The Case for Bash

August 2025

5 min read • 1,048 words

Themes: Programming Recursive

During my time at Heroku, I learned to appreciate the Bash programming language for performing system tasks. I maintained the Python buildpack—the system that automatically detected and deployed Python applications—and the entire thing was written in Bash

> The Python buildpack is roughly 2,000 lines of Bash that handles Python version detection, pip installation, dependency management, Django collectstatic, and dozens of edge cases. It processes hundreds of thousands of deployments daily.

. Not Python, not Ruby, not any of the "modern" languages we champion. Just good old Bash, orchestrating complex deployment workflows for millions of applications.

There's a reason we chose Bash for something so critical. Not shell scripting in general, not zsh or fish with their quality-of-life improvements, but Bash specifically—the GNU Bourne Again Shell that's been installed by default on Linux systems since 1989

> While zsh and fish offer better interactive experiences, Bash remains the POSIX-compliant choice that works identically across Ubuntu, CentOS, Alpine Linux, and macOS. This predictability is invaluable for production systems.

.

The Language That Refuses to Die

Bash is like water—it finds its way into every system, flowing through the cracks of our infrastructure. Every Linux server speaks it. Every Docker container understands it. When you SSH into a burning production system at 3 AM, armed with nothing but desperation and a terminal, Bash is there, waiting.

This ubiquity isn't luck. While we've been building deployment orchestration platforms that require Kubernetes clusters just to say hello world

> A minimal "Hello World" deployment on Kubernetes requires at least: a Deployment manifest, a Service manifest, an Ingress configuration, and often a ConfigMap. That's before considering namespaces, RBAC, or networking policies.

, Bash has been quietly shipping code with the same humble syntax it learned in 1989. It's the cockroach of programming languages—unkillable, omnipresent, and surprisingly capable when you need it most.

The Beauty of Pipes

Consider a simple task: find all Python files in a project, search for a specific import, and count the occurrences. In Bash, it's one line:

```
find . -name "*.py" | xargs grep "import requests" | wc -l
```

That's it. Three commands, two pipes, one result. The data flows naturally from left to right, each command doing one thing well

> This is the Unix philosophy in action: "Write programs that do one thing and do it well. Write programs to work together." Bash didn't invent this—it just makes it effortless.

.

Now try the same thing in Python:

```
from pathlib import Path

count = 0

# Find all Python files recursively.
for py_file in Path('.').rglob('*.py'):

    try:
        # Read file and count matching lines.
        content = py_file.read_text(encoding='utf-8')

        for line in content.splitlines():
            if 'import requests' in line:
                count += 1

    except (IOError, UnicodeDecodeError):
        # Handle files we can't read.
        pass

if __name__ == "__main__":
    print(count)
```

The Python version is more "proper" programming, but it's also more verbose, more error-prone, and honestly, more complex for what should be a simple task. We need explicit encoding handling, exception management for unreadable files, path joining logic. We've taken something that flows naturally and turned it into a state management problem

> There's a cognitive bias in programming where more verbose, "enterprise-ready" solutions are perceived as more professional. Sometimes a one-liner is actually the more sophisticated choice.

.

The Misunderstood Language

People love to hate Bash. "It's unreadable!" they cry, while writing React components that require a PhD in hook dependencies

> Try explaining useEffect dependency arrays to someone who's never seen them. "Well, if you don't include the right dependencies, it might not re-render, but if you include too many, it might re-render infinitely, and also don't forget about useCallback..."

. "It's error-prone!" they shout, ignoring their `node_modules` folder containing more code than the Linux kernel

> The Linux kernel is about 28 million lines of code. A typical React application's node_modules easily exceeds this. We're importing the equivalent of an operating system to display a todo list.

.

Here's the thing: Bash isn't trying to win a beauty contest. It's not auditioning for your startup's tech stack. Bash is the plumber of the programming world—it shows up, connects the pipes, and makes sure your data flows where it needs to go. You don't need to invite it to dinner.

Yes, `[[ ]]` vs `[ ]` is confusing. Yes, `'string'` vs `"string"` is unobvious to newcomers. Yes, variable expansion rules seem designed by someone who hates humanity. But you know what? Regular expressions are also insane, and we still use them. SQL reads like English written by aliens, and we still query databases.

And here's the kicker: tools like shellcheck elevate your Bash programs significantly, catching these gotchas before they bite you. Bash's syntax is the price of admission to a language that runs literally everywhere without asking permission.

The Deployment Reality

Building deployment systems at Heroku taught me that the genius isn't in complex logic but in orchestration. Detect the app type, set up the environment, install dependencies, run build steps. Bash excels at this kind of system coordination because that's literally what it was designed for.

Now, yes, Python is part of the Linux Standard Base, so theoretically you could write system scripts in Python. But here's where theory meets reality: which Python? Python 2.7? 3.6? 3.11? The LSB specifies Python should be there, but not which version. And God help you if you need any package beyond the

standard library—suddenly your simple system script needs pip, virtualenvs, and dependency management. Your "portable" Python script just became a deployment nightmare.

Bash doesn't have this problem. Bash 3.2 from 2006 runs the same scripts as Bash 5.2 from 2022. No pip. No virtualenv. No questioning whether `subprocess.run()` exists in this version or if you need `subprocess.call()`. Just `#!/bin/bash` and go.

The Uncomfortable Truth

Bash is the Latin of programming—dead, supposedly, yet somehow the foundation of everything we build. Every Docker container starts with a Bash command. Every CI/CD pipeline is secretly Bash in a trench coat. Every deployment script, monitoring tool, and system automation eventually reveals itself to be Bash all the way down.

You can hate Bash. You can avoid it. You can write elaborate Python scripts to avoid learning its syntax. But eventually, at some point, you'll type `#!/bin/bash` because you need something to just work, everywhere, without drama.

And it will.

Bash isn't the language we deserve. It's the language we have. And sometimes, that's exactly what we need.