Frequently Answered Questions

Robert S. Muhlestein (rwxrob)

2025-01-01 13:48:34: Perpetually published

Table of Contents

All the questions I'm tired of answering

https://linktr.ee/rwxrob	Download PDF	Download EPUB

Copyright

What is the best tool for the job?

Almost every serious question wanting my opinion on something could have been rephrased into the question: What is the best tool for the job—right now? I've had to add that "right now" part because this list changes from year to year, sometimes even month to month. In tech the only thing constant is that nothing is constant.

It might help to know my preferences:

- Most likely to be required working for a large corporation
- Used and recommended by *actual* professionals (not "influencers")
- Popular because of successful, wide-adoption
- Mature, stable, and easy to use
- As open sourced as is practical (*never* GPLv3)

Operating systems

Job	OS	Why
Development	macOS (Unix)	Largest Unix distro in the world, compatibility, speed
Sound Engineering	macOS (Unix)	Preferred by all DJs, musicians, sound techies
Business	macOS (Unix)	Easiest to use, safest, best-in-class video chat, longest battery life
Gaming	Windows	PC master race, nothing beats Windows for gaming
3D modeling	Windows	SolidWorks only runs on Windows, DirectX standard
Hacking	Windows + WSL2	Required for OSEE, can run all others as VMs
Servers	RedHat Linux	Industry standard
Containers	Ubuntu Linux	Industry standard
Routers	BSD	Industry standard

Job	**Tool**	**Why**
Server VMs	KVM/QEMU	Gold standard
Distro VMs	VMWare	Supported, stable, not Oracle, required by OSEE
Container OS	Ubuntu Linux	Ubiquity, flexibility, support, adoptions
Server OS	RedHat Linux	Ubiquity, market leader
Text editing	`vim`	Ubiquity, Unix filters while editing, muscle-memory

Job	Tool	Why
Development	`nvim`	Dynamic configuration, extensibility, sustainability, cognitive compatibility with `vim`
Command history	`set -o vi`	Searchable history, minimal cognitive switching
Text processing	`perl`	Gold standard for regex, `libpcre`
Terminal	WezTerm	GPU accelerated, configurable, full support
Cut/paste	`screen`/`tmux`	Works between heterogeneous sessionts
Sessions	`screen`/`tmux`	Ubiquity, configurability
Windows/Panes	`screen`/`tmux`	Gold standard for multiplexing
Sharing	`screen`/`tmux`	Allows any two users to share terminal/keyboard
File finder	`find`	Ubiquity, flexibility, extensibility, universal
Containers	`podman`	Not deprecated like Docker company

Best language

Job	Language	Why
Interactive shell	Bash	Ubiquitous, power, self-completion, safest
Shell scripting	Shell	Ubiquitous, compatibility, best license
Windows scripting	PowerShell	Mandatory, piped objects/structs
Text processing	Perl	Fastest to write, dense but contextual and powerful
Machine learning	Python	Pytorch, Tensorflow, NumPy (actually C underneath)
Automation	Python	Ansible, BeautifulSoup, fast to write
Terminal apps	Go	Cobra, Bonzai, Charmbracelet, embedded resources, batteries included
Middleware	Go	Cloud-native standard, Kubernetes, podman, helm, Gin
Multicall monoliths	Go	Bonzai, fastest cross-compilation, easy to write
Embedding resources	Go	Self-contained binaries with embedded file system
Electron alternative	Go	Serve embedded resources to local web browser
Performant terminals	Rust	Fastest execution speed, safer and easier than C
Dynamic configuration	Lua	Ubiquity, embeddable, easy to write and read

Best structured data

Structured data	YAML	Industry standard, readable, maintainable, complex, `yq`
Data endpoints	JSON	Industry standard, safe transfer, complex, `jq`
Data cache	Properties	Fastest/simplest parsing while also human readable
Tables	Delimited	Fastest/simplest parsing while also human readable
High performance	Protobuf	Industry standard, binary
SVG, Documents	XML	Industry standard, best tooling

Avoid CSV

People assuming CSV file format is just a bunch of fields delimited by commas has been the source of an extremely large amount of bugs in the thirty years I have been forced to occasionally use it. The algorithm for parsing a true CSV file is incredibly complicated and requires a well-established package library to be done properly. Alternatives like simple delimited formats do not.

Avoid TOML

TOML (Tom's Obvious, Minimal Language) is praised for its simplicity and readability, making it a popular choice for configuration files. However, when dealing with complex configurations and large data volumes, TOML exhibits several significant limitations.

Hierarchical limitations

TOML supports basic data structures like tables and arrays but struggles with representing deeply nested or complex hierarchies. This can lead to verbose and less maintainable configuration files when configurations become intricate.

Performance with large files

Parsing large TOML files can be slower compared to more optimized formats like JSON or Protocol Buffers. This performance degradation is noticeable in applications that require frequent reads and writes to configuration files.

Memory consumption

Handling extensive TOML configurations may result in higher memory usage, as the parser needs to load and process the entire file into memory. This can be problematic for applications running in memory-constrained environments.

No native inheritance

TOML lacks built-in mechanisms for inheritance or extending configurations. Managing configurations that share common settings or require hierarchical overrides becomes cumbersome and error-prone.

Reusability constraints

Reusing configuration snippets or modularizing configurations requires manual duplication or external tooling, reducing maintainability and increasing the risk of inconsistencies.

No comments in arrays

While TOML allows comments in tables and key-value pairs, adding comments within arrays is not supported. This limitation can impede documentation within complex data structures, making configurations harder to understand.

Minimal error feedback

TOML parsers often provide limited error messages, making it difficult to debug issues in large and complex configuration files. This lack of clarity can slow down development and maintenance processes.

Schema validation

There is no standardized schema definition for TOML, unlike JSON Schema for JSON or YAML schemas. This absence complicates the validation of configurations against expected structures and data types, increasing the potential for runtime errors.

Smaller ecosystem

Compared to JSON (`jq`), YAML (`yq`), or XML (`yq`), TOML has a smaller ecosystem of tools, libraries, and community support for handling complex and large-scale configurations. This scarcity can limit available resources and hinder the adoption of TOML in diverse projects.

Limited editor support

Advanced editor features like intelligent autocompletion, validation, and linting are less mature for TOML. This limitation increases the likelihood of errors in large files and reduces developer productivity.

Why code in Go?

> [YouTube: Sameer] yeah but here in India majority of startups are now preferring go and me being 2024 graduate see a lot of opportunities in go

Why no nano?

The nano editor might be on all Linux systems but it really isn't as good as basic vi for beginners.

When starting out with command-line text editors, many beginners gravitate toward nano because it appears simpler and more user-friendly. However, this simplicity comes at a cost. While vi (or its enhanced versions, vim and nvim) might have a steeper learning curve, they provide a much better foundation for mastering text editing in the long run. Here's why nano isn't the best choice and how it can even create bad habits.

Nano's misleading simplicity

Over-reliance on control key shortcuts: Nano uses control key shortcuts (`Ctrl` + letter) for basic operations like saving, exiting, or searching. While these shortcuts seem straightforward, they often overlap with critical terminal functions like `Ctrl+C` (interrupt) and `Ctrl+S` (pause/lock terminal output), causing confusion when beginners accidentally lock their terminal or interrupt running processes.

Limited Editing Features: Nano lacks advanced features that even basic vi provides, such as:

- Efficient navigation (jumping to the start of a line, word, or character)
- Powerful search and replace
- Undo/redo capabilities that are intuitive and robust

Encourages Dependency: Nano's heavy reliance on displayed shortcuts can prevent users from internalizing commands, making them less adaptable to other tools.

Why basic vi is a better choice

Widely available:

- Vi is included on virtually every Unix-like system, ensuring you can rely on it no matter where you work.
- Mastering vi guarantees you'll always have an editor available, even in minimal environments.

Learn once, apply everywhere:

- The skills learned in vi translate to other powerful editors like vim or neovim, which are used by professionals worldwide.
- Vi's modal editing (switching between insert and command modes) teaches fundamental text-editing concepts applicable in advanced workflows.
- Vi navigation is built-in to all shells (with `set -o vi`) and many other systems because of its ubiquity.

Encourages muscle memory:

By not relying on on-screen prompts, vi forces users to internalize commands quickly, building

long-term proficiency and preserving terminal screen real-estate.

Conclusion

While nano's simplicity might seem appealing at first, it ultimately hinders growth and can foster bad habits. Starting with vi, even at a basic level, sets a strong foundation for effective text editing. Vi's ubiquity, efficiency, and transferable skills make it the better choice for beginners who want to build confidence and adaptability in the command line.

Why Vim?

The benefits of Vim over other editors lie in its unique approach to efficiency, customization, and versatility. Here's a breakdown of why many developers swear by Vim:

Speed and efficiency

Modal editing: Vim's modal nature separates editing, navigating, and selecting text into distinct modes. This allows for precise and fast text manipulation without needing to move your hands away from the keyboard.

Keyboard-driven workflow: Everything in Vim can be done without a mouse, significantly speeding up tasks for those accustomed to its commands.

Commands as composable primitives: Vim commands can be combined in intuitive ways (e.g., daw deletes a word, yap yanks a paragraph). This composability makes it exceptionally powerful for editing text.

Lightweight and ubiquitous

Low resource usage: Vim is highly efficient, making it an excellent choice for older machines or resource-constrained environments.

Available everywhere: Vim comes pre-installed on almost all Unix-like systems, ensuring it's accessible in nearly any server or terminal environment.

Customization and extensibility

Highly configurable: Vim's configuration through .vimrc (or init.lua for Neovim) allows users to create tailored environments for specific workflows.

Plugin ecosystem: Thousands of plugins extend Vim's functionality, from language servers (LSP) for autocompletion and linting to themes like Gruvbox-Material for aesthetic improvements.

Scriptable: Vimscript or Lua (for Neovim) can be used to automate repetitive tasks or create custom commands.

Precision and versatility

Powerful search and replace: Vim's regex-based search and replace (:%s/pattern/replacement/g) works across entire files or custom ranges.

Macros: Record and replay sequences of commands, making repetitive edits easy and consistent.

Versatility: Suitable for quick edits or complex projects, Vim adapts to any programming or text editing need.

Productivity with practice

Muscle memory: Learning Vim takes effort, but once mastered, it becomes second nature, allowing you to edit text faster than with GUI-based editors.

Fewer context switches: Vim can be embedded into many tools (e.g., Git, terminal multipliers like tmux), reducing the need to leave your environment.

Longevity and community

Long history: Vim has been around for decades, and its continued evolution ensures it remains relevant.

Active community: An engaged user base and contributors produce a steady stream of tutorials, plugins, and updates.

Comparison with other editors

- Vim is faster for text manipulation and navigation once learned but lacks a GUI's visual aids.
- GUI editors often have built-in integrations, while Vim relies on plugins for similar features.
- IDEs provide extensive language-specific tooling out of the box but can feel bloated compared to Vim's minimalist approach.
- Vim's plugin ecosystem bridges the gap for features like autocompletion and linting.

Conclusion

In summary, Vim shines for users who prioritize speed, flexibility, and control over out-of-the-box features or visual interfaces. It excels in environments where efficiency and keyboard-driven workflows are paramount. While it has a steeper learning curve than other editors, the productivity payoff makes it a beloved tool for developers, sysadmins, and writers alike.

Why ~/.vimrc?

Using .vimrc as your main configuration file has several advantages for someone who values portability and needs to use Vim across different systems, especially when Neovim might not always be available:

Portability and ubiquity

Vim is ubiquitous: Vim is pre-installed on virtually all Unix-like systems, including Linux distributions and macOS. It's also commonly found on remote servers and even embedded systems. This ensures you can always access a familiar editing environment without needing to install anything extra.

.vimrc compatibility: The .vimrc file is recognized by all standard Vim installations. By sticking to .vimrc, you ensure your configurations work everywhere that Vim is available. Neovim can also read `.vimrc`, either natively or by sourcing it, making it possible to share your configuration between Vim and Neovim.

Simplicity and minimal dependencies

Avoids Neovim-specific features: Neovim introduces some features, such as Lua-based configuration, that are not compatible with standard Vim. While these can be powerful, they may lock you into using Neovim exclusively. By using `.vimrc`, you focus on features and plugins that work universally, avoiding incompatibilities on systems where only Vim is available.

Quick setup: If you frequently work on different machines or servers, having a portable `.vimrc` file that you can quickly copy or source makes it easier to get up and running. You can even store your `.vimrc` in a version-controlled repository (like GitHub), allowing you to fetch it quickly on any machine.

Efficiency and familiarity

Immediate usability: With `.vimrc`, your configurations and workflows are immediately usable on systems with Vim pre-installed. No need to install Neovim or additional dependencies. This is particularly useful in environments like remote servers, where you may not have the permissions or the time to install Neovim.

Standard features: Vim and .vimrc cover the majority of use cases for editing, scripting, and navigating text. For someone who values speed and universality over cutting-edge features, .vimrc is often enough.

Use case scenarios

Remote work: If you SSH into servers or work on machines where you cannot control the software installed, Vim is more likely to be available, and your .vimrc will ensure a consistent editing environment.

Team environments: Some workplaces or collaborative environments may standardize on Vim instead of Neovim, making `.vimrc` the more practical choice. For example, it is common to have a bastion or jump host machine to get to other machines and only authentication from that system can be done for access to services such at Kubernetes.

Emergency situations: If you find yourself on a system without Neovim, being familiar with and having a `.vimrc` setup ensures you won't lose productivity.

Best of both worlds

The best solution for most people is to combine both in a dual configuration that allows all the power of Neovim plugins when available and all the flexibility of a single `.vimrc` when not. In addition to `.vimrc` is a `~/.config/neovim/init.lua` file with `vim.cmd('source ~/.vimrc')` to load the other main file. This way, you can enjoy Neovim's advanced features when it's installed while maintaining a universal configuration for Vim. Adding something like the following to `~/.bashrc` ensures that the right vim is used:

```
_have() { type "$1" &>/dev/null; }
_have vim && alias vi=vim && export EDITOR=vim && export VISUAL=vim
_have nvim && alias vi=nvim && export EDITOR=nvim && export VISUAL=nvim
```

Conclusion

Using `.vimrc` as your main configuration file provides the most flexibility and portability. It ensures you can work comfortably in almost any environment without needing to rely on Neovim-specific features. For someone who values a consistent workflow across systems, `.vimrc` is the pragmatic choice.

Why GitHub?

The case for the most widely used development platform

When it comes to platforms for managing and collaborating on code, there are several contenders, including GitLab, SourceHut, and Bitbucket. However, GitHub remains the most widely used and influential platform for developers worldwide. Here's why GitHub consistently outshines its competitors, even for those who might have reservations about its corporate backing by Microsoft.

The power of popularity

GitHub's dominance in the market isn't just a matter of numbers—it's a testament to its value:

Largest Developer Community: With millions of active users and repositories, GitHub is the go-to platform for collaboration and discovery. The sheer size of its community means more opportunities for networking, learning, and contributing to meaningful projects.

Industry Standard: Most companies, open-source projects, and educational platforms use GitHub as their primary development hub. Knowledge of GitHub is often expected in job applications, making it a must-know platform for developers.

Superior visibility

Searchability: GitHub's advanced search tools make it easy to find projects, issues, and contributions, helping you discover opportunities and solutions faster.

Project Showcase: A GitHub profile serves as a portfolio, showcasing your work to potential employers, collaborators, and contributors. Public repositories on GitHub are more likely to be discovered and gain traction compared to other platforms.

Integration with Ecosystems: GitHub integrates seamlessly with countless third-party tools and services, from CI/CD pipelines to deployment platforms.

Best-in-class command-line tool

GitHub's command-line tool, `gh`, offers unparalleled efficiency for developers who prefer working in the terminal:

Ease of Use: `gh` simplifies common tasks like creating issues, managing pull requests, and cloning repositories—all without leaving the command line.

Efficiency in Workflow: GitHub's CLI provides shortcuts for managing projects and repositories directly, saving time and effort.

Advanced Features: Interactive features like viewing pull request details or switching between branches are intuitive and developer-focused.

Comparing to GitLab, SourceHut, and others

GitLab: While GitLab offers powerful CI/CD features, its interface and community size don't match GitHub's. GitLab's paid tiers are often necessary for features GitHub offers for free, like private repositories.

SourceHut: SourceHut appeals to minimalists but lacks the extensive features, integrations, and user base that GitHub provides.

Bitbucket: Popular for enterprise solutions, Bitbucket trails GitHub in open-source adoption and community engagement.

Addressing Microsoft ownership

For some, Microsoft's acquisition of GitHub in 2018 raised concerns. Many don't know that Microsoft saved GitHub from catastrophic bankruptcy. However, Microsoft has largely honored its commitment to maintaining GitHub's independence and focus on supporting open-source development.

Investment in Features: Since the acquisition, GitHub has introduced numerous improvements, including GitHub Actions, Codespaces, and enhanced security features.

Open-Source Support: GitHub remains home to the world's largest collection of open-source projects, demonstrating its dedication to fostering community-driven development.

Conclusion

While alternatives like GitLab and SourceHut have their merits, GitHub's unparalleled visibility, expansive community, and best-in-class tools make it the platform of choice for developers worldwide. Even for those wary of Microsoft's involvement, GitHub's contributions to open-source development and its continued innovation make it an essential part of any developer's toolkit. Choosing GitHub means joining the largest, most active network of developers and setting yourself up for success in the modern software development ecosystem.

Why bash for beginners?

When learning to use the command line, choosing the right shell is crucial. While there are many shells available, such as Zsh, Fish, and PowerShell, **bash** (Bourne Again Shell) stands out as the best option for beginners. Here's why:

Ubiquity and universality

Default shell on most systems: Bash is official shell of the Linux project and the default shell on the majority of Linux distributions. Zsh is only the default on macOS because Apple refuses to accept the terms of the license change on the bash project (even though they say it is because zsh is "superior"). Bash is widely available, making it the most accessible shell for learners across different platforms.

Cross-platform compatibility: Bash works on Linux, macOS, and Windows (via Git Bash or WSL), ensuring consistent usage regardless of your operating system.

Industry standard: Bash scripts are used in countless production environments, making knowledge of Bash a valuable skill in the job market. Modern bash is far more secure than the ancient POSIX shell scripts from the boomer UNIX era (use of `[[ ]]` that does not allow expansion injection attacks is just one example).

Simplicity and beginner-friendliness

Straightforward syntax: Shell scripting is never really *that* easy, its just powerful. But bash's syntax is relatively simple and easy to grasp for newcomers—especially when compared to monstrosities like PowerShell. Commands and scripting follow logical, minimalistic patterns that are less intimidating for beginners. In fact, every line a person types on the command-line is an ongoing, running, interactive program. Placing any of those same commands into a file and running it becomes a script. In that sense, using bash at all *is* itself "coding."

Rich documentation: Bash has an abundance of resources, including tutorials, man pages, and community support. Most command-line learning materials are written with Bash in mind, making it easier to find help.

No Overwhelming Features: Unlike Zsh or Fish, which come with extra features and plugins that can confuse new users, Bash focuses on core functionality, allowing learners to build a strong foundation. Most importantly, bash generally only uses a single `~/.bashrc` file rather than splitting them all out making it easy to configure and even easier to share that same configuration on any other system by simply copying a single file (which is what `cm init` does).

Portability and practicality

Write once, run Anywhere: Bash scripts are portable and can be executed on almost any Unix-like system without modification—especially if the convenient (albeit slightly less safe and performant) `#!/usr/bin/env bash` idiom is used (which is essentially required to get bash scripts to work on Homebrew macOS machines that put the latest bash binary in a strange place by default).

Widely Used in Scripting: Bash is the default scripting language for many tasks, such as automating workflows and managing servers. Learning Bash provides immediate practical applications in automation and system administration.

Learning Core Unix Tools: By using Bash, beginners gain experience with essential Unix tools (e.g., `ls`, `find`, `grep`), which are foundational for more advanced shell scripting and programming.

Zsh and other shells: why not start there?

Zsh: Zsh offers many advanced features (e.g., plugins, themes, and auto-suggestions) that are appealing for experienced users. However, these extras can distract beginners from mastering core shell concepts. And no one ever writes a "zsh script" for anything because no one can count on zsh being on the system and writing POSIX shell scripts is less secure and more annoying.

Fish: Fish is user-friendly and visually appealing but is not POSIX-compliant, meaning scripts written in Fish are not portable across systems. It introduces unique syntax that can make transitioning to other shells or environments more difficult. More importantly, however, fish promotes terminal muscle memory that does not work when on another system without it, a common occurrence in todays cloud-native, containerized, virtualised world.

PowerShell: PowerShell is powerful and Windows-centric but diverges significantly from Unix-style shells, making it less relevant for those learning Linux or macOS environments.

Conclusion

For beginners, **Bash is the ideal starting point**. Its simplicity, availability, and practicality make it the best choice for learning the command line. By mastering Bash first, you'll build a strong foundation in shell usage and scripting that will serve you well in any technical career.

Why Git Bash?

When working with command-line tools on Windows, developers often choose between Git Bash, Windows Subsystem for Linux (WSL), or a full virtual machine (VM). While each has its strengths, Git Bash is often the preferred choice for its simplicity, speed, and integration with Windows. Here's why Git Bash stands out.

Lightweight and fast

Minimal Setup: Git Bash is easy to install and requires minimal configuration compared to WSL or a VM, which involve additional software and system configurations.

Low Resource Usage: Git Bash consumes very little CPU and memory, making it an ideal choice for quick tasks and lightweight scripting.

Instant Access: Launching Git Bash is near-instantaneous, whereas starting WSL or a VM often involves initialization overhead.

Seamless integration with Windows

Native Windows Compatibility: Git Bash integrates directly with the Windows file system, allowing easy access to files without additional configuration. Windows paths (e.g., `C:\Users\`) are natively accessible, making it straightforward to work with local files unlike WSL which has an entirely different user home directory and plays with file permissions in crazy ways. A file created on Windows is not compatible with one created with WSL even though they are both visible. This is extremely confusing for beginners.

Git and SSH Ready: Git Bash comes preconfigured with Git and SSH, which are essential for developers working with repositories or remote servers. No need for separate installations or bridging tools.

"Git Bash Here" Context Menu: The right-click context menu integration allows users to open Git Bash in any folder directly, speeding up workflows that need bash but started from the Windows desktop.

Simple learning curve

Familiarity for Developers: Developers coming from Unix-like systems find Git Bash familiar, as it uses Bash syntax and tools. Unlike WSL or a VM, there's no need to learn additional commands for managing Linux distributions.

No Dual Environments: With Git Bash, there's no confusion about switching contexts between Windows and Linux environments, as is often the case with WSL or a VM. Plugging in a USB device, like a Yubikey works as expected (which absolutely does not work on WSL and is problematic with a VM).

Portability and maintenance

Standalone tool: Git Bash is self-contained and doesn't rely on external services or subsystems, making it easier to manage and update.

Cross-Machine Usability: Its simplicity makes it easy to replicate on multiple machines without worrying about installing and syncing full Linux distributions or VM images.

VPN networking: When logging into a virtual private network, for work, for example, Git Bash does not need special port forwarding that WSL or VM would require that you might not even be able to do depending on the security policies of your IT department regarding workstations. Git Bash is fundamentally just another application running on Windows so its traffic (from `ssh` say) always works.

Avoiding overhead of WSL and VMs

WSL Drawbacks: WSL requires additional storage and configuration, especially when managing multiple Linux distributions. Some tools in WSL may behave differently from their native Windows or Unix counterparts, leading to inconsistencies. Permissions are inconsistent and USB devices cannot even be used.

VM Drawbacks: VMs require significantly more system resources (CPU, memory, storage) and often have slower performance due to virtualization overhead. Setting up shared folders and networking between the host and the VM can be complex and time-consuming.

Best for most developer workflows

Git Bash strikes the perfect balance for most developers:

For Git and development tasks: Git Bash provides all the essential Unix tools and commands needed for managing repositories and scripting.

For quick access: Its simplicity and speed make it ideal for everyday tasks without the complexity of maintaining a full Linux environment.

Conclusion

While WSL and VMs have their place for specialized needs (e.g., running a full Linux development stack or testing cross-platform applications), Git Bash is often the superior choice for developers seeking a lightweight, fast, and integrated command-line experience on Windows. Its ease of use, seamless compatibility with Windows, and low overhead make it the go-to tool for efficient development workflows.

Where’s more terminal magic?

This book just scratches the surface of what is possible using a terminal—especially bash on a full Unix-like system. Several terminal tools have been left out of this book that are critical but covered in another book: *Terminal Velocity, Mastering the fastest human-computer interface*. These include:

- `tmux` - terminal multiplexer and so much more
- `w3m` - terminal web-browser 10x faster than GUI browsing
- `mods` - Claude AI integration and query from CLI
- `find` - most powerful Unix command for finding and transforming files
- `podman` - run containers from the command line
- `gh` - even more that can be done with the `gh` tool
- `git` - more powerful git functionality
- `nvim` - the power of treesitter and other plugins

Terminal Velocity also covers advanced bash scripting and creating a "dot files" repo using a Bonzai monolith.

Why not Linux for beginners?

Linux is the most important operating system of our time. Every single technologist *must* learn it to be considered serious. Those who do not languish behind those who do. So why not learn it first?

Linux must be installed

There are a tiny fraction of companies that sell computers that come with Linux, but usually they are as expensive as buying a Mac.

Using Linux requires terminal skills

What’s your distro?

I use Unix with a custom desktop manager.

I use Linux, btw—for three decades

Me: "I wish I could get a Sun box at home."

Joey: "You could always use Linux."

It 1994 my Unix mentor from Teleport suggested I learn more about Linux. I got a Pentium II at work to run it even though only the terminal worked. I installed MkLinux on my PowerPC from Apple in 1994. I didn't care if I would never get my Apple operating system back nor play Need for Speed with my cool Thrustmaster wheel ever again. I was permanently forced to use the terminal for *everything* including email over PPP dialup. I've been using Linux almost every day since.

Garage full o' Linux

Within a two years I had a garage full of old computers given new life as Linux servers. I had to clean them and burn CDs with Linux OS on them. It took forever to set them up. In the process I was so anxious I put my servers online in a rush after I got my first domain (tuxzone.net). I got hacked immediately because I forgot to run the hardening software of the day. I was in too much of a hurry to give out free access to Linux so others could learn about it like Joey had done for me. I did manage to catch that hacker right away, however, by setting up Tripwire.

Most powerful workstation

At work (Teleport Internet Services) I was given the most powerful workstation of anyone in the company (that wasn't a Sun box) a Pentium II. Despite all the power it would not run any of the graphic Linux desktops. The best admins in the building, who creating an ISP couldn't get it to work. It had the power, but—as is still the case—Linux desktop environments are just quirky and unstable, then and now.

Linux desktop sucks

People hate me for telling them the truth about Linux desktop. It's horrible—even today—compared to the experience of a Macbook Pro or, yes, even a Windows machine. I've got more years of experience using all Linux DT environments than many people have been alive—including the most recent stuff.

Use the right tool for the job, and the right tool for the desktop is just not Linux and never has been. Perhaps someday, but I've heard people say that for nearly 30 years and it still hasn't happened. Linux server is where it's at, not the desktop.

Linux saved my job

My Linux lab saved my job during the crash of 2000. I took a SysAdmin job because of the skills I had learned managing Linux in my garage.

Decade on Linux laptop

Coming soon …

Thanks awards for teaching Linux

Coming soon …

"Macs are Unix, you know"

Coming soon …

Started a school on Linux

Coming soon …

10-year-olds installed own Linux

Coming soon …

Manjaro crashed my school

Coming soon …

Linux destroyed my laptop

Coming soon …

Among first Linux livestreamers

My first livestreams were from a Linux machine. Not many people could say they were streaming from Linux desktop in 2021. I was constantly fighting with one thing or another. Ultimately, I gave it up. The main reason: I value my time. Here's why:

- Want Streamlabs? Forget it.
- Want StreamAvatars? Nope.
- Want to stream Zwift? Forget about it.
- SolidWorks for some 3D modeling? Can't.
- Prefer *real* professional creative tools like Photoshop, Illustrator, and Premier? Don't make me laugh.
- Gaming from Linux? Seriously? Okay, if you want slow games for no reason.
- What's that screen-ripping? Get used to it. You will fix and driver update will break it again.

People that claim everything "works for them" are lying. They just hide it. When you politely press them for details and get to know them, the truth comes out. I've heard Linux liars since 1996. I don't get it. It's okay to say when something is bad. It doesn't mean you don't support it. The truth is Linux never has and never will be amazing for livestreaming and gaming, not for 99.9% of people, the ones who value their time to do other things.

Linux desktop died live, on stream

Coming soon ...

Livestream community fails to get Zwift working

Coming soon ...

"Skills issue."

Coming soon ...

Why not Alacritty?

I loved Alacritty on Mac for years, but in 2024 it has fallen way behind and contains substantial, show-stopping bugs that make it absolutely unusable for me.

I am so glad `s9tpepper_` from Twitch shared with me why he prefers Wezterm (which has its own section now in *Terminal Velocity*. (Even that italic that I just wrote for that title showed up beautifully, as does actual bold fonts. Wezterm is just so superior to Alacritty there isn't even a comparison at this point.

No support for `Control-[`

While testing it on Windows I noticed a bunch of `[` showing up on the screen using the terminal the same was I have for more than 25 years. This turns out to be because Alacritty has a bug that does not observe the standard `Control-[` as `Esc`. If nothing else were wrong this would be enough for me to never use it again.

Frequent, unexplained crashes

Last week while working on this book and trying to standardize on Alacritty (eventually going with Wezterm instead). Alacritty crashed without explanation or error at least a dozen times. I wasn't doing anything fancy. At least it was a good reminder why to write about tmux saving the day by being able to reattach after restarting.

TOML configuration

I absolutely hate that Alacritty *forces* me to use TOML for configuration for now good reason at all even though it quietly supports YAML still.

(See Why Not TOML.)

Impossibly poor emoji support

Want to use your favorite emoji with the emoji picker on a Mac? Tough. Alacritty doesn't support pasting them inline like every other terminal on the planet.

Failure to implement all ANSI terminal escapes

I abandoned Alacritty once already in 2021 when I realized that some of the terminal animation and coloring using popular ANSI escapes wasn't supported. While I think that is supported now, the fact that it didn't work and now `Control-[` isn't supported suggests the team is not too keen on supporting the actual ANSI standards. All the other major terminals definitely do.

No italic and true bold support

I have waited most of my life for a proper terminal to support thin and think text in the same font. Alacritty doesn't, most of the other modern ones do, including Windows Terminal Preview. That's right, Windows Terminal Preview is *better* than Alacritty and every bit as fast.

Speed

It took using Wezterm for a day or two to realize just how much I was needlessly suffering with terminal display lag on Alacritty. With things like pop-up completion options and such that lag is noticeable and annoying. Such is simply not an issue in all the others. Alacritty can't even compare and increasing speed on that level is not something Alacritty will ever be able to catch up with since it is a core implementation thing. They'd have to rewrite the entire engine.

Why not TOML?

Even though I created the original TOML logo, I have come to loathe it for anything. So many are obsessed with it like I was in 2018, but I have come to realize just how horrible it is for anything substantial. In fact, I won't hate on YAML at all anymore although my preferred way to configure anything these days is Lua. (Can you just imagine if Kubernetes resources had been implemented in Lua?)

Lack of real tool support

TOML has no tools for dynamically altering it from a simple shell script like `yq` does for YAML. The `yq` tool will do some TOML in-place edits, but nothing fancy. I know a tool shouldn't drive a structured data choice, but for me it does. Plus, can we just accept please that YAML has complete won at this point. (I cannot believe I liked TOML enough to make the logo. I really loathe it now.)

Still has significant white space

Lack of adoption

TOML was big with the Rust people for a while and still remains required for Cargo files.

Arrogant community

> Maybe the Kubernetes project should change to TOML.

Why are you still using Perl?

While trends may favor newer tools, the undeniable truth remains: Perl is still the most powerful and versatile language for text processing from the command line. Despite the rise of languages like Python, Perl's unparalleled features and efficiency make it the go-to choice for developers, hackers, and sysadmins who need to process text with precision and speed. It's regular expression DNA is in every other modern language.

Things shell forgot

The `-T` switch is so powerful, unknown and obscure. All it does it test that the file not only exists and is readable, but is also a text file. Such a simple thing, but I use it all the time. All by itself this is a reason to choose Perl over Bash for almost everything I do.

Why Perl rules the command line

A text-processing powerhouse

At its core, Perl was designed for text manipulation. Its regular expression engine is not just powerful; it's legendary. Perl offers features like backreferences, named captures, and advanced pattern matching that other tools struggle to replicate.

Example of extracting and printing matches:

```
perl -ne 'print "$1\n" if /error: (\S+)/' logfile.txt
```

This one-liner scans a log file, extracts the error codes, and prints them with unmatched clarity and efficiency.

One-liners for every occasion

Perl's syntax is compact yet expressive, making it ideal for writing one-liners that handle tasks ranging from simple filtering to complex data transformation.

For example, replacing all instances of "foo" with "bar" in multiple files:

```
perl -pi -e 's/foo/bar/g' *.txt
```

No need for temporary files or complex pipelines—Perl does it all in place.

Flexibility beyond compare

Unlike `awk` or `sed` or `tr` or `grep`, Perl seamlessly bridges the gap between simple text-processing tasks and full-fledged programming. Need to parse comma-delimited files, handle multi-line input, or manipulate strings? Perl has you covered without requiring external libraries or tools.

Example: Parsing a comma-delimited file and extracting specific fields:

```
perl -F, -lane 'print $F[0], $F[2]' data.txt
```

This one-liner uses Perl's split and field syntax to process comma-delimited files effortlessly.

Portability and compatibility

Perl scripts run on virtually every operating system. From Linux to macOS to Windows, Perl's ubiquitous availability ensures that your scripts will work anywhere without modification. Unlike tools like `sed`, which can have differing behaviors across environments, Perl provides consistent functionality no matter the platform.

A legacy of innovation

Perl introduced many features that other tools later adopted, including powerful regex support and flexible file handling. It's not just a language—it's the backbone of countless legacy systems and workflows.

Libpcre: Perl's DNA in modern languages

Perl's influence goes far beyond its interpreter—it's baked into the very fabric of modern programming. The `libpcre` (Perl-Compatible Regular Expressions) library, inspired by Perl's regex engine, is now included in almost every major language and framework.

Examples of libpcre in action:

- **Python:** The `re` module is heavily influenced by Perl's regex syntax.
- **JavaScript:** Many of its regular expression capabilities are directly inspired by Perl.
- **PHP:** Uses libpcre for its regex functionality.
- **Go:** The `regexp` package in Go relies on libpcre-like behavior for its regex processing.
- **Ruby:** Adopts Perl-like regex features for its own implementation.

These languages didn't reinvent the wheel; they borrowed Perl's regex engine for its unmatched power and flexibility. Even when you're not directly using Perl, you're likely benefiting from its innovations.

Sed vs. Perl: The portability advantage

While `sed` is often used for in-place text replacements with its `-i` flag, this feature isn't consistently supported across systems. For example, the behavior of `sed -i` differs between GNU sed and BSD sed (e.g., on macOS), often requiring workarounds like adding an empty string (`''`).

In contrast, Perl's `-p -i -e` syntax works universally and consistently across platforms:

```
perl -pi -e 's/foo/bar/g' *.txt
```

This portability makes Perl the better choice for scripts and one-liners intended for diverse environments. It's better to burn Perl into your one-liner command muscle memory than any of these alternatives.

Implicit printing with `perl -p`

One of Perl's standout features is its ability to process and transform every line of a file with implicit printing using the `-p` flag. This eliminates the need for an explicit `print` statement, making your code cleaner and more concise.

Example: Adding line numbers to a file:

```
perl -p -e '$. .= ": $_"' file.txt
```

Each line is prefixed with its line number, and the result is printed automatically.

Another example: Converting text to uppercase:

```
perl -p -e '$_ = uc($_)' file.txt
```

Every line of the file is transformed to uppercase and printed without additional effort.

The "awk vs. Perl" debate

While awk remains a respected tool for lightweight text processing, it pales in comparison to Perl's versatility and depth. Tasks that require multiple pipelines in awk can often be handled in a single Perl script.

For example, extracting and transforming data with awk:

```
awk -F, '{print $1, $3}' file.txt
```

In Perl, the equivalent is just as simple but far more extensible:

```
perl -F, -lane 'print $F[0], $F[2]' file.txt
```

Best language for Vim and Neovim

Perl's unmatched supremacy in text processing makes it a natural companion to Vim or Neovim for advanced editing workflows. Its concise syntax, powerful regular expressions, and vast array of built-in text-manipulation tools allow users to quickly and efficiently transform content. Pairing Perl with Vim's filter functionality (`!!`) creates a seamless editing experience: you can write transformations or scripts directly within your document, execute them by piping the content to the Perl interpreter, and replace the original text with the output—all without leaving your editor.

This workflow exemplifies the Unix philosophy of combining small, powerful tools to create robust solutions. By leveraging Perl's flexibility and efficiency, you can perform anything from basic substitutions to complex data transformations inline. This eliminates the need for external preprocessing or postprocessing steps, making your editing environment not just a tool for writing but a dynamic, programmable text-processing powerhouse. For anyone serious about crafting or editing text at scale, the synergy between Perl and Vim offers unparalleled power and flexibility.

The "boomer language" myth

First of all, if you use the word "boomer" in the pejorative you have just instantly filtered yourself from the brains of most competent technologists. Only complete dumb-asses say such things. Every generation has had its idiot-detecting terminology and that is one main one today. If you think stuff on 4-chan is worth repeating, you can move along.

The idea that Perl is a "boomer language" is a complete misunderstanding of its power and relevance. While newer languages like Python have gained popularity, Perl remains the tool of choice for those who value:

- **Efficiency:** Do more with less code.
- **Reliability:** Proven in production for decades.
- **Performance:** Optimized for text-heavy workloads.

The reason Perl because the defacto language for *all* backend web development—when there was no JavaScript and no frontend at all—is because the only alternatives were C and POSIX shell, which, as anyone who knows what they are talking about will tell you, is ridiculously insecure and easy to shell-inject. Perl even comes with taint checking (`-T`) and forces the develop to explicitly match any data the originated outside of the script in any way. Python never fucking did that. JavaScript sure didn't. Perl was never intended to do all the web backend processing it was forced to do, it was just that fucking good. Today there are tons of other options for backend web development that are all far superior for that, but *all* of those language suck ass when it comes to text processing from the command line.

Why Perl still matters

Trends come and go, but Perl's utility and efficiency have stood the test of time. When you need to process text on the command line, nothing matches Perl's combination of power, flexibility, and ease of use.

So next time someone questions why you use Perl, just point them to this page and let them know: Perl is not just a tool—it's a way of getting things done right the first time.

Why another Go book?

I've spend several hundred dollars on Go books, always wanting to find one I could just use and recommend to absolute beginners—who have not only never coded before—but didn't know the word "code" could be used as a verb, the type of beginner who thinks "print" involves a printing press or things that run out of toner, those who have never heard the words "Linux" or "GitHub" before in their lives, who think a "terminal" is something to run through at the airport when they are late for a flight. "Go, do you mean like Okay Go, that band from 90s?" These wonderfully clueless people have absolutely zero tech background and never regretted it, until now. Now, for whatever reason, they want to learn to code and, you guessed it, they cannot find a single book out there written for *them*.

I've been a nerd and a geek my entire life and have the photos to prove it (look up the difference). It turns out, I also have a gift for linguistics that includes speaking fluent Russian, French, Creole and English but also translating techno-babble into comprehensive prose and conversation in order to help *anyone* to discover how to learn *anything* in their own way. Since starting SKILSTAK Coding Arts in 2013 and learning and teaching Go in 2014 (when TJ Hallowaychuk famously wrote Farewell to Node) I have noticed all Go books suffer from one or more of the following annoyances:

- Incorrect or out of date material
- Avoid real, practical applications
- Shun the terminal, except to compile
- Assume "some programming" already, usually C
- Lack a free and supportive learning community
- Take themselves too seriously to be memorable or fun
- Ignore how to setup a personal development environment
- Pick poor project ideas that are too big and easily forgotten

Most of these books have some sort of dependency on the Internet to get the accompanying code, but none have any kind of downloadable personal learning assistant app, like the one I developed in Go for this book: Code-Mage.

At the risk of sounding too negative, I'll describe the straw that broke my back. I bought a $40 book to learn about generics that turned out to not cover generics at all—but even worse—contained an entire chapter *claiming* to teach generics that contained a vague proposition for generics that was never adopted by the language. This author and publisher rushed their book out based on false information confusing me and potentially thousands of others. Then they charged another $40 for a "new edition" of the same book without even attempting to give anyone back the money they wasted for the first book. I lost hundreds in premium time wasted. You can probably guess the book, author, and publisher, which is notorious for this kind of fraudulent cash-grab.

Needless to say, I was furious. Most books are written by great people who have put a lot of time and energy into them. At worse they are annoying in their imperfections. But this blatant "screw you, give us your money" made me mad enough to follow my own mantra: *don't get mad, get busy*. I started thinking about what I would put into a book based on what I *knew* hundreds of absolutely non-technical people I had helped to learn Go as a first language would want, and now you are

reading it.

Why is your stream so quiet?

I learned the hard way that muting everything is best when just sharing a terminal in over-the-shoulder mode. For more on that and other lessons, see my perpetually unpublished *Bad Strimmer*.

Why AsciiDoc instead of MkDocs?

I love and use them both a lot.

AsciiDoc is superior for creating books, PDFs and EPUBs. MkDocs doesn't really have any of that.

MkDocs is superior for web sites with have indexed searching in them. This is ideal for technical documentation but doesn't allow that same information to be published easily in any other form but the web.

I'm obsessed with text processing

I'm not exactly sure why but I have always been absolutely obsessed with parsing text in every and any way possible. Sometimes I wonder if this is because I started with Perl for the beginning of my career for most significant stuff. Maybe it's my affinity for syntax and spoken and written languages. Maybe all of that comes from other particular way my semi-autistic brain is wired. I love that stuff.

I've never actually made a compiler but I have made *transpilers* before. For some reason the compiler part doesn't interest me. I'm all about finding the syntax of one thing and transforming it into something else. For example, writing this book in bash code and then transpiling it to an Excalidraw flow chart by implementing an AST parser for bash itself and interpreting it into API commands that generate the flow chart automatically. Now *there's* a project I find so amazing I have a hard time not dropping everything and just working on that right now.