MODERN LINUX IN 2026

A COMPREHENSIVE GUIDE TO THE MODERN COMMAND-LINE ECOSYSTEM

 # ripgrep, fd, bat, eza, fzf, zoxide, btop, dust & Beyond

rg RIPGREP · fd FD · BAT · eza EZA · fzf FZF · ZOXIDE · BTOP · DUST

```
$ rg -n "TODO" src/
src/main.rs:38:    // TODO: refactor this
src/lib.rs:102:    // TODO: handle errors
src/config.rs:17: // TODO: document this
```

```
$ fd -t f -E target
Cargo.toml
src/main.rs
src/lib.rs
src/config.rs
README.md
```

```
$ bat src/main.rs
 1   fn main() {
 2       let msg = "Hello, Linux!";
 4       println!("{}", msg);
 5   }
```

```
$ zoxide query -l
216   ~/projects/modern-linux-book
98    ~/projects/dotfiles
76    ~/projects/infra
42    ~/Downloads
18    ~/projects/old-stuff
```

```
$ eza -lah --icons
drwxr-xr-x   -  howard   4.0K  src
drwxr-xr-x   -  howard   4.0K  target
-rw-r--r--   -  howard   1.1K  Cargo.toml
-rw-r--r--   -  howard    932  README.md
```

```
$ fzf
src/
src/main.rs
src/lib.rs
src/config.rs
tests/
README.md
Cargo.toml
```

```
$ btop
CPU   23%
Mem   3.2G/16G
Load  1.15 1.02 0.87
Uptime  2d 4h 37m
```

```
$ dust -d 1
1.2G  src
456M  target
128M  .git
64M   docs
32M   tests
```

WORK FASTER — Powerful tools that save time every day.

REAL-WORLD WORKFLOWS — See how the tools work together.

MODERN ALTERNATIVES — Replace old Unix utilities with ease.

INSTALL & CONFIGURE — Step-by-step guides for every tool.

MIGRATE WITH CONFIDENCE — Move from classic commands smoothly.

HOWARD LEE

Modern Linux Tooling in 2026

ripgrep, fd, bat, eza, fzf, zoxide, btop, dust & Beyond

Steve Publications

This book is available at https://leanpub.com/modernlinuxtoolingin2026

This version was published on 2026-08-28

Contents

ripgrep, fd, bat, eza, fzf, zoxide, btop, dust & Beyond

This book is a comprehensive guide to the modern Linux command-line ecosystem as it exists in 2026. It covers the tools that have largely replaced decades-old Unix utilities for everyday development and system administration work: ripgrep, fd, bat, eza, fzf, zoxide, btop, dust and their companions. Each chapter explains what a tool is, why it exists, how to install and configure it and how to use it effectively in real-world scenarios. The book demonstrates how these tools compose into powerful workflows, compares them with traditional alternatives, and provides migration guidance for readers moving from classic Unix commands. Whether you are a developer who lives in the terminal or a system administrator managing complex environments, this book will help you work faster and more intelligently on the command line.

Introduction: The New Command Line

A few years ago, searching a codebase meant typing `grep -rn --include='*.py' 'def process_' .` and waiting while grep churned through binary files, build artifacts and every hidden directory it could find. Finding a file required remembering the exact syntax of `find . -name '*.rs' -type f -not -path '*/target/*'`. Listing a directory showed you filenames in monochrome unless you had carefully configured `LS_COLORS`, and reviewing diffs meant squinting at terminal text where insertions and deletions were marked only by plus and minus signs.

Today, the same tasks look different. You type `rg 'def process_'` and get results in milliseconds because ripgrep automatically respects your `.gitignore`, skips binaries, and searches in parallel across all cores. You run `fd '*.rs'` and it finds what you need without a maze of flags. `eza --git` shows you directory contents with colors, icons and Git status indicators already visible. When you want to review a change, delta presents a syntax-highlighted diff with side-by-side viewing and within-line change detection. And when you are not sure exactly what file or directory you need, fzf lets you fuzzy-find it interactively as you type.

This is not science fiction. It is the everyday reality for developers and system administrators who have adopted the modern Linux tooling ecosystem. The tools covered in this book represent a fundamental shift in how command-line utilities are designed and built. They share common characteristics that distinguish them from the classic Unix tools they complement or replace: they are fast, often dramatically so, thanks to modern languages like Rust and Go, plus algorithms optimized for contemporary hardware; they have sensible defaults that match how people actually work rather than how machines worked in 1974; they respect modern project structures including Git workflows and common ignore patterns; and they are designed for the human eye with color, structure and visual clarity built in from the start.

Why This Book Exists

The Unix philosophy remains sound: build small tools that do one thing well and compose them together. But the specific tools that embody this philosophy have evolved. The classic utilities we learned from: `grep`, `find`, `cat`, `ls`, `cd`, `top`, `du`, `sed`: were built for a different era with different constraints and assumptions. Many of them are still excellent at what they do, especially in contexts requiring strict POSIX compliance or maximum portability across heterogeneous systems. But for interactive development work on modern machines, newer alternatives have emerged that are simply better suited to the job.

This book exists because these tools have reached a level of maturity where adopting them is no longer an experiment. ripgrep has over 50,000 stars on GitHub and powers search in Visual Studio Code. fd, bat, eza, fzf, zoxide, btop and dust each have tens of thousands of stars and are packaged by major Linux distributions. They are not fringe projects maintained by a single enthusiast; they are well-maintained, widely used tools with active communities and clear documentation.

The purpose of this book is to give you a thorough, practical understanding of these tools. It is designed as both a learning guide and a long-term reference. You will learn not just what each tool does, but why it works the way it does, when to use it versus traditional alternatives, how to configure it for your workflow, and how to combine it with other tools into productive pipelines.

What Makes Modern Tools Different

Modern command-line tools differ from their classic predecessors in several important ways that go beyond simple feature additions. Understanding these differences helps explain why the ecosystem has shifted and when each type of tool is appropriate.

Speed is the most obvious difference, but it is also the least interesting on its own. ripgrep being faster than grep matters most when you are searching large codebases repeatedly throughout the day; the cumulative time savings compound into real productivity gains. The speed advantage comes from architectural choices: memory-mapped I/O, SIMD-optimized pattern matching, lock-free parallelism and algorithms that avoid backtracking

on complex regular expressions. fd uses similar techniques for filesystem traversal. These are not marginal improvements; benchmarks consistently show order-of-magnitude differences in common scenarios.

Better defaults matter more to daily experience than raw speed. Classic Unix tools were designed for scripting and composability first, which meant their interactive behavior often required flags to be tolerable. `grep` does not search recursively by default on all systems. `find` requires complex syntax even for simple searches. `cat` produces unadorned output that can be difficult to read. Modern tools flip this: ripgrep searches recursively and respects `.gitignore` out of the box. fd uses intuitive glob-style patterns instead of regex for filenames. bat adds syntax highlighting, line numbers and paging automatically. These defaults match how people actually use these tools interactively while preserving scripting capability through explicit flags.

Integration with modern development practices is another distinguishing factor. Git has become ubiquitous, yet classic tools treat repositories no differently than any other directory tree. ripgrep respects `.gitignore` patterns by default. fd does the same and can filter by Git-tracked status. bat integrates with Git to show modification markers alongside file contents. eza displays Git status indicators directly in directory listings. zoxide learns which directories you use most frequently, often within Git repositories. These integrations are not cosmetic; they reduce cognitive load by filtering noise and surfacing relevant information automatically.

Visual design has received attention that classic tools never got. This does not mean excessive decoration or gamification; it means using color, spacing and structure to make output easier to scan and understand. bat uses syntax highlighting themes familiar from editors. eza colors file types, permissions and sizes differently. btop presents system metrics with graphs and structured panels rather than scrolling text. dust shows disk usage as a tree with proportional bar charts instead of raw numbers. These visual improvements are grounded in how humans actually read terminal output: we scan for patterns, we look for outliers, we need context.

The Rust Factor

A significant number of modern CLI tools are written in Rust. This is not a coincidence. Rust offers memory safety without garbage collection, excellent

performance comparable to C and C++, strong concurrency primitives, and a package ecosystem that makes building cross-platform binaries straightforward. For CLI tools, these characteristics translate into programs that are fast, reliable and easy to distribute as single static binaries with no runtime dependencies.

The Rust CLI ecosystem has matured substantially. Crates like `clap` for argument parsing, `regex` for pattern matching, `ignore` for `.gitignore`-aware traversal, and `dirs` for cross-platform path handling have reduced boilerplate and encouraged consistency across tools. Many of the tools covered in this book share underlying dependencies: ripgrep and fd both use the `ignore` crate for Git-aware filtering, for example, which creates a degree of ecosystem coherence.

That said, Rust is not the only language powering modern tooling. fzf is written in Go, which provides similar benefits of fast compilation and single-binary distribution. btop is written in C++, leveraging performance-critical code where it matters. The choice of language matters less than the design philosophy: these tools prioritize correctness, performance, and user experience in ways that reflect modern software engineering practices rather than historical constraints.

What You Will Learn

This book is organized to take you from foundational understanding through advanced usage and real-world workflows. Here is what each section covers:

The first part of the book examines each core tool individually. Chapters on ripgrep, fd, bat, eza, fzf, zoxide, btop and dust provide comprehensive coverage of installation, configuration, syntax, options and practical usage patterns. Each chapter includes realistic examples drawn from development and system administration scenarios, not isolated toy commands.

The second part focuses on composition and integration. You will learn how to combine these tools into powerful workflows: piping fd output into ripgrep for fast file discovery and content search, using fzf with bat previews for interactive file browsing, integrating zoxide with fzf for smart directory navigation, and building monitoring pipelines with btop and dust. A dedicated chapter covers shell integration across Bash, Zsh and Fish, including aliases, functions, completion scripts and configuration best practices.

The third part broadens the scope to additional tools that complement the core ecosystem, such as sd as a sed replacement, delta for Git diffs, yq for YAML processing, skim as an fzf alternative and others, and provides a comparison matrix to help you choose the right tool for each task. A migration guide helps readers transition from classic Unix utilities without breaking existing scripts or workflows.

The final chapters address cross-cutting concerns: performance characteristics and tuning, security considerations when using these tools with untrusted input, scripting best practices for production environments, troubleshooting common issues and maintaining your toolchain as projects evolve.

How to Use This Book

You can read this book linearly from start to finish if you prefer a structured learning path. The chapters build on each other logically, starting with individual tools and progressing to workflows and advanced topics. However, the book is also designed as a reference you can dip into selectively. If you already know ripgrep well but want to understand fzf integration more deeply, jump to the relevant chapter. If you need quick command examples for bat configuration, search for that section directly.

Code examples throughout the book are intended to be copy-paste-ready with realistic file structures and scenarios. When a command depends on specific context, such as being inside a Git repository or having certain files present, this is explained before the example. Shell-specific syntax is clearly identified; Bash, Zsh and Fish behaviors are distinguished where they differ.

Version information is current as of 2026, but tools in this ecosystem evolve quickly. Major version changes that affect commands or configuration formats are noted where relevant. When behavior varies across versions, the range is specified. For the most up-to-date information on any tool, consult its official documentation linked in the references.

A Note on Terminology

Throughout this book, “modern tools” refers to the newer generation of CLI utilities discussed here: ripgrep, fd, bat, and so on. “Classic” or “traditional”

tools refers to the classic Unix utilities they complement or replace: grep, find, cat, ls, cd, top, du, sed, and related commands from GNU coreutils and other established packages. Neither term is meant as a value judgment beyond indicating relative age and design philosophy. Both families of tools have their place, and this book explains when each is appropriate.

When referring to specific distributions, I use shorthand: "Debian/Ubuntu" for Debian-based systems using apt, "Fedora/RHEL" for Red Hat-family systems using dnf or yum, "Arch" for Arch Linux and derivatives using pacman, and "openSUSE" for SUSE-based systems using zypper. Installation instructions note where behavior differs between distributions.

The Journey Ahead

The command line is one of the most powerful interfaces available to developers and system administrators. It provides direct access to system resources, unparalleled flexibility for automation, and a level of control that graphical interfaces cannot match. But power without efficiency is wasted potential. The tools covered in this book exist to make that interface faster, clearer, and more pleasant to use every day.

By the time you finish this book, you will understand not just how to use these tools individually, but how they fit together into a cohesive workflow. You will know when to reach for ripgrep instead of grep, when fd is sufficient versus when find is necessary, when bat adds value versus when cat is more appropriate, and how to compose these tools into pipelines that save you time on repetitive tasks. More importantly, you will understand the design principles behind modern CLI tooling so you can evaluate new tools as they emerge and make informed choices about what fits your workflow.

Let us begin.

Chapter 1: ripgrep: Search at Lightning Speed

ripgrep, invoked as `rg`, is a recursive command-line search tool that has become the default choice for text searching on modern development machines. Written in Rust by Andrew Gallant and first released in 2016, it combines the usability of tools like The Silver Searcher with raw performance that consistently exceeds GNU grep. As of 2026, ripgrep has over 50,000 stars on GitHub and powers built-in search in Visual Studio Code, demonstrating its influence extends far beyond the command line.

This chapter covers ripgrep comprehensively: what problems it solves compared to grep, how to install and configure it, its syntax and options, advanced features like file-type filtering and JSON output, real-world search workflows, and when you might still prefer traditional grep.

Why ripgrep Replaces grep: Performance and Usability Gains

To understand why ripgrep has displaced grep for many users, consider a typical scenario. You are working in a large Python project with 50,000 files totaling several gigabytes, including build artifacts, virtual environments, compiled caches and test data. You need to find all occurrences of a function definition named `process_request`.

With traditional grep, the command looks like this:

```
grep -rn --include='*.py' 'def process_request' .
```

This works, but it has several problems. First, grep will still descend into directories you probably do not want to search: `.git`, `node_modules`, `__pycache__`, build output folders and so on. You can add exclusions with `--exclude-dir`, but remembering all the relevant patterns is tedious. Second,

grep may attempt to search binary files unless you explicitly disable this behavior, producing garbled output or errors. Third, on large codebases, the search takes noticeable time: seconds or even tens of seconds depending on the size and your storage hardware.

With ripgrep, the same task requires only:

```
rg 'def process_request'
```

That is it. ripgrep searches recursively by default, respects `.gitignore` patterns automatically, skips hidden files and directories, avoids binary files, and filters by file type when you ask it to. The search completes in a fraction of the time grep takes on the same codebase.

The performance difference is not marginal. Benchmarks conducted on real-world codebases show ripgrep completing searches 5 to 40 times faster than grep depending on the scenario. On the Linux kernel source tree, which contains approximately 75,000 files totaling around 900 megabytes, a typical search completes in roughly 0.08 seconds with ripgrep versus about 0.67 seconds with GNU grep: an 8x improvement. Against older tools like ack or The Silver Searcher (ag), the gap widens further: ack took over 3 seconds on the same kernel search, making ripgrep nearly 40 times faster.

These numbers come from Andrew Gallant's original benchmarking and have been validated by independent tests using hyperfine on modern hardware with NVMe storage. The speed advantage stems from several architectural choices that distinguish ripgrep from grep:

1. SIMD-accelerated scanning: ripgrep uses CPU vector instructions to scan multiple bytes simultaneously when looking for pattern matches, dramatically reducing the number of comparisons needed.
2. Parallel traversal: ripgrep searches multiple files concurrently using all available CPU cores, whereas traditional grep processes files sequentially.
3. Memory-mapped I/O: ripgrep maps files directly into memory rather than reading them through traditional system calls, reducing overhead.
4. Backtrack-free regex engine: ripgrep uses the finite automata approach from Rust's `regex` crate, which compiles patterns into deterministic machines that guarantee linear-time matching without catastrophic backtracking.

5. Smart skipping: By respecting `.gitignore`, binary detection via NUL bytes and configurable exclusions, ripgrep spends its time only on files likely to contain relevant results.

Beyond raw speed, the usability differences are significant. grep's recursive flag `-r` has historically behaved inconsistently across platforms: GNU grep treats it as recursive search, while some BSD variants differ. ripgrep is always recursive when given a directory argument, removing ambiguity. grep requires explicit flags for most features: `-i` for case-insensitive matching, `-n` for line numbers, `-H` to show filenames and so on. ripgrep provides sensible defaults: it shows filenames automatically when searching multiple files, displays line numbers by default in recursive mode, and handles case sensitivity intelligently with its `--smart-case` option, which treats patterns as case-insensitive only when they contain no uppercase letters.

Installation Across Distributions and Platforms

ripgrep is widely packaged across Linux distributions and available through multiple installation methods. Choose the approach that best fits your system and preferences.

On Debian-based systems including Ubuntu 20.04 and later:

```
sudo apt install ripgrep
```

The package name is `ripgrep` and the executable is `rg`. This has been available in official repositories since Ubuntu 20.04 and Debian 11 (Bullseye).

On Fedora, RHEL, Rocky Linux and AlmaLinux:

```
sudo dnf install ripgrep
```

ripgrep has been in Fedora's official repositories since Fedora 28. For RHEL 8 and derivatives, it is available through the EPEL repository after enabling it with `sudo dnf install epel-release`.

On Arch Linux and derivatives like Manjaro:

```
sudo pacman -S ripgrep
```

On openSUSE:

```
sudo zypper install ripgrep
```

On Alpine Linux:

```
apk add ripgrep
```

For macOS users with Homebrew:

```
brew install ripgrep
```

Windows users have several options. Using winget (built into Windows 10 and 11):

```
winget install BurntSushi.ripgrep.MSVC
```

With Chocolatey:

```
choco install ripgrep
```

With Scoop:

```
scoop install ripgrep
```

If your distribution does not package ripgrep, or you need the latest version before it reaches official repositories, you can install from source using Rust's Cargo package manager. This requires Rust 1.72 or newer:

```
cargo install ripgrep
```

Alternatively, precompiled binaries are available for every release on GitHub at https://github.com/BurntSushi/ripgrep/releases. Download the appropriate tarball for your platform and architecture, extract it, and place the `rg` binary in a directory on your PATH such as `~/.local/bin`.

After installation, verify everything works:

```
rg --version
```

This should display the version number, build information, and supported features. As of 2026, the latest stable release is in the 15.x series.

Shell completion scripts are included with ripgrep for Bash, Zsh, Fish and PowerShell. Installation via package managers typically places these in standard locations where your shell can find them automatically. If installed manually or via Cargo, you may need to source the completion script explicitly. For Bash, add this to your `~/.bashrc`:

```
eval "$(rg --bash-completion)"
```

For Zsh, add to `~/.zshrc`:

```
autoload -U compinit && compinit
# ripgrep completion is typically auto-discovered after installation
```

For Fish:

```
rg --fish-completion | source
# Or save permanently:
rg --fish-completion > ~/.config/fish/completions/rg.fish
```

Core Syntax and Essential Flags

ripgrep's basic syntax mirrors grep's but with better defaults. The fundamental form is:

```
rg PATTERN [PATH...]
```

PATTERN is a regular expression using Rust's regex syntax, which is similar to PCRE (Perl Compatible Regular Expressions) and JavaScript regex. PATH is optional; when omitted, ripgrep searches the current directory recursively. When provided, it can be one or more files or directories to search.

Here are the most essential flags you will use regularly:

- `-i` or `--ignore-case`: Perform case-insensitive matching. Without this flag, matching is case-sensitive by default.
- `-S` or `--smart-case`: Case-insensitive unless the pattern contains uppercase letters. This is often preferable to always-on case-insensitive search because searching for `error` matches `Error`, `ERROR`, and `error`, while searching for `ERROR` matches only exact uppercase occurrences. Many users set this as a default in their configuration file.
- `-F` or `--fixed-strings`: Treat the pattern as a literal string rather than a regular expression. This is faster for simple substring searches and avoids issues with special characters that have meaning in regex syntax.
- `-w` or `--word-regexp`: Match only whole words. The pattern must be bounded by non-word characters on both sides. Searching for `foo` with this flag matches `foo` but not `foobar` or `rafoo`.
- `-n` or `--line-number`: Show line numbers in output. This is enabled by default in recursive searches, so you rarely need to specify it explicitly.
- `-c` or `--count`: Instead of showing matching lines, show the count of matches per file. Useful for quickly understanding how widespread a pattern is across a codebase.
- `--files`: List all searchable files without performing a search. This is useful for seeing what ripgrep considers searchable given its current ignore rules.
- `-H` or `--with-filename`: Always show filenames in output, even when searching a single file. The default behavior omits filenames for single-file searches to reduce noise.
- `-l` or `--files-with-matches`: Print only the names of files containing matches, one per line. This is useful when you want to know which files contain a pattern without seeing every match.
- `-L` or `--files-without-match`: The inverse of `-l`; prints filenames that do not contain the pattern.

Let us walk through some practical examples using realistic scenarios.

Suppose you are working in a web application repository and need to find all TypeScript files that import a utility function called `formatCurrency`. You would run:

```
rg 'import.*formatCurrency' -t ts -t tsx
```

The `-t` flag specifies file types; here we search only TypeScript (`.ts`) and TSX (`.tsx`) files. ripgrep knows common file type associations, so you do not need to specify glob patterns manually. Run `rg --type-list` to see all recognized types.

If you want case-insensitive matching because you are not certain whether the import uses uppercase or lowercase:

```
rg -i 'import.*formatCurrency' -t ts -t tsx
```

To find all files that define a React component named `UserProfile`, searching for the typical class or function declaration patterns:

```
rg '(class|function)\s+UserProfile' -t tsx -l
```

The `-l` flag gives you just the filenames, making it easy to open them in your editor. You can chain this with other commands:

```
code $(rg '(class|function)\s+UserProfile' -t tsx -l)
```

This opens all matching files in Visual Studio Code (assuming `code` is on your PATH).

For searching logs, context lines are often more useful than isolated matches. The `-C` flag shows surrounding context:

```
rg 'ERROR.*timeout' app.log -C 3
```

This displays three lines before and after each match, helping you understand what led to the error. Use `-B` for only preceding lines and `-A` for only following lines.

When you need column numbers along with line numbers: useful for pinpointing exact locations in long lines:

```
rg 'TODO' --column
```

Output looks like `src/main.rs:42:15: // TODO: refactor this function`, where 42 is the line number and 15 is the column.

Advanced Pattern Matching and Context Control

ripgrep supports the full regex syntax from Rust's `regex` crate, which provides a powerful but well-behaved pattern language. Unlike some regex engines, it guarantees linear-time matching by avoiding backtracking entirely. This means your searches will not hang on pathological patterns, even with complex expressions.

Key regex features supported include:

- Character classes: `[a-z]`, `[0-9]`, `\w` (word characters), `\d` (digits), `\s` (whitespace), and their negated forms `[^a-z]`, `\W`, `\D`, `\S`.
- Quantifiers: `*` (zero or more), `+` (one or more), `?` (zero or one), `{n}`, `{n,}`, `{n,m}` for exact ranges.
- Alternation: `(foo|bar)` matches either `foo` or `bar`.
- Anchors: `^` for line start, `$` for line end. Note that ripgrep processes input line by line by default, so these anchor to individual lines. Use `-U` or `--multiline` for true multiline matching where `^` and `$` match the start and end of the entire input.
- Groups and backreferences: `(pattern)` captures text, `\1`, `\2`, etc., refer back to captured groups within the same pattern.
- Lookahead and lookbehind: `(?=pattern)` for positive lookahead, `(?!pattern)` for negative lookahead, `(?<=pattern)` for positive lookbehind, `(?<!pattern)` for negative lookbehind. These allow context-aware matching without consuming characters.

Here are practical examples demonstrating advanced patterns:

Finding Python function definitions that take more than three arguments:

```
rg 'def \w+\([^)]*,[^)]*,[^)]*,[^)]*' -t py
```

This regex matches `def` followed by a function name and parentheses containing at least four comma-separated items.

Searching for TODO comments with an associated ticket or issue number:

```
rg 'TODO.*#[0-9]{4,}'
```

Matches patterns like `TODO #1234 fix this bug`.

Finding unused imports in JavaScript files by looking for imports followed immediately by another import (a heuristic that often indicates the first import is unused):

```
rg -U 'import .+ from .+;\s*import' -t js -t ts
```

The `-U` flag enables multiline mode so the pattern can span across lines.

For context control beyond simple line counts, ripgrep provides several options:

- `--context N` or `-C N`: Show N lines before and after each match.
- `--before-context N` or `-B N`: Show only preceding context lines.
- `--after-context N` or `-A N`: Show only following context lines.
- `--heading`: Group results by file with a header showing the filename, rather than prefixing every line with the filename. This produces cleaner output when reviewing matches within individual files.
- `--no-heading`: Force inline filenames even in recursive mode.

The `--context` flag combined with `--heading` is particularly useful for code review:

```
rg 'FIXME' --heading -C 5
```

This groups all FIXME comments by file and shows five lines of context around each, making it easy to assess what needs attention in each location.

For very long matching lines that would be difficult to read, ripgrep can truncate output while still indicating truncation occurred:

```
rg 'pattern' --max-columns=120
```

Lines longer than 120 characters are truncated in the display, with an ellipsis indicating content was omitted. The `--max-columns-preview` flag provides a similar effect but only for preview purposes; the full line is still available if you need it.

ripgrep also supports searching within compressed files without manual decompression:

```
rg 'error' *.gz *.bz2 *.xz *.zip
```

The `-z` or `--search-zip` flag enables this explicitly, though ripgrep often detects compressed files automatically based on extension. This is invaluable for searching archived logs or packaged data.

Configuration via ripgreprc and Environment Variables

ripgrep supports configuration through a dedicated configuration file, allowing you to set default options without typing them repeatedly or creating complex aliases. Unlike some tools that look for configuration in standard locations automatically, ripgrep uses the `RIPGREP_CONFIG_PATH` environment variable to locate its config file. This explicit approach avoids conflicts and makes configuration portable across systems.

Create a configuration file at a location of your choice, such as `~/.ripgreprc` (conventional but not required), and set the environment variable:

```
# In ~/.bashrc or ~/.zshrc:
export RIPGREP_CONFIG_PATH=~/.ripgreprc
```

The configuration file format is simple: one flag per line, with optional values following an equals sign. Lines starting with # are comments. Here is a practical example configuration for a developer working across multiple projects:

```
# ~/.ripgreprc

# Smart case handling: lowercase patterns match case-insensitively,
# uppercase patterns are case-sensitive
--smart-case

# Show column numbers alongside line numbers
--column

# Include hidden files in searches (useful for dotfiles and config)
--hidden

# Never search these directories regardless of .gitignore
--glob=!.git/
```

```
--glob=!node_modules/
--glob=!dist/
--glob=!build/
--glob=!coverage/
--glob=!target/
--glob=!venv/
--glob=!.venv/

# Truncate very long lines for readability
--max-columns=150
--max-columns-preview

# Custom colors: yellow matches with bold styling
--colors=match:fg:yellow
--colors=match:style:bold
```

With this configuration, every `rg` invocation inherits these settings. Command-line flags always override config file settings, so you can temporarily disable a default by specifying the negated flag:

```
rg --no-hidden 'pattern'    # Search without hidden files despite config
rg --no-config 'pattern'    # Ignore config file entirely for this search
```

The `--debug` flag is useful for understanding what ripgrep is doing with your configuration:

```
rg --debug 'test' . | head -20
```

This prints detailed information about loaded configuration files, applied ignore rules, threading behavior and more. It is invaluable for troubleshooting unexpected search behavior.

ripgrep also respects several environment variables beyond the config path:

- `RIPGREP_CONFIG_PATH`: Path to the configuration file. When unset, no config file is loaded.
- `RIPGREP_MAX_FILE_SIZE`: Maximum file size to search, defaulting to a very large value. Useful for skipping extremely large files like disk images or database dumps. Format accepts human-readable sizes: `10M`, `1G`, etc.
- `RIPGREP_THREADS`: Number of threads to use. Defaults to the number of CPU cores. Set lower if ripgrep is consuming too much CPU during searches.

- `RIPGREP_LOG`: Enable debug logging when set to a non-empty value. Combined with `--debug`, provides detailed diagnostic output.

For project-specific configuration, you can change `RIPGREP_CONFIG_PATH` within a particular directory or use shell functions:

```
# In ~/.zshrc:
project_search() {
    export RIPGREP_CONFIG_PATH="$HOME/projects/myapp/.ripgreprc"
    rg "$@"
}
```

This allows different teams or projects to maintain their own search preferences without affecting global configuration.

Real-World Search Workflows for Codebases and Logs

The true power of ripgrep emerges when you integrate it into daily workflows. Here are practical, realistic scenarios that demonstrate how experienced users leverage ripgrep beyond basic pattern matching.

Finding all environment variable references in a codebase:

```
rg '\b(ENV|env|process\.env)\.' -t js -t ts -t py -t rb
```

This searches for common patterns used to access environment variables across JavaScript, TypeScript, Python, and Ruby files. The `\b` word boundary ensures you match `ENV.` but not `RENEWAL.` or similar false positives.

Locating deprecated API usage before upgrading a library:

```
rg 'deprecated' -i --heading -C 2
```

Searches for deprecation warnings, notices in comments, or documentation markers that indicate code needing updates. The context shows what is deprecated and where it appears.

Auditing hardcoded credentials or secrets (a security-critical task):

```
rg '(password|secret|api_key|token)\s*[:=]\s*["\x27][^"\x27]+["\x27]' -i
↪  --type-add 'src:*.{py,js,ts,yaml,yml,json}' -t src -l
```

This pattern looks for common credential assignment patterns. Combined with `-l`, it lists files containing potential issues without overwhelming output. Note this is a heuristic; comprehensive secret scanning requires dedicated tools like truffleHog or gitleaks, but ripgrep provides quick initial detection.

Searching across multiple repositories simultaneously:

```
rg 'authentication' ~/repos/project-a ~/repos/project-b ~/repos/shared-lib -l
```

ripgrep handles multiple root directories naturally, searching each independently and presenting combined results. This is useful when refactoring shared logic or tracking patterns across a monorepo structure.

Analyzing log files for error patterns over time:

```
rg 'ERROR' /var/log/app/*.log --stats
```

The `--stats` flag provides summary information including total files searched, bytes processed, matches found, and elapsed time. This gives you quantitative data about error frequency across your logs.

For real-time log monitoring similar to `tail -f`:

```
rg 'ERROR|WARN' /var/log/app/current.log --follow
```

The `--follow` flag makes ripgrep continue watching the file as new content is appended, filtering output in real time for matching patterns. This combines the functionality of `tail -f` and `grep` into a single command.

Building a search-and-open workflow with fzf (covered in detail in Chapter 5):

```
rg --files | fzf --preview 'bat --color=always --style=numbers {}'
```

This lists all searchable files, lets you fuzzy-find one interactively, and previews its contents with bat as you navigate. Press Enter to select a file, which outputs the filename for further processing: often piped into an editor:

```
code $(rg --files | fzf --preview 'bat --color=always --style=numbers {}')
```

Searching for code smells and patterns that indicate technical debt:

```
# Find extremely long functions (heuristic: many closing braces suggest depth)
rg '^\s*}\s*$' -t py | awk -F: '{print $1}' | sort | uniq -c | sort -rn | head
↪  -20

# Find copy-pasted code blocks (search for unusually long identical strings)
rg -o '".{50,}"' -t js -t ts | sort | uniq -c | sort -rn | head -20
```

These are more advanced compositions that combine ripgrep with other Unix tools for analysis. The first counts lines containing only closing braces per file as a rough proxy for function complexity in Python. The second finds duplicated string literals longer than 50 characters, which often indicate copy-pasted code that should be refactored into constants.

When to Stick with grep: POSIX Compatibility and Edge Cases

Despite ripgrep's advantages, there are scenarios where traditional grep remains the better choice. Understanding these helps you make informed decisions rather than blindly replacing every grep invocation.

POSIX compliance matters for portable scripts. If you write shell scripts intended to run on arbitrary Unix-like systems: embedded devices, minimal containers, BSD variants, or legacy servers: relying on ripgrep introduces a dependency that may not be available. POSIX specifies grep's behavior precisely; ripgrep is not POSIX-compliant and may behave differently in edge cases. For scripts where portability is paramount:

```
#!/bin/sh
# Portable search using only POSIX tools
if grep -q "pattern" "$file"; then
    echo "Found match"
fi
```

This works everywhere. The equivalent with ripgrep would fail on systems without `rg` installed.

GNU grep offers features that ripgrep does not fully replicate. The `-P` flag for Perl-Compatible Regular Expressions provides advanced capabilities like variable-length lookbehind assertions and recursive patterns that ripgrep's regex engine does not support. If your search patterns rely heavily on PCRE-specific features, grep may be necessary:

```
grep -P '(?<=\bword\b).{0,10}(?=pattern)' file.txt
```

ripgrep supports lookahead but has limitations with certain complex back-reference and lookbehind patterns that PCRE handles.

In-place editing is another area where grep's ecosystem partner sed excels and ripgrep does not compete. ripgrep's `--replace` flag modifies output only; it never changes files on disk. For search-and-replace operations within files, you still need sed or a dedicated replacement tool like sd (covered in Chapter 11):

```
sed -i 's/old/ new/g' file.txt    # grep/sed ecosystem handles this
rg --replace 'new' 'old' file.txt  # This only prints output, does not edit
```

ripgrep is also less suitable for certain streaming scenarios. When processing input from a pipe where you need exact line-by-line behavior matching POSIX specifications, or when integrating with tools that expect grep's specific exit codes and signal handling, traditional grep provides predictable behavior across environments.

Finally, there is the argument of familiarity and muscle memory. If you have used grep for decades and your existing scripts and workflows are built around it, there is no obligation to switch solely because a newer tool exists. ripgrep excels in interactive development work on modern machines; for scripted, portable, or legacy contexts, grep continues to serve its purpose reliably.

A pragmatic approach that many experienced users adopt: use ripgrep for interactive searching during development, and keep grep available for scripting and compatibility scenarios. They coexist peacefully on the same system without conflict.

Chapter 2: fd: Finding Files Without the Pain

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

The Problems with find That fd Solves

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Installation and Quick Start

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Core Syntax: Patterns, Paths, and Filters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Advanced Filtering: Type, Size, Time, Permissions, and More

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Execution Actions and the exec Flag

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Configuration Files and Environment Variables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Real-World File Discovery Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

When find Is Still Necessary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Chapter 3: bat: The Superior cat Alternative

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Beyond cat: What bat Adds to File Viewing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Installation and Syntax Theme Setup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Core Features: Highlighting, Line Numbers, and Paging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Git Integration and Diff Display

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Language Detection and Custom Rules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Configuration via bat-config and Environment Variables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Practical Usage Patterns for Development and Sysadmin Work

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Chapter 4: eza: A Modern ls That Actually Makes Sense

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Why ls Is Hard to Use and How eza Fixes It

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Installation Across Platforms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Core Options: Colors, Icons, Git Status, and Tree View

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Output Formats: Long, Grid, One-Line, and JSON

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Configuration Files and Environment Variables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Realistic Directory Browsing Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Portability Considerations and ls Compatibility

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Chapter 5: fzf: Fuzzy Finding Everything

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

What Fuzzy Finding Is and Why It Transforms CLI Navigation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Installation and Core Concepts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Basic Usage: Interactive Selection from Any Input Stream

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Advanced Options: Preview Windows, Multi-Select, and Keybindings

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Shell Integration for Bash, Zsh and Fish

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Composing fzf with fd, rg, bat, and Other Tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Custom Scripts and Automation Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Chapter 6: zoxide: Intelligent Directory Navigation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

The Problem with cd and How zoxide Solves It

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Installation and Shell Integration Setup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Core Commands: Jump, Add, Query, and List

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Configuration Options and Environment Variables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Combining zoxide with fzf for Interactive Navigation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Practical Navigation Patterns for Large Projects

Chapter 7: btop: Next-Generation System Monitoring

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Beyond top and htop: What btop Brings to Monitoring

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Installation and Dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Interface Layout and Navigation Basics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Process Management and Interactive Controls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Customizing Themes, Layouts, and Display Options

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Real-World Monitoring Scenarios

Chapter 8: dust: Visualize Disk Usage Instantly

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Why du Is Hard to Read and How dust Improves It

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Installation and Quick Start

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Core Options: Depth, Sorting, Filtering, and Output Formats

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Visual Tree Display and Interactive Mode

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Configuration via Environment Variables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Realistic Disk Cleanup and Analysis Workflows

Chapter 9: Workflow Compositions: Combining Tools for Maximum Productivity

The Philosophy of Tool Composition on the CLI

fd plus rg: Fast File Discovery and Content Search

bat plus fzf: Interactive File Browsing with Preview

zoxide plus fzf: Smart Directory Jumping with Selection

Monitoring Workflows: btop, dust, and System Diagnostics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Building Custom Pipeline Scripts for Repetitive Tasks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Chapter 10: Shell Integration Deep Dive: Bash, Zsh and Fish

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Aliases, Functions, and Wrapper Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Bash-Specific Integration Techniques

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Zsh-Specific Features and Plugins

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Fish-Specific Configuration Approaches

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Shared Configuration Strategies for Multiple Shells

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Chapter 11: Beyond the Core Eight: Other Essential Modern Tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

jq and yq for JSON and YAML Processing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

sd as a sed Replacement

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

xargs Alternatives: px and GNU parallel

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Modern Text Processing: skim, delta, and More

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Choosing the Right Tool: Comparison Matrix

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Chapter 12: Migration Guide: From Classic Unix to Modern Tooling

Mindset Shifts When Adopting Modern Tools

Command Translation Reference: grep to rg, find to fd, ls to eza, etc.

Gradual Migration Strategies Without Breaking Existing Scripts

Handling POSIX Compatibility Requirements

Common Pitfalls and How to Avoid Them

Chapter 13: Performance, Security, and Best Practices

Performance Characteristics of Modern Tools vs Traditional Ones

Security Considerations: Untrusted Input, Scripts, and Network Access

Scripting with Modern Tools: Portability and Robustness

Troubleshooting Common Issues

Maintenance and Keeping Up with Updates

Conclusion: The Evolving Command Line

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

What We Have Learned: Core Principles of Modern CLI Tooling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Where the Ecosystem Is Headed

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Building Your Own Modern Workflow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

Final Thoughts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernlinuxtoolingin2026.