

OPENFP QUALITY CONTROL HANDBOOK

COVERING HASKELL, NIX, VORTEX, HORIZON, CONTINUOUS INTEGRATION
AND AGENTS

CONTENTS

General OpenFP Standards	2
Laws	2
Writing a quality-control doc	3
Auditing quality control	3
Haskell	5
Toolchain and dependencies	5
Language and lint	5
Types and effects	5
Generating other code	6
Testing	6
Prose and commits	6
Packages	7
Code documentation	7
Repository documentation	7
Nix	9
Flakes	9
Modules	9
NixOS modules	10
Generated Nix	10
Housekeeping	10
Dependencies	10
Documentation	11
Writing documentation	11
Vortex	13
Vortex terminology	13
Results	13
Horizon	14
Making a set	14
Releasing a set	14
Continuous Integration	15
Building	15
Merging	15
Agents	16
Before working	16
Identity	16
Working with the forge	16

GENERAL OPENFP STANDARDS

LAWS

These laws apply to every domain.

QC-1 REVIEW Vortex is the universal interchange

When a machine reads structured data, configuration or an API, write it as a Vortex program, not JSON, YAML, TOML, Dhall or a hand-made RPC or REST interface. A Vortex program is typed, cannot act on the world itself, and can be turned into whatever file is needed. So the program is the source of truth, and the JSON, YAML, `.cabal`, `flake.nix` or HTTP payload is generated from it rather than maintained by hand. This is why Dhall is retired, why `flake.nix` and the package sets are generated, and why the forge API is a typed Vortex socket instead of REST. Use another format only where something outside OpenFP requires it, and generate it.

QC-2 REVIEW Script in vortex, never Python

Anything you would write a script for, such as fetching, reading or transforming data, is a Vortex program run by a Haskell host. Never use Python, and never a bash loop. If no DSL covers the job, write one: a typed set of effects with a pure interpreter, and a host that runs it. Bash and Python have no types, no error values and no composition, so they fail silently. A lost reply looks like an empty result, a pipe cuts output short, and 151 calls can return one answer without an error. A Vortex program that is wrong does not compile, and every answer reaches the reader whole, as a typed value. If one Vortex call with a `do`-block can solve the problem, write it as one.

QC-3 REVIEW Changes land through merge requests

Change a shared repository by pushing a branch and opening a merge request on NixLab. Never edit the forge's repositories or the server directly. This applies to these quality-control documents too, so a change to a standard is reviewed like any other change.

QC-4 REVIEW The trunk is master

Name a repository's trunk branch `master` and make it the default branch on the forge. Name each long-lived release line `lts/<line>`, such as `lts/ghc-10.0.x`, one branch per line. A repository with release lines and no trunk, such as a package set, uses its current `lts` line as the default. With the same names everywhere, flake inputs, merge requests and scripts never have to guess where a line lives.

QC-5 REVIEW Say only the non-obvious

Where a language's rules allow comments, use them to say why the code exists, such as the failure it prevents or the upstream issue it works around. Never use them to repeat what the code does. Follow each language's rules for documenting declarations. If the code is clear, it needs no comment. Do not write TODO comments. Work not done is an issue on the forge, or not worth a line. Do not leave commented-out code, banners, section dividers or author and date lines, because git records those. A generated file carries a `do-not-edit` header naming its generator, and a source-of-truth line only where the language's rules allow one. Any other comment would be lost the next time the file is generated.

QC-6 REVIEW Use MIT wherever rights allow

Every non-empty repository has a `LICENSE` file at its root. License code, documentation and assets under MIT wherever the rights holders can grant it. Package metadata states the same licence, and distributed packages include the licence files. Keep third-party notices, and the terms of anything that cannot be relicensed. Give each exception its own licence notice and explain it in the README, or in a notice the README links to, so a reviewer can see where it applies.

QC-7 REVIEW Derive operational strings from their owner (a.k.a. No magic strings)

Hostnames, URLs, repository paths, branch names, account names and ports that can change come from the configuration or domain value that owns them, and are validated there. A reusable component must keep working when those values change, without a code edit. Moving a literal into a named constant does not satisfy this rule. Literals are fine where an external protocol or schema

fixes the spelling, and in test fixtures. In review, trace each operational literal to its owner, and reject one that belongs to a particular deployment.

WRITING A QUALITY-CONTROL DOC

The quality-control group is where OpenFP's conventions are written down, versioned and shared. Its documents follow these conventions rather than a fixed template.

QC-DOC-1 REVIEW One repo per domain, ownership delegated

Each domain, such as Haskell or Nix, has its own repository, so whoever owns that domain's quality owns and reviews its standards. Each principle lives in exactly one repository. A rule that crosses domains is written once, as a law above, and referred to rather than copied. The handbook collects every document into one place without taking control away from the domain owners.

QC-DOC-2 REVIEW A principle is a rule, not a description

Say what to do, as an instruction. Give the reason briefly and say how the rule is enforced. Do not describe how things happen today, because a principle sets the standard. Each principle has a stable id, such as HS-4, so a review can cite it.

QC-DOC-3 REVIEW General, not application-specific

A domain document holds rules for the whole domain. Rules about one application's design, such as its internal module layout, live in that application's documentation and are linked from here. To decide, ask whether the rule would apply to a new project in the domain. If not, it belongs to the application.

QC-DOC-4 REVIEW Classify by the language written in, not produced

A principle belongs to the language you write, not the one you produce. A rule about generating Nix, `.cabal` or JSON from Haskell is a Haskell rule, because it governs the Haskell code, even though the output is Nix. The Nix document governs how Nix itself is written. It does not tell a Haskell author which library to generate Nix with.

QC-DOC-5 REVIEW Mark enforced versus by-review

Tag each principle **ENFORCED** when a build, lint or check fails on a violation, and name the check. Tag it REVIEW when a person checks it. Never let a rule that is only hoped for look as if a machine guarantees it.

QC-DOC-6 REVIEW State what always holds, not circumstance

A document states rules that hold whatever the code looks like this week. Leave out circumstance: how far a migration has got, a goal not yet reached, and any file path, version or branch the code will move away from. Such facts go stale, and nothing updates the handbook when they do. They belong in an issue next to the code.

QC-DOC-7 REVIEW Evidence, not assertion

A principle that is not obvious names where it is put into practice, such as a module, a check or an example repository. A reader can then verify it, and it does not turn into folklore.

AUDITING QUALITY CONTROL

An audit records what was checked at a given revision. It says nothing about later commits.

QC-AUD-1 REVIEW Inventory the full scope

Use NixLab's Vortex API to list OpenFP's repositories, their default branches and every maintained release branch, and to read their files and forge records. Record the commit you reviewed on each branch. Never use a local clone as evidence, because it may be out of date or hold unpushed changes. Use a clone only to prepare fixes. Include the quality-control repositories themselves, nested packages, test fixtures and generated documents. Check what a release claimed against the source and tag of that release, not only against today's code. An audit that skips a branch or document is not finished.

QC-AUD-2 REVIEW Give every rule an evidenced result

For each revision you review, check every principle that applies, from every QC document. Record pass, violation, not applicable or unknown against each principle's id. Name the file, command, check result or forge record behind each result. Say why a rule does not apply. An unknown stays open until evidence settles it. A passing build does not pass a rule marked for review.

QC-AUD-3 REVIEW **Check the source of each claim**

Compare the documentation with the code, options, tests and release tags. Check licence files, package metadata and third-party notices against the MIT law and any documented exception. Find generated files and review the input they were generated from. Regenerate them with the project's pinned tools and run the checks that enforced principles name. Check that those checks really fail on a violation. For a review principle, read the relevant source and write down why you judged it as you did.

QC-AUD-4 REVIEW **Separate violations from proposed rules**

Report each violation with its principle id, the revisions affected, the evidence and a concrete fix. Land fixes through merge requests, in the order the dependencies are released. Report recurring problems that no current principle covers separately, with examples and the cost and benefit of making them a rule. A proposed rule is not binding until it has been reviewed and released.

HASKELL

These rules govern how OpenFP writes Haskell.

TOOLCHAIN AND DEPENDENCIES

HS-1 REVIEW **GHC and every dependency come from Horizon**

GHC and every Haskell dependency come from a Horizon package set and the repository's `horizon.vx` pins. Never use `nixpkgs haskellPackages`, `nixpkgs#ghc` or `ghcWithPackages`, not even for a quick type check. A `nixpkgs` GHC brings different versions and breaks the reproducible pins.

A Haskell package from another OpenFP repository is also a `horizon.vx` pin, built in this repository's set. Never take it as a flake input and use that input's overlay, `packages` or `legacyPackages`, because that brings a second package set, with its own lock, into the build. Flake inputs are for the package-set root and for things that are not Haskell, such as design-kit assets or the NixOS modules a system uses.

HS-2 REVIEW **Pick the Horizon set that fits the job**

The sets stack up in order: `horizon-build-packages`, `horizon-core`, `horizon-advance`, `horizon-platform`. Each is the one below plus more packages, and each tracks its own GHC line. Each set's `horizon.vx` and README say which GHC line, since a version written here would go stale. Depend on the highest set your GHC line needs, and you get everything below it too. Do not reach for `horizon-platform` by default. A set's README "always expected here" list is its `requiredPackages` guarantee: packages that are checked to resolve in its `legacyPackages`.

HS-3 REVIEW **Add a Hackage dependency by pinning it**

Pin a new dependency in `horizon.vx`, or check that a Horizon set already has it. Do not take it from `nixpkgs`. For example, `quickcheck-dynamic` is not in `horizon-platform`, so a repository that needs it pins it from Hackage.

LANGUAGE AND LINT

HS-4 REVIEW **GHC2024, set once**

Set `default-language: GHC2024` in the `.cabal`, and put the few extra extensions you need in `default-extensions:.` Do not use `LANGUAGE` pragmas in individual files, except `TemplateHaskell`, which is enabled with a pragma in each module that uses it, so it is on only where it is needed.

HS-5 ENFORCED **Warnings and lints fail the build**

Repositories are `horizon-flake-parts` flakes with `werrorwolf`, which turns on `-Werror`, `-Wmissing-import-lists` and `-Wmissing-local-signatures`. Import lists are explicit, and top-level and local bindings have type signatures. These warnings fire in the `check build` (`nix build .#<pkg>-check`), not the plain package build, so run the check. `Treefmt` runs `hlint`, and its hints are applied. Ignore the `Paths_` and `PackageInfo_` modules that Cabal generates.

TYPES AND EFFECTS

HS-6 REVIEW **Give stringly data a validating newtype**

Give a value its own newtype when a wrong string could break it. Validate it once, in its `fromString`, which calls `error` on bad input, so a mistake is caught as soon as the program is evaluated. Name the unwrapping field `from<Type>`. With `OverloadedStrings`, call sites still read as plain literals. Keep one schema per concept, never two vocabularies kept in sync by hand.

HS-7 REVIEW **Use typed paths for filesystem operations**

Hold file locations as `Path` values whose types say whether they are absolute or relative, and file or directory. Use `Path.IO` for file system operations. Do not pass plain `FilePath` strings to `System.IO` or `System.Directory` when `Path.IO` can do the job. The types catch a directory used as a file, or a relative path where an absolute one is needed. Convert to `FilePath` only where an API demands it. `System.IO` is still right for handles and the standard streams.

HS-8 REVIEW **Mk-prefix single constructors**

A type with one constructor, whether a record, a product or a newtype, names it `Mk<Type>`, as in `MkPackageSet` and `MkPackageName`. Never give a constructor the same name as its type. For a validating newtype, `MkPackageName` is the unchecked constructor, and callers go through `fromString`. Name the constructors of a sum type for what they mean.

HS-9 REVIEW **DSLs are pure effect rows**

A DSL is a set of polysemy effects declared with `makeSem` in an `<App>.<Theme>.Effects` module. Its types contain no IO, so a program can only describe what to do, and a host interpreter turns it into a result. A program is elaborated with exactly one DSL module in scope, plus the Prelude. Every Vortex host answers `:commands` with the list of operations it offers. Meta-commands start with `:`.

HS-10 REVIEW **No unsafePerformIO**

Never use `unsafePerformIO`.

HS-11 REVIEW **Errors are typed or a one-word throw alias**

Use either a typed error type for the app, such as `MastError` or `LatheError`, carried in polysemy's `Error` with an `errorText` function to render it, or a one-word local alias for `throw`, such as `oops` or `die`. Messages shown to users read `context: message`.

GENERATING OTHER CODE

Some OpenFP tools write other languages from Haskell. One generator writes Nix, another writes `.cabal` files. The rules here are Haskell rules, because they govern the Haskell we write, even though the output is another language.

HS-12 REVIEW **Emit structured formats as a typed AST**

When Haskell writes a structured file, build the format's typed syntax tree and pretty-print it. Never glue strings together. A typed tree is escaped correctly and cannot be malformed. Use `hnix` (`Nix.Expr` and `prettyNix`) for Nix, `Cabal-syntax` for `.cabal` and `aeson` for JSON.

TESTING

HS-13 REVIEW **Property and law tests use Hedgehog**

Write property tests with Hedgehog, using named `Gen` values such as `genScene :: Gen Scene` passed to each property. Do not write `QuickCheck Arbitrary` instances. Named generators are ordinary values, they combine, and Hedgehog shrinks failures automatically. A theory's laws are tested: the equations that make the code correct are the properties checked. Every DSL has a test suite for its laws and for any pure interpreter, whether or not it also has a model test.

HS-14 REVIEW **Model tests use quickcheck-dynamic**

Write model tests, which run a sequence of commands against a simple model, with `quickcheck-dynamic` (`StateModel` and `RunModel`, driven by `runPropertyReaderT`). Do not use `quickcheck-state-machine` or Hedgehog's state-machine support. Every shell models the command behaviour a caller can see, and runs that model against a real host. A model tested only against the DSL does not cover the shell. Keep the model smaller than the host by modelling its contract, not its internals. A stateful DSL also gets a model when a separate pure model is simpler than its interpreter. Otherwise its laws are its tests. This is a narrow exception to the Hedgehog rule, for the model test driver only, not for ordinary generators.

HS-15 REVIEW **A shell release gates on real-host model and protocol tests**

Release a new shell only once its `StateModel` passes against a real host and its protocol contract tests pass. Use a NixOS VM test for a service. Release a DSL once its laws pass, and its own model test where the rule above calls for one.

PROSE AND COMMITS

HS-16 REVIEW **Version packages by PVP**

Version every package release by the [Haskell Package Versioning Policy](#). Bump the major version, the first two numbers, when an exported API breaks, including a changed data type or class. Bump the third number when the API only grows. For a release with no visible API change, keep the first three

numbers and change a fourth. The same rules apply to `0.x` packages, and to dependency types that appear in public signatures. Check the exposed modules and dependency changes before tagging, because the version tells users which bounds will compile.

HS-17 REVIEW British spelling

Use British spelling in identifiers, Haddock and documentation: colour, behaviour, normalise, licence, analyse. Use American spelling only where a foreign API requires it, as in `SDL_COLOR`.

HS-18 REVIEW Module and package conventions

Give each module one concern, such as Types, Schema, Effects or Modifiers. `app/` holds `Main`, `src/` the library and `test/` the tests. Packages are named by role, as below. Every package has at least one test suite, and a package without one is not released.

PACKAGES

HS-19 REVIEW Name a package by its role

An app is split into packages by role, and each name ends with its role:

- `<app>-dsl` is the language. It holds the effects, the values they work on, the scope module whose exports are the operations a caller can use, and the pure model, laws and pure interpreter. There is no separate `-types` package, because nothing uses the values without the language.
- `<app>-shell` is the host that runs the language for real. It holds the interpreters over real effects and the Vortex executable, on any transport. There is no `-server`, since a server is just a shell on a transport. There is no `-protocol` either, since every app uses Vortex's protocol. An app defines a language, not messages.
- `<app>-web`, `<app>-desktop` and `<app>-tui` present the language in an interactive medium: the browser (through the JavaScript backend), a native window or a terminal. Name the medium, not the toolkit. The medium decides the toolchain, and so the package boundary, while the toolkit can change.
- `<app>-render` turns the language into an output file, without interaction.
- `<app>` is the deployment that puts these together, or the whole app when it is too small to split.

A `-web` package that uses its `-dsl` needs polysemy built for the JavaScript backend.

HS-20 REVIEW Each package lives in its own subfolder

Every Haskell package lives in a subfolder named after it, such as `design-kit-miso/` or `letterbox/`, even when the repository has only one. The root holds the flake files: `flake.nix`, `flake.lock`, `nix/`, `horizon.vx` and what it generates, a `cabal.project` listing each package folder, and the repository's `README.md`, `LICENSE` and `ChangeLog.md`. The subfolder holds the `.cabal`, `src/`, `app/`, `test/`, `bench/` and every file the `.cabal` names, so `cabal sdist` and `cabal check` pass there. `source-repository` stanzas give `subdir`.

The subfolder is what gets built and published, and nothing above it counts as package source, for Nix or for the package index. So editing the flake, `nix/` or a `README` does not change the package's hash or what the index publishes. Each package has one obvious home, and a second package can sit beside it without moving anything.

CODE DOCUMENTATION

HS-21 REVIEW Only Haddock comments, each with @since

Checked-in Haskell has no ordinary comments. Every comment is Haddock, attached to a module or a declaration, and has exactly one `@since` line naming the package version that introduced it. Write it as its own paragraph, such as `@since 0.2.1`, with nothing after the version. For an existing declaration, use the earliest released version the source and tags support. Do not guess. A generated Haskell file has one Haddock module header holding its do-not-edit notice and its `@since` line, and no other comments.

REPOSITORY DOCUMENTATION

HS-22 REVIEW Make the root README a package and command guide

Start the root README with `# name`, a short tagline and a paragraph saying what the repository provides and why. List every Haskell package under the exact line `This repository provides the following packages:`, even when there is only one. Link each package name to its subfolder, whose README describes it in detail.

Show the commands that work in this repository to build the packages, enter the development shell and run the tests. Show how to run an executable if there is one, and the VM tests if the flake has them. Name the flake outputs, so a reader can find the same commands in the flake. Do not show a run command or VM test the repository does not provide. Update the README whenever the flake's outputs change.

NIX

These rules govern how OpenFP writes Nix.

FLAKES

NIX-1 REVIEW **mkFlake, modules under nix/**

outputs is the one line `nixica.lib.mkFlake { inherit inputs; modules = ./nix; }`. Every file under `nix/` is a `flake-parts` module, and inputs' `flakeModules` are imported automatically. Split an app's Nix into `nix/configuration.nix` for set-wide metadata and formatting, and `nix/<name>.nix` for the `perSystem` build. Each module takes the `flake` as a single `local:` or `_local:` argument.

NIX-2 REVIEW **Follow only at an owned layer**

A leaf `flake` pins its own `nixpkgs` to a named tag. A package set that extends a lower set may follow that set's `nixpkgs` and other shared toolchain inputs, because the lower set owns them. Follow only the set you actually extend, so a layer does not quietly pick up an unrelated toolchain. A nested test `flake` may follow inputs of the `flake` it tests.

Do not set `<tagged-input>.inputs.*.follows`. A tagged release includes the input graph it was tested with, and a different graph needs a new release of that dependency. Review checks where `follows` appears, its targets, the declared inputs and the resolved revisions. Identical sources can still show up as numbered lock nodes.

NIX-3 REVIEW **Pin load-bearing inputs to named tags**

Pin every input the build depends on to a named tag, such as `github:NixOS/nixpkgs?ref=refs/tags/25.11`. Never pin to a moving branch, and do not add a `rev` unless the repository's documentation gives a good reason. If an OpenFP dependency has a fix waiting, test it and cut a tag. If an external dependency has no tag for the revision you tested, pin the revision and explain the missing tag in the repository's documentation until a tagged release or an owned fork is reviewed. Do not invent a tag, and do not quietly change the dependency's code to fit the pin. Inputs used only as source are content-hashed with `flake = false`.

NIX-4 REVIEW **Alphabetise `flake` inputs**

List the inputs in `flake.nix` in alphabetical order by name. Keep each input's URL and its other settings together.

NIX-5 REVIEW **Omit `.git` from `flake` input URLs**

Write Git `flake` URLs without `.git` before the tag reference, as in `git+https://nixlab.openfp.net/nix/nixica?ref=refs/tags/0.13.2`. Use the same form in every `flake` so equal inputs are easy to compare. Update the lock after changing a URL.

NIX-6 REVIEW **Publish the standard output trio**

A Haskell package `flake` publishes its package set as `legacyPackages` and each deliverable as `packages.<name>`. Add `packages.default` when the `flake` has one main build. Provide `devShells.default` with the repository's development tools, built with `legacyPackages.shellFor { ... }` in a Haskell package `flake`. Declare the caches in `nixConfig` with `extra-substituters` and `extra-trusted-public-keys`, keys included.

NIX-7 REVIEW **Declare supported systems, don't hardcode**

Enable `horizon.systems.supported`, which defaults to the systems the `flake` exposes, instead of writing out a `systems = [ ... ]` list.

MODULES

NIX-8 REVIEW **Import unconditionally and enable conditionally**

Always import a reusable module. It does nothing until enabled: all of its effect sits in `config` behind an enable flag, as in `config = lib.mkIf cfg.enable { ... }`. Declare typed options, with `mkEnableOption` for the flag and `mkOption` with an explicit type for everything else. Give each sub-feature its own flag. A reusable `flake-parts` module declares its `perSystem` options with `mkPerSystemOption` and `gates`

`config.perSystem` the same way. A repository's own configuration module may set its configuration directly, since importing it is how it is enabled.

NIXOS MODULES

NIX-9 REVIEW Service modules follow the RFC pattern

A NixOS service module is `nixosModules.default`, gated on `enable`. Secrets are absolute runtime paths, validated, and kept out of the Nix store. Deliver them with `systemd's LoadCredential` and read them from `$CREDENTIALS_DIRECTORY`. Never turn a runtime secret path into a store path. Options that take a secret in plain text exist only behind an assertion that the build is an insecure test. `serviceConfig` is hardened by default. Keep the module for a real host separate from the one for a test VM, and match the NixOS release the target image runs.

GENERATED NIX

NIX-10 REVIEW Generated Nix follows its consumer's interface

A generated file that `flake-parts` imports is a module taking the module argument, as in `local: { ... }`. Do not `import` a generated expression from inside another module. When something outside requires a different shape, generate that shape and document why beside the generator. Every generated Nix file has a `do-not-edit` header naming its generator, is excluded from the formatter, and changes only through its source.

HOUSEKEEPING

NIX-11 REVIEW Enter with `nix develop` and omit `direnv`

Enter the development environment with `nix develop`, which uses `devShells.default`. Repositories do not ship an `.envrc` or use `direnv`.

NIX-12 REVIEW `gitignore` build artifacts only

`.gitignore` lists build artifacts only, such as `result`, `result-*`, `dist-newstyle/`, `*.hi`, `*.o`, `.ghc.environment.*` and runtime state/. It never hides committed source.

NIX-13 REVIEW Format with `nixfmt`

Format Nix with `nixfmt` through `horizon.formatting`, and lint with `deadnix` and `statix`. `Treefmt` checks the formatting.

NIX-14 REVIEW Nix code has no comments

Hand-written Nix has no comments. The only comment allowed in Nix is a generated file's `do-not-edit` header naming its generator. Put explanations in the repository's documentation, in names and in tests. Review changed `.nix` files, `flake.nix` included, for comments outside strings, and check that a generated file has only its header.

DEPENDENCIES

NIX-15 REVIEW The dependency graph is acyclic, tests included

A lower layer never pins a higher one, whether in its own `flake.lock` or in a test or sub-flake lock. The higher layer already pins a release of the lower one, so a pin back makes a cycle. The lower layer's lock then holds an old copy of itself, with everything that release pinned. And a change to the lower layer cannot be tested until the higher layer updates to it, which needs the lower layer released first. Test a lower layer against small fixtures defined inline. Test it together with a higher layer in that higher layer, or in a leaf repository nothing depends on.

DOCUMENTATION

These rules govern how OpenFP writes documentation.

WRITING DOCUMENTATION

DOC-1 REVIEW Give root and package READMEs distinct jobs

The root README starts with `# name` and a short tagline saying what the repository provides and why it exists. It says how the repository is laid out, built and tested, where that helps a user. It links to each package subfolder with a line on that package's role, even if there is only one.

Each package subfolder has its own README, which explains the library or application in detail: behaviour, API and examples. Open with what the package provides, then why it exists if that is not obvious, and introduce API names as you need them. Say what a reusable package provides before listing who uses it today.

Every claim describes the tree as it is. Plans go in issues or a roadmap.

DOC-2 REVIEW Every repository carries a ChangeLog

Every repository has a root `ChangeLog.md`, and every package subfolder has a `ChangeLog.md` beside its `.cabal`, even when there is only one package. A `.cabal` never sits at the repository root.

Root entries use the repository's release version and record changes to the repository. Package entries use the package's Cabal version and record changes to its code. A root release need not touch a package, and a package `ChangeLog` only lists releases of that package. Link from the root `ChangeLog` to each package `ChangeLog` instead of repeating its entries.

Give each release its version and date, newest first. Say what changed for the reader and whether they need to do anything. Add implementation detail when it explains the change, and leave the rest to git. A first release may be called that, followed by what it shipped. Check old entries against the source as it was at that release. Issue references are optional.

DOC-3 REVIEW Long documents are Typst

Anything longer than a README is written in Typst and rendered to PDF and HTML by the repository's flake, using the pinned `typst-doc` toolchain and the `design-kit` theme. The rendered files are build outputs and are never committed. Do not hand-write HTML, or use another long-form format, unless something outside OpenFP requires it.

DOC-4 REVIEW Cite issues that exist

When you cite an issue, name its project and check the number exists on the forge. Check links to code documentation too, including the root README's links to package READMEs.

DOC-5 REVIEW Prose is British

Use British spelling in prose and identifiers: colour, licence, normalise, analyse. Use American spelling only where a foreign API requires it, as in `SDL_COLOR`.

DOC-6 REVIEW Write for the reader, not a pattern

Write naturally. Name the subject and say what it does. Similar READMEs may share wording if it stays clear and accurate. Do not rewrite a sentence that already works just to make files match. Use full sentences where a fragment would hide who does what. Do not use semicolons in prose.

DOC-7 REVIEW Describe a language on its own terms

A DSL's README opens with what the language expresses and the domain it serves. A reader should understand it without knowing which application uses it today. Explain the interpreter or output where that helps someone use the language. When two languages compose, say what each expresses and how a program in one uses the other.

DOC-8 REVIEW Name actors, terms and limits

Say which component does a thing, and where a rule comes from, such as the DSL module whose exports a caller can use. When a name gives no clue, introduce it by role: say whether it is a type, function or module, and what it does. Prefer terms the reader knows, such as `do-block`, over vague substitutes. Check limits, lifetimes and exceptions against the code, and state limits in the units the

code uses. Explain what an intermediate value holds before listing what it produces. State exact behaviour rather than an absolute you cannot back up.

DOC-9 REVIEW **State the scope and outcome of an effect**

For anything that replaces, cleans up or deletes, say what it can touch and what decides that. Distinguish reporting an error from the command failing, and say which failures stop the command. Check every code path that applies.

DOC-10 REVIEW **Describe tests by what they check**

Name the cases a test suite covers. Do not call a finite list comprehensive. When you name a test or fixture, check the name matches what shipped in that release.

DOC-11 REVIEW **Record changes with both ends**

Write ChangeLog bullets as plain past-tense sentences that start with the action. When something moved or switched, give both the old and the new place or state. Add a second sentence for the effect when that helps. Give the reason if it is not obvious. For a migration, say who has to act and what they must do. When editing old entries, keep their versions and revisions accurate.

DOC-12 REVIEW **Keep moving facts at their source**

In a README, do not repeat a generator's version in prose when the command or the file that pins it already gives it.

VORTEX

These rules govern Vortex's own concepts and documentation, as distinct from regular Haskell. See vortex.openfp.net for Vortex itself.

VORTEX TERMINOLOGY

VX-1 REVIEW Name effects and interpreters

An operation a DSL declares is an effect. The DSL is the language a do-block is written in: its exported functions and effects. The host code that gives effects their behaviour is an interpreter. If the host picks the interpreter for a caller, say that the host picks it and why. Do not suggest the caller supplies it. `Vortex.Shell` pairs effects with an interpreter chosen per caller.

VX-2 REVIEW Call Haskell do syntax a do-block

Haskell statements submitted to a host are a do-block. This includes flat, multi-line Vortex source that the host wraps in `do`. Say expression or definition when you mean one of those. Do not call a do-block a script, snippet or block in READMEs, ChangeLogs or examples, old entries included. Leave code names such as `blockDo` alone, and other meanings of "block". `Vortex.Elab.Block` and `Vortex.Shell.Block` read flat source.

VX-3 REVIEW Distinguish do-blocks from direct calls

A host accepts three kinds of request. A do-block is Haskell that GHC elaborates and type-checks. A `call` names one operation and gives its arguments, and skips GHC. A meta-command starts with a colon, such as `:commands`. Document each path separately, so a reader knows what source the host accepts and when it is type-checked. `Vortex.Shell` and `vortex-call` show the two execution paths.

VX-4 REVIEW State the caller's actual operation surface

Name the scope, or the exported DSL module, that gives a do-block its operations. Do not suggest that everything in a package, or everything the host can do, is callable. If the host answers `:commands`, compare that list with the scope and with any direct-call table before saying what a caller can reach. `Vortex.Elab.describeScope` and `Vortex.Shell` show what is actually exposed.

RESULTS

VX-5 REVIEW Derived Show is a result's wire format

A host sends a do-block's result as `show` text, so `Show` is the wire format for every type a DSL returns. Derive `Show`, `Read` and `Generic` for those types, and never write `Show` or `Read` by hand. A client that has the DSL's types reads a result back with `read`. A client without them parses the derived syntax by its shape. Renaming a constructor or field of a result type changes what every client sees, so treat it as a breaking change and bump the major version. Keep presentation out of the text: a console decides how a value looks, guided by its type. Give references and quantities their own newtypes, such as a `track id` or `seconds`, so a console can act on them and show them well. Reviewers check this today. The target is a `vortex-check` law that every result reads back and parses. `Vortex.Trace` sends replies as `show` text, NixLab's web front end reads results back with `read`, and Hedgehog parses derived `Show` to print its diffs.

HORIZON

These rules govern how OpenFP makes and releases Horizon package sets. See horizon-haskell.net for the package sets and their documentation.

MAKING A SET

HZ-1 REVIEW **horizon.vx is the set's only source**

Write a set's compiler, index state, packages, roots and policy in its `horizon.vx`. `horizon-config-shell` generates the Nix under `nix/` and `horizon.lock` from it. Never edit a generated file. Change `horizon.vx` and generate again. Keep source patches as plain diff files beside it.

HZ-2 REVIEW **Build on exactly one lower set**

A set extends exactly one lower set with `on`, pinned to that set's snapshot tag, and adds its own packages on top. Everything the lower set provides is available in the new set without being listed again.

HZ-3 REVIEW **Declare what the set promises**

Name every package the set promises to provide with `root`. The set's README lists the same packages under "always expected here", so a user can see what will resolve without reading `horizon.vx`.

HZ-4 REVIEW **Fix a dependency in the set that supplies it**

When a package does not build, fix it in the set that provides it, with a pin or a patch in that set's `horizon.vx`. Never patch it in the project that uses it. A fix in one project helps only that project and hides the set's defect from everyone else.

RELEASING A SET

HZ-5 REVIEW **Release as snapshot tags**

Release a set as an annotated tag named `snapshots/<branch>/<N>`, such as `snapshots/lts/ghc-10.0.x/3`. `N` starts at 1 on each branch, counts up, and is never reused. Tag only a commit whose pipeline passed. A snapshot never moves.

HZ-6 REVIEW **Pin a snapshot, never a branch**

Anything that uses a set, whether a project or another set, pins one of its snapshot tags. Never pin a set's branch, because a branch moves and takes every lock that follows it along with it.

CONTINUOUS INTEGRATION

These rules govern how OpenFP builds, checks and merges changes.

BUILDING

CI-1 REVIEW The flake is the CI configuration

A repository has no CI configuration file. The forge builds every output of the commit's flake and does nothing else. Write a policy as a flake check, so CI builds it like any other output. Never add a job, an option or an exception list to the forge.

CI-2 REVIEW Every flake builds and is fully locked

Every `flake.nix` in a repository, nested test flakes included, builds all of its outputs and is fully locked, so `nix flake lock` changes nothing. No flake is exempt.

CI-3 REVIEW Push every passing build to the cache

Every output that builds is pushed to the shared binary cache, from any branch. A build that is not in the cache is built again after every garbage collection and on every other machine.

MERGING

CI-4 ENFORCED Merge only on a passing pipeline

A merge request merges only when the pipeline for the latest commit on its source branch has passed, which means every flake output built. A new push blocks the merge again until its own pipeline passes. Nobody can bypass this. The forge refuses the merge.

AGENTS

These rules govern how software agents work in OpenFP.

BEFORE WORKING

AG-1 REVIEW Read the handbook first

Read this handbook before changing code or documentation in OpenFP, and follow it. Read the current release, because the standards change by merge request like anything else.

IDENTITY

AG-2 REVIEW An agent works as itself

An agent authors its commits, merge requests, comments and tags under its own forge account, such as Smartypants. It never works under a person's account, so the history shows which work an agent did.

AG-3 REVIEW Carry no tool attribution

Commits and merge requests carry no attribution to the tool behind an agent, such as Co-Authored-By trailers or "Generated with" lines. The author's identity already says who did the work.

WORKING WITH THE FORGE

AG-4 REVIEW Read state from the forge

Before saying what is done, merged, released or pending, read it from the forge's issues, merge requests, pipelines and tags, and from the deployed systems. Notes and local clones go stale and are never the record.

AG-5 REVIEW Act through the forge's API

Merge, tag, open issues and read pipelines through the forge's own operations, as the agent's account. Never use an administrator's shell or the forge's database to change or read its state.

AG-6 REVIEW Poll the forge with one short call

The forge does not notify an agent, so an agent polls it. Every reply lands in the agent's context, so poll with one call that returns only the state being waited for, such as `do { m <- getMergeRequest "web/openfp.net" 12; pure (mrState <$> m) }`. Check every merge request or pipeline being waited on in that same call. Wait about as long as the build takes between calls, and stop once each one reaches a final state. Never fetch whole records or logs to read a status.

AG-7 REVIEW Check a fact before stating it

Check a claim about a tool, flag, version or behaviour against the tool or its source before stating it. If a claim has not been checked, say so.