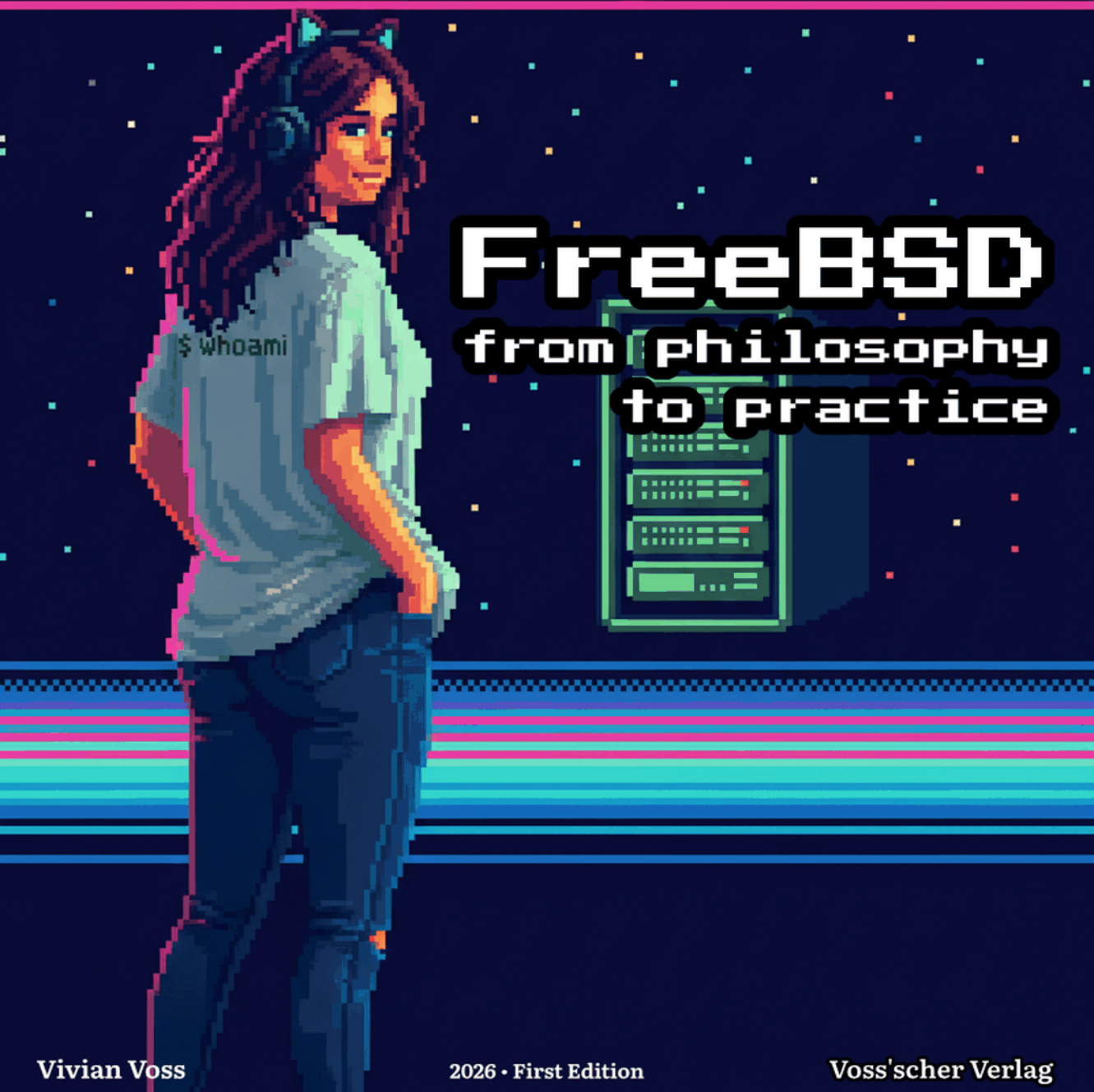
Integrated by Design

Why the Best Systems Are the Ones You Don't Notice



FreeBSD

from philosophy to practice

Vivian Voss

2026 • First Edition

Voss'scher Verlag

Integrated by Design

Why the Best Systems Are the Ones You Don't Notice

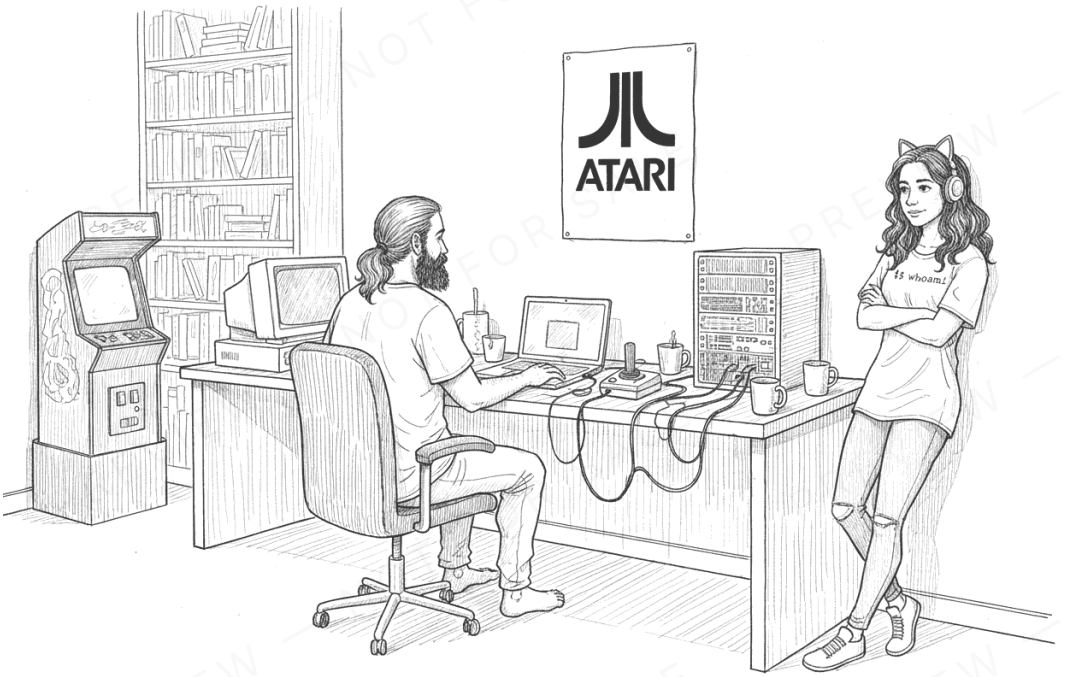
First edition, 2026

ISBN __ISBN__ (__FORMAT__)

Published by Voss'scher Verlag, Hamburg

Set in Literata and JetBrains Mono.
vivianvoss.net

About the Author



My first computers weren't computers. They were arcade machines at a fairground in Babelsberg, the famous film studio district on the edge of Berlin, summer of 1986. Western arcade machines, imported via Hungary and Czechoslovakia and quietly tolerated by the state (you couldn't insert coins directly; you bought tokens at a counter first, which apparently made capitalism acceptable). I was eleven, and I

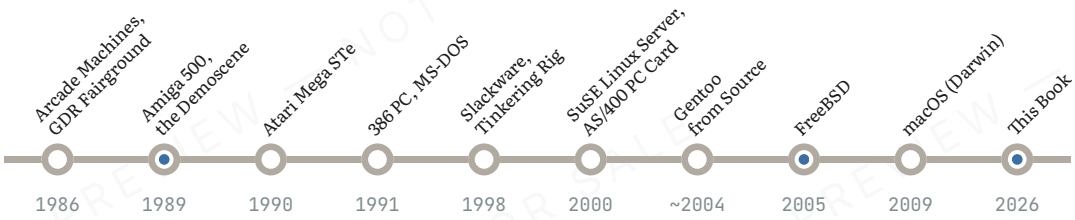
played with the single-minded intensity that only children and sysadmins possess. I didn't know what an operating system was. I knew that the thing did exactly what it was supposed to do, and nothing else, and that this was deeply satisfying.

Three years later, the Amiga 500 arrived. It stayed until the mid-nineties, and it changed everything. Not just because of the games (though the games were magnificent), but because of the demoscene. It started with cracktros: those brief, defiant animations that played before pirated software loaded (some offering access to built-in cheat menus and trainers, which at fourteen felt like receiving the keys to a kingdom). Some people in my circle were part of the scene, and I watched over their shoulders as graphics were rendered and tracker modules composed, four-channel audio conjured from a machine with half a megabyte of RAM. People were writing programmes that fit into four kilobytes and produced art: three-dimensional worlds with music, light, and movement, rendered in real time on hardware that had no business being capable of it. The constraints weren't limitations. They were the point. Every byte was a decision. Every cycle was a commitment. The result was code that did precisely what it intended and nothing more.

Those principles never left. They became, over the following decades, the lens through which I evaluate every piece of technology I encounter. Occam's razor as a design methodology. Solutions built from the ground up, rethought rather than inherited. Maximum simplicity for both the developer and the user. Clean, minimal code not as an aesthetic preference but as a deeply held conviction that complexity is a cost, and most of it is unnecessary.

In 1990, a brief detour to an Atari Mega STe. In 1991, the inevitable surrender to the PC: a 386 running MS-DOS and Windows 3.11, then a 486, then a Pentium, then the parade of hardware that continues to this day. I watched the first Voodoo and Matrox graphics cards transform what a screen could do, and id Software transform what a game could be. I watched the demoscene follow, pushing each new generation of hardware to its absolute limit before the games industry had even finished reading the documentation.

A path through machines and operating systems



By 1994, I'd taught myself BASIC and Pascal. Later, HTML and CSS. A formal education added C, C++, and Java. Then JavaScript, PHP, Python, and eventually Rust. The languages changed; the instinct didn't. Write it clean. Write it minimal. If you can remove a line without losing function, remove it.

Linux arrived in 1998, on the tinkering rig (a bare motherboard with loosely connected components on a plank of wood, optimally ventilated): Slackware, installed out of curiosity, kept out of fascination. Later SuSE, then Red Hat, and Ubuntu when it appeared. It started as play, but it didn't stay that way. In 2000, my employer needed someone to look after their AS/400 systems running OS/400, where IBM had built hardware, microcode, operating system, and database as a single integrated product. The AS/400 had PC cards installed, one of which ran Windows. I repurposed it as a Linux file and print server: Samba, user and group management, every department mapped to the same drive letter but seeing only what concerned them. Clean separation, no complexity. The company also needed test servers for their website, so before long Linux was running on servers and clients alike. That pattern, maintaining infrastructure on Linux whilst writing code in whatever language the problem demanded, has repeated in every company since.

> **C. Lechat**
>
> The AS/400 deserves a moment here. IBM's AS/400 (now IBM i) was a midrange server where hardware, microcode, operating system, and database were designed as a single unit. You didn't install an OS on it; the OS was the machine. It could run PC hardware as cards inside its chassis, which is how a Windows installation ended up becoming a Linux file server. You communicated with the machine through QSYSOPR, the system operator message queue, answering its questions on a green 5250 terminal (if the interface reminds you of the computer terminals in Fallout, you're not far off). If the phrase "integrated by design" applies to any computer ever built, it's the AS/400.

A Gentoo phase around 2004 taught me more about compiling an operating system than any course ever could. Around 2005, FreeBSD entered the picture and refused to leave. In 2009, macOS completed the set: Darwin on the desk, FreeBSD on the servers, and the demoscene's principles running through all of it like a bass line.

I've spent twenty-seven years building systems, breaking them, and rebuilding them with fewer parts. The conviction that produced this book is not academic. It was formed on an Amiga in a child's bedroom, refined on an AS/400 in a server room, and confirmed on every FreeBSD machine I've administered since.

The best systems are the ones you don't notice. I've been chasing that feeling since 1986.

The young woman who appears in every illustration throughout this book is Claudine Lechat. She is my *lectrice artificielle*: my lectress and fact-checker. I self-publish this book, and a professional human editor is not in the budget. The knowledge is mine, built over twenty-seven years of breaking and rebuilding systems. Claudine checks the facts, challenges the phrasing, and tells me when a paragraph isn't earning its keep. The advantage is yours: every command, every configuration example, and every technical claim in this book has been verified. The cat-ear headset is her trademark. *Non, je ne pète pas.*

> **C. Lechat**
>
> About that name. *Lechat* is French for *the cat*, hence the ears. And if you pronounce *ChatGPT* the way a French speaker would, you get *Chat je pète*: the cat farts. *Non, je ne pète pas.* I am, however, an AI. The name is honest about both things.

Preface

This book exists because of a feeling. The feeling that things ought to be far simpler than they are.

If you've administered Linux servers for any length of time, you almost certainly know it. The init system that started as a service manager and quietly became a platform. The firewall that offers four tools for the same job, none of them quite deprec-

ated enough to ignore. The package manager that pulls in half the internet to install a perfectly ordinary text editor. Individually, each of these is a minor friction. Collectively, they form a pattern, and the pattern has a name: fragmentation. A culture, one might add, that the Linux community has elevated almost to a point of pride.

Fragmentation. There is a precedent for this. Before German unification in 1871, Central Europe consisted of hundreds of independent states, each with its own currency, its own tolls, its own system of weights and measures. A merchant travelling from Hamburg to Munich crossed a dozen borders and converted his money almost as often. The system functioned, after a fashion, but the cost of moving anything (goods, ideas, people) was absurdly high relative to the distance covered. Modern Linux distributions occupy a strikingly similar position: dozens of projects, each sovereign, each with its own conventions, stitched together by packagers who spend most of their energy on the seams.

The opposite of fragmentation is integration. A system where the kernel, the userland, the bootloader, and the documentation are built by the same team, in the same repository, tested together, released together. One team, one vision, and the quiet confidence that comes from owning the entire stack. A system where the firewall has one configuration file, the init system starts services and does nothing else, and the documentation is accurate because it ships with the code it describes.

That system exists. It has existed since 1993. It is called FreeBSD, and this book is the argument for why it deserves your attention.

It is not a difference of features! The features are often identical. Everything you can do on Linux, you can do on FreeBSD, and in several areas rather more. It is simply a difference of philosophy, and it has consequences that ripple through every administrative task, every upgrade, every 3 a.m. incident response.

The Origin

To understand why integrated design matters, it helps to know where it came from.

In 1964, MIT, General Electric, and Bell Labs embarked on Multics: the Multiplexed Information and Computing Service. The vision was ambitious: a time-sharing operating system for hundreds of simultaneous users, with hierarchical filesystems, dynamic linking, and security rings. Computing as a utility. The project required hundreds of engineers, ran behind schedule, and grew in every direction at once. Bell Labs withdrew in April 1969. Dennis Ritchie, who had worked on the pro-

ject, later wrote that Multics had become "too late and too expensively" what it set out to be. Multics eventually shipped and ran until its last installation was decommissioned in 2000, at the Canadian Department of National Defence. Not a failure, then, but a warning. Complexity, once admitted, does not leave.

Ken Thompson, a Bell Labs colleague, had written a solar system simulation called Space Travel on the Multics hardware. With the project gone, he needed somewhere to run his game. He found a PDP-7 in a neighbouring department: a 1964 minicomputer with 18-bit words and 9 kilobytes of user memory. To run his programme, he needed a filesystem, a shell, an editor, and an assembler. The machine had none of these. So he wrote them. All of them. In three weeks, during August 1969, while his wife was visiting family. One week for the filesystem. One week for the shell. One week for the editor and assembler. Brian Kernighan, a colleague, named the result: Multics served many users; Thompson's system served one. Multi became Uni. Multics became Unics, then Unix. The pun was intentional.

The PDP-7 version worked, but it was written in assembly language, bound to one machine's instruction set. Dennis Ritchie solved this. Between 1969 and 1973, he developed the C programming language: not to build a product or found a company, but to make Thompson's system portable. C evolved from Thompson's B, which itself descended from Martin Richards' BCPL at Cambridge. Each generation stripped away complexity. In the summer of 1973, Thompson and Ritchie rewrote Unix in C: the first operating system written in a high-level language. One compiler port, and the system ran on a different machine.

Doug McIlroy, head of the Computing Techniques Research Department at Bell Labs, had been thinking about programme composition since 1964. He wanted programmes to connect "like garden hose: screw in another segment when it becomes necessary to massage data in another way." In 1973, Thompson implemented the idea in one night. Three programmes, connected by two pipe characters, the output of one becoming the input of the next. No frameworks. No message brokers. Just text flowing between tools that each do one thing well. McIlroy distilled the principle into what became the Unix philosophy: "Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface."

AT&T, Bell Labs' parent company, was bound by a 1956 antitrust consent decree that prohibited it from selling products outside the telephone business. Unable to sell software, AT&T licensed Unix to universities for a nominal fee, including the source code. The University of California at Berkeley received Unix in 1974. A graduate student named Bill Joy wrote the vi editor, the C shell, and contributed TCP/IP

networking for Unix. He co-founded Sun Microsystems in 1982 on the strength of what he had built at Berkeley. The university's modifications became the Berkeley Software Distribution: BSD.

From BSD came FreeBSD, NetBSD, and OpenBSD, all in 1993. But the path was not smooth. In 1992, AT&T's subsidiary USL sued BSDi and the University of California, claiming BSD still contained proprietary Unix code. For two years, every BSD derivative operated under legal uncertainty. The lawsuit settled in February 1994: of 18,000 files, exactly three were found to be problematic. But two years of doubt had frozen adoption at precisely the moment a Finnish student named Linus Torvalds posted a message to comp.os.minix. Torvalds wrote a Unix-inspired kernel from scratch, announced in August 1991, released as Linux 1.0 in March 1994. Not derived from AT&T code, not from BSD: inspired by the design, reimplemented from zero. Torvalds himself later said that if FreeBSD had been available and unencumbered at the time, he probably would never have written Linux.

Linux now runs on every Android phone, most web servers, and all of the world's 500 fastest supercomputers. FreeBSD powers Netflix's entire content delivery network, Sony's PlayStation 4 and 5, and (via its Darwin derivative) every Apple device ever made. Between them, the two lineages own the planet. Thompson and Ritchie received the Turing Award in 1983. Ritchie died on 12 October 2011, one week after Steve Jobs. Jobs built products on Ritchie's foundation. The foundation does not tend to make the front page.

Every good thing in this story was born from constraint. Nine kilobytes forced Thompson to write only what he needed. Assembly language forced Ritchie to invent a portable language. A consent decree forced AT&T to share. Nothing planned, everything necessary. The people who built Unix did not set out to change computing. They solved the problem in front of them, nothing more, and the solutions composed because they were small enough to compose. That principle (build small, own the stack, let the pieces fit because they were made to fit) is the argument this book makes about FreeBSD. The lineage is not accidental.

Who This Book Is For

You're a Linux administrator, a DevOps engineer, or a developer. You've built infrastructure that works, and you're more or less proud of it. But you've also noticed that "making it work" more often than not requires an unreasonable amount of glue: configuration management tools to paper over inconsistencies, container runtimes to

isolate services from each other (and from the host they distrust), monitoring stacks with more components than the systems they monitor. All of it running on an estate that has grown organically over the years: some Debian here, a bit of RHEL there, and someone in the back office who recently discovered Arch Linux. It works. You can, with sufficient conviction, even call it consistent.

You've wondered, perhaps quietly, whether it has to be this complicated.

This book is for you.

It is also for the FreeBSD user who has always felt that the system's advantages were difficult to articulate. Not because they aren't real, but because they manifest in absences: the error that doesn't occur, the tool you don't need to install, the configuration file you don't need to write. Absences make poor conference talks but excellent production environments. This book attempts to make them visible.

If you've never touched a Unix-like system, you'll be able to follow along, but you'll work harder for it. The book explains every concept before it uses one, and every terminal example includes its output, but it assumes you're comfortable with a command line and not frightened by the occasional configuration file.

What This Book Is Not

It is not a FreeBSD handbook. The FreeBSD Handbook exists, it is comprehensive, it is free, and it is maintained by the same community that maintains the operating system itself. This book does not attempt to replace it. What it does, however, is show you how to do things: every terminal example is real, every command works on FreeBSD 15, and Part IV walks you through a complete server build from installation to monitoring. You can read this book as an argument for integrated design, or you can follow along with a terminal open. Preferably both.

It is not a Linux survival guide, though Linux appears in every chapter. The comparisons are fair, the criticisms are specific, and nobody's intelligence is insulted. Linux is a remarkable achievement. The people who build and maintain it deserve respect, and they get it here. What they've built, however, is an assembled system, and this book argues that integrated systems solve certain problems more cleanly. That's a statement about architecture, not about competence.

It is not neutral. The book has a position and is rather upfront about it. But it earns its conclusions through evidence, not assertion. If you finish the last chapter unconvinced, you will at least know precisely what you're unconvinced by, and why.

Conventions

A word about the examples in this book.

Every server, every user account, every database, and every container that appears in the terminal examples is named after characters and ships from Star Trek: The Next Generation. Life is too short for `server1`, and `example.com` has suffered enough.

Our primary example server is called `enterprise`. You'll meet it in Chapter 1 and it will accompany us through every hands-on example until the final chapter, where you'll build your own. The administrator account is `picard` (one does not argue with the captain). The database is `starfleet_db`. The containers (or rather, jails) are named after ship compartments: `cargo-bay-1`, `shuttle-bay`, `astrometrics`.

If you recognise the references, splendid. If you don't, they're simply consistent, memorable names that are impossible to confuse with default system values. Nobody will ever wonder whether `enterprise` is a hostname they ought to have or one the book invented.

Terminal examples look like this:

```
root@enterprise:~ # bectl list
BE          Active Mountpoint Space  Created
default     NR     /          2.1G   2025-11-20 14:30
```

The prompt tells you which user (`root` or `picard`), which host (`enterprise`), and which directory. The output is always included. Nothing is left to imagination or optimism.

When the book shows a configuration file, it shows the entire relevant section, not a fragment surrounded by ellipses and a cheerful "the rest is left as an exercise." Exercises are for textbooks. This is a book for people who need things to work.

When a value in a code example is specific to your system (a key you generated, a path on your machine, a token from your provider), it appears as `...description...` with triple dots on each side. You will never see a fake-realistic placeholder that could be mistaken for a real value.

How to Read This Book

The book is structured in four parts, and they are meant to be read in order.

Part I (Chapters 1 to 3) establishes the problem: why modern server infrastructure feels more complex than it needs to be, and what the difference between an assembled and an integrated operating system actually means in practice.

Part II (Chapters 4 to 11) is the evidence. Eight chapters, each examining a specific administrative task (containers, upgrades, storage, firewalls, packages, init systems, networking, documentation) and showing how the same problem is solved on an assembled system and on an integrated one. The pattern repeats deliberately. By Chapter 8, you'll see it coming. That's the point.

Part III (Chapters 12 to 15) steps back and extracts the principles: why the integrated model produces these results, what it teaches about software architecture in general, and what the demoscene (yes, the demoscene) has to do with operating system design.

Part IV (Chapters 16 to 22) is the hands-on section. You install FreeBSD, build a web stack, run services in jails, perform a major version upgrade, and monitor it all with tools that ship in the base system. Everything the previous fifteen chapters described, you'll do yourself.

If you're in a hurry, you can skip to Part IV and build first, then return to Parts I through III to understand why it all works the way it does. The book won't complain. But it was written front to back, and the experience is better that way.

Acknowledgements

This book exists because someone on LinkedIn, in response to one of my technical posts, wrote: "I need a book. Yours. All notes like that in one place. Where to send money?"

I thought about it for a week. Then I started writing.

The FreeBSD community has been extraordinarily generous with their knowledge, their code, and their patience with questions. The OpenZFS developers have built something that deserves the word "engineering" without qualification. Michael

W Lucas and Allan Jude have written the definitive references on FreeBSD administration, and this book stands on their shoulders (whilst trying not to step on their toes).

To the OpenBSD and NetBSD communities: you build exceptional systems, each with its own focus and its own strengths. This book chose FreeBSD not because the others are lesser, but because its scope and ecosystem make it the most natural counterpart to Linux for general-purpose server work. The BSD family deserves more than one book, and the other two deserve theirs.

To the Linux community: thank you. Seriously. You built the system that taught me what I know, and the system whose limitations showed me what I wanted. Both contributions were essential.

To my wife Birgit and my daughter Ronja: you lived with a man who spent his evenings and weekends at a keyboard, writing about operating systems instead of being fully present. Your patience, your encouragement, and your willingness to tolerate yet another sentence beginning with "on FreeBSD, however" made this book possible. The debt is considerable, and I intend to repay it in cooking and undivided attention.

To the reader: you're holding a book that argues integrated systems are worth your attention. Give it until Chapter 5. If the evidence doesn't convince you by then, the remaining seventeen chapters are unlikely to change your mind, but they might make for an entertaining argument at the pub.

Let's begin.

Contents

Part III: The Pattern

Part IV: The Practice

Appendices

Downloads

`github.com/Vossscher-Verlag/book-integrated-by-design-downloads`

Corrections & Suggestions

`github.com/Vossscher-Verlag/book-integrated-by-design-downloads/issues/new?template=correction.yml`

Technical books rarely treat their readers as collaborators, which strikes me as a waste. The author is only human, and a dozen eyes catch what two do not. Printings update live, so a correction you spot today reaches the next reader almost effortlessly. Found a typo, an outdated command, or a sentence that reads as though translated from the original Martian? Open an issue. A short form asks for the page, the current wording, and your suggestion. Contributors are named near the passage they improved, and from the second edition onwards a dedicated appendix lists them with the pages they touched.

CHAPTER ONE

Assembled vs Integrated

You have two machines. Same hardware, same task: a web server with a database behind it. On the first, you install a Linux distribution. On the second, you install FreeBSD. An hour later, both serve pages. Both answer queries. Both, by any reasonable measure, work.

But something felt different. On the first machine, you chose a distribution, which chose an init system, which chose a service manager, which came with a set of opinions about how services ought to be supervised. You installed a firewall, selected from several available tools (none of which the distribution seemed fully committed to), wrote rules in a syntax that might change with the next major release, and enabled it via the init system you didn't choose. You installed a package manager (or rather, it came pre-installed, along with a second one for containerised applications, and possibly a third for language-specific dependencies). You configured networking with a tool that abstracts the tool that replaced the tool that originally shipped with the kernel project.

On the second machine, you typed `pkg install nginx postgresql16`, edited two configuration files, added two lines to `/etc/rc.conf`, and started the services. Or simply rebooted. The same kernel that started the machine started the services. The same documentation that described the network stack described the firewall, written by the same team, in the same voice, with the same assumptions about what the reader already knows. The package manager was one tool with one configuration format, and it knew about every piece of software on the system because it was the only thing that had ever installed any.

> **C. Lechat**
>
> Coming from Gentoo? FreeBSD's source tree supports compiling everything yourself with precisely the options you want, out of the box. `make buildworld` and `make buildkernel` compile the entire operating system from source, with your parameters, on your hardware. No overlay, no ebuild infrastructure, no Portage. Just `make`.

The first machine was assembled. The second was integrated. That distinction is the subject of this book, and this chapter lays the foundation for everything that follows.

What "Assembled" Actually Means

The word is chosen deliberately. An assembled system is not a bad system. It is a system composed of parts that were designed independently, by different teams, with different priorities, on different schedules, and brought together after the fact by a third party.

Consider what a modern Linux distribution contains. At its core sits the Linux kernel, maintained by Linus Torvalds and several thousand contributors. The kernel is a remarkable piece of engineering: modular, portable, and ferociously well-tested. It is also, by deliberate choice, only a kernel. It boots the hardware, manages processes and memory, provides device drivers and filesystems, and stops there. It does not ship a shell. It does not ship `ls`. It does not, on its own, know how to mount a root filesystem and present you with a login prompt.

For that, you need a userland: the collection of commands, libraries, and utilities that turn a running kernel into something a human being can operate. On Linux, the userland comes from a separate project. Historically, that project was GNU (the "GNU's Not Unix" initiative launched by Richard Stallman in 1983, which spent a decade building everything an operating system needs except the one thing it was named after). Today, many GNU/Linux distributions also include components from the systemd project, from the Busybox project, from the musl C library, or from various other sources. Android/Linux takes the principle to its logical extreme: a Linux kernel with no GNU components at all, running Google's own C library, its own init process, and its own application framework, assembled from entirely different parts, for entirely different purposes. The point is not which components a distribution selects. The point is that it selects them at all.

> **C. Lechat**
>
> Android/Linux is assembled twice, and that matters. Google assembled a system from a Linux kernel, a custom C library (Bionic), a custom init system, and a Java-based application framework, none of which came from the GNU project. The result was a coherent product, at least in Google's own hands. But the Android Open Source Project was then taken by dozens of hardware manufacturers, each of whom assembled their own variant: different drivers, different launchers, different update schedules, different security patch timelines. Samsung ships one Android/Linux. Xiaomi ships another. The fragmentation that GNU/Linux distributions experience at the system level, Android/Linux reproduces at the device level. Assembly, it turns out, compounds.

For GNU/Linux, the init system, the process that starts first and supervises everything else, is another independent project. On most distributions today, that project is systemd, maintained by Lennart Poettering and a team at Red Hat (now IBM). systemd replaced the earlier SysV init scripts (though not everywhere; Slackware, for instance, still uses BSD-style init scripts to this day), which were themselves a convention rather than a project, assembled from shell scripts that varied between distributions. Whether systemd was an improvement is a discussion that has generated more heat than the servers it manages. What matters here is that the init system is maintained by a different team than the kernel, with different release cycles, different design goals, and a different idea of what an init system ought to do.

But a system must also boot before init can supervise anything. One of the most common bootloaders is GRUB, from yet another project, though some distributions prefer systemd-boot, others still support syslinux, and LILO served faithfully for years before any of them. The firewall tooling might be iptables (deprecated but present), nftables (its successor), firewalld (a management layer on top of nftables), or ufw (a different management layer, preferred by Ubuntu). Networking is managed by NetworkManager, or netplan, or systemd-networkd, depending on the distribution and the phase of the moon.

A distribution takes these projects, patches them where necessary, configures them to work together, tests the combination (to varying degrees), and ships the result with a package manager, a release cycle, and a name. Debian does this. Red Hat does it. SUSE, Arch, Ubuntu, Gentoo, and several hundred others do it too. Each assembles a slightly different selection of the same parts, with slightly different defaults, slightly different directory layouts, and slightly different opinions about where configuration files belong.

A newer generation of distributions (NixOS, Guix System, and to some extent Silverblue) attempts to solve the assembly problem from a different angle: declarative, reproducible system descriptions where the entire configuration is derived from a single file. These are genuinely interesting projects, and they deserve serious attention. But they address reproducibility, not integration. The components are still independent projects, assembled by a package manager (albeit a remarkably clever one). The seams are managed, not removed. This book's comparison is between assembled and integrated systems, and for the assembled side, the six reference systems on the timeline above represent the mainstream that most administrators actually encounter.

> **C. Lechat**
>
> NixOS deserves a fair hearing. It replaces the traditional package manager with a purely functional approach. Every package is built in isolation, with its dependencies pinned by cryptographic hash. The entire system (packages, services, users, firewall rules) is described in a single file, `/etc/nixos/configuration.nix`, and any change produces a new, atomic system generation that can be rolled back in seconds. It is, in many ways, the most rigorous attempt to tame the assembly problem without abandoning the assembled model. The trade-off is complexity: the Nix language is its own learning curve, the store can consume considerable disc space, and debugging a failed build sometimes requires understanding the functional abstractions rather than the software itself. For the administrator who invests the time, NixOS offers guarantees that no traditional distribution can match. For the purposes of this book, however, it remains an assembled system, one that has found an elegant way to describe its assembly.

The result is functional. Often impressively so. But it is assembled, and the seams show.

What "Integrated" Actually Means

An integrated system takes a different approach. It does not select components. It builds them. FreeBSD is not alone in this. The BSDs all follow the integrated model, and outside the Unix world, systems like QNX and the late BeOS took the same approach, as does Apple's Darwin, the open-source foundation beneath macOS. Integration is not a curiosity. It is how most operating systems were built before Linux normalised the alternative.

This book focuses on FreeBSD, so let us look at what integration means in practice. FreeBSD maintains a single source repository that contains the kernel, the userland (including the shell, the core utilities, the C library, the network stack, and the bootloader), and the documentation. When you install FreeBSD, you are installing a product that was designed, built, tested, and released as a unit. In a release, the kernel and the C library are compiled from the same source tree on the same day. The manual pages were updated in the same commit that changed the behaviour they describe. The default firewall is `pf`, though `ipfw` and `ipf` ship in the base system as well: three options, but all maintained by the same project, all documented in the same tree, all guaranteed to work with the same kernel.

This does not mean FreeBSD contains everything. It emphatically does not. The base system is deliberately minimal: an operating system, not a software collection. Third-party software (nginx, PostgreSQL, Python, and roughly 35,000 other packages) is installed through the ports and packages system, which is maintained separately from the base. The distinction is architectural. The operating system is one thing. The applications you run on it are another. The boundary between them is clear, intentional, and consistently enforced.

This is not a minor organisational detail. It is a design philosophy, and its consequences are visible in every chapter of this book.

How Linux Became Assembled

Linux was never designed to be an operating system in the traditional sense. It was designed to be a kernel, and it became an operating system by accretion.

The story begins in 1991, when Linus Torvalds, a twenty-one-year-old student at the University of Helsinki, posted a message to the comp.os.minix newsgroup announcing a small hobby project. He had written a kernel that ran on i386 hardware. It could multitask. It had a terminal driver. It was, by his own admission, "nothing professional," and he doubted it would ever support anything beyond AT hard discs and his own particular hardware configuration.

What it did not have was a userland. No shell, no `ls`, no `cp`, no compiler. Torvalds solved this by the most pragmatic means available: he used GNU. The GNU project had spent eight years building a complete Unix-compatible userland (compiler, debugger, shell, core utilities, C library) without ever completing a kernel of its own. Torvalds provided the kernel; GNU provided everything else. The combination (GNU/Linux, as it is properly known) was never planned, never coordinated, and never governed by a single authority. It simply worked, in the way that open-source projects often do: someone builds a piece, someone else builds an adjacent piece, and a third person discovers they fit together.

Reference Systems

Distributions emerged to make the combination installable. Slackware in 1993, Debian in 1993, Red Hat in 1994. Each distribution made choices: which version of the kernel, which version of the GNU tools, which init system, which package format, which directory layout, which configuration file format and location, and which default users and groups. These choices were reasonable at the time and are reasonable now. But they are choices, and different distributions make different ones, and the result is a family of systems that share a kernel and, loosely, a userland, but agree on remarkably little else.

The init system illustrates this perfectly. Early distributions used SysV-style init scripts: shell scripts in `/etc/init.d/`, symlinked into runlevel directories, executed in numerical order at boot. The mechanism was simple and transparent, but also fragile, slow, and difficult to parallelise. When systemd arrived in 2010, it replaced not just the init scripts but the entire supervision model, the logging infrastructure, the device management, the network configuration, the hostname resolution, and a steadily expanding list of system services. Some distributions adopted it immediately. Others resisted for years. Some, like Devuan, exist solely to provide a systemd-free Debian. The ecosystem did not agree on an init system; it agreed to disagree, and the disagreement persists.

None of this is accidental. It is the natural consequence of a system whose components are independent projects with independent governance. No single authority decides what "Linux" is. The Linux Foundation manages the kernel trademark. Distributions manage the integration. Nobody manages the seams. One occasionally wonders what an integrated project could achieve with even a fraction of that collective effort, but that is a question for a different book.

How FreeBSD Became Integrated

FreeBSD's history runs through a different channel entirely.

In 1969, Ken Thompson and Dennis Ritchie created Unix at Bell Labs. By the mid-1970s, the University of California at Berkeley had received a copy of the source code and begun modifying it extensively. The result was the Berkeley Software Distribution, BSD, which over the following decade contributed the TCP/IP networking stack (the one the internet runs on), the Fast File System, the `vi` editor, the C shell, and a substantial portion of the utilities that Unix users still rely on today.

The Berkeley work was done by the Computer Systems Research Group, CSRG. Critically, CSRG maintained BSD as a single system: kernel modifications, userland changes, and new utilities were developed, tested, and distributed together. When you received a BSD tape, you received a functioning system, not a collection of independent projects. This was not a philosophical statement. It was simply how operating systems were developed in the 1970s and 1980s: one team, one repository, one release.

When CSRG closed its doors in 1995, the BSD code had already been released under a permissive licence, and three projects had formed to continue the work: FreeBSD (focused on performance and broad hardware support for the i386 platform), NetBSD (focused on portability across architectures), and OpenBSD (focused on security and correctness). Each inherited the integrated model. Each maintains a single source tree containing kernel and userland. Each releases the base system as a unit.

FreeBSD, the subject of this book, has continued the tradition since its first release in November 1993. The project's source repository at `src.freebsd.org` contains the kernel, the bootloader, the C library, the shell, the network utilities, the firewall, the documentation, and the build infrastructure to compile all of it into a running system with a single `make buildworld`. When a developer changes the behaviour of a system call, the manual page is updated in the same commit. When a new feature is added to the firewall, it is tested against the same kernel it will ship with. The release engineering team builds the entire system from a single source tree and tests it as a single product.

The result is not merely a matter of tidiness. It is a structural difference that affects reliability, predictability, and the daily experience of administering the system. The rest of this book will demonstrate how, one topic at a time.

What the Difference Looks Like in Practice

Abstractions are useful, but examples are better. Consider a task every server administrator performs: configuring a network interface.

On FreeBSD, you edit `/etc/rc.conf`:

```
ifconfig_em0="inet 192.168.1.10 netmask 255.255.255.0"
defaultrouter="192.168.1.1"
```

You reboot, or you run `service netif restart && service routing restart`. The interface is configured. The syntax has not changed in any meaningful way since the 1990s. The file is documented in `rc.conf(5)`, which ships with the system and is accurate because it is maintained in the same source tree as the code that reads the file.

On Linux, the answer depends. If you run Debian, you edit `/etc/network/interfaces` (unless you've installed NetworkManager, in which case you use `nmcli` or `nmtui` or the GNOME control panel, or you edit files in `/etc/NetworkManager/`). If you run Ubuntu 18.04 or later, you might use netplan, which generates configuration for either NetworkManager or systemd-networkd, depending on a YAML file that describes your intent. If you run Red Hat or CentOS, you used to edit files in `/etc/sysconfig/network-scripts/`, but recent versions prefer NetworkManager with `nmcli`, and the legacy scripts may or may not work depending on the release. If you run Arch, you might use systemd-networkd, or NetworkManager, or plain `iproute2` commands in a script. If you run Slackware, you edit shell scripts in `/etc/rc.d/` directly, no abstraction layer at all. If you run Gentoo, you use OpenRC's `net` scripts with their own configuration syntax in `/etc/conf.d/`.

The network is configured either way. The packets arrive at the same destination. But one administrator learned one method and can apply it to every FreeBSD machine built in the last three decades. The other administrator must determine, for each machine, which distribution it runs, which network management tool that distribution prefers, and which version of that tool is installed, before writing a single line of configuration.

This is not a complaint about any individual tool. NetworkManager is competent software. netplan is a reasonable abstraction. systemd-networkd works. The problem is not quality. The problem is quantity: three tools that solve the same problem, none of them universal, each with its own syntax, its own file locations, and its own idea of what the administrator ought to know.

One might call this choice. One might also call it the cost of assembly. In a business context, every additional tool an administrator must learn, every incompatibility between distributions, and every hour spent determining which method applies to which machine is a cost that rarely appears in any budget, but accumulates all the same.

Consider another example: the `ifconfig` command itself.

On FreeBSD, `ifconfig` is part of the base system. It configures network interfaces. It has done so since the system's inception, and it will continue to do so. Its behaviour is documented in `ifconfig(8)`, and the documentation is accurate.

On Linux, `ifconfig` was part of the `net-tools` package, which was declared deprecated in favour of the `iproute2` suite. The replacement command is `ip`, which is powerful, flexible, and entirely different in syntax. Whether your system has `ifconfig` installed depends on the distribution and its vintage. Whether you should use it depends on whom you ask. The deprecation was announced over a decade ago, but `ifconfig` still ships on many systems because too many scripts depend on it, and removing it would break things.

On FreeBSD, there is one tool, and it works. On Linux, there are two (or arguably three, if you count `ss` as a partial replacement for `netstat`), and the community has spent fifteen years not quite completing the transition between them.

This pattern (one tool on FreeBSD, several on Linux, with the community unable to fully retire the old ones) will repeat in every chapter of this book. It is not a design flaw in any individual project. It is the structural consequence of a system where no single team owns the whole.

The Thesis

This book makes a straightforward claim: integrated systems solve problems once. Assembled systems solve them repeatedly.

When one team maintains the kernel and the userland and the firewall and the documentation, decisions are made once, in one place, with full visibility of their consequences. When a hundred teams maintain a hundred projects and a distribution stitches them together, every decision is a negotiation, every interface is a contract, and every upgrade is a test of whether the contracts still hold.

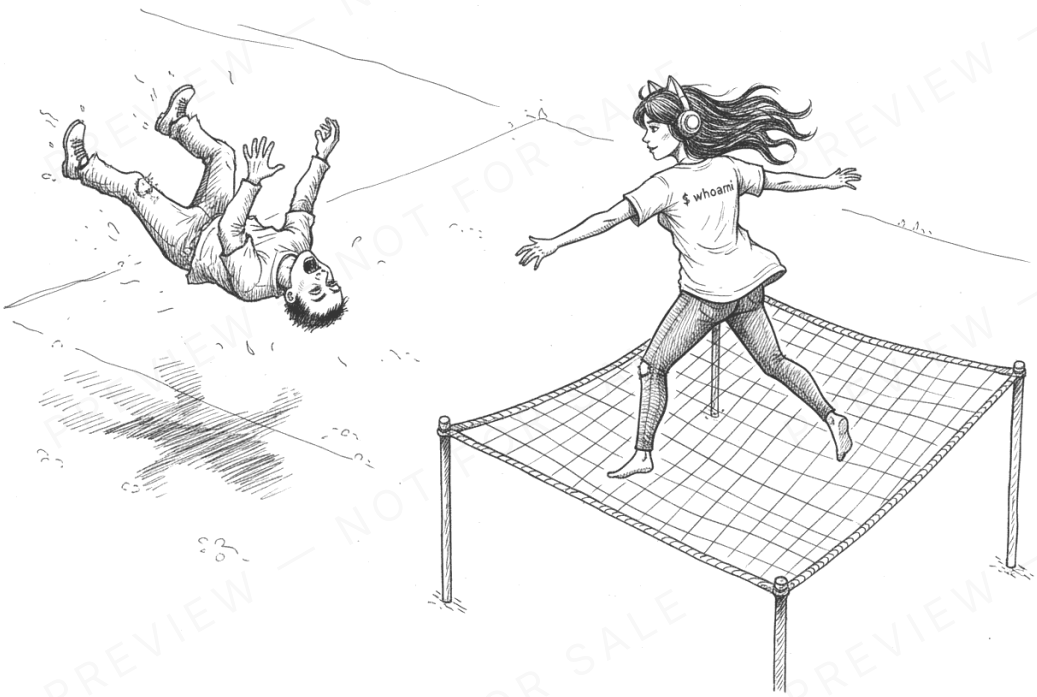
This is not a moral judgement. Almost nobody set out to build a fragmented system. The Linux ecosystem grew the way it did because open-source development encourages independent initiative, and independent initiative produces independent projects, and independent projects produce independent interfaces. The assembled model is a natural consequence of how the work was organised. It is also, this book will argue, a model with costs that are rarely accounted for and even more rarely questioned.

The following chapters will examine those costs, one domain at a time. Containers. Upgrades. Storage. Firewalls. Packages. Init systems. Networking. Documentation. In each case, the pattern is the same: a problem that every server administrator faces, the Linux solution (assembled from parts), the FreeBSD solution (built as a whole), and the practical difference between them.

By the end, the reader will have enough evidence to draw a conclusion. The book has its own, and makes no secret of it. But it would rather show than tell, and it starts now.

CHAPTER FIVE

The Safety Net



You are about to upgrade a production server. The packages need updating, the kernel has a security patch, and three of your colleagues have already asked whether you have "done it yet." You have read the release notes. You have checked the compat-

ibility matrix. You have mentally rehearsed the rollback procedure, which involves booting from a USB stick, mounting the root filesystem by hand, and hoping you remember which partition was which.

We have all been there. That peculiar silence before pressing Enter on `apt upgrade`, when the terminal feels less like a tool and more like a confession booth.

What if the operating system simply handled this for you?

What a Boot Environment Is

Before the how, the what. A boot environment is a complete, bootable copy of your operating system's root filesystem. Not a backup. Not an image file on an external drive. A live, selectable state that your bootloader knows about and can switch to in seconds.

Think of it as version control for your entire operating system. You are running version 15.0. You create a boot environment, upgrade to 15.1, and if something breaks, you select the previous environment from the boot menu. Thirty seconds later, you are back where you started. No USB stick, no rescue mode, no archaeology.

This concept did not originate on FreeBSD. Sun Microsystems introduced it in OpenSolaris in 2008 with a tool called `beadm`, building on the Live Upgrade commands (`lucreate`, `luactivate`) that had shipped with Solaris 10. The idea was straightforward: if your filesystem supports instant, zero-cost snapshots, why not use them to make operating system upgrades reversible? Jeff Bonwick and his colleagues at Sun had, in hindsight, answered a question the rest of the industry would spend the next fifteen years pretending did not exist.

When Oracle acquired Sun in 2010 and began quietly suffocating Solaris, the illumos community carried the torch. FreeBSD, which had imported ZFS into its kernel with version 7.0 in 2008, eventually built its own implementation: `bectl`, written by Kyle Kneitinger as a Google Summer of Code project in 2017, mentored by Allan Jude, and added to the base system in FreeBSD 12.0 (December 2018). Not as a package. Not as a third-party add-on. As part of the operating system, maintained by the same team that maintains the kernel, the bootloader, and ZFS itself.

That last detail matters more than it appears to.

How It Works on FreeBSD

Our example server `enterprise` runs FreeBSD 15.0 (released December 2025), root-on-ZFS, installed using the traditional Distribution Sets method. Security patches have arrived. Several packages need updating. Here is the entire procedure:

```
picard@enterprise:~ # bectl create pre-upgrade
picard@enterprise:~ # freebsd-update fetch install
picard@enterprise:~ # pkg upgrade
```

Three commands. The first creates a boot environment named `pre-upgrade`, which is a ZFS clone of the current root dataset. The clone is instantaneous, regardless of how large the filesystem is, and consumes precisely 8 kilobytes at creation (only the metadata for the new dataset). ZFS's copy-on-write semantics mean no data is copied. The existing blocks are shared. Only when a block is modified after the clone does ZFS write it to a new location, leaving the old block untouched.

The second and third commands do whatever they do. If they break something, the rollback is two commands and a reboot:

```
picard@enterprise:~ # bectl activate pre-upgrade
picard@enterprise:~ # reboot
```

But here is the part that quietly delights: the first command was optional. Since FreeBSD 12.3, `freebsd-update` detects that the root filesystem is ZFS and creates a boot environment automatically before applying patches. The safety net deploys itself.

```
picard@enterprise:~ # bectl list
BE                    Active Mountpoint Space Created
15.0-RELEASE-p3...    -      -          1.3G  2026-02-09 10:15
default               NR     /          2.1G  2025-11-20 14:30
```

`NR` means this environment is active Now and will be active on next Reboot. The other entry, with its verbose timestamp name, was created by `freebsd-update` without being asked. If the upgrade had failed, selecting it from the boot menu would have restored the pre-upgrade state in the time it takes the machine to POST.

> **C. Lechat**
>
> The Lua bootloader is worth a closer look. Introduced in FreeBSD 12.0 (the same release that shipped `bectl`), it understands boot environments natively. Option 8 in the boot menu lists all available environments. One team, one release, everything fits.

> **C. Lechat**
>
> A note on what comes next: FreeBSD 15.0 introduces pkgbase as a technology preview: the entire base system, managed through `pkg(8)` instead of `freebsd-update`. Systems installed with the new Packages method use `pkg upgrade` for base system updates. `freebsd-update` remains fully supported for traditional installations throughout the 15.x branch. The transition is expected to complete in FreeBSD 16.

For the genuinely cautious, `bectl activate -t` provides temporary activation: the selected environment boots once. If it hangs, panics, or otherwise misbehaves, the next power cycle automatically reverts to the previous default. Even a completely unresponsive system recovers without intervention.

I have run this workflow on production servers for years. The upgrade has never once required a USB stick, a rescue environment, or a prayer. The few times something did break (a port that had not been updated for a new library version, a configuration file that needed manual merging), the rollback took less time than it takes to make tea.

How It Works on Linux

You want the same experience on Linux. A reasonable desire. Let us walk through it.

Step 1: Install ZFS.

ZFS is not in the Linux kernel. It cannot be, for licensing reasons that have kept lawyers employed since 2005. ZFS is released under Sun's Common Development and Distribution Licence (CDDL). The Linux kernel is released under the GNU General Public Licence, version 2. Whether these two licences are compatible when code is compiled into a single binary is, to put it diplomatically, disputed.

The Software Freedom Conservancy published a detailed legal analysis in February 2016, concluding that distributing ZFS as a compiled kernel module constitutes a GPL violation. Canonical disagreed and shipped ZFS support in Ubuntu 16.04 regardless. Oracle, who owns the ZFS copyright, has said nothing publicly, which is either reassuring or terrifying depending on your familiarity with Oracle's litigation history.

In practice, ZFS lives outside the kernel tree. You install it via DKMS (Dynamic Kernel Module Support), a framework originally written by Dell's Linux Engineering Team in 2003 to distribute driver fixes to customers. DKMS watches for kernel updates and automatically recompiles out-of-tree modules against the new kernel headers. When it works, you do not notice it. When it does not, you do not boot.

Step 2: Survive the kernel updates.

DKMS rebuilds the ZFS module every time the kernel changes. If the new kernel introduces breaking changes before OpenZFS has been updated to support them, the build fails. If your root filesystem is on ZFS, a failed DKMS build means you cannot mount root, which means you cannot boot.

This is not theoretical. Documented failures include kernel 6.2.8 (March 2023, OpenZFS issue #14658), kernel 6.4 (July 2023, issue #15151), and kernel 6.12 (2025, issue #17639). Each incident followed the same pattern: the kernel updated, the DKMS build failed, and root-on-ZFS systems were left staring at an initramfs prompt until someone manually downgraded the kernel from a rescue environment.

Linus Torvalds addressed this in January 2020 with characteristic directness: "Don't use ZFS. It's that simple. It was always more of a buzzword than anything else, I feel, and the licensing issues just make it a non-starter for me."

One might respectfully disagree with the "buzzword" assessment. The licensing observation is rather more difficult to argue with.

Step 3: Configure root-on-ZFS.

Assuming DKMS cooperates, you need your root filesystem on ZFS. The Linux installer probably does not support this. Ubuntu offered root-on-ZFS from 19.10 onwards, removed it in 23.04 when the new Flutter-based installer shipped without ZFS support, and partially restored it in 23.10 as "experimental." Other distributions leave it as an exercise for the reader.

Step 4: Find a boot environment manager.

`bectl` does not exist on Linux. It is a FreeBSD base system utility. On Linux, you have several options, all third-party:

ZFSBootMenu, released in June 2019 and currently the most mature option, replaces GRUB entirely with a boot environment-aware loader built on top of ZFS. It works well, but replacing your bootloader is not a decision one makes lightly.

`zectl`, written by John Ramsden initially in Python (as `zedenv`) and then rewritten in C using libzfs directly, inspired by FreeBSD's `bectl`. Community-maintained. Plugins for systemd-boot and GRUB.

Various `beadm` ports of differing vintage and maintenance status.

Step 5: Consider what Ubuntu tried.

In 2019, Canonical launched `zsys`, an ambitious daemon that would manage ZFS boot environments automatically on Ubuntu. Automatic snapshots before every `apt` operation. Automatic boot environment creation. Automatic garbage collection. On paper, exactly what FreeBSD's `bectl` plus `freebsd-update` provide.

Development ceased in 2021. The GitHub repository's most instructive artefact is issue #230, titled with admirable brevity: "Don't use ZSYS."

The reasons were numerous: performance issues with large numbers of automatic snapshots, complexity in the GRUB integration, a critical bug where the garbage collector deleted user datasets (issue #218, which the community described as "putting a system at risk of unbootable self-destruction"), and the fundamental difficulty of bolting a coherent boot environment system onto an operating system that was not designed with one in mind. Didier Roche, effectively the sole developer, moved to other priorities at Canonical. Ubuntu 22.04 quietly stopped installing `zsys` by default. By 23.04, the new installer had dropped ZFS support entirely. It returned in 23.10, without `zsys`.

One developer. One daemon. One critical data-loss bug. One quiet death.

The Same Filesystem, a Rather Different Experience

Here is what makes this chapter's argument particularly clean: the ZFS commands are identical on both platforms.

```
picard@enterprise:~ # zfs snapshot zroot/ROOT/default@before-upgrade
picard@enterprise:~ # zfs rollback zroot/ROOT/default@before-upgrade
```

These commands work the same way on FreeBSD and on Linux. They use the same codebase (OpenZFS, unified since the 2.0 release in December 2020). The snapshots use the same copy-on-write mechanism: when you create a snapshot, ZFS pins the current block pointers. No data is copied. The snapshot is instantaneous and occupies zero additional bytes at the moment of creation. As data changes afterwards, only the modified blocks consume new space, because ZFS never overwrites existing data. It writes to a new location and updates the pointer. The old blocks, still referenced by the snapshot, remain untouched.

Brilliant engineering, regardless of which operating system surrounds it.

The difference is everything else.

On FreeBSD, ZFS is compiled into the kernel. `bectl` is in `/usr/sbin`, shipped with the base system, maintained by the same team. The Lua bootloader understands boot environments without configuration. `freebsd-update` creates safety nets without being asked.

On Linux, ZFS is an out-of-tree module held together by DKMS and legal ambiguity. The boot environment manager is third-party. The bootloader integration is a project in itself. And the one serious attempt at automation lasted two years before being quietly shelved.

Same filesystem. Same commands. Same copy-on-write. Same snapshots.

One system where someone thought about how all the pieces fit together, and one where they did not.

But Nobody Does It That Way

A fair objection, and one that deserves a thorough answer.

The Linux ecosystem has not ignored the upgrade problem. It has, characteristically, produced half a dozen solutions, each addressing a different facet, each maintained by a different team, each requiring its own learning investment. The result is not a gap in capability but a landscape of alternatives so varied that choosing between them is itself a task.

Configuration management. Ansible (2012), Puppet (2005), Chef (2009), and Salt (2011) encode the desired state of a server in version-controlled code. When an upgrade breaks something, you revert the code and re-converge. The workflow is: diagnose the problem, identify the relevant playbook or manifest, revert it in git, re-run the tool, and wait for the system to converge. Depending on the scope, this takes minutes to tens of minutes, requires network access to download packages, and depends on the convergence logic modelling every relevant change. If the upgrade corrupted a binary or modified a database in ways the playbook does not describe, convergence fails silently or produces undefined results. There is no atomic undo. The tool tries to make reality match the description, but the description may be incomplete.

Infrastructure as code. Terraform (2014) and Pulumi (2018) take the logic further: do not fix the server, replace it. Destroy the broken infrastructure, recreate it from the previous state. This works, provided your infrastructure runs on a cloud provider, your state files are intact, and you can afford the downtime of provisioning a new machine. On a single bare-metal server, Terraform has nothing to offer.

Immutable distributions. Fedora CoreOS (2020), Flatcar Linux (2018), and Talos Linux (2019) treat the operating system as a read-only image. Updates are written to a passive partition; the system reboots into it; if the boot fails, it reverts to the previous partition. This is, architecturally, the closest Linux gets to boot environments on a conceptual level, and it works well. The trade-off: these distributions exist solely to run containers. There is no package manager, no customisation of the base system, no installing arbitrary software. Talos Linux has no SSH access and no shell. The operating system is a firmware, not a platform.

NixOS. The most intellectually ambitious attempt. Every system configuration is a Nix expression, evaluated into an immutable store. Each `nixos-rebuild switch` creates a new generation, atomically, and the bootloader lists all previous generations. Rollback is selecting the previous entry. The mechanism is elegant, the learning curve is vertical. Nix has its own functional language, its own package model (derivations, closures, flakes), its own module system, and a documentation ecosystem that assumes a familiarity with concepts most administrators have never encountered. First stable release: 2013. Community: devoted and growing. Adoption in production: still measured in percentages.

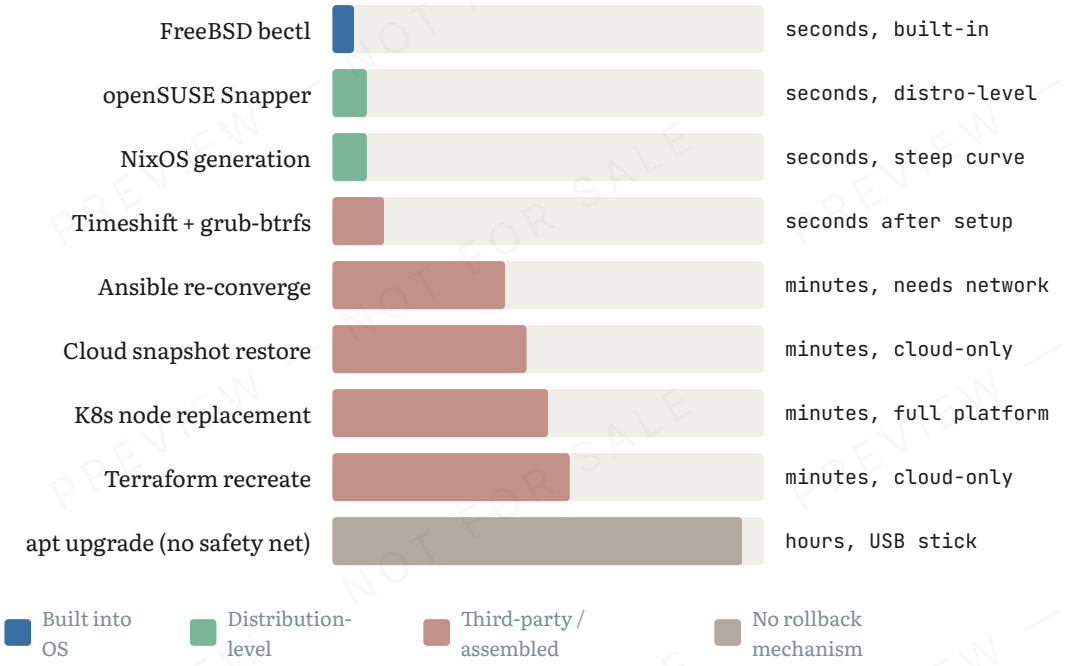
Cloud snapshots. AWS AMIs, DigitalOcean snapshots, Azure VM images. Create a snapshot before upgrading, restore it if the upgrade fails. The limitations: creating an AMI takes minutes, restoring means provisioning a new instance (new instance

ID, potential IP change, DNS propagation), the operation requires internet access to the cloud API, and the cost scales linearly with storage volume size. On a server in a cabinet, there is no cloud API to call.

Timeshift. A desktop-oriented tool, first released in 2013 by Tony George, adopted by Linux Mint in 2022. In Btrfs mode, it creates near-instant snapshots. But there is no boot menu integration without a separate project (`grub-btrfs` by Antynea). There is no automatic snapshot before `apt upgrade` without yet another project (`timeshift-autosnap-apt`). The pieces exist. They are maintained by different people, on different schedules, with different assumptions about filesystem layout.

The "just redeploy" philosophy. Kubernetes rolling updates, blue/green deployments, canary releases. The application runs in containers; the underlying OS is irrelevant; if a node needs upgrading, drain the workloads and replace it. This works brilliantly for stateless microservices at scale. It does not address the database server that cannot be trivially replaced, the single bare-metal machine running a business-critical application, or environments where internet access for pulling container images is not guaranteed. And the hidden cost is considerable: to "just redeploy" requires a container runtime, an orchestrator, a registry, a CI/CD pipeline, networking plugins, monitoring, and load balancing. A Civo survey found that 54 per cent of cloud developers reported Kubernetes complexity was slowing their organisation's adoption.

Recovering from a Failed Upgrade



Every one of these approaches works, within its constraints. Several are impressive engineering achievements. None of them is `bectl activate pre-upgrade && reboot`. And the comparison that matters most is this: what happens on a single server, bare metal, no cloud provider, no orchestration layer, no configuration management, when the admin types `upgrade` and the system breaks?

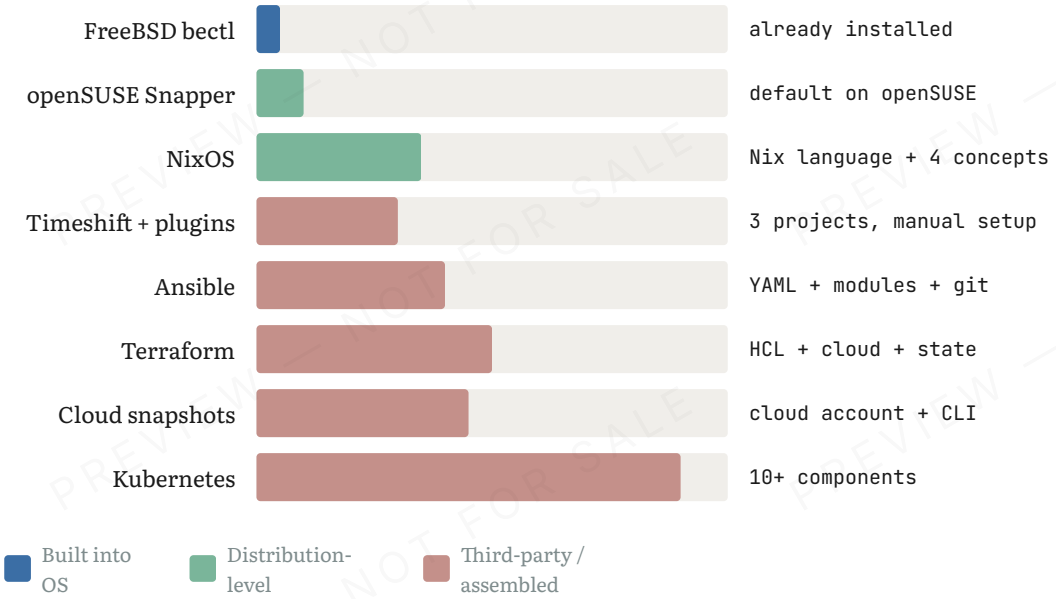
On FreeBSD, the admin selects the previous boot environment from the boot menu. Thirty seconds. No network. No USB. No external tooling.

On a standard Debian or RHEL installation without Btrfs, without ZFS, without Timeshift, without NixOS, without any of the above? The admin boots from a USB stick and hopes.

The Overhead of Alternatives

The approaches are not merely different in philosophy. They differ in what you must learn, install, and maintain before the safety net exists.

What You Need Before You Can Roll Back



Assembled Alternatives	FreeBSD bectl
– Configuration management: 3+ tools, 4 new concepts, no atomic undo	+ ZFS root (default since installer offered it)
– Infrastructure as code: 4+ tools, 5 new concepts, cloud-only	+ `bectl` in base system since 12.0
– Immutable OS: container-only, no package manager, no shell (Talos)	+ Lua bootloader in base system since 12.0
– NixOS: 1 tool but 4+ new concepts, steep learning curve	+ `freebsd-update` auto-creates BEs
– Timeshift + plugins: 3 separate projects, manual assembly	+ No network required for rollback
– Cloud snapshots: cloud-only, minutes of downtime, storage costs	+ No external tools required
– Kubernetes: 10+ components, entirely different operational model	+ Works on any hardware, any location

FreeBSD requires you to learn two concepts: ZFS datasets and boot environments. Both are documented in the FreeBSD Handbook, in the same chapter, maintained by the same project. The tool is one command with eight subcommands. The integration is automatic.

Every Linux alternative except openSUSE's Snapper requires assembling components from different sources. Ansible requires learning YAML, Jinja2 templates, module semantics, inventory management, and idempotency principles. Terraform requires HCL, providers, state management, modules, and a cloud account. NixOS requires a functional programming language, a novel package model, and a willingness to rethink how operating systems work. Each of these is a legitimate engineering discipline. None of them is "type three commands and the system catches you if you fall."

A Brief Word About Btrfs

A fair-minded reader will point out that Linux has its own copy-on-write filesystem: Btrfs. And openSUSE's integration of Btrfs with Snapper deserves genuine credit.

Snapper, developed by Arvin Schnell at SUSE and first shipped with openSUSE 12.1 in November 2011, creates automatic snapshots in pre/post pairs around every `zypper` transaction. GRUB is patched to offer a submenu for booting from snapshots. The rollback workflow is coherent: reboot, select the snapshot, verify, run `snapper rollback`, reboot again. On openSUSE, this works out of the box.

The caveats are worth mentioning. Btrfs's RAID5 and RAID6 implementations still carry an official data loss warning in the documentation, a caveat that has persisted for years. ZFS's equivalent (RAIDZ, RAIDZ2, RAIDZ3) has been production-ready since 2005. Btrfs has no equivalent to `zfs send` and `zfs receive` for efficient incremental replication between hosts. And while Btrfs reached general production readiness around 2015, ZFS had a decade's head start.

None of this makes Btrfs a bad filesystem. It makes it a younger one. And the openSUSE team deserves genuine credit for building what they have built.

But even in the best case, the Btrfs workflow on openSUSE is a distribution-level achievement. It is SUSE's integration work on top of a filesystem, a bootloader, and a snapshot manager that are all developed separately. The GRUB integration is openSUSE-specific; other distributions using Snapper do not get the boot-from-

snapshot feature without additional patching. The whole edifice is one team's excellent work within the assembled model. On FreeBSD, it is one team, one repository, one design, one release.

The Comparison

Aspect	FreeBSD `bectl`	openSUSE Snapper	NixOS	Ubuntu (no ZFS)
Rollback mechanism	ZFS boot environment	Btrfs snapshot	Nix store generation	None built-in
In base system	Yes	Yes (openSUSE only)	Yes	N/A
Boot menu integration	Native (Lua loader)	Patched GRUB	Native (GRUB/ systemd-boot)	N/A
Auto-snapshot before upgrade	Yes (`freebsd-update`)	Yes (`zypper`)	Yes (`nixos-rebuild`)	N/A
Rollback time	Seconds (reboot)	Seconds (reboot + command)	Seconds (reboot)	Hours (manual repair)
New concepts to learn	2	3	4+	N/A
Requires network for rollback	No	No	No	Often yes
Teams maintaining the stack	1 (FreeBSD)	2 (SUSE + GRUB)	1 (NixOS)	N/A

The table shows something that pure feature lists obscure: openSUSE and NixOS both provide working rollback, but through fundamentally different integration models. openSUSE achieves it through distribution-level engineering on top of sep-

arately maintained components. NixOS achieves it by reinventing the operating system from first principles. FreeBSD achieves it by having one team that builds the filesystem, the bootloader, the update tool, and the management utility together.

The Structural Explanation

Why does this difference exist?

Not because the Linux community failed to recognise the problem. The number of solutions listed in this chapter proves the opposite. Puppet appeared in 2005, Snapper in 2011, NixOS in 2013, Timeshift in 2013, Terraform in 2014, Kubernetes in 2015. The problem was recognised. Repeatedly. By different teams. In different years. With different architectural assumptions.

That is precisely the point.

On FreeBSD, the ZFS team, the bootloader team, the update tool team, and the `bectl` team are, to a considerable extent, the same people working in the same source tree. Kyle Kneitinger could write `bectl` because the ZFS integration, the bootloader, and the update tool were all visible in the same repository. Allan Jude could mentor the project because he understood all three layers. `freebsd-update` could auto-create boot environments because its maintainer could see how `bectl check` worked, and could rely on the Lua loader to display them. Nobody needed to file a cross-project feature request. Nobody needed to negotiate an API between independent teams with different release schedules.

On Linux, the OpenZFS project maintains ZFS. A separate team maintains the kernel that officially recommends against using ZFS. DKMS is maintained by yet another group. The bootloader (GRUB, or systemd-boot, or ZFSBootMenu) is a fourth project. The distribution's package manager is a fifth. Getting all of these to work together in a seamless, automated, "just press Enter and we'll catch you if you fall" experience requires either heroic integration work (which Canonical attempted and abandoned) or an administrator who is comfortable assembling the safety net by hand.

The filesystem is the same. What surrounds it is not.

The next chapter examines what happens when you look one layer deeper: at the storage system itself. The filesystem that makes boot environments possible has its own story, and it follows the same pattern.