Setting up

juliaup, environment variables, editors, JETLS, self-hosted extras

Linux, August 2026. Current stable Julia is **1.12.6**, LTS is **1.10.11**, and **1.13** is in release candidates.

Toolchain

```
curl -fsSL https://install.julialang.org | sh
```

That installs `juliaup` and the current stable Julia into `~/.juliaup`, and puts a `julia` shim on your PATH. Nothing else is required — Julia binaries are self-contained tarballs with their own LLVM, BLAS and libgit2. No compiler toolchain, no system BLAS, no Python.

Two environmental notes that aren't obvious:

- **Headless plotting.** The default `GR` backend links against X libraries and will try to open a window. If the machine has no display, `export GKSwstype=100` makes it write to file instead. Without it you get an obscure failure deep inside a plotting call, and you will not guess why.
- **Don't install Julia from your distro's repository.** Distro packages lag badly and confuse `juliaup`'s shim. If one is already installed, remove it.

Version management: juliaup, not asdf

```
juliaup add lts              # a channel, not a pin — tracks LTS as it moves
juliaup add 1.13-rc
juliaup add nightly
juliaup status
juliaup update               # updates every channel you track
juliaup default release

julia +lts                   # run a specific channel ad hoc, no switching
julia +1.10 --project -e 'using Pkg; Pkg.status()'
```

`julia +channel` is the feature that makes this worth using over a generic version manager. Testing a package against LTS and stable in the same terminal is one token of difference, not an `asdf set` dance and back. Channels also mean patch releases arrive with `juliaup update` and no version strings to edit.

The reproducibility argument for `.tool-versions` is weaker here than elsewhere, because Pkg already covers it a layer down: `Project.toml` carries a `[compat]` bound on `julia` itself, and `Manifest.toml` pins the entire resolved dependency graph.

If you want one tool across all your runtimes anyway, there's a community plugin:

```
asdf plugin add julia https://github.com/rkyleg/asdf-julia.git
asdf install julia 1.12.6
asdf set julia 1.12.6            # post-Go-rewrite command; `asdf local` is gone

mise use julia@1.12.6            # mise reads asdf's plugin registry
```

What that costs you: channels, `julia +ver`, and same-day patch releases (you wait for the plugin maintainer). It also adds a shim layer that occasionally confuses tools which shell out to `julia` — the `jetls` and `runic` executables installed by `Pkg.Apps` are the usual casualties.

The hybrid worth adopting: juliaup for the toolchain, direnv (or `mise`'s `[env]` block) for per-project environment.

```
# .envrc in a project root
export JULIA_PROJECT=@.
export JULIA_NUM_THREADS=auto
```

Then bare `julia` in that directory is already in the right project with the right thread count, and Zed picks the `.envrc` up automatically.

Environment variables

Variable	Why it matters
`JULIA_NUM_THREADS=auto`	Default is **1**. Everything "multithreaded" silently runs serially otherwise.
`JULIA_PROJECT=@.`	Activates the nearest `Project.toml`, walking upward. Without it you're installing into a global environment.
`PATH` += `~/.julia/bin`	**Required.** `Pkg.Apps` installs `jetls`, `runic`, `jlfmt` here.
`JULIA_DEPOT_PATH`	Relocate the depot. Colon-separated; the first entry is where writes go.
`JULIA_PKG_SERVER`	Point at a self-hosted package server (see below).
`JULIA_EDITOR`	What `@edit` and `InteractiveUtils.edit` open. `"zed --wait"`, `"nvim"`, etc.
`GKSwstype=100`	Headless plotting, as above.

The depot

`~/.julia` holds registries, package source, and precompilation caches — large, written constantly, and mmap'd.

- **Never put it on a network mount.** SMB in particular: precompile caches are lock-sensitive and you get corruption plus glacial startup. Local disk only. Back up `Project.toml`/`Manifest.toml` instead — everything else is reconstructible with `] instantiate`.
- Give each VM or container its own depot rather than sharing one. Multiple Julia *versions* share a depot fine (they namespace by version); multiple *machines* do not.
- `] gc` reclaims space from package versions nothing references any more. It grows to tens of gigabytes if you never run it.

Editors

The landscape, honestly

`julia-vscode` remains the most featureful integration by a distance: inline evaluation, a workspace/variable explorer, an integrated plot pane, a profiler viewer, and the only working step debugger. Everything below trades some of that away.

Editor	LSP	Inline eval	Plot pane	Debugger	Notes
VSCode + julia-vscode	LanguageServer.jl	yes	yes	yes	the benchmark
Zed	both, JETLS via dev extension	yes, via Jupyter kernel	via ZedPlotPane or Jupyter	no	fast, remote-over-SSH is excellent
Neovim	either, trivially configured	via a terminal plugin	terminal or file-watch	no	best LSP flexibility; most assembly required
Helix	either, four lines of TOML	no	no	no	zero-config, zero-extension
Emacs + eglot	either	julia-repl / julia-snail	julia-snail	no	julia-snail is the closest thing to the VSCode experience outside VSCode
Pluto.jl	no	inherent	inherent	no	not an editor; a different way to work (see below)

Zed and Neovim both have vim keybindings if that's the draw — Zed's vim mode is good, and it also ships a Helix mode.

The new language server

JETLS is built on the compiler's own machinery — JET.jl for abstract interpretation, JuliaLowering.jl for byte-precise, macro-aware lowering. That gets you type-sensitive diagnostics and go-to-definition that works through macros, neither of which `LanguageServer.jl` can do structurally. It's officially experimental, and the pace of change is high.

```
# needs Julia >= 1.12.2 and < 1.14 — check with `julia --version` first
julia -e 'using Pkg; Pkg.Apps.add(;
    url="https://github.com/aviatesk/JETLS.jl", rev="release")'
jetls --help          # verifies installation and that ~/.julia/bin is on PATH
```

Three things to know before you commit to it:

1. Installation precompiles JETLS and its dependencies — several minutes, once. Do this *before* first opening a `.jl` file in your editor, or the LSP `initialize` request can time out while it compiles.
2. **The `jetls` app is pinned to the Julia version that installed it.** After every `juliaup update` to a new minor, re-run the install command or the binary stops working. Put it in a shell function so you don't forget.
3. JETLS analyses by *running* parts of your code, including dependencies. Don't point it at a repository you don't trust.

To pin a version rather than track the release branch: `rev="2026-08-07"` (release tags are dates).

Update the same way you installed:

```
julia -e 'using Pkg; Pkg.Apps.add(;
    url="https://github.com/aviatesk/JETLS.jl", rev="release")'
```

Zed

The published extension:

1. `zed: extensions` from the command palette, search `julia`, install, restart.
2. Make sure `julia` is on PATH — the extension looks in standard locations.

That gives you syntax, tasks, and `LanguageServer.jl`. Configure its linting in `~/.config/zed/settings.json`:

```
{
  "lsp": {
    "julia": {
      "settings": { "julia.lint.missingrefs": "none" }
    }
  }
}
```

JETLS in Zed needs the dev-extension route, because Zed can only register a language server through an extension — unlike Helix or Neovim, you can't point it at an arbitrary LSP binary from settings. The JETLS-enabled fork lives on a branch:

```
# prerequisite: Rust via rustup specifically (a package-manager Rust will NOT work;
# Zed compiles extensions to wasm32-wasip2 and needs rustup to add the target)
git clone -b avi/JETLS https://github.com/aviatesk/zed-julia ~/src/zed-julia-jetls
```

Then in Zed: command palette ⟶ `zed: install dev extension` ⟶ select that directory. It overrides the published Julia extension, and the extensions page will say so. Check the branch README for current specifics — it moves, and it will eventually be merged upstream, at which point all of this collapses to "install the extension".

Debugging a failed install: `zed: open log`, or relaunch with `zed --foreground` for verbose output.

Inline evaluation in Zed. The zed-julia README says there's no inline execution, and for the extension's own integration that's true — but Zed has a separate built-in REPL that speaks Jupyter kernels, and Julia is a supported language:

```
using Pkg; Pkg.add("IJulia")     # registers a Julia kernelspec with Jupyter
```

Then `repl: run` (bind it, or `ctrl-shift-enter`) evaluates the selection, line, or block and renders the result inline beneath it. `repl: sessions` shows what's running; `repl: refresh kernelspecs` after installing a new kernel.

Understand what this is, though: a **Jupyter kernel**, not your `julia --project` session. State lives in the kernel, Revise isn't in the loop by default, and it's a separate process from any terminal REPL you have open. It's excellent for exploration and plots; it is not the edit-save-and-the-running-session-updates workflow. For that, keep a terminal REPL with Revise and use the send-selection bindings below.

Extra threads for the kernel:

```
using IJulia
IJulia.installkernel("Julia (auto threads)", env = Dict("JULIA_NUM_THREADS" => "auto"))
```

Terminal REPL send-selection bindings, for the Revise workflow:

```
// ~/.config/zed/keymap.json
[
  {
    "context": "Terminal",
    "bindings": { "ctrl-shift-`": "terminal_panel::ToggleFocus" }
  },
  {
    "context": "Editor && mode == full",
    "bindings": {
      "ctrl-shift-`": "workspace::OpenInTerminal",
      "ctrl-shift-f11": ["action::Sequence", [
        "editor::SelectEnclosingSymbol",
        "editor::CopyAndTrim",
        ["workspace::SendKeystrokes", "ctrl-` ctrl-shift-v enter ctrl-shift-`"]
      ]]
    }
  }
]
```

Adjust the paste keystroke to your terminal. Start the REPL with `julia --project -t auto`.

Plot pane without Jupyter:

```
using Pkg; Pkg.add("ZedPlotPane")
using ZedPlotPane      # load BEFORE Plots or any other plotting package
```

First plot creates `~/.cache/zed-julia/current-plot.png`; drag that tab into a side pane and it updates in place.

Zed as Julia's editor, so `@edit` opens the right file at the right line:

```
# ~/.julia/config/startup.jl
atreplinit() do repl
    InteractiveUtils.define_editor("zed") do cmd, path, line, column
        `$cmd $path:$line:$column`
    end
end
```

with `export JULIA_EDITOR="zed --wait"`.

Remote work. If the machine you compute on is headless, run Zed locally and use its SSH remote projects. The LSP, terminal, and tasks all execute on the remote host; only the UI is local. Install juliaup, the depot and `jetls` **on the remote machine** — the local Zed needs no Julia at all. This is materially better than the alternatives for a headless workstation.

Neovim

The most flexible LSP setup of the lot, because you point it directly at the binary.

```
-- JETLS (Neovim 0.11+)
vim.lsp.config("jetls", {
  cmd = { "jetls", "serve" },
  filetypes = { "julia" },
  root_markers = { "Project.toml" },
})
vim.lsp.enable("jetls")
```

`--stdio` instead of the default socket if you're editing over TRAMP-like remote schemes; sockets are the more stable transport otherwise.

Worth installing alongside:

- **`julia-vim`** — the LaTeX-to-Unicode tab substitution (`\alpha<tab>` becomes α) that the REPL has built in. Without it, typing mathematical Julia is painful. Also block-wise `%` matching for `if`/`end`.
- **`iron.nvim`**, **`vim-slime`**, or **`toggleterm`** — send-region-to-REPL. Slime is the simplest and works with tmux, which pairs well with a long-lived Julia session.
- **ZedPlotPane's equivalent**: `using UnicodePlots` for terminal plots, or watch the PNG with an image-capable terminal.

Helix

Four lines and no extension system to fight:

```
# ~/.config/helix/languages.toml
[[language]]
name = "julia"
language-servers = ["jetls"]

[language-server]
jetls = { command = "jetls", args = ["serve"], timeout = 60 }
```

The `timeout = 60` matters: Helix aborts `initialize` after 20 seconds by default, and JETLS's first startup compiles enough Julia code to blow past that on a cold cache. Without it you get `request 0 timed out` and no server.

Emacs

```
(with-eval-after-load 'eglot
  (add-to-list 'eglot-server-programs
               '(((julia-mode :language-id "julia")
                  (julia-ts-mode :language-id "julia"))
                 "jetls" "serve" "--socket" :autoport)))
```

Use `--stdio` rather than the socket if you edit over TRAMP. Add `julia-snail` for REPL integration — it's the closest thing to the VSCode experience outside VSCode, with module-aware evaluation and a plot pane.

Formatting, linting, analysis

```
julia -e 'using Pkg; Pkg.Apps.add("Runic")'          # opinionated, zero config
julia -e 'using Pkg; Pkg.Apps.add("JuliaFormatter")' # configurable, installs `jlfmt`
```

Both land in `~/.julia/bin`. **Runic** is JETLS's default and has no options at all, which ends the formatting argument permanently — pick it unless you have a specific reason not to. To switch, put a `.JETLSConfig.toml` in your project root:

```
formatter = "JuliaFormatter"     # or "Runic" (default)
```

Static analysis and package hygiene are covered in section 17, but install them now so they're there: `JET.jl` (type-level bug finding, the Dialyzer analogue), `Aqua.jl` (ambiguities, piracy, stale deps). JETLS also ships a diagnostic CLI, so the same analysis runs in CI without an editor.

REPL setup

Detail is in section 20; the minimum to do today:

```
julia -e 'using Pkg;
    Pkg.activate("v$(VERSION.major).$(VERSION.minor)"; shared=true);
    Pkg.add(["Revise", "OhMyREPL", "BenchmarkTools"])'
```

These go in the **shared** default environment, not in any project, so they're available everywhere without being anyone's dependency.

```
# ~/.julia/config/startup.jl
try
    using Revise
catch e
    @warn "Revise failed to load" exception = (e, catch_backtrace())
end
```

Revise is not optional. Julia's first-call compilation makes restarting a session expensive, so the workflow is: keep one REPL alive for hours, edit files, let Revise apply the changes. The exception is redefining a `struct`, which still requires a restart.

Self-hosted, Julia-adjacent

If you run your own infrastructure, four of these earn their keep.

Pluto.jl — reactive notebooks; the Livebook analogue, and a close one. No hidden state: the program state is always fully described by the visible code, and changing a cell re-runs its dependents. It manages a package environment per notebook automatically, and notebooks are plain `.jl` files that also run as scripts, which behaves better in git than `.livemd` or `.ipynb`.

```
using Pkg; Pkg.add("Pluto")
using Pluto; Pluto.run(host = "0.0.0.0", port = 1234)
```

A Pluto server is arbitrary code execution by design — private network only. For a container deployment, bake the depot and precompilation into the image layer; that's the slow part, and doing it at build time turns a two-minute cold start into seconds.

PlutoSliderServer.jl — runs only the `@bind` parts of notebooks, statelessly, over plain HTTP GET. Visitors get working sliders and recomputed results; they can't edit code, and there's no per-visitor Julia session. It watches a git repo and re-exports on push. This has no Livebook equivalent, and it's what makes Pluto interesting as a hosted service rather than a local tool. `PlutoPages.jl` goes further and renders a directory of notebooks into a static site.

LocalRegistry.jl + LocalPackageServer.jl — a private registry (just a git repo) so `] add MyInternalThing` works, plus a package server that serves it *and* caches the General registry. Point `JULIA_PKG_SERVER` at it and every machine on your network pulls each package once. Small, TOML-configured, and it makes CI and multi-VM setups noticeably faster.

Genie / Stipple — Phoenix and LiveView, respectively: full-stack MVC with an ORM, plus a reactive UI layer holding state server-side and syncing over JSON to a Vue3/Quasar client. Genie Builder, the visual editor, is a VSCode plugin — irrelevant here; the low-code API is fine hand-written.

Also worth knowing about: **Oxygen.jl** (Plug-shaped minimal HTTP), **Franklin.jl** (static site generator with live Julia evaluation in pages), **Documenter.jl** (the docs standard), **IJulia** (Jupyter kernel — already needed above for Zed's REPL), **Dagger.jl** (distributed task scheduler).

Verification

```
juliaup status
julia --version                      # 1.12.6
julia +lts --version                 # 1.10.11
echo $PATH | tr ':' '\n' | grep .julia/bin
which jetls runic
julia -e 'using Revise; println("revise ok")'
julia -t auto -e 'println(Threads.nthreads())'
julia -e 'using IJulia; println("kernel ok")'
```

In the editor: open a `.jl` file, confirm the LSP reports as running, hover a function and check you get a docstring, and introduce a deliberate type error to confirm diagnostics arrive.