://cdn.aerynos.dev/stream/volatile/x86_64/stone.index [10]
(disabled)
- local = file:///home/ermo/.cache/local_repo/x86_64/stone.index [100]
(disabled)
```

## Enabling moss-format repositories [↗](#)

However, when testing locally built packages, they *must* be built against the `local-x86_64` boulder build profile, which in turns relies on the `volatile` repository via the boulder `local-x86_64` build profile.

Hence, when testing locally built packages, you may need to **temporarily** enable the `volatile` repository for moss to resolve from.

```
sudo moss repo enable volatile
sudo moss repo enable local
moss repo list
# output
- unstable = https://cdn.aerynos.dev/stream/unstable/x86_64/stone.index [0]
- volatile = https://cdn.aerynos.dev/stream/volatile/x86_64/stone.index [10]
- local = file:///home/ermo/.cache/local_repo/x86_64/stone.index [100]
```

## Building recipes using the local-x86\_64 profile [🔗](#)

To actually build a recipe, it is recommended that new packagers start out by building nano.

```
# Go into the root of the AerynOS recipe directory
gotoaosrepo
# change to the directory holding the nano recipe
chpkg nano
# bump the release number in the nano recipe
just bump
# check the difference between the local state and the upstream recipe state
git diff
# build the bumped nano recipe
just build
# check the difference between the local state and the upstream recipe state
git status
# move the newly built .stone build artifacts to the local repository
just mv-local
# list the build artifacts present in the local repository
just ls-local
```

Note that the basic packaging workflow in AerynOS assumes that you are using a local repository.

If you are building multiple package recipes, you will need to `just build` and `just mv-local` for each package recipe sequentially.

# Updating the installed system state

Testing your package(s) is now as simple as:

- Enabling (or disabling) the relevant moss-format repositories with:

```
sudo moss repo enable/disable <the repository>
```

- Updating moss' view of the enabled moss-format repository indices with:

```
sudo moss sync -u
```

## Cleaning the local repository

Often, it will be prudent to clean out the local repository after the associated recipe PR has been accepted upstream.

```
gotoaosrepo
just clean-local
sudo moss repo disable volatile
sudo moss repo disable local
sudo moss sync -u
```

This will sync the the local system to a new installed system state made only from the upstream unstable moss-format .stone package repository state.

This will effectively make the new system state “forget” the nano version installed from the local repository in the previous system state.

## Ending notes

If you have made it this far, congratulations! You should now understand the basic workflow of packaging and managing repositories with AerynOS.

Tip: execute `just -l` to see a list of supported just ‘recipes’, which are common actions that have been automated by the AerynOS developers.

# Preparing for packaging

This page details prerequisite steps required before either creating a new package recipe or updating an existing package recipe. If you have not yet followed the [prerequisites](#) steps and [Basic Packaging Workflow](#), follow those steps first before proceeding.

## Update your clone of the recipes repository

As a reminder, you want to ensure you have the volatile repository enabled and fully updated on your system.

Whilst all the information can be found in the prior pages, to recap, the commands will be:

```
sudo moss repo enable volatile
sudo moss sync -u
gotoaosrepo
git switch main
gh repo sync yourusername/yourfork -b main
git pull
```

## Switch to a new git branch

When conducting any packaging work, it is a good idea to separate out your work in a different branch. This allows you to isolate changes you make from one package in a separate branch to changes you make to a different package in a second branch and so on. This additionally is helpful as it keeps your work separate to any underlying changes made to the main recipes repository, more easily allowing you to rebase your work if needed.

```
git checkout -b "branch-name"
```

Change “branch-name” to whatever description you feel comfortable with. Our general convention is to use the format add-packagename or update-packagename depending on whether you are adding a new package or updating an existing one.

You can check what branch you are on and what branches you have in your repository with the following command:

```
git branch -a
```

# Creating a new package recipe

This guide details the process of creating a new package recipe that is not yet present in the [AerynOS repository](#). We will use Nano as the running example, but the same steps apply to any new package.

Before creating the package recipe yourself, please double check that there isn't already an outstanding PR for the package you want to include. Please also check if someone has created a new package request issue in the [AerynOS recipes repository](#).

## Prepare your workspace

Prior to starting, ensure you have followed the [prerequisites](#) set up process, the [Basic Packaging Workflow](#) and updated your system in accordance with [Preparing for Packaging](#) guide.

If you have not done this, follow those steps first before proceeding.

## Scaffold the recipe directory

Prior to starting, you need to create the directory structure for your recipe. In our example, we will create a recipe for the Nano text editor. Each recipe is stored in its own directory within the recipes repository you already have downloaded to your computer. In this case, we will create a directory called nano in the n directory:

```
gotoaosrepo
mkdir -p n/nano
cd n/nano
```

## Fill the recipe step by step

The rest of this guide shows how to create a recipe and to replace any missing metadata by pulling information from upstream Nano.

### Step 1 - Collect upstream metadata

- Search for “GNU Nano download” to locate the upstream homepage: <https://www.nano-editor.org/>.
- Note the latest release number (8.7 at the time of writing) and the canonical download link.
- Record any prerequisites listed in upstream build instructions—these become candidates for `builddeps` later.

The Nano “bleeding edge” page lists the following tools you should keep in mind:

Package	Minimum Version
autoconf	2.69
automake	1.14
autopoint	0.20
gcc	5.0
gettext	0.20
git	2.7.4
groff	1.12
make	(any version)
pkg-config	0.22
texinfo	4.0

## Step 1 - Use boulder to help create the recipe

We use `boulder` to help create the recipe using the `boulder recipe new` command. This command will generate a skeleton recipe for you to fill in. `boulder` will read the contents of the source code of the package you are trying to add and automatically create a `stone.yaml` recipe file and a `monitoring.yaml` file.

```
boulder recipe new "upstream URL"
```

In the example of Nano, to create a recipe based on version 8.7, you would use the following command:

```
boulder recipe new https://www.nano-editor.org/dist/v8/nano-8.7.tar.xz
```

This command does the following:

1. Creates a new `stone.yaml` in your current directory for the package
  - Populates as many of the fields in the `stone.yaml` file as it can automatically identify
  - Checks the Sha256sum of the source code and inputs this in the recipe
2. Creates a new `monitoring.yaml` file in your current directory for the package
  - Populates as many of the fields in the `monitoring.yaml` file as it can automatically identify

Using Nano as an example, the generated `stone.yaml` file will look like this:

```
#
# SPDX-FileCopyrightText: © 2025- AerynOS Developers
#
# SPDX-License-Identifier: MPL-2.0
#
name      : nano
version   : 8.7
release   : 1
homepage  : https://www.nano-editor.org/dist/v8
upstreams :
  - https://www.nano-editor.org/dist/v8/nano-8.7.tar.xz :
afd287aa672c48b8e1a93fdb6c6588453d527510d966822b687f2835f0d986e9
summary   : UPDATE SUMMARY
description : |
  UPDATE DESCRIPTION
license   :
  - GFDL-1.2-invariants-or-later
  - GFDL-1.2-no-invariants-or-later
  - GFDL-1.2-or-later
  - GPL-3.0-or-later
  - GFDL-1.2-no-invariants-only
  - GFDL-1.2-invariants-only
  - GFDL-1.2-only
  - GPL-3.0-only
builddeps :
  - pkgconfig(ncurses)
  - pkgconfig(ncursesw)
setup     : |
  %configure
build     : |
  %make
install   : |
  %make_install
```

The second is a `monitoring.yaml` file which we will address later in this guide.

## Step 2 - Add core metadata fields

The `boulder recipe new` command has already made an attempt to populate the name, version, release and homepage. Please review these and correct them if necessary.

In this case, the following changes need to be made:

- Correct the homepage to <https://www.nano-editor.org/>.
- Update the summary to reflect the GNU Text Editor.
- Fill in the description field with a brief description of Nano.

```
name      : nano
version   : 8.7
release   : 1
homepage  : https://www.nano-editor.org/
upstreams :
  - https://www.nano-editor.org/dist/v8/nano-8.7.tar.xz :
    afd287aa672c48b8e1a93fdb6c6588453d527510d966822b687f2835f0d986e9
summary   : GNU Text Editor
description : |
  Nano is a small and simple text editor for use on the terminal.
  It copied the interface and key bindings of the Pico editor but
  added several missing features: undo/redo, syntax highlighting,
  line numbers, softwrapping, multiple buffers, selecting text by
  holding Shift, search-and-replace with regular expressions, and
  several other conveniences.
```

### Release numbering

Keep release at 1 when you introduce a brand-new package. We subsequently incrementally increase it by 1 each time we submit an update to our recipe repository.

## Step 3 - Declare / correct the license

Find the license in upstream's repository (often COPYING, LICENSE, or package metadata). Convert it to an [SPDX identifier](#).

Nano uses GPL-3.0-or-later.

```
license   :
  - GPL-3.0-or-later
```

## **SPDX Licence identifier**

The SPDX License List is a list of commonly found licenses and exceptions used in free and open or collaborative software, data, hardware, or documentation. The SPDX License List includes a standardized short identifier, the full name, the license text, and a canonical permanent URL for each license and exception.

The purpose of the SPDX License List is to enable efficient and reliable identification of such licenses and exceptions in an SPDX document, in source files or elsewhere.

## **Step 4 - Translate prerequisites into build dependencies**

Map each upstream requirement to the package name that exists in AerynOS. Use `pkg-config()` helpers when libraries provide `.pc` files. Toolchain components like `gcc` and `make` are already available inside the build environment, so you do not have to list them.

<b>Upstream prerequisite</b>	<b>Recipe build dependency</b>
<code>ncurses</code>	<code>pkgconfig(ncursesw)</code>
<code>zlib</code>	<code>pkgconfig(zlib)</code>
<code>libmagic</code>	<code>pkgconfig(libmagic)</code>
<code>autoconf, automake</code>	already provided by the sandbox

Add them to `builddeps`:

```
builddeps :
- pkgconfig(libmagic)
- pkgconfig(ncursesw)
- pkgconfig(zlib)
```

## **Step 5 - Fill in build steps**

Nano follows the GNU autotools flow, so uses the standard macros (%configure, %make, %make\_install). These have already been populated by boulder recipe new so do not need to be adapted. You can consult the [macros](#) documentation for variations and additional guidance.

## Step 6 · Review the finished recipe

Combining all the prior steps gives you a complete stone.yaml:

```
#
# SPDX-FileCopyrightText: © 2025- AerynOS Developers
#
# SPDX-License-Identifier: MPL-2.0
#
name      : nano
version   : 8.7
release   : 1
homepage  : https://www.nano-editor.org/
upstreams :
  - https://www.nano-editor.org/dist/v8/nano-8.7.tar.xz :
afd287aa672c48b8e1a93fdb6c6588453d527510d966822b687f2835f0d986e9
summary   : GNU Text Editor
description : |
  Nano is a small and simple text editor for use on the terminal.
  It copied the interface and key bindings of the Pico editor but
  added several missing features: undo/redo, syntax highlighting,
  line numbers, softwrapping, multiple buffers, selecting text by
  holding Shift, search-and-replace with regular expressions, and
  several other conveniences.
license   :
  - GPL-3.0-or-later
builddeps :
  - pkgconfig(libmagic)
  - pkgconfig(ncursesw)
  - pkgconfig(zlib)
setup     : |
  %configure
build     : |
  %make
install   : |
  %make_install
```

# Update/correct the `monitoring.yaml` file

Release monitoring keeps automated eyes on your package. More details around our monitoring file can be found on our [Monitoring](#) page.

As mentioned earlier in this guide, the `boulder recipe new` command has already attempted to create a `monitoring.yaml` file for you.

In the case of Nano, it wasn't able to uniquely identify the project so the output was not as valuable and needs to be corrected.

For reference, its output is as below:

releases:

id: ~ # <https://release-monitoring.org/> and use the numeric id in the url of project

rss: ~

security:

cpe:

- vendor: gnu  
product: nano
- vendor: nano\_arena\_project  
product: nano\_arena
- vendor: viz  
product: nano\_id
- vendor: lenovo  
product: thinkpad\_x1\_nano\_gen\_1\_firmware
- vendor: nvidia  
product: jetson\_nano
- vendor: lenovo  
product: thinkpad\_x1\_nano\_gen\_2\_firmware
- vendor: lenovo  
product: thinkpad\_x1\_nano\_gen\_2
- vendor: lenovo  
product: thinkpad\_x1\_nano\_gen\_1
- vendor: nxp  
product: mifare\_ultralight\_nano\_firmware
- vendor: jtekt  
product: nano\_cpu\_tuc-6941\_firmware
- vendor: jtekt  
product: nano\_10gx\_tuc-1157\_firmware
- vendor: autelrobotics  
product: evo\_nano\_drone\_firmware
- vendor: jtekt  
product: nano\_safety\_rs01ip\_tuu-1087
- vendor: jtekt  
product: nano\_safety\_rs00ip\_tuu-1086
- vendor: netshieldcorp  
product: nano\_25\_firmware
- vendor: ledger  
product: nano\_x\_firmware
- vendor: ledger  
product: nano\_s\_firmware
- vendor: jtekt  
product: nano\_cpu\_firmware

```
- vendor: jtekt
  product: nano_2et_firmware
- vendor: jtekt
  product: nano_10gx_firmware
- vendor: nxp
  product: mifare_ultralight_nano
- vendor: nxp
  product: i.mx_8m_nano
- vendor: nvidia
  product: jetson_nano_2gb
- vendor: jtekt
  product: nano_safety_tuc-1085
- vendor: jtekt
  product: nano_cpu_tuc-6941
- vendor: jtekt
  product: nano_2et_tuu-6949
- vendor: jtekt
  product: nano_10gx_tuc-1157
- vendor: autelrobotics
  product: evo_nano_drone
- vendor: netshieldcorp
  product: nano_25
- vendor: ledger
  product: nano_x
- vendor: ledger
  product: nano_s
- vendor: jtekt
  product: nano_cpu
- vendor: jtekt
  product: nano_2et
- vendor: jtekt
  product: nano_10gx
- vendor: magzter
  product: nano_digest
```

Update monitoring.yaml once you know the upstream identifiers:

1. Search for the project on <https://release-monitoring.org/> and copy the numeric id.
2. Add an RSS or Atom feed URL if upstream publishes one; otherwise leave rss: ~.
3. Check the National Vulnerability Database for a CPE string (<https://nvd.nist.gov/products/cpe/search>). If none exists, leave it as ~.

Using this information we can correctly identify the id as 2046 and that there is no rss feed or cpe string. The `monitoring.yaml` file should look like this:

```
releases:
  id: 2046           # Release Monitoring ID for nano
  rss: ~           # Replace when upstream publishes a feed
security:
  cpe: ~           # Update if upstream changes identifiers
```

## Build and test the package

Once you have made the relevant changes to the package, you will need to build it locally. Refer to the [Building and Testing packages](#) page on guidance of how to do this.

# Updating an existing package recipe

This guide details the process of updating a package that is already present in the [AerynOS repository](#). We will use GNU Nano as the running example, but the same steps apply to any existing package.

Before updating the package yourself, please double check that there isn't already an outstanding PR for the update you want to make. Please also check if someone has created an update request issue in the [AerynOS recipes repository](#).

## Prepare your workspace

Prior to starting, ensure you have followed the [prerequisites](#) set up process, the [Basic Packaging Workflow](#) and updated your system in accordance with [Preparing for Packaging](#) guide.

If you have not done this, follow those steps first before proceeding.

## Simple updates to a package

To update a package to a newer available version, navigate to the relevant folder within your local recipe repository on your system. If you have already navigated to the local recipe repository, then by way of example, to navigate to the Nano package folder, you would use the command:

```
chpkg nano
```

## Bumping a package

If there are changes to dependencies of a package, but not to the package itself, you need to increase the release number within the `stone.yaml` recipe file by one. This will allow you to rebuild the package and test it against the newer dependences to make sure everything is working. You can do this by using the following command:

```
just bump
```

which is a shortcut for

```
boulder recipe bump
```

Please note each time you do this, you will increase the release number by one, so do not use this command multiple times for one package update.

## Updating a package version [🔗](#)

If you need to update the package version itself, you can use the following command:

```
boulder recipe update --ver "version name" --upstream "upstream URL"  
stone.yaml -w
```

In the example of nano, to update to version 8.7, you would use the following command:

```
boulder recipe update --ver 8.7 --upstream https://www.nano-  
editor.org/dist/v8/nano-8.7.tar.xz stone.yaml -w
```

This command does the following:

1. Updates the version of the package within the recipe
2. Updates the upstream location of the source code
3. Checks the Sha256sum of the source code and inputs this in the recipes
4. Bumps the release number by 1

## Wider updates to a package [🔗](#)

If there are further changes required to the `stone.yaml` recipe file, you can either use a text editor (such as Nano itself) or a code editor (such as Zed which is pre-installed on AerynOS) to make changes those changes. Guidance on how to make changes to a `stone.yaml` file are covered in the [Creating a new package recipe](#) page.

## Build and test the package

Once you have made the relevant changes to the package, you will need to build it locally. Refer to the [Building and Testing packages](#) page on guidance of how to do this.

# Building and testing packages

This guide will walk you through the process of building and testing packages locally on your system, regardless of whether they come from new package recipes or existing ones you are updating.

## Build the package

Once you have created or updated a package recipe, you will need to build it locally. If you are only updating one package, you can either keep your local repository disabled prior to building the package. If you prefer to keep it enabled, make sure there are no other packages indexed locally that could interfere with your new package build.

### Note

Please ensure you have followed the steps in the [Preparing for Packaging](#) guide to ensure your volatile repository is enabled.

The command to build the updated package is:

```
just build
```

If the package is successfully built, you will need to move it to your local repository. You can do this using the following command:

```
just mv-local
```

If you have not yet enabled the local repository, you do this with the following command:

```
sudo moss repo enable local
```

You will then need to sync the local repository using the command:

```
sudo moss sync -u
```

Note, if you already have an older version of the package installed, you will be asked if you want to update to the new local version you have just built. If you have not yet installed this package, you would install it as normal using the command:

```
sudo moss install "package name"
```

Once you have tested the package, you can make a submission for including the update in the repository.

## **How to submit pull requests**

To find guidance on how to submit a pull request (PR), you can refer to our [submit a pull request](#) page. s

# Submitting a PR

## Submitting packages to AerynOS repository

Once you have prepared your package, you can submit it to the AerynOS repository by creating a pull request (PR). There are certain guidelines to follow when submitting a PR:

### Naming Pull Requests

To keep git summaries readable, AerynOS requires the following git summary format

- name: Add at v<version>
- name: Update to v<version>
- name: Fix <...>
- [NFC] name: <description of no functional change commit>

#### **No Functional Change**

NFC refers to “No Functional Change”, which means that the commit does not introduce any new functionality or behavior, so a recipe does not need to be rebuilt as part of the PR process.

### Content of Pull Request descriptions

Git commits should be self-contained and self-explanatory. They serve as documentation for the changes made to a codebase so that others can understand and review them and refer back to them later down the line. It is important to provide high quality git commit messages so that you or other contributors can understand the changes you are making and why.

While you know what you’re doing in the moment, other contributors may not. As time goes by, bisecting changes becomes more difficult if commit messages give you no clue as to why you made a change or what regressions might be caused if you alter it.

## Commit message format

The recommendation for commit messages is:

- Short summary written in the imperative mood
- A few sentences or bullet points with the key changes this commit introduces
- Link to full changelog (if applicable). If this commit updates the recipe several versions, consider splitting the changelog out into version bullet point entries in ascending order (newest change last).
- Test plan demonstrating that you have actually confirmed the changes work on your local system
- If the change resolves an issue, include a Resolves line with the issue number (Where `issuenum` is the issue number of the package request/update).

The last point about the test plan is particularly important, as it ensures that the changes have been tested and verified before being merged into the main codebase. There is an explicit agreement that you take ownership of the quality of the changes/updates you submit, and that you understand that if there are issues, you are likely to be the first person consulted to fix said issues.

### **The imperative mood**

Git commits should be written in the [imperative mood](#). This means that the commit message should start with a verb in the present tense, such as “Add”, “Update”, or “Fix”. This makes the commit message more concise and easier to understand.

## Example commit message

An example commit message for the AerynOS recipe repository is structured as follows:

brobdingnar: Update to v1.2.3

Write a suitable short summary of the changes if relevant, including potentially a list of things like:

- foo
- bar
- baz

Full changelog [here](the.uri)

**Test Plan**:

- Build and install the updated package
- Confirm functionality of changes

**Resolves**:

(If applicable for the recipes repository) Resolves aerynos/recipes#issuenumbr

# Checking for package updates

## Use ent to check for package updates [🔗](#)

This guide will walk you through the process using `ent`, a tool built by the AerynOS team to check for package updates. `ent` checks recipes against upstream sources to determine whether updates are available.

`ent` is not installed on your system by default. To install `ent` using `moss`, use the following command:

```
sudo moss install ent
```

## How ent works [🔗](#)

`ent` scans the current directory and all subdirectories beneath it. It inspects each recipe `monitoring.yaml` file and compares the referenced `stone.yaml` recipe upstreams to determine whether newer versions are available.

Because `ent` operates relative to the directory in which it is executed, you can control the scope of the update check by choosing where to run the command within the recipes repository.

## Running update checks [🔗](#)

To check for updates across all recipes, run the following command from the root of the recipes repository:

```
gotoaosrepo  
ent check updates
```

You can also run this command from more specific locations:

- **Repository root** Checks all recipes in the repository.

- **Letter directory (for example, f/)** Checks only recipes whose names start with that letter.
- **Specific recipe directory (for example, f/firefox/)** Checks only that single recipe.

For example, running the command from `f/firefox/` will check only the Firefox recipe for available updates.

### **What is ent?**

ent queries an upstream site for package release info *every time you run it*. Please be mindful of not running it gratuitously so as to remain a good ecosystem citizen.

# Recipes

## Overview



Introduction to the `stone.yaml` format

## Upstreams



Configuring where the recipe finds the 'sources' required for a build to work

## Metadata



Keys and options to tweak the metadata for a recipe

## Monitoring



Create and maintain monitoring.yaml so release automation and security alerts stay accurate.

## Build dependencies



Build dependency types

## Package definitions



Manage dependencies, subpackages and more

## Triggers



Triggers are system actions that run during package installation

## System Accounts



Stateless management of packaging-based system accounts

# Overview

Simply put, a recipe is some metadata to describe a software package, and the associated *instructions* required to build that package in a reproducible fashion. Doing so allows us to automate builds, and provide software updates. At a surface level, our `stone.yml` recipe format has an awful lot in common with other packaging systems.

## A basic recipe

How might a `stone.yml` look like for a very trivial package, such as the [Nano editor](https://www.nano-editor.org/)?

```
name      : nano
version   : 8.7
release   : 38
homepage  : https://www.nano-editor.org/
upstreams :
  - https://www.nano-editor.org/dist/v8/nano-8.7.tar.xz :
afd287aa672c48b8e1a93fdb6c6588453d527510d966822b687f2835f0d986e9
summary   : GNU Text Editor
description : |
  Nano is a small and simple text editor for use on the terminal.
  It copied the interface and key bindings of the Pico editor but
  added several missing features: undo/redo, syntax highlighting,
  line numbers, softwrapping, multiple buffers, selecting text by
  holding Shift, search-and-replace with regular expressions, and
  several other conveniences.
license   :
  - GPL-3.0-or-later
builddeps :
  - binary(msgfmt)
  - pkgconfig(libmagic)
  - pkgconfig(ncursesw)
  - pkgconfig(zlib)
setup     : |
  %configure
build     : |
  %make
install   : |
  %make_install
```

..It really is that simple. However, do not let the simplicity of the format fool you, boulder has a lot of hidden powers.

# Upstreams

The majority of packages are built using upstream release sources. While it is possible to create packages manually from local assets, the bulk of packages take an upstream tarball and build it.

## Plain sources

A plain source is one that simply has an upstream URI and can be unpacked in some fashion, i.e. a tarball. The hash must be provided for the upstream and accompanied by the SHA256 sum.

```
upstreams:  
  - uri: $hash
```

```
upstreams  
  - uri:  
    hash: $hash
```

## Additional options

Key	Type	Description
hash	string	SHA256 of the upstream source
stripdirs	string	Number of directories to remove from archive root
unpack	boolean	Whether to automatically unpack archive or not
unpackdir	string	Force a different directory name when unpacking

## Git sources

A git source may be used, when providing either a tag or ref. In AerynOS we forbid the use of branch names in packaging, as they may mutate and break subsequent builds. Ideally a full git ref should be used.

## **Caution**

Git repositories do work well with boulder right now, however some submodule based builds are under active testing.

```
upstreams:  
  - git|uri: $ref
```

```
upstreams:  
  - git|uri:  
    ref: $ref
```

## **Additional options**

<b>Key</b>	<b>Type</b>	<b>Description</b>
ref	string	git ref when using git source
unpackdir	string	Override clone target directory

# Metadata

Recipes provide basic metadata to support discovery and automation.

Certain data is purely for naming, others are purely functional and *some* are used for our integration tooling. By having a well defined format with strongly typed keys, we're able to build in automatic update checking, for example. Most importantly, we need users to be able to find the software!

## Mandatory keys [🔗](#)

The following metadata keys are absolutely essential.

### name [🔗](#)

Set the source name of the package. As closely as possible, this should match the upstream name. This is used as the basename of the package when subpackages are automatically generated, for example:

```
name: zlib
```

Could generate `zlib`, `zlib-devel`, `zlib-dbginfo`, etc.

### version [🔗](#)

This string tells users what version they are using, and isn't used at all for any kind of version comparison logic in the tooling. It is essentially a freeform string. It **should** be identical to the upstream identifier so that we can detect new releases automatically of the source project.

### release [🔗](#)

A monotonically incrementing integer. This field is bumped whenever we need to issue a new build ("release") of a package as an update to users. Without incrementing this field, no build is scheduled.

### homepage [🔗](#)

Web presence for the upstream project.

## **license**

Either a string or list of strings denoting all applicable licenses, using their [SPDX](#) identifier. Required for basic compliance.

# Monitoring

Every recipe should ship a `monitoring.yaml` so our tooling can watch for upstream releases and security issues. Use this reference to populate the file consistently and to find the data required for each field.

## File layout

A minimal monitoring file includes release tracking and optional security metadata:

```
releases:
  id: 00000
  rss: https://example.com/project/releases.atom
security:
  cpe:
    - vendor: example
      product: project
```

Indent with two spaces and keep related comments inline so CI and reviewers can follow your reasoning.

## Release tracking

`releases.id` : Numeric identifier from [release-monitoring.org](https://release-monitoring.org) (Anitya). Look up the upstream project and note the number in the URL, for example `https://release-monitoring.org/project/300958` for Python.

`releases.rss` : URL to an Atom/RSS feed for new releases. Use `~` if no feed exists.

## Common feed patterns

- **GitHub**: `https://github.com/<org>/<repo>/releases.atom` or `.../tags.atom`
- **GitLab / KDE Invent**: append `/-/tags?format=atom` to the project URL, for example `https://invent.kde.org/plasma/plasma-desktop/-/tags?format=atom`
- **PyPI**: no feed is required; prefer `~` and rely on the Anitya ID
- **Freedesktop GitLab**: `https://gitlab.freedesktop.org/<path>/-/tags?format=atom`

- **Custom sites:** many upstreams publish a `releases.xml/atom.xml` file; link directly when available

## Ignore patterns

Use `releases.ignore` to skip versions our repo does not track. Provide a short comment and regular expressions that match the releases to drop.

```
releases:
  id: 320206
  ignore:
    # Qt 6 builds are out of scope for qt5 packages
    - ^6\.
  rss: ~
```

Prefer anchored expressions (`^ / $`) to avoid false positives.

For reference, `^` means “begins with”, while `$` means “ends with”.

## Security metadata

`security.cpe` : List of Common Platform Enumeration entries to watch in the NVD feed. Search [nvd.nist.gov](https://nvd.nist.gov) for vendor and product strings. Add every applicable CPE when upstream ships multiple identifiers.

`security.ignore` : Optional list of CVE IDs or regexes our package should ignore (for example, CVEs that only affect optional components).

If no CPE exists, set the value to `~` and add a dated comment noting the last time you checked.

```
security:
  cpe: ~
  # No known CPE as of 2024-09-01
```

Assuming that the [repository helper script](#) has been sourced for your shell, you should be able to use the `cpesearch` function to search for related CPEs for the package given as the argument.

Example:

cpsearch urllib3

## Where to find the data

1. **Start with release-monitoring.org**: search for the upstream name.
2. **Collect feeds**: confirm the `releases.atom` or `/-/tags?format=atom` URL opens in a browser. Use `curl` or `wget -q0- <feed>` locally when you need to double-check.
3. **Identify CPE strings**: search the NVD catalog or reuse values from similar recipes. Many projects share vendor IDs (for example, both upstream `python` and the `urllib3` package provide CPEs).
4. **Document exceptions**: add comments whenever you set ignore patterns or leave fields empty so future maintainers understand the decision.

## Example templates

### GitHub project with security feed

```
releases:
  id: 4078
  rss: https://github.com/urllib3/urllib3/releases.atom
security:
  cpe:
    - vendor: urllib3
      product: urllib3
    - vendor: python
      product: urllib3
```

### GitLab project with prerelease filter

```
releases:
  id: 5440
  ignore:
    # Track the current stable branch only
    - 258.*
  rss: https://gitlab.freedesktop.org/systemd/systemd/-/tags?format=atom
security:
  cpe:
    - vendor: systemd\_project
      product: systemd
```

## No CPE available

```
releases:
  id: 19755
  rss: ~
security:
  cpe: ~
  # No CPE published as of 2023-03-23
```

Keep monitoring files in sync with upstream changes. When a project moves or renames releases, update the ID and feed so our automated tooling continues to work.

# Build dependencies

Every build of a recipe by `boulder` will create an entirely new root, with only the absolute minimum support dependencies in place. In order to build most software, you will need to add to the `builddeps` key in `stone.yml`. Luckily, our tooling supports more than one kind of dependency.

Note that AerynOS packages are also capable of storing **providers** that make the following kinds of dependencies work.

## `$name` - standard deps

Simply listing a name will create a dependency on that package name. This is discouraged as automatically resolved providers offer a far more resilient system.

```
builddeps:  
  - some-package
```

## `binary()` - Standard binaries

Got a hard requirement for an executable in `/usr/bin`, such as `grep`?

```
builddeps:  
  - binary(grep)
```

## `sysbinary()` - System binaries

Need an executable only found in `/usr/sbin`?

```
builddeps:  
  - sysbinary(mount)
```

## `pkgconfig()` - PkgConfig / pkgconf

Trivially map package names to standard `pkgconfig` names (`.pc` files):

```
builddeps:  
- pkgconfig(ncurses)  
- pkgconfig(zlib)
```

## pkgconfig32() - 32-bit PkgConf

Much like pkgconfig - specifically designed for .pc files installed to /usr/lib32/pkgconfig in 32-bit builds:

```
builddeps:  
- pkgconfig32(x11)
```

## cmake() - CMake modules

Work with many C++/CMake builds much more easily by using the CMake module names

```
builddeps:  
- cmake(Qt5OpenGL)
```

# Package definitions

A recipe build can result in a number of packages being produced from a single source, through an automatic splitting system. Certain subpackages are already defined in the boulder project to ensure consistency of package splitting and names, whereas some may be explicitly defined in a recipe to fine-tune the results.

Every recipe also contains a **root package definition**, i.e the default target. This is merged with the standard metadata.

## Package metadata

### summary

A brief, one line description of the package based on its contents.

### description

A more in depth description of the package, usually sourced from a README or project description.

### rundeps

A list of manually specified runtime dependencies. These may be added to ensure that one split package depends on another, or to add a hard dependency that is not accounted for by the automatic systems.

Example:

```
rundeps:  
  # Depend on subpackage in this set ending with `~devel`  
  - "%(name)-devel"  
  - filesystem
```

## Defining a subpackage

Additional packages may be defined by extending the packages set, and matching a set of paths to include in that subpackage.

For example:

```
packages:  
  - "%(name)-tools":  
    summary: Cool tools package  
    description: |  
      Provides a cool set of tools!  
    paths:  
      - /usr/bin/extra-tool
```

Note that automatic dependencies and providers still work with subpackages, so binary deps will resolve without having to manually specify those.

## Overriding defaults [🔗](#)

To override splitting in the root package, for example, to avoid `-devel` subpackage when building a headers-only package, you could do:

```
paths:  
  - /usr/include
```

To add to a predefined package, such as `-docs`:

```
packages:  
  - "%(name)-docs":  
    paths:  
      - /usr/share/custom-docs
```

# Triggers

## Overview [→](#)

Triggers match filesystem paths to actions

## Transaction triggers [→](#)

Transaction triggers run in confinement to finish package configuration tasks

# Overview

AerynOS supports the use of triggers, or actions, that run at the end of package installations. Given the significantly different architecture of AerynOS, these triggers may not be quite what you are used to in other distributions or package managers.

## Basic mechanism [🔗](#)

After a new transaction is formed and moss has identified all the paths used to compose a filesystem, the staging tree is built as the basis of the new `/usr`. Any trigger files (under `/usr/share/moss/triggers`) will be loaded, and any **matching** triggers will be executed at the appropriate stage.

Note that trigger logic is based on glob-style path matches and are not incremental. Our triggers were so designed to avoid the uncontrolled execution of arbitrary scripts, instead relying on logical matching of patterns to handlers.

## Capturing globs [🔗](#)

Our triggers use special string tokens to permit capturing groups from a glob-style string. At this stage we support `*` and `?` glob characters only, compiling to a regex internally. Support is planned for braces.

```
/usr/lib/(GROUP_NAME:PATTERN)/dir
```

The parenthesis begin a non-greedy capture group named `GROUP_NAME` containing pattern `PATTERN`. For example:

```
/usr/share/icons/(name:*)/index.theme
```

This creates a capture group identified by name matching `*` in `/usr/share/icons/*/index.theme`. As such, the path `/usr/share/icons/hicolor/index.theme` with name being set to `hicolor`.

This is a powerful mechanism that allows us to control handler execution without relying on interim scripts.

Consider this example:

```
/usr/lib*/(libname:lib*.so.*)
```

This will only match `lib*.so.*` glob, and set `libname` to `libz.so.1` for `/usr/lib32/libz.so.1`, but will not match for `/usr/lib64/libz.so`.

These globs are then used for string substitution in the arguments passed to handlers.

# Transaction triggers

Transactional scope triggers (tx triggers) are run after the new filesystem transaction has been blitted to disk, and just before the new /usr tree is activated. These triggers run within a specialized container and have read-write access to the new /usr tree, but only have read-only access to the /etc directory.

Transaction triggers must be installed in /usr/share/moss/triggers/tx.d with a .yaml suffix.

## Sample trigger

This simple trigger will run `depmod -a 6.6.15` when any files are installed to `/usr/lib/modules/6.6.15/`. Note that identical commands (after expansion) will be collapsed automatically to a single run.

```
name: depmod
description: |
    Update kernel module dependencies

# Define all of our handlers
handlers:
  depmod:
    # Run `depmod` with these arguments
    run: /usr/sbin/depmod
    args: ["-a", "${version}"]

paths:
  # Set up a match
  "/usr/lib/modules/(version:*)/*" :
    # Run these handlers for this match.
    handlers:
      - depmod
    type: directory
```

To install this trigger in your recipe:

```
%install_file %(pkgdir)/trigger.yaml %  
(installroot)/usr/share/moss/triggers/tx.d/gdk_pixbuf.yaml
```

# System Accounts

## Groups



Stateless management of system group accounts

## Overview



Stateless management of AerynOS user accounts

## Users



Stateless management of system user accounts

# Groups

Refer to the [JSON Group Record](#) documentation for information on all supported fields.

## Example

Within the package tree `./pkg` add `gdm.group`:

```
{
  "groupName" : "gdm",
  "gid" : 21,
  "disposition" : "system"
}
```

Note that these are the minimum required set of fields, and `disposition` should always be set to `system`.

In your recipe's `install` section, you must install the file by group name *and* by gid to the `%(libdir)/userdb` directory:

```
%install_file %(pkgdir)/gdm.group %(installroot)%
(libdir)/userdb/gdm.group
ln -s gdm.group %(installroot)%(libdir)/userdb/21.group
```

# Overview

As a stateless distribution, AerynOS does not permit the modification of `/etc/passwd` and `co` by packages or triggers. Instead, we integrate `nss-systemd` and `userdb`.

## **Caution**

The use of `nss` means that user accounts and groups defined by this mechanism are only available to packages using the correct `glibc` APIs. Statically linking with `musl` or directly reading `/etc/passwd`, `/etc/group`, etc, will not reveal these accounts.

The main benefit with this approach is ensuring that we do not directly mutate system files, and that unlike the `sysusers` mechanism, removal of a package ensures these system user and group definitions are no longer available.

# Users

System accounts should *always* be marked as locked. Refer to the [JSON User Record](#) documentation for information on all supported fields.

In AerynOS we only ship user definitions without privileged or signature fields.

## Example

Within the package tree `./pkg add gdm.user:`

```
{
  "userName" : "gdm",
  "realName" : "GNOME Display Manager",
  "uid" : 21,
  "gid" : 21,
  "disposition" : "system",
  "locked" : true
}
```

Note that these are the minimum required set of fields, and `disposition` should always be set to `system`. Also note that `homeDirectory` may need setting for some packages.

In your recipe's `install` section, you must install the file by username *and* by uid to the `%(libdir)/userdb` directory:

```
%install_file %(pkgdir)/gdm.user %(installroot)%(libdir)/userdb/gdm.user
ln -s gdm.user %(installroot)%(libdir)/userdb/21.user
```

# Macros

Every `stone.yaml` build has access to a number of **action** and **definition** macros. With these, we can ensure a greater degree of integration and consistency in our packaging, and vastly simplify common tasks to reduce maintainer burden.

## **autotools**

autotools macros



## **cargo**

Rust project builds



## **cmake**

cmake build system



## **meson**

meson build system



## **Miscellaneous**

misc helpers



## **perl**

perl module packaging



## **python**

python packaging



# autotools

The autotools macros are used for projects that supply a Makefile, and potentially a ./configure script.

## %configure

Perform ./configure with the default options

## %make

Perform a make

## %make\_install

Install results of build to the destination directory

## %reconfigure

Re autotools-configure a project without an autogen.sh script

## %autogen

Run autogen.sh script, attempting to only configure once

# cargo

When building “pure” [Rust](#) packages with the cargo build tool, ensure you use the `%cargo*` macros to allow boulder to control the various tuning options and debuginfo behavior.

## **%cargo\_set\_environment**

Set environmental variables for Cargo build

## **%cargo\_fetch**

Fetch dependencies

## **%cargo\_build**

Build the rust project

## **%cargo\_install**

Install the built binary

## **%cargo\_test**

Run tests

# **cmake**

## **%cmake**

Perform cmake with the default options in a subdirectory

## **%cmake\_unity**

Perform cmake with unity build enabled

## **%cmake\_build**

Build the cmake project

## **%cmake\_install**

Install results of the build to the destination directory

## **%cmake\_test**

Run testsuite with ctest

# **meson**

## **%meson**

Run meson with the default options in a subdirectory

## **%meson\_unity**

Run meson with unity build enabled

## **%meson\_build**

Build the meson project

## **%meson\_install**

Install results of the build to the destination directory

## **%meson\_test**

Run meson test

# Miscellaneous

## %install\_bin

Install files to %(bindir)

Example usage

```
%install_bin nano
```

## %install\_dir

Create a new empty directory with the default permissions

Example usage

```
%install_dir %(installroot)%(datadir)/pkgname/docs
```

## %install\_exe

Macro to install a file with default executable permissions

## %install\_file

Macro to install a file without executable permissions

Example usage

```
%install_file %(pkgdir)/helper.file %(installroot)%(datadir)/pkgname/pkgfile
```

## %patch

Patch the upstream sources using an input patch file.

Example usage

```
%patch %(pkgdir)/${file}
```

```
# If you need to override -p#, add it after ${file}
```

```
%patch %(pkgdir)/some.patch -p3
```

## **%tmpfiles**

Create a tmpfiles.d file for the package with given content

## **%sysusers**

Create a sysusers.d file for the package with given content

# perl

## %perl\_setup

Setup perl with ExtUtils::MakeMaker from stdlib

# **python**

## **%python\_setup**

Perform python setup and build with the default options

## **%python\_install**

Install python package to the destination directory

## **%pyproject\_build**

Build a wheel for python PEP517 projects

## **%pyproject\_install**

Install wheel to destination directory

## **%python\_compile**

Compile .pyc bytecode files from any miscellaneous .py files in the install directory.

# Developers

## **Caution**

This documentation is only a stub and serves as a placeholder for future content. In time, the full format and payloads of `of moss` will be documented, along with other technologies such as `blsforme`, `os-info`, etc.

AerynOS includes some bespoke technologies and formats that are used to package, distribute, and introspect deployed software. This section of the documentation provides an overview of these technologies and formats.

### **Stone Format**



An overview of the Stone format

# Stone Format

The Stone format is a binary format designed to be type-safe and version-aware. It is used to package and distribute software in AerynOS, in fact both the packages themselves and the index file of repositories use the Stone format.

Anything encoded in the Stone format is called a *stone*. Each *stone* is composed of a Prelude (the global header) and zero or more payloads, each with its own sub-header. No limit is set for the length of a *stone*, but it will always be at least 32 bytes long, that is the size of the Prelude.

To completely encode or decode a given *stone*, the Stone version must be taken into account, as different versions may support different contents. The version is stored in the Prelude, as explained in the pages to come. For now, it is sufficient to remember that each *stone* targets exactly one version.

Described below is the general (that is, version-agnostic) layout of a *stone*.

```
%%{
  init: {
    'packet': {
      'showBits': false
    }
  }
}%%
packet
+32: "Prelude – 32 bytes"
+16: "Payload 1 header – 32 bytes"
+16: "Payload 1 records – Variable length"
+16: "Payload 2 header – 32 bytes"
+16: "Payload 2 records – Variable length"
+32: "[...]"
```

The first payload, if present, immediately follows the Prelude.

The fundamental types of the Stone format are integer numbers and strings.

Type	Format	Size (bytes)	Description	Abbreviation
Signed integer	Big-endian	Variable	Integer number with a sign. When negative, two's complement is used. The range of values goes from $-2^{(n-1)}$ to $2^{(n-1)}-1$ , where $n$ is the number of bytes.	uint
Unsigned integer	Big-endian	Variable	Integer number without a sign. The range of values goes from 0 to $2^{(n)}-1$ , where $n$ is the number of bytes.	int
String	UTF-8	Variable	String of text, without the NULL termination.	str
Blob	Undetermined	Variable	Meaningless or context-dependent array of bytes.	blob

The length of a field is documented where it is known a priori. When this is not possible, it is always paired with another field that reveals its actual length. The position of the latter depends on the version of the Stone format in use.

The documentation will delve into the details of each section in the next pages.

### **Prelude**



The version-agnostic header of Stones

### **Overview**



Overview of the Stone v1 format

# Prelude

*Stones* are encoded with a version agnostic header, the Prelude, ensuring that version-specific fields can be handled separately from version and format detection. This is a 32-byte header at the start of the *stone*.

## Fields

Field	Type	Size (bytes)	Description
magic	str	4	Always 0x006d6f73.
data	Version-dependent	24	Version-specific header of the <i>stone</i> .
version	uint	4	Version number, i.e. 1.

## magic

It's the [magic number](#) of the Stone format.

The magic field always contains ['\0', 'M', 'O', 'S']. It is defined after AerynOS's package manager: moss.

In the Rust language it is defined as:

```
pub const STONE_MAGIC: &[u8; 4] = b"\0mos";
```

## data

The content of the data field depends on the Version field. Documentation about Stone versions is available in the next pages.

## version

A number that uniquely identifies the version of the Stone format in use.

# Overview

The v1 of the Stone format is the version currently employed by AerynOS, and is the first revision of our format.

## Tip

The contents below extend the version-agnostic components of the Stone format. Readers are encouraged to read the [Stone format overview](#) first.

v1 revolves around the concept of records.

A payload contains one or more records, all of the same type, which specified in the payload's sub-header; the record's type describes the information it carries.

Records within a payload may be compressed as a whole archive (not individually) using [Zstandard](#).

With the exception of the [Content](#) record, all records have a fixed size determined by their type, or have a preamble that reveals the final size. The Content record is unique in that it spans the entire payload and must be the only record it contains.

```
%%{
  init: {
    'treemap': {
      'showValues': false
    }
  }
}%%
treemap-beta
"Payload 1"
  "Header": 32
  "Records"
    "Record 1": 8
    "Record 3": 8
    "Record 2": 8
    "Record 4": 8
"Payload 2"
  "Header": 32
  "Records"
    "Record 1": 8
    "Record 2": 8
"...": 47
```

## Header



The header of the Stone v1 format

## Payload's sub-header



The content of Payload's sub-header

## Attribute



The format of the Attribute record

# Header

The v1 header is contained in [Prelude](#)'s 24-byte Data field. It contains fields to denote the type of the *stone* as well as the number of payloads.

## Fields [↗](#)

Field	Type	Size (bytes)	Description
num_payloads	uint	2	Number of payloads within the <i>stone</i> .
padding_chk	blob	21	Corruption check (fixed content).
type	uint	1	Denotes the <i>stone</i> 's type.

## padding\_chk [↗](#)

padding\_chk contains a fixed array as a mild corruption check: [0, 0, 1, 0, 0, 2, 0, 0, 3, 0, 0, 4, 0, 0, 5, 0, 0, 6, 0, 0, 7].

## type [↗](#)

type denotes the intended use of a *stone*. Applications may consider the value of this field when executing logic, but it doesn't affect the way a *stone* is encoded or decoded.

Value	Type	Description
1	Binary	Standard package.
2	Delta	Currently unused.
3	Repository	Index of a repository of packages.
4	BuildManifest	Build-time artefact containing the <i>yield potential</i> of a package.

# Payload's sub-header

Described below is the format of the 32-byte-long payload sub-header. This format is unique to v1.

Field	Type	Size (bytes)	Description
stored_size	uint	8	Compressed size, in bytes, of the records. If records are not compressed, stored_size is equal to plain_size.
plain_size	uint	8	Uncompressed size, in bytes, of records.
checksum	blob	8	Checksum of records, computed using <a href="#">XXH3_64bits</a> .
num_records	uint	4	Number of records contained in the payload.
version	uint	2	Version of the payload data format. Unused.
record_kind	uint	1	Kind of records that the payload contains.
compression	uint	1	Compression algorithm used for the records.

## record\_kind

record\_kind is an enum. The following table lists all possible values:

Value	Name
1	<a href="#">Meta</a>
2	<a href="#">Content</a>
3	<a href="#">Layout</a>
4	<a href="#">Index</a>
5	<a href="#">Attributes</a>

## compression

Value	Name
1	Uncompressed
2	Zstd

# Attribute

The Attribute record is a general-purpose key-value pair. The format of the key and the value is undetermined: applications will use them according to the context they operate in. The Attribute record has no fixed length, but it is at least 16 bytes long.

<b>Field</b>	<b>Type</b>	<b>Size (bytes)</b>	<b>Description</b>
key_size	uint	8	Size, in bytes, of the key field.
value_size	uint	8	Size, in bytes, of the value field.
key	blob	Variable	General-purpose key.
value	blob	Variable	General-purpose value.

# Content

The Content record contains a blob. Its size is indicated in the Payload sub-header, as this kind of record spans the entire payload. There must be exactly one Content record inside a payload.

# Index

The Index record isolates a region of a Content record. This is especially useful when the Content's blob is composed of heterogenous data that must be managed separately.

The Index record does not specify which Content record it refers to; applications may use custom logic to disambiguate the correct record (e.g., a *stone* may be encoded with Index records followed by the Content they refer to). The trivial case is a *stone* with only one Content record.

Field	Type	Size (bytes)	Description
start	uint	8	The offset in the Content blob where this region begins. This is a zero-based index: the first byte is addressed with 0.
end	uint	8	The offset in the Content blob where this region ends. This last byte is included in the range. This is a zero-based index: the first byte is addressed with 0.
hash	blob	16	Checksum of the content of the region, computed using <a href="#">XXH3_128bits</a> .

Illustrated below is an example of two Index records addressing the same Content.

```
block
  columns 3
```

```
block
columns 1
  block:Index1:1
  columns 1
    start1["start = 1"]
    end1["end = 3"]
    cap1["hash"]
  end
```

```
space
  block:Index2:1
columns 1
  start2["start = 1"]
  end2["end = 6"]
  cap2["hash"]
end
end
```

```
space
```

```
  block:Content
  columns 1
    c0["0x41"]
    c1["0x65"]
    c2["0x72"]
    c3["0x79"]
    c4["0x6E"]
    c5["0x4F"]
    c6["0x53"]
  end
```

```
  start1 --> c1
  end1 --> c3
```

```
start2 --> c1
end2 --> c6
```

# Layout

The Layout record contains metadata of a file or directory that should be written to the mass memory. When combined with a Content record (and possibly an Index record), it is possible to write the file or directory on disk in a reproducible way.

It is composed of a 32 bytes long header-like section and two fields of variable length.

Field	Type	Size (bytes)	Description
uid	uint	4	User ID of the owner of the file.
gid	uint	4	Group ID of the owner of the file.
mode	uint	4	File's mode and permission bits.
tag	blob	4	Unused.
source_length	uint	2	Length of the source blob.
target_length	uint	2	Length of the target string.
file_type	uint	1	Type of file to be written.
padding	blob	11	Unused.
source	blob	Specified by source_length	"source" of the file. Its meaning varies depending on file_type.
target	str	Specified by target_length	Path where the file should be written to.

## file\_type

file\_type is an enum that specifies the type of file that should be written on disk.

Value	Name	Description	Meaning of source
1	Regular	Regular file of data.	<a href="#">XXH3_128bits</a> hash of the file. The source is a hash because our package manager, Moss, uses a <a href="#">CAS</a> to store files. This hash may be

<b>Value</b>	<b>Name</b>	<b>Description</b>	<b>Meaning of source</b>
			also used to match an Index record, in order to identify a region of a Content record to read. In fact, the Index record also contains a XXH3_128bits-based hash.
2	Symlink	Symbolic link to another file.	String. Original file path to link somewhere else.
3	Directory	Empty directory.	Unused.
4	CharacterDevice	<u>Character device</u> .	Unused.
5	BlockDevice	<u>Block device</u> .	Unused.
6	FIFO	<u>POSIX FIFO</u> .	Unused.
7	Socket	<u>POSIX communication socket</u> .	Unused.

# Meta

The Meta record carries information about the *stone* itself or other [binary\\_packages](#), hence the name “meta”. It is composed of a 8 bytes long header-like section, plus a value of variable length. The information it carries is strongly typed.

Some of its fields can only be found in Repository *stones*, since they are information about packages to download.

Field	Type	Size (bytes)	Description
length	uint	4	Length, in bytes, of the value that follows the header-like section.
tag	uint	2	What information this record carries about the <i>stone</i> .
kind	uint	1	The base type of the information, for proper encoding and decoding.
padding	blob	1	Unused.
value	blob	length	Value of the information. The encoding depends on the value of <i>kind</i> .

## tag

tag is an enum.

Value	Name	Description
1	Name	Name of the <i>stone</i> , which is typically the name of a package to install or remove.
2	Architecture	Target hardware architecture of the package.
3	Version	Version of the package.
4	Summary	Succinct description of a package.

<b>Value</b>	<b>Name</b>	<b>Description</b>
5	Description	Full length description of the package.
6	Homepage	Web homepage of the project being packaged.
7	SourceID	ID of the source package, used for grouping.
8	Depends	One dependency of the package.
9	Provides	One capability, or the name, of the package. See further paragraphs for details.
10	Conflicts	One capability, or name of another package, that conflicts with this package.
11	Release	Release number of the package. It differs from Version in that Version identifies a state of the upstream project. Release is an increasing number that relates to the package.
12	License	One license under which the upstream project is released, in <a href="#">SPDX format</a> .
13	BuildRelease	Uniquely identifies a rebuild of a <i>stone</i> . It is an increasing number.
14	PackageURI	URL relative to the base repository URL, where to download a package from. It is specific to a Repository <i>stone</i> .
15	PackageHash	SHA-256 hash of a whole package <i>stone</i> . It is specific to a Repository <i>stone</i> .
16	PackageSize	Size, in bytes, of a package when installed on disk (assumed when the CAS is empty, thus without de-duplication). It is specific to a Repository <i>stone</i> .
17	BuildDepends	Build-time dependency of a package.
18	SourceURI	Upstream URL for the source archive of a package.
19	SourcePath	Relative path for the source within the upstream URL.

Value	Name	Description
20	SourceRef	Git commit or ref of the upstream source.

## kind [↗](#)

kind is an enum.

Value	Name	Description
1	Int8	int, 1 byte long.
2	UInt8	uint, 1 byte long.
3	Int16	int, 2 bytes long.
4	UInt16	uint, 2 bytes long.
5	Int32	int, 4 bytes long.
6	UInt32	uint, 4 bytes long.
7	Int64	int, 8 bytes long.
8	UInt64	uint, 8 bytes long.
9	String	String.
10	Dependency	<a href="#">PackageReference</a> structure.
11	Provider	<a href="#">PackageReference</a> structure.

## PackageReference [↗](#)

As explained before, the encoding format of value is explicated by the kind enum. Most of the kinds are a characterization of a [base format](#), but there is one outstanding kind: PackageReference. Below is the format of PackageReference.

Field	Type	Size (bytes)	Description
kind	uint	1	Kind of package one depends on, or provides.

Field	Type	Size (bytes)	Description
name	str	length - 1	Value of the reference.

## kind [↗](#)

PackageReference's kind field is an enum.

Value	Name	Description
0	PackageName	A package depends on, or provides, a package name.
1	SharedLibrary	A package depends on, or provides, a shared library (*.so file).
2	PkgConfig	A package depends on, or provides, a <a href="#">pkg-config</a> target.
3	Interpreter	A package depends on, or provides, a program interpreter (i.e. an ELF binary with PT_INTERP set).
4	CMake	A package depends on, or provides, a <a href="#">CMake</a> target.
5	Python	A package depends on, or provides, a Python module.
6	BinaryDep	A package depends on, or provides, a binary file in /usr/bin.
7	SystemBinary	A package depends on, or provides, a binary file in /usr/sbin.
8	PkgConfig32	Like PkgConfig, but emul32-compatible.