The Essential Neovim Setup for Full-Stack Development in 2025

You're missing several game-changing plugins and workflows that would dramatically improve your development experience across React Native, Symfony, Android native, and Python projects. The gap between your minimal setup and what the community consistently recommends as essential is substantial, particularly in LSP configuration, debugging, language-specific tooling, and productivity enhancements. Your foundation with Lazy.nvim, LazyGit, Telescope, and Neo-tree is solid, but you're essentially missing the entire middle layer that transforms Neovim from a text editor into a proper development environment—LSP intelligence, autocompletion, debugging capabilities, testing integration, and the workflow optimizations that experienced users consider non-negotiable.

The most critical missing piece is a complete LSP and completion stack. Without nvim-lspconfig, mason.nvim, and nvim-cmp, you lack intelligent code completion, go-to-definition, inline diagnostics, and all the IDE-like features that make modern development productive. LiveCode247 +3↗ This becomes especially painful in your polyglot environment where context switching between TypeScript, PHP, Java, Kotlin, and Python without language server support means constantly leaving Neovim for documentation lookups and losing flow state. The second major gap is debugging infrastructure—nvim-dap with language-specific adapters would let you debug React Native apps, PHP Symfony applications, Android code, and Python scripts directly in your editor, which the community consistently identifies as a transformative addition.

What makes exceptional Neovim setups in 2025

The most successful Neovim configurations share remarkably consistent patterns that go beyond plugin lists. LazyVim, NvChad, and AstroNvim—the three most popular distributions with 20k+, 27k+, and 13.8k+ stars respectively—all prioritize blazing fast startup times between 20-70ms despite loading 100+ plugins through aggressive lazy loading strategies. github↗Neovimcraft↗ They achieve this through event-based loading (VeryLazy, BufRead, InsertEnter), filetype-specific activation, and command-based triggers that defer everything non-essential until actually needed. NvChad reaches startup times as low as 0.02 seconds with 93% of plugins lazy-loaded, demonstrating that feature-richness and performance aren't mutually exclusive. github↗GitHub↗

These exceptional setups employ a modular architecture that treats the main distribution as an importable plugin via lazy.nvim, allowing users to layer custom configurations on top without breaking updates. GitHub↗Project-Awesome↗ The file structure follows a consistent pattern with `lua/config/` for core settings and `lua/plugins/` for plugin specifications, each focused on a single responsibility. Vineeth↗ LazyVim's approach of automatically loading any Lua file in the plugins directory exemplifies this philosophy, where adding functionality means dropping a new file rather than editing monolithic configuration. LazyVim +2↗

What truly distinguishes these setups is their emphasis on built-in Neovim features before reaching for plugins. They leverage native LSP aggressively, use Treesitter for syntax awareness universally, and prefer Lua-based plugins that integrate cleanly with Neovim's API. Stack Exchange↗github↗ The community consensus in 2025 has shifted decisively away from "distributions" toward "starter configs" that educate users— Stack Exchange↗kickstart.nvim's single-file, heavily-commented approach represents this philosophy, where understanding your configuration becomes as important as having features. github↗Khuedoan↗

The LSP and completion foundation you need immediately

Building a proper LSP stack should be your first priority because it fundamentally transforms how you interact with code. The modern approach requires three interconnected plugins: mason.nvim for managing language server installations, mason-lspconfig.nvim as the bridge to nvim-lspconfig, and nvim-lspconfig itself for server configurations. GitHub +3↗ Mason provides a unified interface for installing LSP servers, DAP adapters, linters, and formatters across 200+ tools with simple `:Mason` commands or automatic `ensure_installed` configuration. GitHub +2↗ This replaces the previous nightmare of manually tracking down and installing language servers for each language, which was a major pain point in earlier Neovim LSP setups.

The completion engine choice in 2025 centers on nvim-cmp, which remains the de facto standard with 9,000+ GitHub stars and an extensive ecosystem of sources. Medium↗ While blink.cmp has emerged as a faster alternative written in Rust and is now LazyVim's default, nvim-cmp's maturity, stability, and comprehensive source support make it the safer choice for production workflows. github↗Barbarian Meets Coding↗ You'll need cmp-nvim-lsp for LSP completion, cmp-buffer for text from open buffers, cmp-path for filesystem paths, and a snippet engine—LuaSnip is the 2025 standard, pure Lua with no Python dependencies, supporting dynamic snippets with Lua functions and maintaining compatibility with VSCode snippet formats Cj↗Evesdropper↗ through friendly-snippets. github↗Medium↗

The critical innovation in 2025 LSP configuration is the LspAttach autocommand pattern, which replaces the older on_attach approach. Neovim↗Stack Exchange↗ Instead of defining keybindings within each server's setup function, you create a single autocommand that fires when any LSP server attaches to a buffer. Neovim↗GitHub↗ This centralizes all LSP keybindings, avoids repetition across server configs, and works elegantly with dynamically registered capabilities. The pattern looks like setting up an autocommand for the LspAttach event that receives buffer and client information, then configures buffer-local keymaps for gd (go to definition), K (hover docs), leader-rn (rename), and leader-ca (code actions). Neovim↗ This approach has become the community standard because it separates concerns cleanly—server configurations handle server-specific settings while the autocommand handles universal keybindings. Stack Exchange↗

For your specific language needs, you'll install tsserver for TypeScript/JavaScript/React, intelephense for PHP/Symfony, jdtls for Java/Android, kotlin_language_server for Kotlin, pyright for Python, yamlls for YAML, and jsonls for JSON. GitHub↗ The handlers pattern in mason-lspconfig automates setup for all installed servers with a default handler while allowing server-specific overrides—this means you configure lua_ls to recognize the vim global once, and all other servers get sensible defaults automatically. Medium↗

Treesitter transforms syntax understanding completely

nvim-treesitter provides syntax highlighting, code folding, and indentation using abstract syntax tree parsing rather than regex patterns, fundamentally improving how Neovim understands code structure. github +2↗ The essential configuration enables highlight and indent modules while ensuring parsers are installed for your languages—tsx, typescript, javascript, php, java, kotlin, python, yaml, and json. Unlike traditional syntax highlighting that breaks with complex nested structures, Treesitter parses code correctly even in deeply nested JSX components or complex PHP templates, which becomes especially noticeable in React Native components with multiple levels of conditional rendering.

The game-changing addition is nvim-treesitter-textobjects, which provides syntax-aware text objects that transform code editing. LiveCode247↗ Once configured, you can select entire functions with af (around function) or if (inner function), jump between methods with]m and [m, GitHub↗ swap function parameters with dedicated keybindings, and perform structural edits that understand code semantics. github↗Project-Awesome↗ For someone working across multiple languages, this creates a consistent editing vocabulary—af selects a JavaScript function, a PHP method, a Python function, or a Kotlin function with the same keystrokes, dramatically reducing cognitive load during context switches.

Additional Treesitter plugins enhance specific workflows. nvim-treesitter-context shows the current function or class signature at the top of the screen, invaluable when working in long files where you lose track of which method you're editing. Project-Awesome↗ nvim-ts-autotag automatically closes and renames HTML and JSX tags, essential for React and React Native work where manually tracking closing tags in complex component hierarchies causes constant friction. Devas↗github↗ nvim-treesitter-refactor provides smart renaming and highlights all usages of the identifier under your cursor, complementing LSP rename functionality.

Language-specific tooling for your entire tech stack

React Native development uses the same TypeScript/JavaScript infrastructure as standard React with tsserver providing type checking and completion. The critical setup detail is ensuring filetypes are correctly set—typescriptreact for tsx files and javascriptreact for jsx files—because Neovim won't apply correct LSP or Treesitter parsing otherwise. Devas↗ For debugging React Native applications in both Expo and bare workflows, use nvim-dap with Microsoft's vscode-js-debug adapter, which replaces the deprecated vscode-chrome-debug and supports modern JavaScript debugging protocols. The configuration sets up pwa-node adapters for Node.js debugging and pwa-chrome for browser attachment, allowing you to set breakpoints in Neovim and debug Metro bundler output directly.

Symfony and PHP development centers on the Intelephense versus Phpactor debate, where community consensus in 2025 heavily favors Intelephense for primary LSP duties. Intelephense provides superior autocomplete, better performance on large codebases like Symfony applications, and more stable type inference, though its premium version costs twenty-five dollars for features like variable renaming and enhanced code actions. Phpactor offers excellent refactoring capabilities and is fully open source, making the optimal approach using both—Intelephense as the primary LSP with Phpactor configured for code actions only. GitHub↗ For Symfony-specific support, configure Intelephense with Symfony stubs via php-stubs/symfony-stubs installed through Composer, and ensure composer.json is in your root directory for dependency detection. Daniele↗ Debugging PHP uses nvim-dap with vscode-php-debug adapter and Xdebug 3.x configured with mode=debug and client_port=9003.

Android native development requires eclipse.jdt.ls for Java, installed via brew or Mason and requiring Java 21+ to run while supporting projects on Java 8+. The nvim-jdtls plugin provides essential enhancements over basic lspconfig, including organize imports, extract refactorings, and generate constructors. Tamerlan +2↗ Critical configuration details include setting root_dir to detect gradlew and configuring multiple Java runtimes in settings because jdtls itself needs Java 21 but your Android projects likely use Java 11 or 17. Kotlin support comes through kotlin-language-server installed via SDKMAN or npm. The major limitation is jdtls does NOT support Kotlin files natively, so mixed Java/Kotlin Android projects require both servers running simultaneously, with cross-language navigation working imperfectly. Stack Overflow↗ For build integration, vim-android or android-nvim plugins provide Gradle wrapper support with commands like AndroidBuild and AndroidInstall, though many developers keep Android Studio available for complex debugging and emulator management.

Python development in 2025 strongly favors Pyright as the LSP, offering the fastest performance, best type checking based on Pylance, and excellent support for modern Python features including match statements and Python 3.12+ syntax. Nilsso↗ The alternative python-lsp-server provides more comprehensive features and better machine learning library support but with slower performance and higher memory usage. The critical Python challenge is virtual environment detection—LSP servers must find the correct interpreter in projects using poetry, pyenv, pdm, conda, or standard venv. The solutions include venv-selector.nvim for interactive switching, whichpy.nvim for automatic detection, or manual configuration that checks VIRTUAL_ENV environment variable and falls back to poetry env info. Stack Overflow↗ For formatting and linting, Ruff has emerged as the 2025 standard, combining extremely fast linting and formatting in a single Rust-based tool that replaces both Black and isort while maintaining compatibility with their configurations.

YAML and JSON development is transformed by SchemaStore.nvim, which provides 400+ schemas from the SchemaStore catalog with automatic detection by filename. This integration eliminates manual schema configuration, giving you validation for package.json, tsconfig.json, GitHub Actions workflows, Kubernetes manifests, Docker Compose files, and countless other formats out of the box. github↗ The configuration passes require('schemastore').json.schemas() to jsonls settings and similar for yamlls, with options to selectively enable only specific schemas or add custom schemas for proprietary formats. GitHub↗NeovimCraft↗ For formatting, prettier or prettierd via conform.nvim handles JSON while yamlfmt or prettier works for YAML files. Stack Overflow↗

Debugging and testing infrastructure you're completely missing

nvim-dap provides Debug Adapter Protocol support, enabling full debugging capabilities for JavaScript, TypeScript, PHP, Java, Kotlin, and Python directly in Neovim. Dotfyle↗ The setup complexity is high—each language requires specific adapter configuration—but mason-nvim-dap simplifies installation by automatically setting up common adapters through Mason. AstroNvim +2↗ The essential additions are nvim-dap-ui for a professional debugging interface with variables, watches, stack traces, and console output in floating windows, plus nvim-dap-virtual-text to show variable values inline next to code during debugging sessions. github↗ The combination transforms debugging from print-statement driven to proper breakpoint debugging with step-through execution.

For JavaScript and TypeScript including React Native, vscode-js-debug provides the most comprehensive debugging support with configurations for both launching Node.js programs and attaching to running processes like Metro bundler. Theosteiner↗GitHub↗ PHP debugging requires debugpy installed via pip with configuration pointing to your virtual environment's Python interpreter. Tamerlan↗ Java debugging integrates through jdtls with java-debug bundles added to init_options, while Android-specific debugging typically falls back to Android Studio for device/emulator management despite Neovim handling the code-level debugging.

neotest modernizes testing workflows with a tree view of test suites, inline status indicators, integrated output panels, and diagnostic integration that shows test failures at the exact source line. The framework supports 20+ languages through

adapters—neotest-jest for JavaScript/React, neotest-phpunit for PHP/Symfony, neotest-python for pytest, and neotest-java for Java. github↗ The critical advantage over vim-test is neotest's ability to show per-test results, run nearest test under cursor, display output in floating windows, and integrate with nvim-dap for debugging tests. The summary window provides a hierarchical view of all tests with pass/fail status, while watch mode automatically reruns tests on file changes. For someone working across multiple frameworks, this unified testing interface with consistent keybindings across languages dramatically improves test-driven development workflows.

Formatting and linting migrate to conform and nvim-lint

The null-ls.nvim plugin was archived in 2023, creating a critical migration path that every Neovim user must navigate. The modern approach splits functionality between conform.nvim for formatting and nvim-lint for linting, both actively maintained and designed as official replacements. github +4↗ conform.nvim provides lightweight yet powerful formatting with 180+ built-in formatters, preserves extmarks and folds by calculating minimal diffs, and enables range formatting for all formatters regardless of whether they natively support it. github +2↗ The configuration maps formatters to filetypes—prettier for JavaScript/TypeScript, stylua for Lua, black or ruff for Python, php-cs-fixer for PHP—with format_on_save option that falls back to LSP formatting when no formatter is configured. FFFF↗

nvim-lint provides asynchronous linting with 100+ built-in linters, integrating cleanly with Neovim's diagnostic system to show errors inline. github↗ The typical pattern attaches linters to BufWritePost and TextChanged events with appropriate debouncing to avoid performance issues. For your stack, you'd configure eslint for JavaScript/TypeScript, phpcs or phpstan for PHP, checkstyle for Java, and ruff or pylint for Python. The combination of conform.nvim plus nvim-lint provides complete formatting and linting coverage while remaining significantly more performant than the deprecated null-ls approach.

For projects using prettier, installing prettierd provides a daemon-based version that formats files 10-20x faster than regular prettier by avoiding Node.js startup costs on every invocation. Similarly, ruff in Python land handles both linting and formatting in a single Rust binary with execution times measured in single-digit milliseconds compared to Black's 50-100ms typical formatting time. These performance improvements become critical in format-on-save workflows where any delay disrupts the editing flow state.

Git integration beyond LazyGit alone

You already have LazyGit for complex git operations, but gitsigns.nvim provides essential inline git integration that LazyGit doesn't offer—git decorations in the sign column showing added, removed, and changed lines, hunks navigation with next/previous hunk commands, stage and unstage individual hunks, preview diffs in floating windows, and inline blame annotations. github↗ The performance is excellent with async operations ensuring no editor blocking, Hacker News↗ and the visual feedback of seeing exactly which lines changed since last commit becomes indispensable for tracking work-in-progress changes.

The workflow pattern combines gitsigns for local changes, LazyGit for commits and branch management, and telescope.nvim git pickers for navigation. Bahadiraydin↗ Telescope provides git_files for fuzzy finding changed files, git_commits for searching commit history, git_branches for branch switching, and git_status for reviewing all changes. Track Awesome List↗ diffview.nvim adds best-in-class diff viewing and merge conflict resolution with split views showing both sides of conflicts Dlvhdr↗Dotfyle↗ with clean choose-left, choose-right, and choose-both commands. This replaces the frustrating experience of manually editing conflict markers with a visual interface that shows context and lets you make informed decisions.

git-conflict.nvim provides simpler conflict resolution for straightforward conflicts where you just need to quickly choose one side without reviewing full context, while gitlinker.nvim adds quality-of-life features for generating GitHub or GitLab permalinks to the current line for sharing in code reviews. The recommended stack uses gitsigns always for inline visibility, LazyGit for interactive staging and commits, diffview for complex merges and diffs, and Telescope git pickers for navigation—this covers all git workflows without leaving Neovim.

Productivity plugins that fundamentally change workflows

Harpoon stands out as the single most transformative productivity plugin according to community feedback, created by ThePrimeagen and designed for marking 4-5 frequently accessed files per project for instant navigation. Unlike global

marks that save line positions, Harpoon provides project-specific file bookmarks that remember your last position in each buffer. github↗ The workflow involves marking files with leader-a as you encounter them during focused work, then using Ctrl-h/j/k/l or leader-1/2/3/4 for instant access. The power emerges when working intensively on a small set of files—marking your React Native component, its test file, the API service, and the type definitions lets you jump between them without fuzzy finding or navigating directories. ThePrimeagen's core insight is "Harpoon is a very specific and very precise instrument—you get one chance" meaning it's for files you actively work with, not a general bookmark system.

The community-recommended navigation pattern forms a triangle: Telescope for discovery when you don't know where something is, Harpoon for frequent files during focused work, and Ctrl-o/Ctrl-i jump list for navigating back through your history. GitConnected↗ This covers all navigation needs without cognitive load—you discover files once with Telescope, mark them if they become important, then navigate via Harpoon for the rest of that feature's development. Buffer management becomes simpler because you're not trying to manage 20 open buffers; instead you close buffers liberally with :bd and trust that reopening via Telescope or Harpoon takes under a second.

which-key.nvim provides discoverability for keybindings through popups that appear as you type leader sequences, showing all available completions with descriptions. GitHub +3↗ This becomes essential as your configuration grows beyond 50 keybindings because remembering every mapping is impossible. github↗ The modern v3 API uses the .add() method to define groups and bindings with descriptions, organizing commands logically— Will Code for Beer↗leader-f for find (Telescope commands), leader-g for git operations, leader-l for LSP actions, leader-b for buffers. GitHub↗ The descriptions you write during initial configuration pay dividends for months afterward when you remember "there's a command for this" but can't recall the exact keys.

Motion plugins enhance navigation within visible text, with flash.nvim emerging as the 2025 standard over older hop.nvim and even the excellent leap.nvim. flash.nvim integrates search highlighting with labels, meaning when you use / or ? to search, matching locations show labels for instant jumping. github +2↗ The plugin supports Treesitter integration for jumping to syntax nodes, multi-window operation, GitHub↗NeovimCraft↗ and remote operations where you can execute commands at distant locations. For experienced Vim users, flash.nvim reduces the keystrokes for precise navigation from multiple operations (search, next, next, next) to a single search plus one or two label characters.

Oil.nvim represents a paradigm shift in file management by making directories editable as buffers—you navigate to a directory, see files as buffer lines, and use normal Vim commands to rename (cw), delete (dd), or create files (o for new line, then write filename). This approach has dramatically lower cognitive load than learning tree view keybindings because you already know Vim editing commands. github↗Grundy↗ The community consistently describes oil.nvim as a "game changer" with sentiment like "being able to edit your directories in a buffer is revolutionary" and "no learning curve whatsoever." LibHunt↗ For bulk file operations like renaming a dozen test files to match a new naming convention, editing the directory buffer and saving is vastly faster than clicking through tree view operations.

Performance optimization and lazy loading mastery

Modern Neovim should start in under 50ms with 15-30 plugins, and under 100ms with 50+ plugins if lazy loading is configured correctly. Miblog↗ The lazy.nvim profiler accessed with :Lazy profile shows exactly which plugins take longest to load and which events trigger loading, making optimization data-driven rather than guesswork. lazy.nvim↗ The VeryLazy event loads plugins after the initial UI renders, typically 50-100ms after startup, providing the optimal balance for non-critical functionality like status line enhancements, which-key, or git integration that doesn't need to be available before you've even opened a file. Miblog↗

The most common performance bottleneck is loading everything at startup, particularly heavy plugins like nvim-cmp with all sources, Treesitter with all parsers, and telescope with all extensions loading immediately. The fixes involve event-based triggers—Treesitter loads on BufRead (when actually reading a file), completion loads on InsertEnter (when you actually start typing), and Telescope loads on first command invocation. Miblog↗ For languages, install only Treesitter parsers you actively use rather than all 100+ available parsers; each parser adds 5-15ms to initial load time when configured to eagerly install.

LSP lazy loading requires care because overly aggressive lazy loading causes servers to not attach when opening files directly from command line like nvim file.py. The safe approach uses event = { "BufReadPre", "BufNewFile" } rather than BufReadPost, ensuring the LSP server starts before buffer content loads. An alternative approach keeps LSP plugins loading at startup but defers heavy operations like format-on-save configuration or diagnostic setup to autocommands that only fire after files are open.

Disable built-in Neovim plugins you don't use by setting loaded_netrw, loaded_gzip, loaded_tar, loaded_zip, and similar global variables to 1 before any plugins load. lazy.nvim's performance.rtp.disabled_plugins option automates this with a curated list of commonly unused built-ins. LazyVim↗ Each disabled plugin saves 3-10ms from startup, and disabling 10-15 unused plugins compounds to 30-100ms savings. The measurement tool is nvim --startuptime startup.log which produces a timestamped log showing exactly which scripts load and how long each takes. Neovim↗GitHub↗

Keybinding organization prevents cognitive overload

Experienced users organize keybindings with consistent leader key prefixes that form mnemonic groups—space-f for find operations with Telescope, space-g for git commands, space-l for LSP actions, space-b for buffer operations, space-w for window management, space-t for terminal or testing. This pattern lets muscle memory develop because you know all git operations live under space-g, so space-gs probably relates to git status, space-gc to commits, space-gb to branches. The descriptions provided to which-key document these patterns, creating a self-documenting configuration.

The modern vim.keymap.set() API introduced in Neovim 0.7 defaults to noremap behavior, preventing infinite loops without explicitly setting noremap = true. The API accepts mode as first argument (n for normal, v for visual, i for insert, or a table like {"n", "v"} for multiple modes), the key sequence, the command or function to execute, and an options table with buffer for buffer-local mappings and desc for which-key descriptions. Von Heikemen↗github↗ Buffer-local keymaps in the LspAttach autocommand prevent LSP keybindings from conflicting with global mappings and ensure LSP commands only work in buffers with attached servers.

The critical mistake to avoid is mapping over useful Vim defaults without understanding what you're losing. Keys like * for searching word under cursor, # for backward search, % for matching bracket, and Ctrl-o/Ctrl-i for jump list navigation are so fundamental that remapping them causes constant friction. The safe approach prefixes custom mappings with leader or uses alternative keys like space or comma that have minimal default functionality.

Anti-patterns to avoid in 2025 configurations

The endless tinkering trap catches many users who spend more time configuring than actually coding, constantly trying new plugins, colorschemes, and keybindings without giving each configuration enough time to develop muscle memory. The solution involves forcing artificial stability—pick a base configuration, use it for a minimum of two weeks before making changes, then add one plugin per week maximum while documenting why you added it and what problem it solves. Quarterly reviews audit plugins you haven't consciously used in three months, which are candidates for removal.

Plugin bloat accumulates when you add multiple solutions to the same problem—three file explorers, two fuzzy finders, five colorschemes, multiple session managers, overlapping completion sources. The diagnostic uses :Lazy profile to identify plugins you've loaded but not actually used in days of work, indicating functionality you thought you'd want but don't actually need. The clean approach picks one solution per category based on community recommendations and your specific needs rather than installing everything mentioned in blog posts.

Configuration anti-patterns from the VimScript era persist in many configs, using vim.cmd("set number") instead of vim.opt.number = true, or vim.cmd("nnoremap ...") instead of vim.keymap.set(). The Lua APIs are more expressive, type-safe, and integrate better with modern plugins that expect Lua configuration. Another common mistake involves including options in sessionoptions, which causes vim settings to override plugin configurations when restoring sessions, breaking plugins in subtle ways that are difficult to debug. Ayman Bagabas↗

LSP configuration mistakes include not setting capabilities before server setup (causing completion to not work), forgetting to actually install language servers via Mason (resulting in "no LSP attached" errors), wrong root directory detection where the server doesn't recognize your project structure, and multiple formatters fighting over the same filetype. GitHub↗ The debugging approach uses :LspInfo to check server status, :Mason to verify installations, and :checkhealth lsp to identify configuration issues. GitHub↗

What should you add to your setup right now

Your immediate next steps start with the LSP foundation—install mason.nvim, mason-lspconfig.nvim, and nvim-lspconfig, then configure servers for tsserver, intelephense, jdtls, kotlin_language_server, pyright, yamlls, and jsonls through the handlers pattern that auto-configures all installed servers. GitHub↗ Add nvim-cmp with cmp-nvim-lsp,

LuaSnip, and friendly-snippets to get working autocompletion across all languages. GitHub +2↗ This single addition transforms your editing experience from manual typing to intelligent completion with documentation, parameter hints, and snippet expansion.

Follow with nvim-treesitter and nvim-treesitter-textobjects, enabling parsers for tsx, typescript, javascript, php, java, kotlin, python, yaml, and json. The syntax-aware text objects this enables—af for around function,]m for next method, ac for around class—provide consistent editing vocabulary across all languages in your polyglot stack, dramatically reducing the mental context switching cost when jumping between React Native TypeScript and Symfony PHP.

Add the modern formatting and linting stack with conform.nvim configured for prettier/prettierd on JavaScript/TypeScript, php-cs-fixer on PHP, google-java-format on Java/Kotlin, and ruff on Python. Pair this with nvim-lint running eslint, phpcs, checkstyle, and ruff respectively. This enables format-on-save workflows and inline linting diagnostics that catch errors before running code.

Install gitsigns.nvim for inline git status—the visual feedback of seeing which lines changed transforms how you track work in progress and stage hunks. Add harpoon for project-local file marks and which-key.nvim for keybinding discoverability. These three productivity additions combined take your workflow from basic navigation to the efficiency level experienced users operate at consistently.

Consider nvim-dap with nvim-dap-ui and language-specific adapters if debugging is regular part of your workflow—the ability to set breakpoints in React Native components, step through Symfony controllers, or debug Android activity lifecycle directly in Neovim eliminates constant context switching to browser DevTools or Android Studio debugger. Add neotest with adapters for jest, phpunit, and pytest if you practice test-driven development, as the inline test status and quick test running becomes essential for red-green-refactor workflows.

The mini.nvim suite provides lightweight alternatives for common functionality—mini.surround for vim-surround functionality, mini.comment for smart commenting, mini.pairs for auto-pairs, mini.ai for extended text objects—all in a single dependency with zero external requirements and consistent API. github +2↗ This consolidation reduces plugin count and maintenance burden while providing solid implementations of essential editing enhancements.

Recommended configuration approach for your stack

Start with kickstart.nvim or a minimal LazyVim configuration as a base rather than building entirely from scratch—these provide sensible defaults, proper lazy loading patterns, and educational comments explaining each section. Jarv +2↗ Fork the starter, read through every line, remove what you don't need, and customize from there. Khuedoan↗ This approach takes 4-6 hours initially but provides a solid foundation you understand completely rather than a mystery configuration copied from unknown sources.

Organize your configuration with a personal namespace in lua/yourname/ containing init.lua, options.lua, keymaps.lua, autocmds.lua, and plugins/ directory with one file per plugin category—lsp.lua, treesitter.lua, completion.lua, telescope.lua, git.lua. This structure scales well as you add functionality because each file has single responsibility and you know exactly where to find configurations. Use local variables everywhere to avoid global namespace pollution and prefer vim.opt/vim.keymap.set/vim.api.nvim_create_autocmd Lua APIs over vim.cmd VimScript commands. Von Heikemen↗

Set your leader key to space at the very top of init.lua before loading any plugins, as some plugins read leader during initialization and setting it afterward causes keybindings to break subtly. Medium↗DEV Community↗ Configure which-key with logical grouping—f for find, g for git, l for LSP, b for buffers, w for windows—and provide descriptions for every keymap using the desc option. This documentation pays dividends months later when you remember "there's a command for this" but can't recall the exact keys.

Use lazy.nvim's lazy = false only for essential plugins like colorschemes, setting priority = 1000 to ensure they load first. Load git plugins, Treesitter, and productivity tools on VeryLazy event. Load LSP stack on BufReadPre/BufNewFile events. Load completion on InsertEnter. Load file explorers and git interfaces on command invocation. lazy.nvim↗ This pattern should result in startup times under 50ms with 15-20 plugins or under 100ms with 30-40 plugins.

Profile regularly with :Lazy profile and nvim --startuptime startup.log to identify bottlenecks before they become problematic. Neovim↗ Run :checkhealth monthly to catch configuration issues, update plugins quarterly to get bug fixes and new features, and audit unused plugins every six months to prevent accumulation of functionality you thought you'd

want but never actually use. Back up your entire configuration to git from day one, committing each change with descriptive messages that explain what problem you were solving—this creates an audit trail for understanding why each piece exists.

Final workflow recommendations

The optimal workflow pattern for your polyglot stack combines Telescope for broad discovery across your entire project, Harpoon for 4-5 core files during focused feature work, and Ctrl-o/Ctrl-i jump list for navigating your edit history. When starting work on a feature, use Telescope find_files or git_files to discover the relevant files, mark them with Harpoon as you identify the key ones, then jump between marked files with instant keystrokes while using LSP's gd (go to definition) and Ctrl-o (jump back) for code exploration.

Git operations layer LazyGit for complex interactive operations like rebasing or resolving hairy conflicts, gitsigns for inline visibility of changes and staging individual hunks without leaving your code, and Telescope git pickers for navigation across branches, commits, and changed files. This combination means you rarely leave Neovim during development but can drop into LazyGit's powerful interface when needed.

Code navigation relies on LSP's gd for definitions, gr for references, gi for implementations, and K for hover documentation, all enhanced by Treesitter text objects that let you select, delete, or change entire functions (af/if), classes (ac/ic), or parameters with syntax awareness. The flash.nvim motion plugin accelerates navigation within visible text, using search integration to label matches for instant jumping. This multi-layered approach means the right navigation tool is always one or two keystrokes away regardless of whether you're jumping across files, within files, or within visible text.

Testing workflows with neotest provide a unified interface across JavaScript, PHP, and Python tests with consistent keybindings—leader-tt to run nearest test, leader-tf to run all tests in file, leader-ts to show test summary. The inline status indicators and diagnostic integration mean test failures appear at the exact source line, while the integrated output panel shows assertion failures and error messages. For debugging failing tests, neotest integrates with nvim-dap to set breakpoints and step through test execution, completing the test-driven development cycle entirely within Neovim.

Your configuration should aim for 15-25 plugins total for a minimal but powerful setup, or 30-40 plugins for a comprehensive IDE experience. Beyond this point, diminishing returns set in where additional plugins add complexity faster than they add value, and startup time degrades despite lazy loading. The guiding principle is "only what you actually use"—if you haven't consciously invoked a plugin's functionality in two weeks of active development, it's a candidate for removal.

The learning investment requires 2-4 weeks of daily use to become comfortable, 1-2 months to reach previous IDE productivity levels, and 6-12 months to achieve expert-level efficiency where Vim motions and plugin workflows become automatic. The compound returns on this investment accrue over years as the configuration remains relevant through Neovim updates and language changes, skills transfer across different projects and languages, and the lightweight setup works identically on local machines, remote servers, and cloud development environments. The community's consistent message is that the initial frustration period pays substantial dividends—as one user put it, "Neovim in 2025: Still my daily driver. There is something satisfying about using a tool that truly gets out of the way, that lets you think about code."