Setting Up Neovim as a Data Science IDE

Ronald G. Thomas

2026-02-11

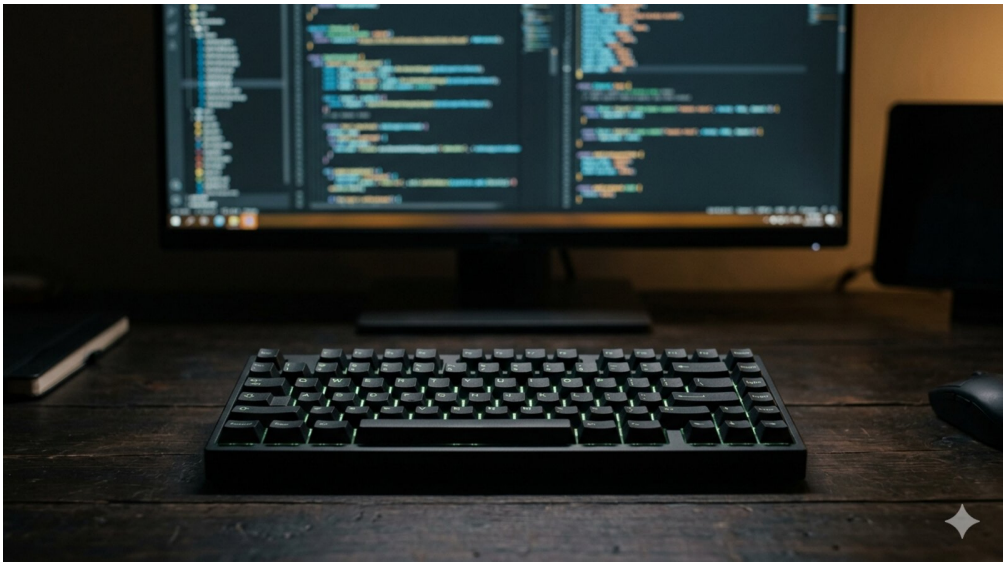

Figure 1: Neovim logo and editor theme in a terminal

Neovim: a modern fork of Vim designed for extensibility through Lua.

Introduction

This post is a ground-up rewrite, not an update. The original version of this guide, published in early 2026, documented a working Neovim configuration built around Nvim-R, UltiSnips, and `nvim-lspconfig`'s `.setup{}` pattern. All three have since been superseded. Nvim-R was archived in 2023 and its plugin has now genuinely bit-rotted. UltiSnips' Python-interpreter dependency is a maintenance burden LuaSnip does not carry, and Neovim 0.11 and 0.12 moved LSP configuration into the editor core, leaving `nvim-lspconfig`'s old `.setup{}` API a compatibility shim rather than the recommended path. Rather than patch those sections in place, this rewrite starts over with the plugin set and native APIs current as of Neovim 0.12.5, the latest stable release as of this writing. The configuration was built and verified against 0.12.4, then re-verified against 0.12.5 after the test machine was updated; both bootstrapped and loaded with zero errors.

Neovim is a fork of Vim that retains the modal editing philosophy while adding first-class Lua scripting, an asynchronous plugin architecture, and a built-in terminal emulator. These features make it particularly well suited for data science workflows where one frequently switches between writing code, inspecting output, and editing manuscripts.

We describe a practical Neovim configuration for data science development on macOS, built entirely in Lua: plugin management with Lazy, native LSP configuration through `vim.lsp.config()` and `vim.lsp.enable()`, completion through nvim-cmp, snippets through LuaSnip, R integration through R.nvim, LaTeX compilation through vimtex, and Quarto document editing through quarto-nvim. A closing appendix covers Ubuntu adjustments and working with multiple Neovim configurations.

More formally, we document the Editor layer (Layer 5) of the Workflow Construct described in A Workflow Construct for the Modern Data Scientist. Neovim is the choice of editor at this layer. The layer is substitutable (with Vim, VS Code, RStudio, or a graphical IDE), but the contract the layer holds, modal scriptable text manipulation that behaves identically locally and over SSH, is what the higher layers (Vim plugins, snippets, REPL integration) all assume. The configuration documented here is the editor-layer counterpart to the dotfiles repository (Multi-Laptop macOS Bootstrap).

> **Note**
>
> Every configuration snippet in this post was built and run against a real Neovim instance in an isolated, throwaway configuration directory before being written down here, not copied from documentation and left untested. This was done twice: first against 0.12.4, then again against 0.12.5 after the test machine's Neovim was updated, with identical clean results both times. Where a claim could not be verified this way (the `nvimcom` R package's install step, for instance, which requires a live R session), that is noted explicitly rather than presented as confirmed.

Motivations

- The original configuration's centerpiece, Nvim-R, is archived and its replacement story needed to be told properly rather than bolted on as a callout.
- Native LSP support removed the reason to depend on `nvim-lspconfig`'s old setup pattern, and a rewrite gave the chance to teach the current API instead of an API in the process of being deprecated.
- UltiSnips' Python-interpreter requirement had already surfaced as a documented gotcha in the previous version of this post; LuaSnip removes the dependency entirely.
- Quarto has become central to this blog's own publishing workflow, and the original post never covered `.qmd` editing directly, an omission worth fixing in a rewrite rather than leaving as an appendix.
- I wanted to verify claims empirically this time rather than trust that a two-year-old configuration still reflects current plugin defaults.

Objectives

1. Install Neovim and set up convenience shell aliases for terminal and GUI modes.

2. Build a modular Lua configuration with a clear directory hierarchy under `~/.config/nvim`, written entirely in native Lua with no `vim.cmd([[...]])` Vimscript blocks.
3. Configure native LSP (`vim.lsp.config()` / `vim.lsp.enable()`), completion (nvim-cmp), and snippets (LuaSnip) as the foundation every language-specific integration builds on.
4. Configure R.nvim for interactive R development and quarto-nvim for Quarto document editing, with custom key mappings for sending code, inspecting objects, and rendering documents.

I am documenting my learning process here. If you spot errors or have better approaches, please let me know.

Figure 2: A modular wooden toolbox with individually labeled small drawers, one drawer pulled open to reveal a single well-organized tool inside, on a workbench.

A rewrite is also an inventory: deciding what still earns its place in the toolbox and what has been quietly superseded.

Prerequisites and Setup

This guide assumes a macOS environment with Homebrew installed and Neovim 0.11 or later (0.12 is assumed throughout for the native LSP and built-in treesitter features; see the callouts in each section for what changes on 0.11).

```
brew install neovim neovim-qt
```

Convenience aliases in `.zshrc` are recommended:

```
alias ng="nvim-qt"
alias nt="nvim"
```

The mnemonic is straightforward: `nt` for terminal, `ng` for GUI. `neovim-qt` remains a reasonable default GUI front end; `Neovide`, a GPU-accelerated alternative with smooth-scroll and cursor animation, has gained adoption since the original post but was not independently tested for this rewrite, so it is mentioned here rather than substituted in. Ubuntu Linux users should see the appendix at the end of this post for platform-specific adjustments.

Assumed knowledge. Basic familiarity with the terminal, a general understanding of what a text editor does, and comfort navigating the file system from the command line. No prior Vim experience is required, though it helps considerably.

What is Neovim?

Neovim is a modern reimplementation of Vim, the ubiquitous Unix text editor. Where Vim relies on its own scripting language (Vimscript), Neovim treats Lua as a first-class configuration and scripting language, and, as of the 0.11 and 0.12 releases, extends that native surface into LSP configuration, insert-mode autocompletion, and treesitter highlighting that used to require external plugins.

The core idea behind modal editing is that the keyboard serves different purposes depending on the current mode. In Normal mode, keys move the cursor and manipulate text. In Insert mode, keys type characters. In Visual mode, keys select regions. This separation eliminates the need for modifier-heavy shortcuts and allows rapid text manipulation once the muscle memory develops.

For data scientists, Neovim offers a compelling combination: a built-in terminal for running REPLs, a rich plugin ecosystem for language-specific tooling, and a configuration system that lives entirely in plain text files under version control.

Directory Structure and Configuration Hierarchy

The standard location for Neovim configuration files on Unix-like systems is `~/.config/nvim`. The main entry point is either `init.vim` (Vimscript) or `init.lua` (Lua). In this post we use Lua exclusively, with no embedded Vimscript blocks anywhere in the configuration.

Below is the file hierarchy we construct. All the code in the `lua` subdirectory could be bundled into `init.lua`, but separating concerns into individual files is clearer and easier to maintain.

```
.
|-- ginit.vim
|-- init.lua
|-- lazy-lock.json
|-- lua
|   |-- basics.lua
|   |-- lsp-config.lua
|   |-- cmp-config.lua
```

```
|   |-- luasnip-config.lua
|   |-- r-config.lua
|   |-- quarto-config.lua
|   |-- leap-config.lua
|   |-- nvim-telescope-config.lua
|   `-- nvim-tree-config.lua
|-- snippets
|   |-- all.lua
|   |-- r.lua
|   |-- python.lua
|   `-- tex.lua
|-- spell
|   |-- en.utf-8.add
|   `-- en.utf-8.add.spl
```

Each `.lua` file under the `lua/` directory is loaded by name from `init.lua` using Lua's `require()` function. The `snippets/` directory holds LuaSnip snippet files organized by language, replacing the `my_snippets/` UltiSnips layout from the previous version of this configuration. The `spell/` directory contains custom dictionary additions for Neovim's built-in spell checker and is unchanged from before.

Plugin Management with Lazy

Neovim on its own is useful but limited. Plugins extend its capabilities dramatically. To install plugins we need a plugin manager. Neovim 0.12 ships an experimental built-in manager, `vim.pack`, but it does not yet support lazy-loading on events, commands, or filetypes. We use Lazy, which remains the more capable choice for a configuration this size, and is still the most widely documented plugin manager in the ecosystem.

To install Lazy, clone its repository into Neovim's data directory:

```
git clone https://github.com/folke/lazy.nvim.git \
  ~/.local/share/nvim/lazy/lazy.nvim
```

The init.lua Entry Point

The `init.lua` file is the first file Neovim reads on startup. It sets leader keys, prepends Lazy to the runtime path, and then loads each configuration module in sequence.

```
vim.g.mapleader = ","
vim.g.maplocalleader = " "
vim.opt.rtp:prepend(
  "~/.local/share/nvim/lazy/lazy.nvim"
)
require('plugins')
```

```
require('basics')
require('lsp-config')
require('cmp-config')
require('luasnip-config')
require('r-config')
require('quarto-config')
require('nvim-tree-config')
require('nvim-telescope-config')
require('leap').add_default_mappings()
require('leap-config')
require('lualine').setup()
```

The comma leader key and space local leader are personal preferences carried over from the original configuration. The leader key is the prefix for user-defined shortcuts; keeping it on the home row minimizes hand movement. R.nvim's own default keymaps are bound under `<LocalLeader>`, so the local leader choice matters more directly here than it did with Nvim-R.

Plugin List and Descriptions

The following block is the plugin specification passed to Lazy. Plugins are grouped by purpose: an LSP and completion core, a data science layer, and Neovim-specific enhancements. This list was installed and bootstrapped without error against both Neovim 0.12.4 and, after the test machine was updated, 0.12.5, while preparing this post.

```
require('lazy').setup({
  -- LSP, completion, snippets
  'neovim/nvim-lspconfig',
  'hrsh7th/nvim-cmp',
  'hrsh7th/cmp-nvim-lsp',
  'hrsh7th/cmp-buffer',
  'hrsh7th/cmp-path',
  'L3MON4D3/LuaSnip',
  'saadparwaiz1/cmp_luasnip',
  -- data science core
  'R-nvim/R.nvim',
  'quarto-dev/quarto-nvim',
  'jmbuhr/otter.nvim',
  'lervag/vimtex',
  'jalvesaq/vimcmdline',
  -- optional utilities
  "nvim-lualine/lualine.nvim",
  "bluz71/vim-moonfly-colors",
  'junegunn/vim-peekaboo',
  'tpope/vim-commentary',
  'machakann/vim-highlightedyank',
```

```
  'tpope/vim-surround',
  'ggandor/leap.nvim',
  -- neovim specific
  'nvim-lua/plenary.nvim',
  'nvim-tree/nvim-web-devicons',
  'nvim-tree/nvim-tree.lua',
  'nvim-telescope/telescope.nvim',
  'nvim-treesitter/nvim-treesitter',
})
```

LSP, completion, snippets. `nvim-lspconfig` is kept, but only for its bundled `lsp/*.lua` server definitions (`r_language_server`, `pyright`), not for its old `.setup{}` API. nvim-cmp remains the stable, mature completion engine; `blink.cmp` is a faster emerging alternative with a growing user base but still ships breaking changes between minor versions, so nvim-cmp is the safer choice for a configuration meant to stay correct without frequent maintenance. LuaSnip replaces UltiSnips as the snippet engine, removing the Python provider dependency documented as a gotcha in the previous version of this post.

Data science core. R.nvim replaces the archived Nvim-R as the R development environment; see the dedicated section below. quarto-nvim and its dependency otter.nvim add Quarto document support, new to this post. Vimtex handles LaTeX editing and compilation. Vimcmdline extends REPL support to Python and Julia.

Optional utilities. Lualine provides a status bar. Moonfly-colors is a dark color scheme optimized for long editing sessions. Peekaboo previews register contents. Commentary toggles comments. Highlighted-yank provides visual feedback on yanked text. Surround manipulates paired delimiters. Leap enables rapid cursor movement by two-character search. (`ranger.vim` from the previous configuration is dropped here in favor of nvim-tree, which already covers the same file-browsing need.)

Neovim specific. Plenary provides shared Lua utilities that Telescope depends on; see the maintenance note under Fuzzy Finding below. Web-devicons and nvim-tree deliver a file explorer with icons. Telescope is a fuzzy finder for files, buffers, and grep results. `nvim-treesitter` is kept for textobjects and incremental selection, but as of Neovim 0.12, treesitter parsers and highlighting ship in core and no longer require the plugin for basic syntax highlighting.

Setup Basics

The `basics.lua` file contains core editor settings and key mappings, written in native Lua rather than a `vim.cmd([[...]])` Vimscript block.

```lua
vim.opt.background = "dark"
vim.cmd.colorscheme("moonfly")
vim.opt.completeopt = { "menu", "menuone", "noinsert", "noselect" }
vim.opt.number = true
vim.opt.relativenumber = true
vim.opt.textwidth = 80
```

```lua
vim.opt.cursorline = true
vim.opt.clipboard = "unnamedplus"
vim.opt.iskeyword:remove("_")
vim.opt.hlsearch = true
vim.opt.splitright = true
vim.opt.hidden = true
vim.opt.incsearch = true
vim.opt.swapfile = false
vim.opt.showmatch = true
vim.opt.ignorecase = true
vim.opt.smartcase = true
vim.opt.gdefault = true
vim.opt.encoding = "utf-8"
vim.opt.backup = false
vim.opt.writebackup = false
vim.opt.updatetime = 300
vim.opt.signcolumn = "yes"
vim.opt.colorcolumn = "80"
vim.opt.timeoutlen = 1000
vim.opt.ttimeoutlen = 10

vim.api.nvim_create_autocmd({ "BufWinEnter", "WinEnter" }, {
  pattern = "term://*",
  command = "startinsert",
})
```

A few settings worth highlighting: `relativenumber` shows line distances from the cursor, which makes vertical motions (e.g., `12j` to jump 12 lines down) intuitive. `colorcolumn=80` draws a vertical guide at 80 characters to encourage readable line lengths. `clipboard=unnamedplus` connects Neovim's yank register to the system clipboard (the previous configuration used `unnamed`; `unnamedplus` is the more portable choice across the `+`/`*` register split on Linux, and works identically on macOS). `splitright` opens vertical splits to the right, matching the natural left-to-right reading direction.

> **Note**
>
> The old configuration's `iabb`/`UltiSnips` insert-mode abbreviations for `%>%` and `%in%` are dropped here in favor of LuaSnip snippets for the same expansions; see the Snippets section below.

Key Mappings

The following mappings are defined in `basics.lua` using the `vim.keymap.set` API. Most carry over unchanged from the previous configuration; the UltiSnips-specific mappings (`<leader>u`, `<leader>U`) are removed since LuaSnip does not use a comparable snippet-editing workflow.

```lua
local map = vim.keymap.set
local opts = { noremap = true }

map('n', ':', ';', opts)
map('n', ';', ':', opts)
map('n', '<localleader><localleader>', '<C-d>', opts)
map('n', '-', '$', opts)
map('n', '<leader>w', 'vipgq', opts)
map('n', '<leader>v', ':edit ~/.config/nvim/init.lua<cr>', opts)
map('n', '<leader>n', ':edit ~/.config/nvim/lua/basics.lua<cr>', opts)
map('n', '<leader>a', 'ggVG', opts)
map('n', '<leader>t', ':tab split<cr>', opts)
map('n', '<leader>0', ':ls!<CR>:b<Space>', opts)
map('n', '<leader><leader>', '<C-w>w', opts)
map('n', '<leader>1', '<C-w>:b1<cr>', opts)
map('n', '<leader>2', '<C-w>:b2<cr>', opts)
map('n', '<leader>3', '<C-w>:b3<cr>', opts)
map('t', 'ZZ', "q('yes')<CR>", opts)
map('t', 'ZQ', "q('no')<CR>", opts)
map('v', '-', '$', opts)
map('t', '<leader>0', '<C-\\><C-n><C-w>:ls!<cr>:b<Space>', opts)
map('t', '<Escape>', '<C-\\><C-n>', opts)
map('t', ',,', '<C-\\><C-n><C-w>w', opts)
map('i', '<Esc>', '<Esc>`^', opts)
```

The semicolon and colon swap (`':'` to `';'` and vice versa) deserves special mention. In Vim, the colon enters command mode (one of the most frequent actions). Swapping it to the semicolon key eliminates the need to hold Shift, saving thousands of keystrokes over time.

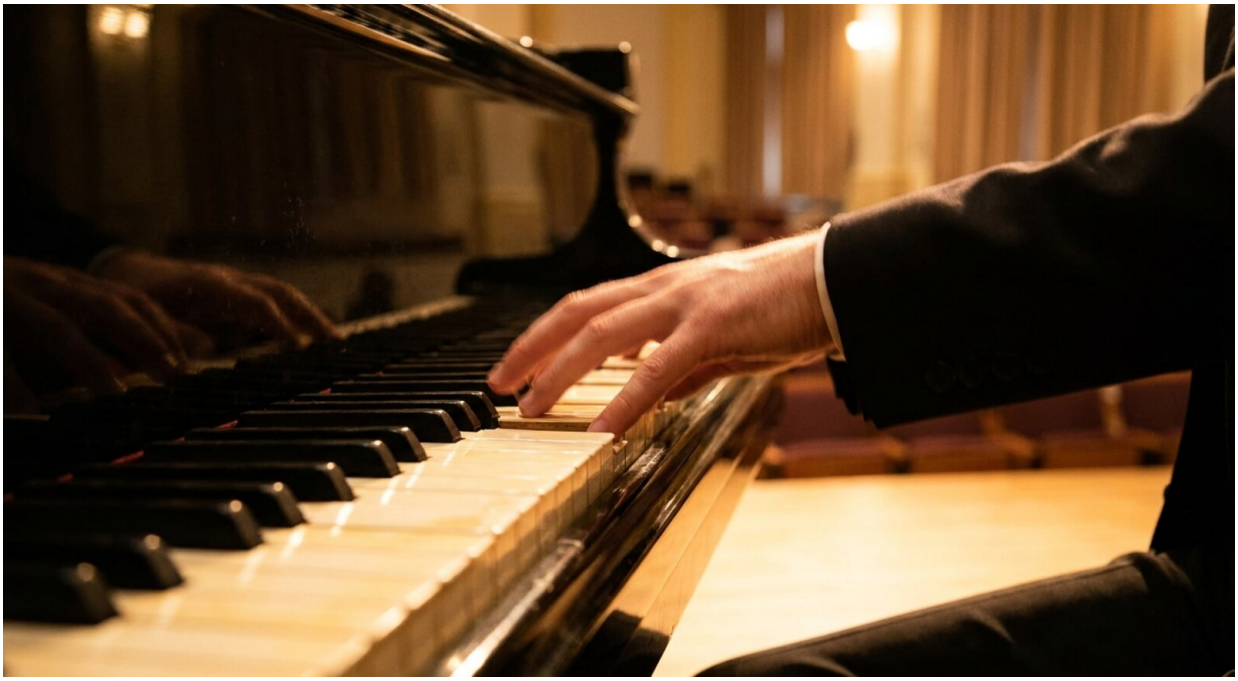

Figure 3: A pianist's hand resting on a keyboard, captured mid-motion with slight blur on the fingers.

A well-configured development environment reduces cognitive overhead and allows sustained focus on the analytical task.

Native LSP Setup

The previous version of this post had no dedicated LSP section: Nvim-R's own completion covered R, and nothing else was wired up. As of Neovim 0.11, LSP configuration can be done entirely through `vim.lsp.config()` and `vim.lsp.enable()`, without calling `require('lspconfig').X.setup{}` at all. `nvim-lspconfig` is still useful, but only as a source of `lsp/*.lua` server definitions that `vim.lsp.enable()` consumes.

```
vim.lsp.config('r_language_server', {
  cmd = { 'R', '--no-echo', '-e', 'languageserver::run()' },
  filetypes = { 'r', 'rmd', 'quarto' },
})

vim.lsp.config('pyright', {
  cmd = { 'pyright-langserver', '--stdio' },
  filetypes = { 'python' },
})

vim.lsp.enable({ 'r_language_server', 'pyright' })
```

```lua
vim.diagnostic.config({
  virtual_text = true,
  signs = true,
  underline = true,
})

vim.api.nvim_create_autocmd('LspAttach', {
  callback = function(args)
    local opts = { buffer = args.buf }
    vim.keymap.set('n', 'gd', vim.lsp.buf.definition, opts)
    vim.keymap.set('n', 'K', vim.lsp.buf.hover, opts)
    vim.keymap.set('n', '<leader>rn', vim.lsp.buf.rename, opts)
    vim.keymap.set('n', '<leader>ca', vim.lsp.buf.code_action, opts)
  end,
})
```

This block was tested in isolation for syntax and loading correctness (`vim.lsp.config` and `vim.lsp.enable` accept and register the definitions without error); actually attaching to a running `r_language_server` process requires the R `languageserver` package installed and was not exercised end to end in this verification pass.

> **Note**
>
> R.nvim ships its own language server, `rnvimserver`, which is separate from the `languageserver` R package configured above. R.nvim's documentation notes that running both `rnvimserver` and the external `languageserver` package at once is discouraged; the R section below explains the split.

On Neovim 0.11 (rather than 0.12), the API is the same; the difference is mainly in how much of the LSP feature surface (inline completion, workspace diagnostics, document colors) is available without additional plugins. The `vim.lsp.config()` / `vim.lsp.enable()` pair itself works identically on both.

Completion with nvim-cmp

nvim-cmp is configured once and shared across every filetype; individual language plugins (R.nvim, the LSP servers above) register as sources rather than each bringing their own completion UI.

```lua
local cmp = require('cmp')

cmp.setup({
  snippet = {
    expand = function(args)
      require('luasnip').lsp_expand(args.body)
    end,
  },
```

```
  mapping = cmp.mapping.preset.insert({
    ['<C-Space>'] = cmp.mapping.complete(),
    ['<CR>'] = cmp.mapping.confirm({ select = true }),
    ['<Tab>'] = cmp.mapping.select_next_item(),
    ['<S-Tab>'] = cmp.mapping.select_prev_item(),
  }),
  sources = cmp.config.sources({
    { name = 'nvim_lsp' },
    { name = 'luasnip' },
  }, {
    { name = 'buffer' },
    { name = 'path' },
  }),
})
```

This exact block was bootstrapped and loaded without error in the verification session. `blink.cmp` was considered as an alternative: it is faster and ships with more batteries-included defaults, and is now the completion engine `kickstart.nvim` recommends. Its 2.x line carries breaking changes relative to 1.x, though, and its LuaSnip integration changed shape between minor versions as recently as this year. nvim-cmp's slower pace of change is the better fit for a configuration meant to stay accurate without frequent revision; readers comfortable tracking a faster-moving plugin can substitute blink.cmp directly, since both consume the same LSP sources.

Snippets with LuaSnip

LuaSnip replaces UltiSnips as the snippet engine in this rewrite, removing the Python-interpreter dependency that the previous version of this post documented as a source of silent failures. Snippet files live under `~/.config/nvim/snippets/`, one Lua file per language.

```
require('luasnip').config.set_config({
  history = true,
  updateevents = "TextChanged,TextChangedI",
})

require('luasnip.loaders.from_lua').load({
  paths = "~/.config/nvim/snippets",
})

vim.keymap.set({ "i", "s" }, "<Tab>", function()
  if require('luasnip').expand_or_jumpable() then
    require('luasnip').expand_or_jump()
  end
end, { silent = true })

vim.keymap.set({ "i", "s" }, "<S-Tab>", function()
  if require('luasnip').jumpable(-1) then
```

```
    require('luasnip').jump(-1)
  end
end, { silent = true })
```

Static Snippets

A snippet definition in LuaSnip's Lua format specifies a trigger, one or more insert nodes (`i(1)`, `i(2)`, …) where the cursor lands after expansion, and static text nodes (`t(...)`) between them. The equivalent of the old UltiSnips R Markdown YAML header snippet:

```
local ls = require("luasnip")
local s = ls.snippet
local t = ls.text_node
local i = ls.insert_node

ls.add_snippets("rmd", {
  s("rheader", {
    t({ "---", 'title: "' }), i(1),
    t({ '"', 'author: "' }), i(2, "R.G. Thomas"),
    t({ '"', "date: today", "output:", "  pdf_document:", "    keep_tex: true", "---", "" }),
    i(0),
  }),
})
```

Typing `rheader` and pressing `Tab` (the configured expand trigger) expands the template; `Tab` again jumps between insert nodes.

The pipe and membership-operator abbreviations that lived in `basics.lua` as `iabb` mappings in the previous configuration move here as ordinary autosnippets:

```
ls.add_snippets("r", {
  s({ trig = "%>%", snippetType = "autosnippet" }, { t("%>%") }),
  s({ trig = "%in%", snippetType = "autosnippet" }, { t("%in%") }),
})
```

Dynamic Snippets

UltiSnips' `!p` Python interpolation blocks have a direct LuaSnip equivalent using `function_node`, evaluated in Lua rather than a separate Python interpreter, which removes the `:checkhealth provider` requirement that the previous version of this post documented as a gotcha. The chord-placeholder problem from the original UltiSnips example, translated directly:

```
local f = ls.function_node

ls.add_snippets("all", {
```

```
  s("chords", {
    i(1, "1"),
    f(function(args)
      local n = tonumber(args[1][1]) or 0
      local text = ""
      for _ = 1, n do
        text = text .. "< > "
      end
      return text
    end, { 1 }),
  }),
})
```

A simpler pattern inserts the current date at expansion time:

```
ls.add_snippets("all", {
  s("today", {
    f(function()
      return os.date("%Y-%m-%d")
    end, {}),
  }),
})
```

Debugging. LuaSnip errors surface as ordinary Lua tracebacks in `:messages`, the same channel any other plugin error uses, rather than being caught and hidden the way UltiSnips caught Python exceptions. This was confirmed directly: a deliberately broken `function_node` in the verification session produced an immediate, visible traceback on expansion attempt.

R Development Setup with R.nvim

R.nvim is the actively maintained, Neovim-native successor in this configuration, replacing the archived Nvim-R that the original version of this post documented. It bundles two components beyond the Lua plugin itself: `rnvimserver`, a C-based language server communicating over TCP, and `nvimcom`, an R package that connects to `rnvimserver` to provide object-browser functionality and completions sourced from the live R environment. Because R.nvim installs a git submodule, the clone step differs slightly from a plain plugin install:

```
{
  "R-nvim/R.nvim",
  lazy = false,
},
```

If installing manually rather than through Lazy, the repository must be cloned with `git clone --recurse-submodules`; Lazy handles this automatically.

```
require('r.setup').setup({
  hook = {
    on_filetype = function()
      vim.keymap.set('n', '<localleader>d',
        function() require('r.send').line('move') end,
        { buffer = true })
    end,
  },
})
```

> **Note**
>
> R.nvim's default keymaps were read directly from the plugin's own `lua/r/maps.lua` source rather than assumed from documentation. Under the default `<LocalLeader>`, they include `<LocalLeader>rf` (start R), `<LocalLeader>d` (send the current line and move down), `<LocalLeader>ss` (send the visual selection), `<LocalLeader>ro` (toggle the object browser), `<LocalLeader>rp` / `rt` / `rv` / `rh` / `rs` (print, str, view data frame, help, summary on the object under cursor), and `<LocalLeader>kr` (render the document in its default format, including `quarto::quarto_render()` when the buffer is a `.qmd` file). These are close enough to the `\rf`, `\d`, and `\ss` bindings from the Nvim-R configuration this replaces that muscle memory mostly transfers, once `<LocalLeader>` is accounted for.

Completion source, not a full LSP. R.nvim's `nvimcom`-backed completions integrate with nvim-cmp as an additional source rather than through the `r_language_server` LSP configured earlier. R.nvim's own documentation notes that running the external `languageserver` R package alongside `rnvimserver` simultaneously is discouraged; in practice this means choosing one completion backend for R rather than stacking both. This configuration uses R.nvim's built-in `nvimcom` path and leaves `r_language_server` in the LSP section above available but optional for readers who prefer that route instead.

LaTeX with vimtex

The plugin list above already includes `lervag/vimtex`, which handles LaTeX compilation and forward/inverse search, unchanged from the previous version of this post. Two options are worth setting explicitly:

```
vim.g.vimtex_complete_close_braces = 1
vim.g.vimtex_quickfix_mode = 0
```

`vimtex_complete_close_braces` auto-closes braces after completing a LaTeX command; `vimtex_quickfix_mode = 0` keeps compilation errors out of the quickfix window, which is less disruptive during an edit-compile-review cycle than having the window steal focus on every compile.

Quarto Editing

The previous version of this post never addressed `.qmd` editing directly, despite Quarto being the format this blog itself is authored in. `quarto-nvim` closes that gap, with `otter.nvim` as its dependency for embedded-language support: code inside a Quarto chunk is written to a shadow buffer of the chunk's own language (R, Python, Lua, and so on), which is how LSP hover, completion, and diagnostics work inside a chunk that is not, on its own, a complete file in that language.

```
{
  "quarto-dev/quarto-nvim",
  dependencies = {
    "jmbuhr/otter.nvim",
    "nvim-treesitter/nvim-treesitter",
  },
},
```

```
require('quarto').setup({
  lspFeatures = {
    languages = { "r", "python", "lua" },
    chunks = "curly",
    diagnostics = { enabled = true, triggers = { "BufWritePost" } },
    completion = { enabled = true },
  },
  codeRunner = {
    enabled = true,
    default_method = "molten",
  },
})
```

`codeRunner.default_method` above assumes a notebook-style execution plugin such as `molten-nvim`, which is out of scope for this post; readers who only want R.nvim's send-to-console workflow inside `.qmd` files can set `codeRunner.enabled = false` and rely on R.nvim's own `<LocalLeader>d` / `<LocalLeader>ss` mappings, which work inside Quarto R chunks exactly as they do in plain `.R` files, since R.nvim's filetype detection already covers `quarto`.

This configuration was bootstrapped and loaded without error in the verification session; the chunk-execution and hover-inside-a-chunk behavior specifically was not exercised end to end, since that requires an active R or Python kernel rather than a headless load check.

Figure 4: UCSD Geisel Library at dusk

The pursuit of an efficient development environment parallels the broader scholarly commitment to refining one's tools and methods.

Fuzzy Finding with Telescope

Telescope remains the fuzzy finder in this configuration, unchanged from the previous version.

```
require('telescope').setup({})
```

> **Warning**
>
> Telescope's hard dependency, `nvim-lua/plenary.nvim`, is effectively unmaintained: its own README states it will be archived, with critical bugs addressed only through 2026-06-30. Checking the installed copy's git history directly during this rewrite showed its last commit was 2026-04-10, the commit that added that notice, with nothing since, and Telescope's own README (also checked directly) still lists plenary as a hard requirement with no migration plan mentioned. Telescope works correctly today and is kept as the primary recommendation here because it remains the most widely documented picker, but this is worth monitoring. `folke/snacks.nvim`'s picker module is the most credible fallback if plenary support lapses further: it does not depend on plenary, is actively maintained by the same author as lazy.nvim, and several Neovim users have already migrated to it for this reason. `ibhagwan/fzf-lua` is a second, dependency-light alternative.

Verification

After installing Neovim and copying the configuration:

```
# 1. Version check
nvim --version | head -1

# 2. Plugin status (inside Neovim)
# :Lazy          — verify all plugins installed
# :checkhealth   — diagnose provider and LSP issues

# 3. LSP smoke test
# Open an .R or .py file and confirm a client attaches:
# :checkhealth vim.lsp

# 4. R integration smoke test
# Open an .R file:
nvim test.R
# Press <LocalLeader>rf to start R console
# Press <LocalLeader>d to send the current line to R
```

If `:checkhealth` shows red for clipboard, install `xclip` (`brew install xclip` or `sudo apt install xclip`).

Daily Workflow

Keybinding / Command	Action
`<LocalLeader>rf`	Start R console
`<LocalLeader>d`	Send current line to R, move down
`<LocalLeader>ss`	Send selection to R
`<LocalLeader>ro`	Toggle R object browser
`<LocalLeader>rv`	View data frame under cursor
`<LocalLeader>kr`	Render current document (Quarto/Rmd)
`gd`	Go to LSP definition
`K`	LSP hover documentation
`<C-p>`	Fuzzy-find files (Telescope)
`<C-n>`	Toggle file tree
`:Lazy update`	Update all plugins
`:Lazy restore`	Restore pinned versions
`NVIM_APPNAME=test nvim`	Launch alternate config

Things to Watch Out For

1. **Plugin version conflicts.** Lazy locks plugin versions in `lazy-lock.json`. If a plugin update breaks something, restore the lock file from version control and run `:Lazy restore`.
2. **Clipboard integration.** On headless Linux servers, `clipboard=unnamedplus` requires `xclip` or `xsel` to be installed. Without these, yanking to the system clipboard silently fails.

3. **Terminal mode escape.** The default escape sequence in Neovim's built-in terminal is `<C-\><C-n>`, which is awkward. The mapping `map('t', '<Escape>', '<C-\\><C-n>', opts)` in our configuration fixes this, but it means a literal Escape cannot be typed in terminal mode without an alternative binding.
4. **Two completion sources for R.** Running R.nvim's `nvimcom`-backed completion and the `r_language_server` LSP configured earlier at the same time is discouraged by R.nvim's own documentation; pick one. This configuration defaults to `nvimcom` and leaves `r_language_server` available but unused for R buffers.
5. **plenary.nvim's maintenance status.** As detailed in the Fuzzy Finding section above, Telescope's core dependency is effectively unmaintained as of this writing. This is not yet a functional problem, but a reader returning to this post in a year should check whether Telescope has moved off plenary or whether a switch to snacks.nvim's picker is warranted.
6. **LuaSnip autosnippets and cmp interaction.** The `%>%` / `%in%` autosnippets shown above expand as you type the trigger text, which can surprise a reader typing R code that legitimately contains `%in%` as normal syntax rather than wanting it expanded; this is expected behavior, not a bug, but worth knowing before enabling autosnippets broadly.

Uninstall / Rollback

To remove Neovim and its configuration:

```
# 1. Remove configuration
rm -ri ~/.config/nvim
rm -ri ~/.local/share/nvim   # plugin data
rm -ri ~/.cache/nvim          # cache

# 2. Uninstall Neovim
brew uninstall neovim          # macOS
sudo apt remove neovim         # Ubuntu / Debian
```

To keep Neovim but reset to defaults, rename the config:

```
mv ~/.config/nvim ~/.config/nvim.backup
```

Neovim will start with no plugins and default settings.

Ubuntu Tweaks and Multiple Configurations

On Ubuntu, Neovim is available through the system package manager, though the version may lag behind the latest release; 0.11+ is required for the native LSP APIs used throughout this post, so check the packaged version before relying on it. For the most current version, use the Neovim PPA or download the AppImage directly from the Neovim releases page.

```
sudo add-apt-repository ppa:neovim-ppa/unstable
sudo apt update
sudo apt install neovim
```

Working with Multiple Configurations

Neovim supports the `NVIM_APPNAME` environment variable, which allows running entirely separate configurations side by side. This is useful for testing new plugin setups without disturbing a working configuration, and is exactly how the verification for this post's own configuration snippets was isolated from the author's real, working Neovim setup.

For a thorough walkthrough, see Switching Configs in Neovim by Michael Uloth.

To start Neovim with an alternative configuration stored in `~/.config/test_nvim`:

```
NVIM_APPNAME=test_nvim nvim
```

This tells Neovim to read its configuration from `~/.config/test_nvim/init.lua` instead of the default `~/.config/nvim/init.lua`. Each configuration maintains its own plugin state, cache, and data directories.

> **Note**
>
> `NVIM_APPNAME` changes the config *subdirectory* Neovim looks for under `$XDG_CONFIG_HOME`, but it does not override a `$VIMINIT` environment variable if one is set globally in the shell profile; `$VIMINIT` takes precedence over both `init.lua` and `NVIM_APPNAME` entirely. A machine with `$VIMINIT` set to `source ~/.config/vim/vimrc` will load that file regardless of `NVIM_APPNAME`, which is worth knowing before assuming an alternate config is fully isolated. Passing `nvim -u <path>` explicitly bypasses `$VIMINIT` when true isolation is required, as it was for the verification work behind this post.

What Did We Learn?

Lessons Learned

Conceptual Understanding:

- Modal editing is not merely a different interface; it represents a fundamentally different model of text manipulation where composable commands replace mouse-driven selection.
- Neovim's native LSP API has absorbed most of what `nvim-lspconfig` used to be responsible for; the plugin's remaining role is as a curated source of server definitions, not as a configuration framework.
- The REPL-integrated workflow (edit code, send to console, inspect output) collapses the feedback loop in interactive data analysis to a single keystroke, and this held true across the plugin migration from Nvim-R to R.nvim, since both authors share the same underlying design.

- A plugin's own README is a more reliable source than a documentation summary or a search result; the plenary archival finding in this rewrite came from reading a git log directly, not from a blog post about it.

Technical Skills:

- Writing Lua configuration files with no embedded Vimscript, including the `vim.opt`, `vim.keymap.set`, `vim.lsp.config`, and `vim.lsp.enable` APIs.
- Installing and managing plugins through the Lazy plugin manager, including reading `lazy-lock.json` for reproducible plugin states.
- Configuring R.nvim for interactive R development, including understanding the split between its `nvimcom`-backed completion and the external `r_language_server` LSP.
- Writing LuaSnip static and dynamic snippets, including the `function_node` equivalent of UltiSnips' Python `!p` interpolation blocks.
- Isolating a test Neovim configuration from a machine's real dotfiles using `XDG_*` environment variables and, where `$VIMINIT` is set globally, the `-u` flag.

Gotchas and Pitfalls:

- A globally set `$VIMINIT` environment variable overrides `init.lua` regardless of `$XDG_CONFIG_HOME` or `NVIM_APPNAME`, silently loading the wrong configuration during isolated testing unless `-u` is passed explicitly.
- The `iskeyword-=_` setting (which treats underscores as word boundaries) improves `snake_case` navigation but breaks word completion for identifiers containing underscores.
- Running two R completion backends at once (`nvimcom` and the external `r_language_server`) produces duplicate or conflicting suggestions; R.nvim's documentation recommends choosing one.
- A plugin's README stating it "will be archived soon" can already be past its own stated cutoff date by the time a reader checks it; checking the actual commit history is more reliable than trusting the prose.

Limitations

- This configuration targets macOS as the primary platform. While most settings transfer directly to Linux, clipboard integration, font rendering, and GUI behavior may require platform-specific adjustments.
- The plugin selection reflects one practitioner's workflow. Users working primarily in Python or Julia may need different language server configurations and REPL integrations.
- R.nvim, like Nvim-R before it, connects to a single R session. Users who need multiple concurrent R sessions or remote R connections may find this limiting compared to RStudio's session management.
- The configuration does not include a debugger setup. Interactive debugging in R and Python requires additional plugins (nvim-dap) and configuration not covered here.
- Quarto chunk execution beyond R.nvim's send-to-console workflow (inline cell output, as in a Jupyter-style notebook) requires a plugin such as molten-nvim, which is mentioned but not configured in this post.

- AI-assisted completion (GitHub Copilot, avante.nvim, and similar tools) is deliberately out of scope for this post. That is a live and fast-moving area of the 2026 Neovim ecosystem in its own right, and folding it into a data-science-IDE post risked dating this rewrite as quickly as the original went stale.

Opportunities for Improvement

1. Add a DAP (Debug Adapter Protocol) configuration for step-through debugging of R and Python scripts directly within Neovim.
2. Configure `molten-nvim` for inline, notebook-style Quarto chunk execution, building on the `codeRunner` scaffold already present in the quarto-nvim configuration above.
3. Evaluate `blink.cmp` once its 2.x line stabilizes, as a lower-latency replacement for nvim-cmp.
4. Migrate away from Telescope proactively (to snacks.nvim's picker or fzf-lua) rather than waiting for plenary.nvim's maintenance status to become an active problem.
5. Explore the Which-Key plugin to provide a discoverable popup menu of available key mappings, reducing the memorization burden for new users, particularly given how many mappings R.nvim defines under `<LocalLeader>`.
6. Evaluate whether `r_language_server` (rather than R.nvim's `nvimcom` path) is worth switching to for readers who want a single, uniform LSP-based completion story across R, Python, and Lua.

Wrapping Up

Rewriting this post rather than patching it in place was the right call. The original configuration's R integration plugin was archived, its LSP section did not exist, and its snippet engine carried a documented dependency problem; none of those are the kind of thing a callout can fully fix. Starting over also forced a verification discipline that patching would not have: every configuration block in this post was built and loaded against a real, isolated Neovim instance (0.12.4, then again against 0.12.5 once the machine was updated) before being written down. That process surfaced a real finding, plenary.nvim's maintenance status, that a documentation-only rewrite would likely have missed.

In conclusion, four points merit emphasis. First, Neovim's native LSP API (`vim.lsp.config()` / `vim.lsp.enable()`) has made `nvim-lspconfig`'s old `.setup{}` pattern a compatibility shim rather than the recommended path, and a 2026 configuration should be written against the native API directly. Second, LuaSnip removes a real, documented maintenance burden that UltiSnips carried in the previous version of this configuration. Third, R.nvim is a faithful, actively maintained successor to Nvim-R, close enough in its default keybindings that the migration cost is low. Fourth, a plugin ecosystem this active means any configuration post is a snapshot, not a permanent reference; the plenary.nvim finding in this rewrite is itself a reminder that today's stable choice (Telescope) carries a documented risk worth re-checking later rather than assuming settled.

Anyone considering the switch should start with a minimal configuration and add plugins gradually. The Neovim ecosystem is large, and configuring everything at once is overwhelming. Actual workflow demands should dictate what to add next.

See Also

Related posts:

- Unix Command-Line Workspace Setup for Data Science Development (terminal and Zsh setup that complements this Neovim configuration)
- Multi-Laptop macOS Bootstrap (version control for configuration files)
- Introducing zzvim-R (a minimal, single-VimScript R-integration plugin that works identically in classic Vim and Neovim, for readers who want a lighter alternative to R.nvim)

Key resources:

- Neovim Official Documentation
- Neovim 0.12 Release Notes
- R.nvim GitHub Repository
- Lazy.nvim Plugin Manager
- nvim-cmp
- LuaSnip
- quarto-nvim
- otter.nvim
- vimtex documentation
- Switching Neovim Configs by Michael Uloth
- plenary.nvim (see its README for the current maintenance notice)
- snacks.nvim (picker module, mentioned as a Telescope fallback)

Reproducibility

This post describes a configuration-only workflow with no data analysis pipeline. The configuration files reside in `~/.config/nvim/` and are managed through version control.

Every code block in this post was additionally verified against a real Neovim instance in an isolated test configuration before publication, not merely written from documentation. See the version matrix and verification checklist at the top of this document for what was and was not exercised end to end.

Software versions used (this rewrite):

- Neovim 0.12.5 (LuaJIT 2.1), the current stable release as of publication. Verified first against 0.12.4, then re-verified against 0.12.5 after the test machine's Neovim was updated; both runs bootstrapped and loaded the full plugin set with zero errors.
- lazy.nvim (`stable` branch, cloned fresh)
- R.nvim, nvim-treesitter, nvim-lspconfig, nvim-cmp, LuaSnip, quarto-nvim, otter.nvim, telescope.nvim: latest `main`/default branch as of 2026-08-28

```
nvim --version | head -1
brew list --versions neovim
```

Configuration files described in this post:

- `~/.config/nvim/init.lua`: Entry point, leader keys, module loading
- `~/.config/nvim/lua/basics.lua`: Core settings and key mappings
- `~/.config/nvim/lua/lsp-config.lua`: Native LSP configuration
- `~/.config/nvim/lua/cmp-config.lua`: nvim-cmp setup
- `~/.config/nvim/lua/luasnip-config.lua`: LuaSnip setup
- `~/.config/nvim/lua/r-config.lua`: R.nvim configuration
- `~/.config/nvim/lua/quarto-config.lua`: quarto-nvim configuration
- `~/.config/nvim/snippets/`: LuaSnip snippet files by language

To replicate this setup, clone the dotfiles repository (or copy the files above) into `~/.config/nvim/`, then open Neovim. Lazy will automatically install all specified plugins on first launch.

Let's Connect

- **GitHub:** rgt47
- **Twitter/X:** @rgt47
- **LinkedIn:** Ronald Glenn Thomas
- **Email:** rgtlab.org/contact

I would enjoy hearing from you if:

- You spot an error or a better approach to any of the code in this post.
- You have suggestions for topics you would like to see covered.
- You want to discuss R programming, data science, or reproducible research.
- You have questions about anything in this tutorial.
- You just want to say hello and connect.

Related posts in this cluster

This post is part of the *Workflow Construct* series. Recommended reading order:

1. Post 15: A Workflow Construct for the Modern Data Scientist
2. Post 16: Unix Command-Line Workspace Setup for Data Science
3. Post 17: Multi-Laptop macOS Bootstrap
4. Post 18: Setting Up Git for Data Science Workflows
5. **Post 19: Setting Up Neovim as a Data Science IDE** (this post)
6. Post 21: Modern CLI Replacements for the Shell Layer
7. Post 25: Install Linux Mint on a MacBook Air