



DEBIAN LINUX ADMINISTRATION
MASTERY SERIES

Debian 13.7 Linux Administration

A Complete Practical Guide to Server Management,
Networking, Security, Automation, and Troubleshooting



Server
Management



Networking



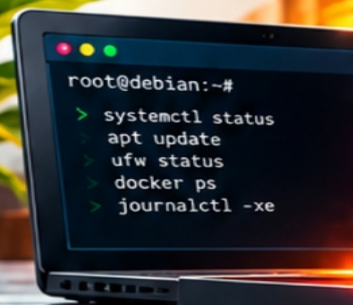
Security



Automation



Troubleshooting



George West

Debian 13.7 Linux Administration

**A Complete Practical Guide to Server
Management, Networking, Security, Automation,
and Troubleshooting**

George West

Copyright

© 2026 by the George West. All rights reserved.

No part of this publication may be reproduced, distributed, transmitted, stored in a retrieval system, or used in any form or by any means electronic, mechanical, photocopying, recording, scanning, or otherwise without the prior written permission of the copyright holder, except for brief quotations used in reviews, criticism, commentary, or other uses permitted by applicable copyright law.

This book is intended for educational and informational purposes. The author has made reasonable efforts to ensure that the information presented is accurate and useful at the time of publication. However, server software, operating systems, packages, network services, security technologies, configuration methods, and technical standards are continuously evolving.

The commands, scripts, configurations, deployment procedures, and troubleshooting techniques presented in this book are provided for educational and practical guidance. They should be evaluated and tested in an appropriate environment before being used on production systems.

Disclaimer

The author and publisher make no representations or warranties, express or implied, regarding the accuracy, completeness, reliability, suitability, or availability of the information contained in this publication.

Server administration involves potentially disruptive operations, including modifying system configuration, changing firewall rules, managing storage, installing services, changing permissions, modifying network settings, deploying applications, and performing system maintenance. Incorrect implementation may result in data loss, service interruption, security vulnerabilities, or system failure.

You are solely responsible for evaluating and validating any command, script, configuration, security recommendation, deployment procedure, or troubleshooting technique before applying it to your environment.

Always maintain current backups and, where appropriate, test configuration changes in a development, staging, or isolated environment before applying them to production systems.

The security practices described in this book are general educational guidance and should not be considered a complete security program, penetration test, vulnerability assessment, compliance certification, or substitute for professional security advice.

The author is not responsible for any loss, damage, downtime, data loss, security incident, configuration error, service interruption, or other consequence arising directly or indirectly from the application or misuse of information contained in this book.

Table of Contents

Introduction Building and Managing Debian 13.7 Servers

- The Role of Debian in Modern Server Infrastructure
- Server Administration vs. Desktop Linux Administration
- Planning a Debian Server Environment
- Physical, Virtual, VPS, and Cloud Deployment Options
- Server Administration Principles and Operational Workflows
- Building a Safe Testing and Production Environment

Chapter 1 Debian Server Installation and Initial Configuration

- 1.1 Preparing Hardware, Virtual Machines, and VPS Environments
- 1.2 Installing Debian 13.7 for Server Workloads
- 1.3 Partitioning, Filesystems, and Storage Planning
- 1.4 Initial User, Network, Hostname, and Time Configuration
- 1.5 Post-Installation Server Configuration and Validation

Chapter 2 Server Administration and Service Management

- 2.1 Managing Users, Groups, and Administrative Privileges
- 2.2 Package and Repository Management on Servers
- 2.3 Managing Processes and Systemd Services
- 2.4 Configuring Service Startup and Recovery
- 2.5 Managing Server Logs and Operational Information

Chapter 3 Debian Server Networking

- 3.1 Server Network Architecture and Configuration
- 3.2 Static IP Addresses, Routes, and DNS Configuration
- 3.3 Network Interfaces, Ports, and Connectivity
- 3.4 Network Diagnostics and Troubleshooting
- 3.5 Designing Reliable Server Network Configurations

Chapter 4 SSH and Remote Server Management

- 4.1 Deploying and Configuring OpenSSH
- 4.2 SSH Key-Based Authentication
- 4.3 Managing Administrative Access and Privileges
- 4.4 SSH Hardening and Attack Reduction
- 4.5 Secure Remote Administration Workflows

Chapter 5 Web and Application Server Administration

- 5.1 Deploying a Web Server on Debian
- 5.2 Configuring Nginx and Apache HTTP Server
- 5.3 Virtual Hosts, TLS, and HTTPS
- 5.4 Deploying Applications and Managing Services
- 5.5 Diagnosing Web Server and Application Problems

Chapter 6 Storage, Databases, and Server Data Management

- 6.1 Server Storage Architecture and Planning
- 6.2 Managing Disks, Filesystems, LVM, and Mounts
- 6.3 Deploying and Managing Database Services
- 6.4 Storage Monitoring, Capacity Planning, and Maintenance
- 6.5 Backup Strategies for Server Data

Chapter 7 Server Security and Hardening

- 7.1 Building a Debian Server Security Baseline
- 7.2 Firewall Configuration and Network Access Control
- 7.3 SSH, Account, and Privilege Hardening
- 7.4 Security Updates, Auditing, and Vulnerability Management
- 7.5 Intrusion Detection, Logging, and Security Monitoring

Chapter 8 Containers and Modern Debian Server Workloads

- 8.1 Understanding Containers and Containerized Applications
- 8.2 Installing and Configuring Docker on Debian
- 8.3 Managing Images, Containers, Networks, and Volumes
- 8.4 Running Multi-Container Applications
- 8.5 Container Security, Maintenance, and Troubleshooting

Chapter 9 Automation and Configuration Management

- 9.1 Automating Server Administration with Bash**
- 9.2 Scheduled Administration with cron and systemd Timers**
- 9.3 Automating Updates, Backups, and Maintenance**
- 9.4 Introduction to Configuration Management Concepts**
- 9.5 Building Repeatable Server Provisioning Workflows**

Chapter 10 Monitoring, Logging, and Performance Engineering

- 10.1 Server Monitoring Fundamentals**
- 10.2 CPU, Memory, Disk, and Network Monitoring**
- 10.3 Centralized and Persistent Log Management**
- 10.4 Performance Analysis and Bottleneck Identification**
- 10.5 Capacity Planning and Server Optimization**

Chapter 11 Backup, Recovery, and Troubleshooting

- 11.1 Designing a Reliable Debian Server Backup Strategy**
- 11.2 File, Configuration, and Database Backups**
- 11.3 System Recovery and Disaster Recovery Procedures**
- 11.4 Troubleshooting Boot, Network, Storage, and Service Failures**
- 11.5 Building a Structured Linux Troubleshooting Methodology**

Chapter 12 Production Debian Server Projects

- 12.1 Building a Secure Web Server**
- 12.2 Deploying a Database-Backed Application Server**
- 12.3 Building a Containerized Debian Application Environment**
- 12.4 Implementing Monitoring, Backups, and Automated Maintenance**
- 12.5 Performing a Complete Production Server Review**

Conclusion Operating Reliable Debian Servers

- From Initial Deployment to Production Operations
- Maintaining Security and Reliability
- Automating Repetitive Administration
- Developing Long-Term Server Management Practices
- Continuing Your Debian and Linux Administration Journey

Preface

Every production server tells a story about the decisions its administrator made before anything went wrong. The partition layout chosen during installation, the SSH configuration written on day one, the firewall rules drafted in a hurry before a deadline—these choices sit quietly in the background of a running system until the moment they matter. This book exists because that moment always comes, and because the gap between "knowing Linux" and "operating a Linux server responsibly" is wider than most learning material admits.

Debian 13.7 Linux Administration is the second book in the Debian Linux Administration Mastery Series. The first book, *Debian 13.6 Linux Mastery*, built the reader's command of the Debian operating system itself—its package management philosophy, its filesystem hierarchy, its init system, its user and permission model, and the broad landscape of tools that make Debian one of the most trusted distributions in the world. This book assumes that foundation. It does not re-teach what a shell is, how `apt` resolves dependencies, or what a `systemd` unit file looks like in its most basic form. Instead, it takes a reader who already understands Debian and asks a different question: what does it take to run Debian as a server that other people, applications, and businesses depend on?

That question has a very different shape than "how do I use Linux." A desktop installation that occasionally misbehaves is an inconvenience. A server that occasionally misbehaves is an incident—one with stakeholders, downtime, possibly data loss, and a post-mortem. Server administration is therefore not simply "Linux administration, but on a machine without a monitor." It is a discipline built around reliability, security, observability, and recoverability, practiced by people who have learned—often the hard way—that a system which merely "works" is not the same as a system that can be trusted.

This book was written to close that gap directly. It walks through the complete lifecycle of a Debian 13.7 server: from the moment hardware or virtual infrastructure is provisioned, through installation and hardening, into the ongoing work of running services, managing storage and databases,

securing remote access, automating maintenance, monitoring health, and critically preparing for the day something fails. Because something eventually does.

Who This Book Is For

This book is written for readers who are past the beginner stage of Linux but have not yet spent years running production infrastructure. That includes:

- Linux system administrators moving from desktop or workstation support into server operations
- Debian users who have completed foundational learning (such as Book 1 in this series) and want to apply that knowledge professionally
- DevOps and infrastructure engineers who need a systematic reference for Debian-specific server practices
- Web and application developers who are increasingly responsible for the servers their code runs on
- Students and self-taught professionals preparing for system administration, DevOps, or infrastructure engineering roles
- IT professionals supporting VPS, cloud, or on-premises Debian deployments who want to formalize ad hoc knowledge into a coherent operational practice

If you can already navigate a Linux shell, edit files with a text editor, understand basic permissions, and have some familiarity with package management, you have what you need to start. Where a concept from Book 1 is essential to a topic in this book, it will be briefly reintroduced and then immediately connected to the server-administration task at hand not re-taught from scratch.

What This Book Is Not

This is not a certification study guide, and it does not attempt to cover every Linux distribution or every possible server role. It is not a theoretical networking textbook, a database development manual, or a general DevOps philosophy book. It focuses specifically and deliberately on Debian 13.7 as a server operating system, and on the practical, day-to-day work of deploying, configuring, securing, automating, monitoring, and troubleshooting real services running on it. Where broader concepts are necessary TCP/IP fundamentals, relational database theory, container architecture they are covered only to the depth required to operate Debian servers competently, not exhaustively.

How This Book Is Organized

The book follows the natural lifecycle of a server. It begins with an introduction that frames what server administration actually demands, distinguishes it clearly from desktop administration, and walks through the planning decisions that precede any installation. From there, the chapters proceed in the order a working administrator would actually encounter them:

Installation and initial configuration come first, because nothing else matters until a system exists and is minimally trustworthy. From there, the book covers day-to-day administration—users, packages, processes, services, and logs—because these are the mechanics every later chapter depends on. Networking and SSH follow, because a server is useless if it cannot be reached, and dangerous if it can be reached by the wrong people. Web and application services, storage and databases, and security hardening form the operational core of the book, reflecting the reality that most Debian servers exist to run something—a website, an API, a database—behind a security posture that has to be deliberately built, not assumed.

Later chapters address the concerns that separate hobbyist administration from professional operations: containerized workloads, automation and configuration management, monitoring and performance engineering, and backup, recovery, and troubleshooting. The book closes with production-style projects that integrate everything into realistic, multi-service environments, followed by a conclusion that ties the operational disciplines together into a coherent practice.

Every chapter follows a consistent philosophy, even where its structure adapts to its subject matter: explain what a concept is and why it matters operationally, show the real commands and configuration required to implement it, explain the important details of that configuration, show how to verify that it worked, and explain what can go wrong and how to investigate it. Nothing in this book is presented as "configure the appropriate settings"—every procedure is shown with the actual commands,

files, and directives involved, because vague instructions are not useful to someone operating a real server under real constraints.

A Note on Safety and Production Systems

Server administration is inherently a discipline of consequence. Many of the commands and procedures in this book can disrupt services, modify network access, alter storage, or affect data if applied carelessly or in the wrong environment. Throughout the book, potentially destructive operations are explicitly flagged, and the text consistently emphasizes testing in non-production environments, maintaining current backups, and verifying changes before considering them complete. This is not a formality—it reflects how experienced administrators actually work. Chapter by chapter, this book tries to model not just the commands of a competent Debian administrator, but the judgment.

A Word on Debian 13.7

Debian releases are known for stability, careful curation, and a conservative approach to change—qualities that make Debian a natural choice for production server infrastructure. This book is aligned specifically with Debian 13.7, and commands, package names, configuration file locations, and service behaviors reflect that version. Where behavior has meaningfully changed from earlier Debian releases, or where a procedure depends on version-specific tooling, this is called out explicitly rather than assumed. The underlying concepts in this book—networking, security, storage, automation, monitoring—will remain useful well beyond this specific release, but the concrete implementation details are written to be accurate for the system you are actually likely to be running.

With that foundation in place, it's time to begin.

Introduction : Building and Managing Debian 13.7 Servers

The Role of Debian in Modern Server Infrastructure

Debian occupies a distinct position in the landscape of server operating systems. It is neither a commercial distribution backed by a single vendor's release cycle and support contracts, nor a rolling-release system optimized for developers who want the newest package versions at the cost of stability. Debian is a community-governed project whose stable branch — the branch this book concerns itself with — is built around a deliberately conservative philosophy: packages are tested extensively before release, security patches are backported rather than bundled with disruptive version bumps, and the overall system is engineered to change as little as possible once it reaches production.

This philosophy is precisely what makes Debian attractive for server infrastructure. A production server's value comes from its predictability. An administrator who configures an Nginx virtual host, a PostgreSQL cluster, or a systemd service on Debian 13.7 can reasonably expect that configuration to keep behaving the same way for the multi-year support lifetime of that release, barring deliberate upgrades. Compare this to distributions where package versions shift more aggressively — the operational cost of that unpredictability, in the form of unexpected behavior changes, broken configuration syntax, or subtle regressions, is a cost that production environments are rarely willing to absorb.

Debian's package ecosystem, managed through `dpkg` and `apt`, is also a significant operational asset. The Debian archive is enormous, extensively tested for interdependency, and organized around a security-update infrastructure that has been running reliably for decades. When you install `nginx`, `postgresql`, `docker.io`, or virtually any other server-relevant package on Debian, you inherit not just the software but a maintenance and security

process behind it — one that many administrators trust more than ad hoc installation from upstream vendor repositories.

This is why Debian is disproportionately common beneath the surface of the internet. It runs behind countless VPS providers as a default image, powers a significant share of self-hosted infrastructure, and serves as the base layer for other distributions (Ubuntu Server being the most visible descendant). Understanding Debian server administration is not a niche skill — it is close to a lingua franca of Linux server operations, transferable with minor adaptation to Debian-derived systems and broadly informative even on unrelated distributions, because the underlying concepts — `systemd`, networking, package management logic, filesystem layout — are shared across most modern Linux server environments.

Choosing Debian for a server is, in effect, choosing to prioritize stability, security process maturity, and long-term predictability over bleeding-edge software versions. That tradeoff defines much of what "good" Debian server administration looks like throughout this book: preferring stable, well-tested configuration paths, understanding how Debian's release and update model affects long-term maintenance, and building operational practices around a system that is designed to be dependable rather than novel.

Server Administration vs. Desktop Linux Administration

It is worth being explicit about how fundamentally different server administration is from desktop or workstation administration, because the habits that serve a desktop user well can actively work against a server administrator.

Availability is the primary constraint. A desktop system can be rebooted at will, left in a broken state overnight, or reinstalled from scratch with minimal consequence beyond personal inconvenience. A server, particularly one running production workloads, is often expected to maintain near-continuous availability. Every reboot, every service restart, every configuration change carries an implicit cost measured in downtime, and administrators must plan changes — especially disruptive ones — around

maintenance windows, redundancy, and rollback procedures rather than applying them reactively.

There is no local user sitting at the console. Desktop administration assumes a human physically present, able to see graphical error dialogs, interact with a display manager, or use a recovery boot menu. Server administration overwhelmingly happens remotely, over SSH, often with no graphical environment installed at all. This has cascading implications: network configuration becomes safety-critical (a misconfigured firewall or network interface can sever your only means of access), logging and remote diagnostics become the primary way problems are understood, and out-of-band access (via a hosting provider's console, IPMI, or a hypervisor's virtual console) becomes an essential fallback that desktop users never need to think about.

Security posture is fundamentally more adversarial. A desktop machine on a home network faces a relatively narrow threat model. A server especially one with any public-facing service faces continuous, automated scanning and attack attempts from the moment it receives a public IP address. SSH brute-force attempts, port scans, and exploitation attempts against known vulnerabilities are a constant background condition of running a server on the internet, not an edge case. This reshapes priorities: minimizing exposed services, hardening authentication, applying security updates promptly, and monitoring for intrusion become core, everyday responsibilities rather than occasional concerns.

Services, not applications, are the unit of concern. A desktop administrator thinks in terms of applications a user runs interactively. A server administrator thinks in terms of long-running services web servers, databases, application backends, containers that must start reliably at boot, restart automatically on failure, log their behavior for later inspection, and run under appropriately scoped privileges. systemd's role shifts from "the thing that starts my desktop environment" to "the primary contract I use to make software behave like a reliable service."

Change is deliberate, not exploratory. On a desktop, an administrator might install and remove software casually, experimenting freely. On a production server, every package installation, every configuration change,

and every service restart is or should be a deliberate, documented action, ideally tested somewhere other than production first. This book will repeatedly return to the discipline of testing changes safely before they reach systems that matter, because this distinction is one of the clearest lines between administering Linux and operating Linux infrastructure.

Recognizing these differences early sets the tone for everything that follows in this book. Every chapter assumes a server administrator's mindset: availability-conscious, remote-access-aware, security-conscious by default, service-oriented, and disciplined about change.

Planning a Debian Server Environment

Effective server administration begins well before the first command is typed. Planning decisions made before installation shape the server's entire operational life, and poor early decisions—an undersized disk partitioning scheme, an unclear network topology, an unplanned security boundary—tend to become expensive or disruptive to correct later.

Define the server's role clearly. A server dedicated to a single, well-defined purpose—a web server, a database server, a reverse proxy—is easier to secure, monitor, and troubleshoot than a general-purpose machine running many unrelated services. This "single responsibility" principle isn't a rigid law (many small deployments legitimately run several services on one machine for cost reasons), but it should be a deliberate choice, not an accident of convenience. Whenever multiple services must share a machine, plan explicitly for how they will be isolated from one another—through dedicated service accounts, container boundaries, resource limits, or network segmentation—rather than allowing them to coexist without any separation.

Estimate resource requirements realistically. CPU, memory, storage, and network bandwidth requirements should be estimated from the actual expected workload, not guessed. A database server has very different memory and disk I/O characteristics than a static web server.

Underestimating resources leads to performance problems and emergency scaling; wildly overestimating wastes budget, particularly relevant on billed cloud and VPS infrastructure. Where the workload is genuinely unknown in

advance, favor infrastructure that can be resized or scaled with minimal disruption this consideration alone often decides whether physical hardware, a VM, or cloud infrastructure is the right choice.

Plan storage architecture up front. Decisions about disk partitioning, filesystem choice, and the use of LVM (Logical Volume Manager) are far easier to get right during installation than to retrofit afterward. Chapter 1 and Chapter 6 cover this in technical depth, but the planning question belongs here: how much storage does the operating system need, how much do logs need (and how will they be prevented from filling the root filesystem), how much do applications and databases need, and how will that storage grow over time?

Plan network topology and addressing. Determine, before installation, whether the server will use a static IP address or rely on DHCP with a reservation, what DNS records will point to it, whether it sits behind a load balancer or reverse proxy, what other systems it needs to communicate with (and over which ports), and what its position is relative to any existing firewall or network segmentation. A server whose network identity is planned in advance is dramatically easier to integrate into monitoring, DNS, certificate issuance, and firewall policy later.

Decide on the security boundary early. Will this server be directly reachable from the public internet, or only from an internal network or VPN? Will SSH be exposed on the default port to the world, restricted to specific source addresses, or placed behind a bastion host? These decisions influence firewall design, SSH hardening choices (Chapter 4), and the overall attack surface of the system, and are far easier to design correctly from the outset than to bolt on after the server is already running production workloads.

Plan for maintenance and lifecycle from day one. How will this server receive security updates? Who is responsible for it? How will its configuration be recorded, so that it can be rebuilt if lost? Will infrastructure be managed as reproducible configuration (even informally, through documented scripts) or treated as a unique, hand-configured machine? The single most common failure mode in small-scale server administration is the creation of a "pet" server manually configured,

undocumented, and irreplaceable whose loss becomes a crisis precisely because no one can reliably recreate it. This book's chapters on automation (Chapter 9) and backup (Chapter 11) exist specifically to help readers avoid this trap.

Physical, Virtual, VPS, and Cloud Deployment Options

Debian 13.7 can be deployed across a wide range of infrastructure types, and the right choice depends on workload, budget, scaling needs, and operational requirements. Understanding the tradeoffs between these options is a foundational planning skill.

Physical (bare-metal) servers offer the highest level of performance predictability and complete control over hardware, with no virtualization overhead and no "noisy neighbor" effect from other tenants sharing the same physical resources. They are appropriate for workloads with intensive, consistent resource demands large database clusters, high-throughput storage systems, latency-sensitive applications and for organizations with existing data center relationships or colocation arrangements. The tradeoffs are significant: capital cost, lead time to provision or replace hardware, and the operational burden of physical maintenance, out-of-band management (via IPMI or similar), and hardware failure response fall entirely on the administrator or their organization.

Virtual machines, whether self-hosted on a hypervisor (KVM, Xen, VMware, Proxmox) or provisioned through a cloud provider, decouple the operating system from physical hardware. A VM can be provisioned in minutes, resized (within limits) without hardware replacement, snapshotted for backup or rollback, and migrated between physical hosts for maintenance without necessarily interrupting the guest. Self-hosted virtualization gives an organization control over the hypervisor layer useful for cost control and data sovereignty while introducing the operational responsibility of maintaining that hypervisor infrastructure itself.

VPS (Virtual Private Server) hosting is a commercial offering built on virtualization, where a provider manages the physical infrastructure and

hypervisor, and the customer receives a virtual machine with root access, typically billed monthly with fixed or scalable resource tiers. VPS hosting is extremely popular for small-to-medium Debian deployments because it removes hardware and hypervisor management entirely while still providing full root-level control of the guest operating system precisely the level of control this book assumes throughout. The tradeoff is a dependency on the provider's reliability, network, and support quality, and typically less granular control over the underlying physical resources than dedicated hardware would offer.

Cloud infrastructure (AWS EC2, Google Compute Engine, Azure Virtual Machines, and similar) extends the VPS model with a much broader ecosystem: software-defined networking, managed load balancers, snapshot and image management, auto-scaling groups, managed DNS, and integration with a wide range of adjacent managed services (object storage, managed databases, monitoring). Debian is a well-supported guest OS across all major cloud providers. Cloud infrastructure is particularly suited to workloads that need to scale elastically, integrate with other managed services, or be provisioned and torn down programmatically as part of automated infrastructure workflows. The tradeoffs are cost complexity (cloud billing can be significantly harder to predict than flat-rate VPS billing) and a steeper learning curve around the provider's specific tooling and service model.

Table 0.1 summarizes the practical tradeoffs an administrator should weigh when choosing among these options.

Deployment Type	Provisioning Speed	Cost Model	Scalability	Operational Burden	Typical Use Case
Physical / Bare Metal	Slow (days–weeks)	High capital cost	Limited, hardware-bound	High (hardware + OS)	High-performance, latency-sensitive, or compliance-bound workloads
Self-Hosted VM	Fast (minutes)	Hardware + operational cost	Bound by host capacity	Moderate–High (hypervisor + OS)	Internal infrastructure, labs, cost-controlled private clouds
VPS	Fast (minutes)	Predictable monthly fee	Limited tiered resizing	Low–Moderate (OS only)	Small–medium production servers, personal and business hosting
Cloud (IaaS)	Very fast (seconds–minutes)	Usage-based, variable	Elastic, near-unlimited	Moderate (OS + cloud config)	Scalable, integrated, or programmatically managed infrastructure

Regardless of which deployment type is chosen, this book's practices apply directly at the guest operating system level: once Debian 13.7 is running—whether on bare metal, a self-hosted VM, a VPS, or a cloud instance—the administration principles, commands, and configurations covered in the following chapters are almost entirely identical. The meaningful differences are at the edges: how the server is initially provisioned and accessed (Chapter 1), how out-of-band recovery works if SSH access is lost, and how storage and networking are exposed by the underlying platform (a cloud provider's virtual disks and software-defined networking behave somewhat differently from a bare-metal RAID array, for instance, a distinction touched on in Chapter 6).

Server Administration Principles and Operational Workflows

Beyond specific commands and configurations, effective server administration rests on a small number of durable principles that recur throughout this book.

Idempotent, repeatable actions. Wherever possible, prefer administrative actions that can be safely repeated without unintended side effects, and prefer configuration that can be reapplied from a known source (a script, a

configuration management tool, a documented procedure) rather than relying on manual, one-off changes that exist only in a single administrator's memory. Chapter 9 develops this principle in depth through shell automation and an introduction to configuration management concepts.

Least privilege. Every user account, service account, application process, database role, and firewall rule should be granted the minimum access necessary to perform its function no more. This principle appears throughout the book: in user and sudo configuration (Chapter 2), SSH key management (Chapter 4), database user privileges (Chapter 6), container configuration (Chapter 8), and firewall design (Chapter 7). Excess privilege is not a hypothetical risk; it is the single most common factor that turns a minor security incident into a catastrophic one.

Change with a rollback plan. Before making a consequential change to a production system, an experienced administrator asks: how will I undo this if it goes wrong? This might mean taking a configuration backup before editing a file, snapshotting a VM before a major upgrade, testing a firewall rule change in a way that cannot lock out remote access, or verifying a database migration against a backup first. This book flags destructive or high-risk operations explicitly and consistently models this rollback-aware thinking.

Verify, don't assume. A command returning without an error is not the same as a command succeeding in its intended purpose. Throughout this book, procedures are followed by explicit verification steps checking service status, inspecting logs, testing connectivity, confirming expected behavior because assuming success is one of the most common sources of latent, undiscovered misconfiguration in real infrastructure.

Evidence before action in troubleshooting. When something breaks, the instinct to immediately start changing configuration is usually counterproductive. Chapter 11 develops a full troubleshooting methodology, but the underlying principle belongs here: gather evidence (logs, error messages, service status, resource utilization) and form a hypothesis about the actual cause before making changes, rather than changing settings speculatively and hoping the problem resolves.

Documentation as an operational asset, not an afterthought. Every nonstandard configuration decision, every deviation from a default, and every manual intervention should be recorded somewhere durable—whether in comments within configuration files, a runbook, or a version-controlled repository of infrastructure scripts. A server's configuration that exists only in the current administrator's memory is a significant operational liability the moment that administrator is unavailable.

These principles are not abstract ethics—they are the practical difference between an administrator who can confidently operate infrastructure under pressure and one who is perpetually one mistake away from a serious incident.

Building a Safe Testing and Production Environment

Because so much of server administration involves operations that can disrupt or damage a running system, this book consistently assumes—and strongly recommends—that readers build and use a safe, non-production environment for learning and experimentation before applying any given procedure to infrastructure that matters.

Use virtualization for a personal lab. A Debian 13.7 virtual machine, run locally through KVM/QEMU, VirtualBox, or a similar hypervisor, provides a fully isolated environment in which every command in this book—including partitioning disks, configuring firewalls, and even intentionally breaking services to practice troubleshooting—can be performed without any risk to real infrastructure. Snapshots (a feature of essentially all desktop virtualization tools) allow a lab VM to be reset to a known-good state in seconds, which is invaluable when practicing destructive or hard-to-reverse operations.

Treat a first VPS or cloud instance as a lab, not production, until it has been deliberately hardened and verified. It is extremely common—and reasonable—for readers to work through this book using a small, inexpensive VPS instance rather than a local VM, in order to get realistic exposure to a genuinely remote, internet-facing system. If this is your approach, treat that instance explicitly as a lab environment during the

learning process: avoid pointing real DNS records, real users, or real data at it until you have consciously decided it is ready to serve as a production system, following the hardening and validation practices developed throughout the book, especially in Chapter 1's post-installation validation and Chapter 7's security baseline.

Separate environments by purpose as your skills and infrastructure grow. As covered briefly earlier in this introduction, professional operations distinguish between development environments (where new configuration is written and initial testing happens, often locally), staging or testing environments (which mirror production as closely as practical, used to validate changes before they are exposed to real users or data), and production environments (which serve real traffic or real workloads, and whose changes should be the most conservative and well-verified of all). Even a single-person or small-team operation benefits enormously from maintaining at least a basic separation between "where I try things" and "what real users depend on" the discipline of that separation is often more valuable than the specific tooling used to implement it.

Never practice destructive commands directly against a system you cannot afford to lose. This guidance recurs throughout the book at points where genuinely dangerous operations are introduced disk partitioning, firewall rule changes that could sever remote access, database privilege modifications, container network changes. Readers are strongly encouraged to internalize this as a standing rule rather than a chapter-specific warning: if an operation could plausibly destroy data, cut off access, or disrupt a running service, its first execution should happen somewhere recoverable.

With planning principles, deployment context, and a safe learning environment established, the book now turns to the first concrete task every Debian server administrator faces: taking a piece of hardware or virtual infrastructure and turning it into a properly installed, initially configured Debian 13.7 server.

Chapter 1 : Debian Server Installation and Initial Configuration

1.1 Preparing Hardware, Virtual Machines, and VPS Environments

Before the Debian installer runs, the underlying infrastructure needs to be prepared and understood, because the choices available during installation and the steps required immediately after depend heavily on what kind of infrastructure the system is being installed onto.

Preparing Physical Hardware

For a bare-metal installation, preparation involves several concrete steps beyond simply having a machine available:

Verify hardware compatibility. Debian supports an extremely broad range of hardware, but server-specific components RAID controllers, network interface cards, out-of-band management controllers (iDRAC, iLO, IPMI) occasionally require verification that appropriate drivers are included in the installer or available as firmware packages. Check the target hardware's specifications against the Debian hardware compatibility documentation before beginning, particularly for RAID controllers, which sometimes require proprietary drivers not included in the default installer image.

Prepare installation media. Download the official Debian 13.7 installer image appropriate to the hardware's architecture (almost always amd64 for modern server hardware) from the Debian project's distribution mirrors. For server installations, the "netinst" (network install) image is generally preferred over the full DVD image it is small, and pulls current packages directly from the network during installation, avoiding the need to

separately apply updates immediately after install. Write the image to a USB drive using a tool appropriate to your workstation's operating system:

```
bash
```

```
# On a Linux workstation, identify the target USB device carefully
```

```
lsblk
```

```
# Write the image (replace /dev/sdX with the correct device this is destructive)
```

```
sudo dd if=debian-13.7.0-amd64-netinst.iso of=/dev/sdX bs=4M status=progress conv=fsync
```

Warning: The `dd` command above is destructive to whatever device is specified as `of=`. Double- and triple-check the target device identifier with `lsblk` before running it writing to the wrong device can silently destroy data on an unrelated disk, including your workstation's own drive.

Plan out-of-band access. For physical servers, especially those in a remote data center or colocation facility, verify and configure out-of-band management (IPMI, iDRAC, iLO, or a vendor equivalent) before you need it. This is what allows you to access a virtual console, power-cycle the machine, and even mount virtual installation media remotely if physical access is inconvenient or impossible. Losing physical or remote console access to a bare-metal server well after it is racked is one of the more painful lessons in server administration plan for it in advance.

Configure RAID at the hardware level (if applicable). If the server uses a hardware RAID controller, configure the desired RAID level (RAID 1 for redundancy on boot/OS disks is common; RAID 10 or RAID 6 arrangements are typical for data volumes) through the controller's pre-boot configuration utility before beginning the Debian installation, since the installer will see the resulting RAID volume as a single block device.

Preparing a Virtual Machine

For self-hosted virtualization (KVM/QEMU, VirtualBox, Proxmox, VMware, Hyper-V), preparation involves:

Download the Debian 13.7 netinst ISO, as with physical hardware.

Create the virtual machine with appropriate resource allocations. A minimal Debian server VM used for learning can function with 1–2 vCPUs, 1–2 GB of RAM, and 10–20 GB of disk, though production workloads

(especially databases) should be sized according to actual expected demand, as discussed in the Introduction. Attach the ISO as a virtual optical drive.

Configure virtual networking deliberately. Most hypervisors offer several networking modes NAT (the VM shares the host's network identity, with port forwarding needed for inbound access), bridged (the VM receives its own address on the physical network, behaving like a separate machine), and host-only or internal networks (isolated from the outside world, useful for lab environments). For a server VM that needs to be reachable like a real server, bridged networking is usually the most representative choice; for an isolated lab, an internal or NAT network is safer and simpler.

Enable virtualization extensions in the host BIOS/UEFI, if not already enabled (Intel VT-x/VMX or AMD-V), as KVM-based virtualization depends on these hardware extensions.

Example: creating a KVM virtual machine from the command line using virt-install :

bash

```
sudo virt-install \
  --name debian-server \
  --ram 2048 \
  --vcpus 2 \
  --disk path=/var/lib/libvirt/images/debian-server.qcow2,size=20 \
  --os-variant debian12 \
  --network bridge=br0 \
  --graphics none \
  --console pty,target_type=serial \
  --location 'debian-13.7.0-amd64-netinst.iso' \
  --extra-args 'console=ttyS0,115200n8 serial'
```

This example uses a text console over serial (`--graphics none` , `console=ttyS0`), which mirrors how many server installations are actually performed headless, over a console redirected to text and is good practice to become comfortable with, since it mirrors VPS and cloud provisioning workflows closely.

Preparing a VPS or Cloud Instance

VPS and cloud environments typically abstract away the installer entirely: the provider offers a pre-built Debian 13.7 image, and "installation" consists of selecting that image, choosing a resource tier, and configuring initial access—most commonly an SSH key—through the provider's dashboard or API, rather than running the Debian installer interactively.

Key preparation steps specific to VPS/cloud deployment:

Generate an SSH key pair on your local workstation before provisioning, so that it can be supplied to the provider at creation time rather than relying on password authentication from the outset:

```
bash
ssh-keygen -t ed25519 -C "admin@debian-server" -f ~/.ssh/debian_server_ed25519
```

This creates an Ed25519 key pair (a modern, strong key type, preferable to older RSA keys for new deployments) at `~/.ssh/debian_server_ed25519` and `~/.ssh/debian_server_ed25519.pub`. Chapter 4 covers SSH key authentication in full depth; at this stage, the important point is having the key pair ready to supply during provisioning.

Select an appropriately sized resource tier based on the planning work described in the Introduction—undersized instances are easy to resize upward on most providers, but this still causes a disruption (typically a reboot), so reasonably accurate initial sizing is worth the effort.

Understand the provider's default access model. Most providers deliver a fresh Debian instance with root SSH login enabled and your supplied public key already installed, or provide a temporary root password via their dashboard. Either way, treat this initial state as provisional—Chapter 2 and Chapter 4 cover creating a dedicated administrative user and disabling direct root login, which should happen very early in the server's life, before it is exposed to any real usage.

Note the provider's networking and firewall model. Cloud and VPS providers frequently offer a network-level firewall (a "security group" or equivalent) in addition to anything configured within the guest OS. Understanding both layers—and not assuming that an in-guest firewall rule is your only protection, or conversely that a provider security group makes an in-guest firewall unnecessary—is an important habit to build from the

very first deployment. This book treats both layers as complementary throughout, particularly in Chapter 7.

1.2 Installing Debian 13.7 for Server Workloads

This section walks through the Debian 13.7 installer with a server-specific lens—the choices that matter for a machine that will run unattended, remotely administered production workloads, as opposed to a desktop workstation.

Booting the Installer

Boot from the prepared installation media (physical USB, virtual ISO, or provider-supplied image). At the initial boot menu, server installations should generally select **"Install"** rather than **"Graphical install"**—the text-based installer is lighter weight, behaves identically over serial or remote console connections, and avoids depending on a graphical environment that a headless server has no reason to carry.

Language, Location, and Keyboard

The installer will prompt for language, geographic location (which influences default locale and timezone suggestions), and keyboard layout. For servers administered exclusively over SSH from varied locations, English (or your organizational standard) with a standard keyboard layout is typical; these choices mainly affect locale defaults and have minimal server-specific implications beyond ensuring the system clock's timezone reflects intended operational practice (commonly UTC for server fleets, to simplify log correlation across systems in different physical time zones—this is revisited in section 1.4).

Network Configuration During Installation

The installer will attempt to configure networking automatically via DHCP. For a server intended to have a stable, predictable address, this is often sufficient at install time even if a static configuration will be applied afterward (covered in Chapter 3)—many administrators prefer to complete installation with DHCP for simplicity, then convert to static addressing post-installation once the final network plan is confirmed. If the installer

prompts for a hostname and domain name, supply the server's intended hostname deliberately and consistently with your naming scheme (this book uses `debian-server` as a running example hostname throughout, consistent with the Introduction's emphasis on consistent example environments).

Setting the Root Password and Creating a User Account

The installer prompts to set a root password and optionally create a non-root user account. For server installations, best practice is to:

1. Set a strong root password (even if root login will later be disabled over SSH, as covered in Chapter 4, a root password provides a necessary fallback for console-based recovery).
2. Create a non-root administrative user account during installation this account will be granted `sudo` privileges in Chapter 2, and will become the primary account used for day-to-day administration.

Disk Partitioning

The installer offers "Guided" partitioning (with several sub-options) or "Manual" partitioning. For server workloads, manual partitioning or guided partitioning with manual review before confirming is strongly preferred over accepting an entirely automatic single-partition layout, because server storage planning (detailed in section 1.3) has real operational consequences that a fully automatic layout does not account for.

Package Selection

Near the end of installation, the installer presents a "tasksel" screen offering broad software collections (Desktop environments, Web server, SSH server, standard system utilities, etc.). For a server installation:

- **Deselect any desktop environment options.** A server has no need for a graphical desktop, and including one adds unnecessary attack surface, resource consumption, and maintenance burden.
- **Select "SSH server"** so that remote access is available immediately after first boot this is particularly important for VPS/cloud/headless installations where there may be no other practical access method afterward.
- **Select "standard system utilities."**
- Leave role-specific software (web server, database server, container runtime) unselected at this stage; these will be installed deliberately in later chapters (5, 6, and 8 respectively) with full configuration control, rather than accepting the installer's default, minimally configured versions.

GRUB Bootloader Installation

The installer will offer to install the GRUB bootloader to the appropriate disk. Accept this on the primary boot disk unless you have a specific reason (such as a pre-existing multi-boot configuration, which is essentially never relevant for a dedicated server) to do otherwise.

First Boot

After installation completes and the system reboots, log in either at the physical/virtual console or, if SSH was installed and networking is functioning, via SSH from your workstation:

```
bash  
ssh root@<server-ip-address>
```

At this point, you have a minimally installed Debian 13.7 system. The remainder of this chapter and much of Chapter 2 covers turning this minimal install into a properly configured, validated server.

1.3 Partitioning, Filesystems, and Storage Planning

Storage planning deserves particular care during installation because, while some adjustments are possible later (particularly with LVM, covered in Chapter 6), getting the initial layout wrong is one of the more disruptive mistakes to correct on a running server.

Why Partitioning Matters More on Servers

On a desktop system, a single root partition containing everything is a common and reasonable simplification. On a server, several operational concerns argue strongly for a more deliberate, multi-partition (or multi-logical-volume) layout:

Log growth should not be able to fill the root filesystem. If `/var/log` shares a filesystem with `/` and something begins logging excessively (a misbehaving service, a security incident generating enormous authentication log volume, a debug flag left enabled), an unbounded, unisolated log directory can fill the entire root filesystem, at which point the system may be unable to write to essential locations, potentially halting services or even preventing clean shutdown. Isolating `/var/log` onto its own partition or logical volume caps this risk to that partition alone.

Temporary files should not threaten system stability. Similarly, `/tmp` can fill unexpectedly due to application behavior; isolating it limits the blast

radius.

Data and OS should be separable. Application data, database storage, and user-uploaded content (commonly under `/var/lib` for many services, or a dedicated mount point such as `/data`) benefit from being on separate storage logically, and ideally physically from the base operating system. This makes OS reinstalls, backups, and storage-tier decisions (fast SSD for databases, larger cheaper storage for bulk data) much cleaner.

LVM adds flexibility that raw partitions do not. Using LVM (Logical Volume Manager) instead of, or on top of, raw disk partitions allows logical volumes to be resized (in many cases while mounted), new physical storage to be added to a volume group without disrupting existing volumes, and snapshots to be taken for backup purposes. Chapter 6 covers LVM administration on a running system in depth; this section addresses using LVM during initial installation.

A Representative Server Partitioning Scheme

While exact requirements vary by workload, the following scheme represents a reasonable, defensible baseline for a general-purpose Debian 13.7 server, expressed as LVM logical volumes atop a single physical volume (adjust sizes according to actual disk capacity and workload):

Mount Point	Suggested Size (Example)	Filesystem	Purpose
<code>/boot</code>	512 MB – 1 GB	ext4	Kernel and bootloader files (kept outside LVM for boot simplicity)
<code>/</code> (root)	10–20 GB	ext4	Base operating system and installed packages
<code>/var</code>	10–20 GB (or more)	ext4	Variable data, including package caches and some service data
<code>/var/log</code>	5–10 GB	ext4	Isolated log storage
<code>/tmp</code>	2–5 GB	ext4	Isolated temporary storage
<code>/home</code>	2–5 GB (servers rarely need much)	ext4	User home directories
swap	Equal to RAM up to ~4 GB, less beyond that	swap	Virtual memory overflow
<code>/data</code> or <code>/var/lib/<service></code>	Remaining capacity	ext4/xfs	Application/database data, sized to workload

`/boot` is conventionally kept as a plain partition outside LVM, since GRUB's ability to read LVM volumes directly, while supported, is more failure-prone during early boot than reading a straightforward partition – this is a widely followed convention worth respecting even though LVM-on-`/boot` is technically possible.

Filesystem Choice

ext4 is the default, well-understood, and thoroughly battle-tested choice for the great majority of Debian server use cases, offering solid reliability, mature tooling, and good general-purpose performance. **XFS** is a strong alternative particularly suited to large files and high-throughput, parallel I/O workloads (common in some database and large-media use cases), and is worth considering specifically for large data volumes with demanding I/O patterns. Unless a workload has a specific, identified reason to prefer XFS or another filesystem, ext4 is the sound default choice for this book's examples and for most real deployments.

Performing Manual/LVM Partitioning in the Installer

Within the Debian installer's partitioning step, select "Manual" (or "Guided – use entire disk and set up LVM" as a starting point, followed by manual review of the proposed logical volumes). The installer allows you to:

1. Create a physical partition for `/boot` (ext4, ~512MB–1GB, flagged bootable).
2. Create a physical partition consuming the remainder of the disk, marked as a physical volume for LVM.
3. Within that physical volume, create a volume group (commonly named after the hostname or purpose, e.g., `vg_debian-server`).
4. Within that volume group, create logical volumes matching the scheme above (`lv_root`, `lv_var`, `lv_var_log`, `lv_tmp`, `lv_home`, `lv_swap`, and optionally `lv_data`), assigning each an appropriate filesystem and mount point.

After installation, the resulting layout can be inspected with:

```
bash
lsblk
df -h
```

```
sudo vgs  
sudo lvs  
sudo pvs
```

`lsblk` shows the block device hierarchy (physical disks, partitions, and LVM logical volumes); `df -h` shows mounted filesystem usage in human-readable form; `vgs`, `lvs`, and `pvs` show LVM volume groups, logical volumes, and physical volumes respectively. These commands, and the deeper LVM operations they support (extending volumes, adding physical storage), are covered comprehensively in Chapter 6.

Storage Planning Principles Going Forward

Two principles from this section recur throughout the book. First, isolate anything that can grow unpredictably (logs, temporary files, user-uploaded content) from the filesystems that the operating system needs to remain functional. Second, prefer LVM over raw partitioning on any server where storage needs might grow or shift over time—which, in practice, is nearly every server that survives more than a few months in production—because the cost of using LVM from the start is minimal, while the cost of migrating a raw-partitioned running system to LVM later is substantial.

1.4 Initial User, Network, Hostname, and Time Configuration

With Debian 13.7 installed, several foundational configuration tasks should be completed before the server is considered ready for any real use—even lab use.

Verifying and Setting the Hostname

Confirm the hostname set during installation, or set/correct it if necessary:

```
bash
hostnamectl status
```

To set or change the hostname:

```
bash
sudo hostnamectl set-hostname debian-server
```

Ensure `/etc/hosts` includes an entry mapping the hostname to `127.0.1.1` (Debian's conventional loopback mapping for the machine's own hostname, distinct from `127.0.0.1`), which prevents various tools (including some that resolve the local hostname before DNS is available) from failing:

```
127.0.0.1 localhost
127.0.1.1 debian-server.example.com debian-server
```

Verify with:

```
bash
hostname
hostname -f
getent hosts debian-server
```

`hostname -f` should return the fully qualified domain name if one is configured; `getent hosts` confirms that name resolution for the hostname itself works correctly, which matters for some services and logging tools that expect to resolve their own hostname reliably.

Creating the Administrative User (if not done during installation)

If a non-root user was not created during installation, create one now and grant it `sudo` privileges (full detail on user and privilege management follows in Chapter 2; this is the minimum needed to move off direct root usage immediately):

```
bash
sudo adduser adminuser
sudo usermod -aG sudo adminuser
```

`adduser` creates the account interactively (prompting for a password and optional account details); `usermod -aG sudo` adds the user to the `sudo` group, which on a default Debian installation grants members permission to run commands as root via `sudo`, governed by the configuration in `/etc/sudoers` (examined closely in Chapter 2).

Verify `sudo` access works correctly before relying on it, ideally in a separate session so that your current root or console session remains available as a fallback if something is misconfigured:

```
bash
su - adminuser
sudo whoami
# Should output: root
```

Basic Network Verification

Confirm the network configuration applied during installation is functioning as expected:

```
bash
ip addr show
ip route show
resolvectl status # or: cat /etc/resolv.conf, depending on resolver configuration
ping -c 4 debian.org
```

`ip addr show` lists configured interfaces and their addresses; `ip route show` displays the routing table, confirming a default gateway is present; the resolver check confirms DNS resolution is functioning; and the `ping` test confirms actual internet connectivity end-to-end. A full treatment of static IP configuration, routing, and DNS follows in Chapter 3 at this stage, the goal is simply confirming that the installer's networking configuration is functional before proceeding further.

Time Synchronization

Accurate system time is essential for TLS certificate validation, log correlation across systems, authentication protocols (including SSH and Kerberos-based systems), and general operational sanity. Debian 13.7 uses `systemd-timesyncd` by default for basic NTP client functionality. Verify it is active:

bash

```
systemctl status systemd-timesyncd  
timedatectl status
```

`timedatectl status` reports whether the system clock is synchronized, which NTP service is in use, and the currently configured timezone. If the timezone was not set correctly during installation, correct it for server fleets, UTC is a very common and recommended standard, since it avoids daylight-saving-time ambiguity when correlating logs across systems or with external services:

bash

```
sudo timedatectl set-timezone UTC
```

For environments requiring more precise or more resilient time synchronization than `systemd-timesyncd` provides (common in environments running their own NTP infrastructure, or requiring redundant time sources), `chrony` is a mature alternative available in the Debian repositories:

bash

```
sudo apt update  
sudo apt install chrony  
sudo systemctl status chrony
```

Installing `chrony` will typically disable `systemd-timesyncd` automatically to avoid conflict; verify this with `systemctl status systemd-timesyncd` afterward, and disable it explicitly if it remains active:

bash

```
sudo systemctl disable --now systemd-timesyncd
```

Locale Configuration

Confirm the system locale is set appropriately, particularly if the installer's language/location prompts were answered with defaults that don't match operational needs:

bash

```
locale  
sudo dpkg-reconfigure locales
```

`dpkg-reconfigure locales` presents an interactive menu to select which locales to generate and which to set as the system default for most server

environments, `en_US.UTF-8` (or the organizationally appropriate equivalent) is sufficient, and unnecessary additional locales can be safely left unselected to keep the system minimal.

1.5 Post-Installation Server Configuration and Validation

Before considering the base installation "done," a systematic validation pass should confirm the server is in a known, sound, and minimally-hardened state the platform onto which every later chapter's configuration will be layered.

Updating the System

Apply all available package updates immediately, even on a freshly installed system, since installer images are sometimes built somewhat before their corresponding release's ongoing security patches:

```
bash
sudo apt update
sudo apt full-upgrade -y
sudo reboot
```

`apt update` refreshes the local package index against configured repositories; `apt full-upgrade` (preferred over plain `upgrade` for handling dependency changes, including kernel updates, more robustly) applies all available updates; the subsequent reboot ensures a newly installed kernel (if any) is actually running, rather than leaving the system running an older kernel in memory while a newer one sits installed but unused.

After reboot, confirm the running kernel version:

```
bash
uname -r
```

Reviewing Installed Packages and Enabled Services

Get a baseline understanding of what is installed and running, both for security awareness and for future troubleshooting reference:

```
bash
```

```
dpkg -l | less
systemctl list-units --type=service --state=running
```

`dpkg -l` lists all installed packages; piping through `less` allows this potentially long list to be reviewed page by page. `systemctl list-units --type=service --state=running` shows every currently active systemd service on a freshly installed, minimal server, this list should be short and entirely explainable; anything unexpected merits investigation before proceeding.

Verifying SSH Access Independently

Before disconnecting your initial console/root session, open a **second, independent** SSH session using the newly created administrative user account and confirm both login and `sudo` access work correctly:

```
bash
ssh adminuser@<server-ip-address>
sudo apt update
```

This step matters more than it might initially seem: administrators have locked themselves out of servers by making an assumption about access that turned out to be wrong, then closing their only working session. Keeping an existing, known-working session open while validating a new access path is a small habit that prevents a genuinely common and painful mistake one that recurs as a theme throughout Chapter 4's SSH hardening material, where the stakes of this kind of verification are considerably higher.

Checking Disk and Filesystem State

Confirm the partitioning scheme from section 1.3 applied as intended, and that no filesystem is unexpectedly near capacity even this early:

```
bash
df -h
lsblk
sudo vgs && sudo lvs
```

Reviewing the Firewall State

Debian does not enable a firewall by default. Confirm the current (likely permissive) state, without yet configuring firewall rules in depth that is the

dedicated subject of Chapter 7:

```
bash
```

```
sudo nft list ruleset
```

An empty or default output here confirms no firewall is currently restricting traffic expected at this stage, but an important baseline to record, and a clear reminder that the server is currently unprotected beyond whatever network-level protections (a cloud security group, for instance) exist outside the guest OS itself.

Confirming Time, Hostname, and Logging Are Functioning Together

As a final validation step, generate a test log entry and confirm it is timestamped correctly and retrievable via `journalctl`, tying together the hostname, time synchronization, and logging configuration validated throughout this section:

```
bash
```

```
logger "Post-installation validation check on $(hostname) at $(date)"  
journalctl -n 5 --no-pager
```

The output should show your test message, correctly timestamped in the configured timezone, attributed to the correct hostname confirming that the logging and time infrastructure that every later chapter's troubleshooting and monitoring guidance depends on is functioning correctly from the very start.

Summary Checklist

At the end of Chapter 1, a properly prepared Debian 13.7 server should satisfy the following baseline:

- Hostname is set correctly and resolves via `/etc/hosts`
- A non-root administrative user exists with verified `sudo` access
- Basic network connectivity (address, gateway, DNS) is confirmed functional
- System time is synchronized and timezone is deliberately set
- All available package updates have been applied, and the system has been rebooted onto the current kernel
- Storage is partitioned according to a deliberate plan (ideally LVM-based), with logs and temporary files isolated from the root filesystem
- SSH access via the administrative user has been independently verified in a separate session
- The current (likely absent) firewall state has been explicitly checked and recorded as a known baseline

This baseline is the foundation the remainder of the book builds on. Chapter 2 picks up immediately from here, developing full administrative practice around users, packages, processes, and services the everyday mechanics of running a Debian server responsibly.

Chapter 2 : Server Administration and Service Management

2.1 Managing Users, Groups, and Administrative Privileges

User and privilege management is the control layer beneath every other security and operational decision on a Debian server. Every service that runs, every file that is written, and every administrative action that is taken happens under the identity of some user account, and the discipline with which those accounts are created, scoped, and audited determines how contained a mistake or compromise remains.

Understanding the Debian User Model

Every account on a Debian system is represented by an entry in `/etc/passwd` (basic identity information: UID, GID, home directory, shell) and `/etc/shadow` (password hashes and aging policy, readable only by root). Groups are defined in `/etc/group`. Two categories of accounts matter operationally:

- **Human administrative accounts** accounts tied to a specific person, used for interactive login and `sudo`-elevated administration.
- **Service accounts** non-interactive accounts created specifically to run a particular daemon or application, typically with no login shell and no password, existing solely to own files and processes with appropriately scoped permissions.

Inspect the current state of accounts on a system with:

```
bash
cat /etc/passwd
getent passwd
```

`getent passwd` is generally preferable to reading `/etc/passwd` directly, since it correctly reflects any additional identity sources configured through NSS (Name Service Switch, in `/etc/nsswitch.conf`) such as LDAP though on most standalone Debian servers the two will show identical output.

Creating and Managing Human Administrative Accounts

Use `adduser` (Debian's interactive, higher-level wrapper) for creating human accounts, since it handles home directory creation, default file copying from `/etc/skel`, and password setup in one guided flow:

```
bash
sudo adduser jsmith
```

This prompts for a password and optional details (full name, room number, etc. largely vestigial fields that can be left blank on a server). The lower-level `useradd` command offers more explicit control and is preferable for scripted account creation:

```
bash
sudo useradd -m -s /bin/bash -c "J. Smith, Infrastructure Team" jsmith
sudo passwd jsmith
```

Here, `-m` creates the home directory, `-s /bin/bash` sets the login shell, and `-c` sets a comment/description field. Always follow non-interactive `useradd` calls with an explicit `passwd` step (or disable password login entirely in favor of SSH keys, as covered in Chapter 4) `useradd` alone leaves the account locked with no usable password.

Modifying and Removing Accounts

```
bash
sudo usermod -aG developers,docker jsmith # add to supplementary groups
sudo usermod -L jsmith                    # lock the account (disable password login)
sudo usermod -U jsmith                    # unlock
sudo chage -l jsmith                      # view password aging/expiration policy
sudo userdel -r jsmith                    # remove account and home directory
```

Warning: `userdel -r` removes the user's home directory and mail spool irreversibly. Confirm any needed data has been backed up or reassigned before running it, and never run it against an account still owning

production data or actively running processes without first verifying those dependencies.

Always use `-aG` (append to groups) rather than plain `-G` with `usermod` when adding supplementary group membership. plain `-G` *replaces* the user's entire supplementary group list, a common and disruptive mistake that silently removes existing group memberships.

Understanding and Configuring sudo

`sudo` allows designated users to execute commands with elevated privileges, governed by `/etc/sudoers` and the drop-in directory `/etc/sudoers.d/`. Never edit `/etc/sudoers` directly with a standard text editor. Always use `visudo`, which validates syntax before saving and prevents a malformed file from locking out all `sudo` access:

```
bash
sudo visudo
```

On Debian, membership in the `sudo` group (not `wheel`, as on some other distributions) is the default mechanism granting full `sudo` privileges, via this line typically present in `/etc/sudoers`:

```
%sudo  ALL=(ALL:ALL) ALL
```

For finer-grained privilege delegation—for example, granting a monitoring account permission to restart only a specific service, without full root access—create a dedicated file under `/etc/sudoers.d/`:

```
bash
sudo visudo -f /etc/sudoers.d/monitoring-restart
# Allow the 'monitor' user to restart only the nginx and myapp services, no password prompt
monitor ALL=(root) NOPASSWD: /usr/bin/systemctl restart nginx, /usr/bin/systemctl restart myapp
```

This is a meaningful example of least privilege in practice: `monitor` gains exactly the capability it needs and nothing more, rather than blanket `sudo` access. Always verify a new `sudoers` file with `visudo -c` after creating it:

```
bash
sudo visudo -c
```

Auditing Privileged Access

Periodically review which accounts have administrative privilege, and confirm it matches expectations:

```
bash
getent group sudo
sudo grep -r "" /etc/sudoers.d/
sudo journalctl _COMM=sudo --since "24 hours ago"
```

The `journalctl` query surfaces recent `sudo` invocations system-wide, useful both for routine audit and for incident investigation (a theme developed further in Chapter 7's security monitoring section and Chapter 11's troubleshooting methodology).

2.2 Package and Repository Management on Servers

While Book 1 covers `apt` fundamentals, server package management carries distinct operational stakes: unattended or careless package operations can introduce instability, security gaps, or unexpected service disruption on systems that are expected to remain reliable.

Understanding Debian's Repository Structure

A server's package sources are defined in `/etc/apt/sources.list` and files under `/etc/apt/sources.list.d/`. A typical Debian 13.7 (codename **trixie**) server configuration looks like:

```
deb http://deb.debian.org/debian trixie main contrib non-free-firmware
deb http://deb.debian.org/debian trixie-updates main contrib non-free-firmware
deb http://security.debian.org/debian-security trixie-security main contrib non-free-firmware
```

- `trixie` the base stable release archive.
- `trixie-updates` point-release updates that accumulate between full releases.
- `trixie-security` security patches, maintained separately and updated promptly outside the normal point-release cadence, and the most operationally important line for a production server.

Confirm the currently configured sources:

```
bash
```

```
cat /etc/apt/sources.list
ls /etc/apt/sources.list.d/
apt-cache policy
```

Core Package Operations for Server Use

bash

```
sudo apt update           # refresh package index
sudo apt list --upgradable # show what would be updated
sudo apt upgrade -y       # apply updates, conservative (won't remove packages)
sudo apt full-upgrade -y  # apply updates, allows package removal/replacement if needed
sudo apt install <package> # install a package
sudo apt remove <package>  # remove package, keep configuration files
sudo apt purge <package>   # remove package and its configuration files
sudo apt autoremove --purge # remove unused dependency packages
```

For servers, `apt full-upgrade` is generally preferred over plain `upgrade` for routine maintenance, since it correctly handles situations where an update requires a dependency to be removed and replaced (common with kernel and library transitions) rather than holding the affected package back silently.

Holding Packages at a Specific Version

Occasionally, an administrator needs to prevent a specific package from being upgraded—for example, to avoid an update known to break compatibility with an application until it can be tested:

bash

```
sudo apt-mark hold postgresql-15
sudo apt-mark showhold
sudo apt-mark unhold postgresql-15
```

Held packages should be tracked deliberately and revisited—an indefinitely held package silently accumulates security exposure over time, which is precisely the kind of debt this book's principles (Introduction, "documentation as an operational asset") warn against leaving undocumented.

Investigating Package Contents and Provenance

Before installing unfamiliar packages, especially from third-party repositories, verify their contents and origin:

bash

```
apt-cache show <package>
apt-cache policy <package>
dpkg -L <package>          # list files installed by an already-installed package
dpkg -S /path/to/file      # find which package owns a given file
```

Adding Third-Party Repositories Safely

Server software occasionally requires packages from outside the standard Debian archive (Docker's official repository, for instance, covered in Chapter 8). Add such repositories deliberately and with proper key verification, never by disabling signature checking:

bash

```
# Example pattern: add a vendor's GPG key and repository correctly
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://example-vendor.com/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/vendor.gpg
sudo chmod a+r /etc/apt/keyrings/vendor.gpg

echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/vendor.gpg]
https://example-vendor.com/debian trixie stable" | \
  sudo tee /etc/apt/sources.list.d/vendor.list

sudo apt update
```

This pattern a dedicated keyring file referenced explicitly via `signed-by`, rather than a globally trusted key limits the scope of trust granted to a third-party repository to only the packages it signs, and is the current recommended practice over the deprecated `apt-key` command.

Automating Security Updates

For servers that cannot be manually patched on a tight cadence, `unattended-upgrades` automates the application of security updates:

bash

```
sudo apt install unattended-upgrades apt-listchanges
sudo dpkg-reconfigure --priority=low unattended-upgrades
```

Review and adjust `/etc/apt/apt.conf.d/50unattended-upgrades` to confirm it targets the `-security` suite (enabled by default) and to decide whether automatic reboots (for kernel updates) are appropriate for this server:

```
Unattended-Upgrade::Automatic-Reboot "false";
Unattended-Upgrade::Automatic-Reboot-Time "03:00";
```

Automatic reboots carry availability tradeoffs discussed further in Chapter 9's automation coverage for many production servers, applying security packages automatically while deferring reboots to a controlled maintenance window is the more conservative and generally preferred configuration.

Verifying Package Integrity

bash

```
sudo apt-get check
sudo dpkg --audit
sudo debsums -c      # requires: sudo apt install debsums; verifies installed files against package
checksums
```

`debsums -c` compares currently installed files against the checksums recorded at package installation time, flagging any that have been modified a useful integrity check both for routine hygiene and as part of incident investigation.

2.3 Managing Processes and Systemd Services

Debian 13.7, like all current Debian releases, uses **systemd** as its init system and service manager. Competent process and service management is arguably the single most-used skill in day-to-day server administration.

Inspecting Running Processes

bash

```
ps aux                # full process listing
ps aux --sort=-%mem | head    # top memory consumers
pgrep -f nginx          # find PIDs matching a pattern
pstree -p              # process tree with PIDs
```

`ps aux` columns of particular operational relevance: %CPU and %MEM (resource consumption), STAT (process state S sleeping, R running, Z zombie, D uninterruptible sleep, the last of which often indicates a process stuck on I/O), and START / TIME (when the process started and accumulated CPU time).

Signaling and Terminating Processes

bash

```
kill -TERM <pid>      # graceful termination request (default signal)
```

```
kill -HUP <pid>    # request config reload, for daemons that support it
kill -9 <pid>      # SIGKILL - immediate, forceful termination (no cleanup)
pkill -f "pattern" # signal all processes matching a name/argument pattern
```

Always attempt `SIGTERM` (the default, graceful signal) before resorting to `SIGKILL`. A process killed with `SIGKILL` has no opportunity to close files cleanly, flush buffers, or release locks for a database process in particular, this can risk data corruption. Reserve `SIGKILL` for processes that are confirmed unresponsive to graceful termination.

The systemd Service Model

systemd organizes managed entities into **units** services, sockets, timers, mount points, and more described by unit files, typically located in `/lib/systemd/system/` (packages) or `/etc/systemd/system/` (local overrides and custom units, which take precedence).

Core service management commands:

bash

```
sudo systemctl status nginx    # current status, recent log lines, process info
sudo systemctl start nginx
sudo systemctl stop nginx
sudo systemctl restart nginx   # stop then start
sudo systemctl reload nginx    # apply config without dropping connections (if supported)
sudo systemctl enable nginx    # start automatically at boot
sudo systemctl disable nginx
sudo systemctl enable --now nginx # enable and start in one step
sudo systemctl is-active nginx
sudo systemctl is-enabled nginx
```

The distinction between `restart` and `reload` matters operationally: `reload` (where a service supports it Nginx and many other daemons do) applies configuration changes without dropping existing connections or fully re-initializing the process, making it the preferred choice for routine configuration updates on a service actively handling traffic.

Listing and Filtering Units

bash

```
systemctl list-units --type=service    # all loaded service units
systemctl list-units --type=service --state=running # only running services
systemctl list-unit-files --type=service --state=enabled # services enabled at boot
systemctl --failed                     # units that failed to start
```

Running `systemctl --failed` periodically and certainly whenever investigating unexpected server behavior surfaces any service that has entered a failed state, which is one of the most immediately useful diagnostic habits in day-to-day server administration.

Writing a Custom systemd Service Unit

Server administrators frequently need to run a custom application (a script, a compiled binary, an interpreted application) as a properly managed systemd service rather than an ad hoc background process. Create a unit file under `/etc/systemd/system/`:

```
bash
```

```
sudo nano /etc/systemd/system/myapp.service
```

```
ini
```

```
[Unit]
```

```
Description=My Application Service
```

```
After=network-online.target
```

```
Wants=network-online.target
```

```
[Service]
```

```
Type=simple
```

```
User=myappuser
```

```
Group=myappuser
```

```
WorkingDirectory=/opt/myapp
```

```
ExecStart=/opt/myapp/venv/bin/python /opt/myapp/app.py
```

```
Restart=on-failure
```

```
RestartSec=5
```

```
Environment=APP_ENV=production
```

```
StandardOutput=journal
```

```
StandardError=journal
```

```
# Resource and security hardening
```

```
NoNewPrivileges=true
```

```
PrivateTmp=true
```

```
ProtectSystem=strict
```

```
ReadWritePaths=/opt/myapp/data
```

```
[Install]
```

```
WantedBy=multi-user.target
```

Key directives explained:

- **After= / Wants=network-online.target** ensures the service starts only after networking is genuinely available, important for applications that connect to a database or external service immediately on startup.
- **Type=simple** the most common service type, appropriate for a process that runs in the foreground and does not fork/daemonize itself.
- **User= / Group=** runs the service under a dedicated, non-root service account (created beforehand with `sudo adduser --system --no-create-home myappuser`), following the least-privilege principle established in the Introduction.
- **Restart=on-failure** with **RestartSec=5** automatically restarts the service after a crash, waiting 5 seconds between attempts, providing basic self-healing without needing external tooling.
- **NoNewPrivileges**, **PrivateTmp**, **ProtectSystem=strict** systemd's built-in sandboxing directives, restricting the service's ability to escalate privileges, isolating its temporary files, and making the filesystem read-only except for explicitly listed writable paths (`ReadWritePaths`). This is a practical, low-effort application of defense-in-depth that Chapter 7 revisits in more depth.

After creating or modifying a unit file, reload systemd's configuration and start the service:

bash

```
sudo systemctl daemon-reload
sudo systemctl enable --now myapp.service
sudo systemctl status myapp.service
```

`daemon-reload` is required any time a unit file is created or edited. `systemd` caches unit definitions, and skipping this step is a very common cause of "I edited the file but nothing changed" confusion.

2.4 Configuring Service Startup and Recovery

Beyond simply defining a service, production reliability depends on how that service behaves during boot, failure, and dependency changes.

Boot Ordering and Dependencies

systemd's dependency directives control ordering and coupling between units:

- **After= / Before=** pure ordering; does not imply the referenced unit must succeed or even be present.
- **Requires=** a hard dependency; if the required unit fails, this unit is also stopped/failed.
- **Wants=** a soft dependency; the wanted unit is started alongside this one, but its failure does not prevent this unit from running.
- **Bindsto=** an even stronger coupling than **Requires=**; this unit is stopped immediately if the bound-to unit stops, useful for units tightly coupled to a specific resource (a mounted device, for instance).

A database-dependent application, for example, might reasonably express:

ini

[Unit]

Description=Application depending on local PostgreSQL

After=postgresql.service network-online.target

Wants=network-online.target

Requires=postgresql.service

This ensures the application unit will not even attempt to start until PostgreSQL is active, and will be stopped if PostgreSQL later fails.

Automatic Restart Behavior

systemd's restart directives allow fine control over self-healing behavior:

ini

[Service]

Restart=on-failure

RestartSec=5

StartLimitIntervalSec=60

StartLimitBurst=5

StartLimitIntervalSec and **StartLimitBurst** together prevent a crash-looping service from restarting indefinitely and consuming resources here, if the service fails more than 5 times within a 60-second window, systemd stops

attempting restarts and leaves the unit in a failed state, requiring manual intervention. This is an important safeguard: unrestricted automatic restart of a genuinely broken service can mask a serious problem (or actively worsen it, for example by repeatedly attempting and failing a database connection) rather than surfacing it for investigation.

Restart policy options include `no` (default, no automatic restart), `on-success`, `on-failure`, `on-abnormal`, `on-watchdog`, `on-abort`, and `always`. For most production services, `on-failure` strikes the right balance automatic recovery from crashes, without masking a deliberate `systemctl stop`.

Startup Ordering at the Target Level

`systemd` organizes the overall boot sequence around **targets** (roughly analogous to traditional runlevels). The default target for a server is normally `multi-user.target` (full multi-user, non-graphical operation):

```
bash
systemctl get-default
sudo systemctl set-default multi-user.target
```

Confirming a server's default target is `multi-user.target`, not `graphical.target`, is a useful sanity check, particularly on systems that may have had a desktop-oriented installation profile applied inadvertently a graphical target attempts to start a display manager and related services that serve no purpose on a headless server and represent unnecessary resource use and attack surface.

Handling a Failed Service

When `systemctl --failed` reveals a failed unit, the standard investigation sequence is:

```
bash
sudo systemctl status myapp.service
sudo journalctl -u myapp.service --since "10 minutes ago"
sudo journalctl -u myapp.service -b # since last boot
```

`systemctl status` typically shows the most recent log lines and the specific failure reason (a non-zero exit code, a crash signal, a failed dependency); `journalctl -u <unit>` retrieves the full, filtered log history for that specific unit,

which is usually necessary to understand the actual root cause beyond the summary shown by `status`. This workflow is developed further as part of the systematic troubleshooting methodology in Chapter 11.

Timers as a cron Alternative

`systemd` timers provide a modern alternative to `cron` for scheduled tasks, with tighter integration into service management, dependency handling, and logging. A brief introduction here; Chapter 9 covers timers in full depth alongside `cron`.

ini

```
# /etc/systemd/system/myapp-cleanup.timer
```

```
[Unit]
```

```
Description=Run myapp cleanup daily
```

```
[Timer]
```

```
OnCalendar=daily
```

```
Persistent=true
```

```
[Install]
```

```
WantedBy=timers.target
```

ini

```
# /etc/systemd/system/myapp-cleanup.service
```

```
[Unit]
```

```
Description=myapp cleanup task
```

```
[Service]
```

```
Type=oneshot
```

```
User=myappuser
```

```
ExecStart=/opt/myapp/scripts/cleanup.sh
```

bash

```
sudo systemctl enable --now myapp-cleanup.timer
```

```
systemctl list-timers
```

`Persistent=true` ensures a missed run (for example, if the server was powered off at the scheduled time) executes as soon as possible after the system is back up, rather than waiting silently for the next scheduled occurrence – an important detail for maintenance tasks that must not be silently skipped.

2.5 Managing Server Logs and Operational Information

Logs are the primary evidence base for understanding server behavior after the fact, and Debian 13.7's logging infrastructure centered on **journald**, systemd's logging component is a core operational tool every administrator must be fluent in.

Understanding journald

journald collects log data from systemd-managed services, the kernel, and applications that log via the systemd journal API or standard syslog interfaces, storing it in a structured, indexed binary format queried through `journalctl`.

Verify journald's storage configuration:

bash

```
sudo systemctl status systemd-journald
journalctl --disk-usage
```

By default, Debian configures journald for **volatile** (in-memory, cleared on reboot) or **persistent** (written to `/var/log/journal/`, surviving reboots) storage depending on whether that directory exists. For production servers, persistent storage is strongly preferred, since reboots are exactly the moments an administrator most needs surrounding log context:

bash

```
sudo mkdir -p /var/log/journal
sudo systemd-tmpfiles --create --prefix /var/log/journal
sudo systemctl restart systemd-journald
```

Core journalctl Usage

bash

```
journalctl          # full log, oldest first
journalctl -e       # jump to end (most recent)
journalctl -f       # follow, live tail
journalctl -u nginx.service # filter to a specific unit
journalctl -p err    # filter by minimum priority (err and above)
journalctl --since "2026-09-20 08:00" # time-range filtering
journalctl --since yesterday --until now
journalctl -k       # kernel messages only
journalctl _PID=1234 # filter by specific PID
journalctl --disk-usage # storage consumed by the journal
```

Priority filtering (`-p`) accepts standard syslog severity levels: `emerg` , `alert` , `crit` , `err` , `warning` , `notice` , `info` , `debug` . Combining time-range and unit filters is especially useful for incident investigation:

```
bash
journalctl -u nginx.service --since "09:00" --until "09:30"
```

Controlling Journal Retention

Unbounded journal growth can consume significant disk space over time. Control retention in `/etc/systemd/journald.conf` :

```
ini
[Journal]
Storage=persistent
SystemMaxUse=1G
SystemMaxFileSize=100M
MaxRetentionSec=30day
```

`SystemMaxUse` caps total journal disk consumption; `MaxRetentionSec` enforces a maximum age for retained entries regardless of size. After editing, apply changes with:

```
bash
sudo systemctl restart systemd-journald
```

This configuration directly supports the storage-isolation principle from section 1.3 bounding journal size is a second, complementary layer of protection against log-driven disk exhaustion, alongside isolating `/var/log` onto its own filesystem at the partition level.

Traditional Syslog-Style Logs

Many services (and some administrators' habits) still rely on traditional flat-file logs under `/var/log/` , either written directly by the application or forwarded there from `journald` via `rsyslog` (commonly installed alongside `journald` on Debian for compatibility with software expecting classic syslog behavior):

```
bash
ls -la /var/log/
sudo tail -f /var/log/auth.log
sudo tail -f /var/log/syslog
```

`/var/log/auth.log` in particular deserves routine attention – it records authentication events, including SSH login attempts and `sudo` invocations, and is one of the first places to check both during routine security review and during active incident investigation, a theme returned to in Chapter 7.

Log Rotation

Flat-file logs under `/var/log/` are managed by `logrotate`, configured via `/etc/logrotate.conf` and per-application drop-in files under `/etc/logrotate.d/`:

```
bash
cat /etc/logrotate.d/nginx
```

A typical entry:

```
/var/log/nginx/*.log {
    daily
    missingok
    rotate 14
    compress
    delaycompress
    notifempty
    create 0640 www-data adm
    sharedscripts
    postrotate
        systemctl reload nginx > /dev/null 2>&1 || true
    endsript
}
```

Key directives: `daily` rotates logs once per day; `rotate 14` retains 14 rotated copies before deletion; `compress / delaycompress` compress older rotated logs while leaving the most recent rotated copy uncompressed (useful for tools still actively reading it); and the `postrotate` block ensures Nginx reopens its log file handles after rotation, rather than continuing to write to a now-renamed file descriptor.

Test a `logrotate` configuration without waiting for its schedule:

```
bash
sudo logrotate -d /etc/logrotate.d/nginx # dry run, shows what would happen
sudo logrotate -f /etc/logrotate.d/nginx # force immediate rotation
```

Establishing a Routine Log Review Habit

Beyond reactive troubleshooting, competent server administration includes periodic, deliberate log review checking `journalctl --failed` equivalents, scanning `/var/log/auth.log` for unexpected authentication attempts, and confirming that expected services are logging normally rather than silently going quiet (itself sometimes a sign of a stalled or crashed process that hasn't formally entered a "failed" systemd state). This habit, formalized further through centralized logging and monitoring in Chapter 10, is what allows an administrator to detect emerging problems before they become outages a direct application of the "verify, don't assume" principle established in the Introduction, applied continuously rather than only at the moment of a specific change.

With users, packages, processes, services, and logs now under systematic control, the server has moved from a freshly installed machine to an actively, deliberately administered system. Chapter 3 turns to the network layer this server depends on for every external interaction it will ever have.

Chapter 3 : Debian Server Networking

3.1 Server Network Architecture and Configuration

A server's network configuration is arguably its single most safety-critical subsystem: a networking mistake, unlike most other misconfigurations, can simultaneously break the service the server exists to provide and sever the administrator's only means of remotely fixing it. This section establishes how Debian 13.7 manages networking and the architectural thinking that should precede any configuration change.

Debian's Networking Stack

Debian 13.7 supports multiple network configuration mechanisms, and understanding which one is actually active on a given system is the essential first step before making any change:

- **ifupdown** (traditional, configuration in `/etc/network/interfaces`) the long-standing Debian default, still widely used, especially on servers and minimal installations.
- **systemd-networkd** a lighter-weight, systemd-native network manager, increasingly common on minimal and cloud-oriented installations.
- **NetworkManager** primarily oriented toward desktop and laptop use cases (Wi-Fi roaming, VPN integration), rarely the right choice for a static server, but occasionally present on installations that included desktop-oriented package selections.

Determine which is active:

```
bash
systemctl status networking      # ifupdown
systemctl status systemd-networkd # systemd-networkd
systemctl status NetworkManager # NetworkManager
```

```
ls /etc/network/interfaces.d/ 2>/dev/null
ls /etc/systemd/network/ 2>/dev/null
```

For a server installed via the standard Debian installer (as in Chapter 1), `ifupdown` is the most common default, and this chapter's primary examples use it, with `systemd-networkd` shown as an alternative where relevant.

Inspecting Current Network State

Before changing anything, build a clear picture of the current configuration:

```
bash
ip addr show      # interfaces and assigned addresses
ip link show      # interface state (up/down), MAC addresses
ip route show     # routing table
ip -6 addr show   # IPv6 addresses specifically
cat /etc/resolv.conf # current DNS resolver configuration
```

Identify the primary network interface name (commonly `eth0` on older configurations or predictable names like `enp1s0`, `ens18` on modern kernels using `systemd`'s predictable network interface naming):

```
bash
ip -brief link show
```

Architectural Considerations Before Configuring

Before assigning addresses or writing configuration files, resolve several architectural questions, echoing the planning guidance from the Introduction:

Single or multiple interfaces? Many servers—particularly VPS and cloud instances—have a single network interface handling all traffic. Servers with more demanding requirements (dedicated storage networks, separate management and public-facing traffic) may have multiple interfaces, each requiring distinct configuration and firewall treatment.

IPv4 only, or dual-stack IPv4/IPv6? Debian fully supports IPv6, and an increasing share of infrastructure—particularly cloud providers—offers IPv6 addressing by default. Decide deliberately whether IPv6 should be configured, and if so, ensure firewall rules (Chapter 7) are applied consistently across both protocol families, a commonly overlooked gap.

Static or dynamic addressing? As discussed in Chapter 1, DHCP is convenient during installation, but production servers overwhelmingly benefit from static addressing (or a DHCP reservation functioning equivalently) a server's IP address is frequently referenced by DNS records, firewall rules, monitoring configuration, and other systems, and an address that can silently change on lease renewal introduces fragile, hard-to-diagnose failure modes.

3.2 Static IP Addresses, Routes, and DNS Configuration

Configuring a Static IP with ifupdown

Edit `/etc/network/interfaces` (or, preferably, a file under `/etc/network/interfaces.d/` for cleaner organization on more complex configurations):

```
bash
sudo nano /etc/network/interfaces
# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface  static configuration
auto enp1s0
iface enp1s0 inet static
    address 192.0.2.10/24
    gateway 192.0.2.1
    dns-nameservers 9.9.9.9 149.112.112.112
    dns-search example.com
```

Key directives: `auto enp1s0` ensures the interface is brought up automatically at boot; `address` specifies the IP address and prefix length (CIDR notation); `gateway` sets the default route; `dns-nameservers` and `dns-search` configure resolver behavior via `resolvconf` integration (if installed) or a compatible mechanism.

Warning: Applying network configuration changes over an active SSH session carries real risk of losing access if the configuration is incorrect. Before restarting networking, ensure you have console access (physical, virtual, or provider-based out-of-band access, as discussed in Chapter 1) as a fallback, and consider testing the syntax and reviewing the change carefully before applying it.

Apply the configuration:

```
bash
```

```
sudo systemctl restart networking
```

On many modern Debian installations using `systemd-networkd` or where `ifupdown`'s full restart is considered too risky for a live interface, a more surgical approach using `ip` directly, followed by making the change persistent, is often safer:

```
bash
```

```
# Immediate, non-persistent change (verify connectivity before making it permanent)
```

```
sudo ip addr add 192.0.2.10/24 dev enp1s0
```

```
sudo ip route add default via 192.0.2.1
```

Test connectivity thoroughly before finalizing the persistent configuration file, and only then restart networking (or reboot) to confirm the persistent configuration reproduces the working state achieved manually.

Configuring Static IP with `systemd-networkd`

If `systemd-networkd` is the active mechanism, create a `.network` file:

```
bash
```

```
sudo nano /etc/systemd/network/10-static-enp1s0.network
```

```
ini
```

```
[Match]
```

```
Name=enp1s0
```

```
[Network]
```

```
Address=192.0.2.10/24
```

```
Gateway=192.0.2.1
```

```
DNS=9.9.9.9
```

```
DNS=149.112.112.112
```

```
Domains=example.com
```

```
bash
```

```
sudo systemctl restart systemd-networkd
```

DNS Resolver Configuration

Modern Debian systems typically manage `/etc/resolv.conf` dynamically through `systemd-resolved` or `resolvconf`, rather than as a hand-edited static file. Verify which mechanism is in effect:

```
bash
```

```
ls -la /etc/resolv.conf
resolvectl status
```

If `/etc/resolv.conf` is a symlink to `/run/systemd/resolve/stub-resolv.conf` or similar, `systemd-resolved` is managing resolution, and DNS servers should be configured through the interface's network configuration (as shown above), not by editing `/etc/resolv.conf` directly, since any manual edit would be overwritten.

Verify DNS resolution functions correctly after configuration:

```
bash
resolvectl query debian.org
dig debian.org
nslookup debian.org
```

`dig` provides the most detailed output, including query time and the specific resolver used – valuable when diagnosing DNS-related problems, a common category covered further in section 3.4.

Configuring Routes Beyond the Default Gateway

For servers with multiple networks (a management network and a public network, for example), additional routes may be necessary:

```
bash
sudo ip route add 10.10.0.0/16 via 192.0.2.254 dev enp1s0
ip route show
```

To make additional routes persistent under `ifupdown`, add a `post-up` directive within the relevant `iface` stanza in `/etc/network/interfaces`:

```
iface enp1s0 inet static
    address 192.0.2.10/24
    gateway 192.0.2.1
    post-up ip route add 10.10.0.0/16 via 192.0.2.254 dev enp1s0
```

3.3 Network Interfaces, Ports, and Connectivity

Understanding TCP and UDP Ports in a Server Context

Every network service listens on a specific combination of IP address, port, and protocol (TCP or UDP). Understanding what is actually listening and

on which interfaces is fundamental both to normal operation and to security review.

bash

```
sudo ss -tulnp
```

This command, `ss` (socket statistics, the modern replacement for the older `netstat`), with flags `-t` (TCP), `-u` (UDP), `-l` (listening sockets only), `-n` (numeric addresses/ports, avoiding slow reverse-DNS lookups), and `-p` (show the owning process, requires root) is the single most useful command for understanding a server's current network exposure.

Example output interpretation:

```
Netid State Local Address:Port Peer Address:Port Process
tcp LISTEN 0.0.0.0:22      0.0.0.0:*      users:(("sshd",pid=612,fd=3))
tcp LISTEN 127.0.0.1:5432    0.0.0.0:*      users:(("postgres",pid=890,fd=5))
tcp LISTEN 0.0.0.0:80      0.0.0.0:*      users:(("nginx",pid=1200,fd=6))
```

This distinction matters critically: `0.0.0.0:22` means SSH is listening on **all** interfaces, reachable from any network the server is connected to (including, on a cloud/VPS instance, the public internet, unless blocked by a firewall). `127.0.0.1:5432`, by contrast, shows PostgreSQL bound only to the loopback interface reachable only from processes on the same machine, not from the network at all. This is a deliberate and common security pattern (elaborated on in Chapter 6 and Chapter 7): binding sensitive services to loopback or to a private/internal interface only, rather than relying solely on firewall rules to restrict access to a service that is technically listening publicly.

Checking Specific Port Availability

bash

```
sudo ss -tulnp | grep :443
```

```
sudo lsof -i :443 # requires: sudo apt install lsof
```

Testing Outbound and Inbound Connectivity

bash

```
# Outbound connectivity to a remote host/port
```

```
nc -zv google.com 443
```

```
curl -Iv https://debian.org
```

```
# Local port reachability test (run from another machine)
nc -zv <server-ip> 22
```

`nc -zv` (netcat, zero-I/O mode, verbose) attempts a TCP connection without sending data, reporting success or failure – useful for quickly confirming whether a port is reachable without needing a full application-level client. `curl -Iv` performs a verbose HEAD request, useful for diagnosing HTTPS/TLS-layer connectivity specifically, a technique returned to in Chapter 5.

Interface-Level Diagnostics

```
bash
ip -s link show enp1s0 # interface statistics: packets, errors, drops
ethtool enp1s0         # link speed, duplex, driver info (requires: sudo apt install ethtool)
```

`ip -s link show` is particularly useful for diagnosing performance problems at the link layer – a steadily incrementing error or drop counter on an interface points toward a physical, driver, or switch-level problem rather than anything in the server's own configuration.

3.4 Network Diagnostics and Troubleshooting

Network problems are among the most common – and most disruptive – issues a server administrator encounters, precisely because they often affect the very access path needed to diagnose them. This section builds a systematic diagnostic approach, anticipating the fuller troubleshooting methodology developed in Chapter 11.

A Layered Diagnostic Approach

Diagnose network problems from the bottom of the stack upward, confirming each layer before assuming a higher layer is at fault:

1. Physical/link layer – is the interface up?

```
bash
ip link show enp1s0
```

Look for `state UP` in the output; `state DOWN` indicates a link-layer problem (cable, virtual NIC misconfiguration, switch port issue) that must be

resolved before anything above it can function.

2. IP layer does the interface have the expected address?

```
bash
ip addr show enp1s0
```

3. Routing is there a valid path to the destination?

```
bash
ip route get 8.8.8.8
```

`ip route get` shows exactly which route and source address the kernel would use to reach a given destination extremely useful for confirming routing decisions match expectations without generating actual traffic.

4. Local connectivity (same subnet)

```
bash
ping -c 4 192.0.2.1 # gateway
arp -n             # or: ip neigh show confirms ARP resolution on the local segment
```

5. Remote connectivity (beyond the local network)

```
bash
ping -c 4 9.9.9.9    # a known, reliable external IP bypasses DNS to isolate DNS-layer issues
traceroute 9.9.9.9    # or: mtr 9.9.9.9 for continuous, combined ping+traceroute output
```

Pinging a known IP address directly (bypassing hostname resolution) is a deliberate technique to distinguish DNS problems from actual routing/connectivity problems if the IP ping succeeds but a hostname ping fails, the problem is isolated specifically to DNS resolution, not general connectivity.

6. DNS resolution

```
bash
dig debian.org
dig @9.9.9.9 debian.org # query a specific resolver directly, bypassing local config
resolvectl status
```

7. Port/service-level connectivity

```
bash
```

```
nc -zv <target> <port>
curl -v https://<target>/
```

Common Network Problems and Their Signatures

Symptom	Likely Cause	Diagnostic Command
Interface shows state DOWN	Cable/virtual NIC issue, driver problem	ip link show
Interface up, no IP address	DHCP failure, misconfigured static config	ip addr show , check DHCP client logs
Can ping gateway, not beyond	Routing misconfiguration, upstream ISP/provider issue	ip route show , traceroute
Can ping by IP, not by hostname	DNS resolver misconfiguration	dig , resolvectl status
Connection refused on expected port	Service not running, or bound to wrong interface	ss -tulnp
Connection times out (no refusal)	Firewall (local or network-level) silently dropping traffic	Check nft list ruleset , provider security group
Intermittent packet loss	Physical layer issue, network congestion, MTU mismatch	mtr , ip -s link show

The distinction in the table between "connection refused" and "connection times out" is a particularly valuable diagnostic signal: a refused connection means a host was reached but nothing is listening (or a firewall actively rejected it with an RST), while a timeout typically indicates traffic is being silently dropped somewhere along the path — often a firewall rule with a default-drop policy, whether local (Chapter 7) or at the network/provider level.

Capturing Traffic for Deeper Analysis

For problems that resist the diagnostics above, packet capture provides ground truth:

```
bash
sudo apt install tcpdump
sudo tcpdump -i enp1s0 port 443 -n -c 50
```

This captures 50 packets on interface `enp1s0` filtered to port 443, with `-n` disabling reverse-DNS lookups (which can otherwise slow capture and clutter output). For more complex analysis, capture to a file and analyze with Wireshark on a workstation:

```
bash
sudo tcpdump -i enp1s0 -w /tmp/capture.pcap port 443
```

Warning: Packet capture can expose sensitive data including credentials transmitted in cleartext protocols. Use capture judiciously, secure or delete capture files promptly after analysis, and never capture broader traffic than necessary for the diagnostic task at hand.

3.5 Designing Reliable Server Network Configurations

Documenting Network Configuration

Given the operational risk network misconfiguration carries, maintaining accurate, current documentation of a server's network configuration addresses, gateway, DNS servers, any static routes, and the specific mechanism (`ifupdown` , `systemd-networkd`) in use is a direct application of the Introduction's "documentation as an operational asset" principle, and materially reduces both the risk and the recovery time of future network changes.

Designing for Resilience

Avoid single points of DNS failure. Configure at least two DNS resolvers, ideally operated by different providers, so that a single resolver outage does not fully break name resolution.

Understand your provider's network redundancy model. For VPS and cloud deployments, understand whether the provider offers redundant network paths, and whether a floating/reserved IP mechanism exists that could allow rapid failover to a replacement instance in the event of a hardware failure a concept more fully relevant in the production project work of Chapter 12, but worth understanding at the planning stage.

Plan MTU consistently, especially in virtualized/tunneled environments. Certain network configurations VPNs, some cloud provider overlay networks, certain container networking modes covered in Chapter 8 can introduce MTU mismatches that manifest as intermittent, hard-to-diagnose connectivity problems (often specifically with larger

packets, while small packets like ICMP pings succeed normally). Where such symptoms appear, checking and explicitly setting MTU consistently across the relevant path is a diagnostic step worth remembering:

bash

```
ip link show enp1s0 | grep mtu
sudo ip link set dev enp1s0 mtu 1450
```

Establishing a Pre-Change Safety Habit for Network Configuration

Given the risk of losing remote access, adopt a consistent discipline before any network configuration change on a production server:

1. Confirm out-of-band/console access is available and working *before* starting.
2. Record the current, working configuration (`ip addr show` , `ip route show` , resolver configuration) before changing anything.
3. Where possible, test the new configuration non-persistently first (applying with `ip` directly, rather than editing the persistent configuration file immediately), verifying it works before making it permanent.
4. Keep an existing SSH session open while testing connectivity through a new configuration, rather than closing it and hoping the new configuration works.
5. If using a scheduled/delayed reboot as a safety net for risky changes, consider scheduling an automatic revert:

bash

```
# Schedule a reboot in 5 minutes as a safety net  cancel it once the change is confirmed working
sudo shutdown -r +5 "Scheduled safety reboot after network config change"
# If the change succeeds and access is confirmed, cancel the pending reboot:
sudo shutdown -c
```

This "dead man's switch" pattern scheduling an automatic reboot that reverts to a last-known-good configuration if you fail to cancel it is a genuinely valuable technique for higher-risk network changes on remote systems, and is worth having in an administrator's habitual toolkit specifically for this kind of high-consequence, hard-to-reverse operation.

With reliable, well-understood networking established, the server is ready for the remote administration channel every subsequent chapter depends on:

SSH, covered next in full depth.

Chapter 4 : SSH and Remote Server Management

4.1 Deploying and Configuring OpenSSH

SSH (Secure Shell) is the near-universal mechanism for remote Debian server administration, and its configuration represents one of the highest-leverage security decisions an administrator makes it is simultaneously the primary legitimate access path to the server and the primary target of automated attacks against it.

Installing and Verifying OpenSSH Server

If not already installed during initial setup (Chapter 1):

```
bash
sudo apt update
sudo apt install openssh-server
sudo systemctl status ssh
sudo systemctl enable --now ssh
```

Confirm the service is listening as expected:

```
bash
sudo ss -tulnp | grep ssh
```

Understanding sshd_config

The SSH daemon's primary configuration file is `/etc/ssh/sshd_config`. Always back up before editing, and critically **validate syntax before restarting the service**, since a malformed configuration can prevent SSH from starting at all, potentially locking out all remote access:

```
bash
sudo cp /etc/ssh/sshd_config /etc/ssh/sshd_config.bak.$(date +%Y%m%d)
sudo nano /etc/ssh/sshd_config
sudo sshd -t      # test configuration syntax before applying
sudo systemctl reload ssh
```

`sshd -t` parses the configuration and reports any syntax errors without affecting the running daemon. Run this after every edit, before reloading, without exception.

Baseline Configuration Directives

Several directives deserve attention even before the deeper hardening covered in section 4.4:

```
# /etc/ssh/sshd_config
```

```
Port 22
AddressFamily any
ListenAddress 0.0.0.0
```

```
Protocol 2
LogLevel VERBOSE
```

```
PermitRootLogin yes      # will be hardened to 'no' in 4.4, once key auth is confirmed
PasswordAuthentication yes # will be hardened to 'no' in 4.4, once key auth is confirmed
```

```
X11Forwarding no
PrintMotd no
```

```
ClientAliveInterval 300
ClientAliveCountMax 2
```

`LogLevel VERBOSE` provides more detailed authentication logging than the default, valuable for both routine audit and incident investigation.

`ClientAliveInterval` / `ClientAliveCountMax` configure server-side keepalive checks, disconnecting genuinely dead sessions after roughly 10 minutes of unresponsiveness ($300 \text{ seconds} \times 2$), which helps prevent orphaned sessions from consuming resources indefinitely.

Important sequencing note: This section deliberately leaves `PermitRootLogin` and `PasswordAuthentication` at their permissive defaults. These are hardened in section 4.4, *after* key-based authentication (section 4.2) has been configured and independently verified. Tightening these settings before key authentication is confirmed working is a common and entirely avoidable cause of SSH lockouts.

4.2 SSH Key-Based Authentication

Public-key authentication is both more secure and more convenient than password authentication, and is the foundation the rest of this chapter's hardening builds on.

Understanding SSH Key Pairs

An SSH key pair consists of a private key (kept secret, never shared, ideally protected by a passphrase) and a public key (safely shareable, installed on servers you wish to access). Authentication works by the server challenging the client to prove possession of the private key corresponding to a public key it has been told to trust the private key itself is never transmitted.

Generating a Key Pair (Client-Side)

If not already done during provisioning (Chapter 1):

```
bash
ssh-keygen -t ed25519 -C "adminuser@debian-server" -f ~/.ssh/id_ed25519_debian_server
```

When prompted, set a strong passphrase protecting the private key an unprotected private key that is stolen or copied grants immediate access, while a passphrase-protected key requires the passphrase in addition to the key file itself, an important additional layer particularly for keys used from laptops or other devices that could be lost or stolen.

Ed25519 is recommended for new keys over the older RSA algorithm, offering equivalent or stronger security with smaller key sizes and faster operations. Where compatibility with older systems requires RSA, use at least 4096 bits:

```
bash
ssh-keygen -t rsa -b 4096 -C "adminuser@debian-server" -f ~/.ssh/id_rsa_debian_server
```

Installing the Public Key on the Server

```
bash
ssh-copy-id -i ~/.ssh/id_ed25519_debian_server.pub adminuser@<server-ip>
```

ssh-copy-id handles creating ~/.ssh/ with correct permissions on the remote server and appending the public key to ~/.ssh/authorized_keys . If ssh-copy-id is unavailable, the equivalent manual process is:

```
bash
```

```
cat ~/.ssh/id_ed25519_debian_server.pub | ssh adminuser@<server-ip> \  
'mkdir -p ~/.ssh && chmod 700 ~/.ssh && cat >> ~/.ssh/authorized_keys && chmod 600  
~/.ssh/authorized_keys'
```

The explicit permission settings matter: SSH refuses to use an `authorized_keys` file (or `.ssh` directory) with overly permissive permissions, as a security measure against tampering by other local users. `700` for the directory, `600` for the file, are the required baseline.

Verifying Key-Based Authentication Before Disabling Passwords

This verification step is not optional. Open a new terminal (leaving any existing session open) and confirm key-based login succeeds:

```
bash
```

```
ssh -i ~/.ssh/id_ed25519_debian_server adminuser@<server-ip>
```

Confirm you were logged in without being prompted for a password (only, if configured, the key's own passphrase). Only after this is confirmed working should password authentication be disabled in section 4.4.

Managing Multiple Keys with SSH Config

For administrators managing several servers, an SSH client configuration file (`~/.ssh/config`) simplifies key management and connection details:

```
Host debian-server  
  HostName 192.0.2.10  
  User adminuser  
  IdentityFile ~/.ssh/id_ed25519_debian_server  
  IdentitiesOnly yes  
  Port 22
```

With this configuration, `ssh debian-server` connects using exactly the specified user, key, and port, without needing to specify them on every invocation.

4.3 Managing Administrative Access and Privileges

Restricting SSH Access to Specific Users or Groups

Beyond authentication method, `sshd_config` allows explicit control over *who* may connect via SSH at all – a valuable additional layer, particularly on systems with many local accounts that don't all need remote access:

```
AllowUsers adminuser deployuser  
# or, for group-based control:  
AllowGroups sshusers
```

Using `AllowGroups` with a dedicated group (created via `sudo groupadd sshusers` and populated with `sudo usermod -aG sshusers <user>`) is generally more maintainable than an `AllowUsers` list that must be edited every time access changes, since group membership can be managed through normal user administration commands rather than repeated `sshd_config` edits.

Separating Interactive Administrative Access from Service Accounts

As established in Chapter 2, service accounts should not have SSH login capability at all. Confirm this explicitly:

```
bash  
sudo usermod -s /usr/sbin/nologin myappuser
```

Setting a service account's shell to `/usr/sbin/nologin` (or `/bin/false`) prevents interactive login (including via SSH) while still allowing the account to own processes and files – an important least-privilege boundary distinct from, and complementary to, `AllowUsers` / `AllowGroups` restrictions at the SSH layer.

Restricting sudo Access Over SSH for Specific Contexts

For particularly sensitive systems, some organizations restrict full interactive `sudo` shells over SSH in favor of specific, audited command delegation (as introduced in section 2.1's `sudoers.d` example). This is a design choice with real usability tradeoffs, and is generally appropriate only for narrowly-scoped service or automation accounts, not for general human administrative access, where a full audited shell with logging (see section 4.5) is typically the more practical and equally accountable approach.

Session and Login Auditing

```
bash  
last -a # recent login history
```

```
lastb # failed login attempts (requires appropriate log access)
sudo journalctl -u ssh --since "24 hours ago" | grep -i "accepted\\|failed"
```

Reviewing `last` and authentication logs periodically and certainly after any suspicious activity is suspected is a direct, practical application of the security monitoring principle developed fully in Chapter 7.

4.4 SSH Hardening and Attack Reduction

With key-based authentication verified and working (section 4.2), the following hardening steps meaningfully reduce SSH's attack surface. Apply each change incrementally, testing access after each one, rather than making every change simultaneously.

Disabling Password Authentication

```
PasswordAuthentication no
KbdInteractiveAuthentication no
ChallengeResponseAuthentication no
```

This eliminates the entire class of brute-force password-guessing attacks against SSH by far the most common form of automated attack observed against internet-facing SSH servers. Test again with `sudo sshd -t`, then reload:

```
bash
sudo sshd -t && sudo systemctl reload ssh
```

Before closing your current session, open a fresh connection attempt to confirm key-based login still works and password attempts are correctly rejected.

Disabling Direct Root Login

PermitRootLogin no

With a properly configured administrative user and `sudo` access (Chapter 2) confirmed working, direct root SSH login serves no legitimate purpose and represents a significant risk reduction—it eliminates root as a directly attackable account over SSH entirely, forcing any legitimate administrative access (and any attacker who somehow obtained valid credentials) through a named, audited user account first.

Changing the Default SSH Port (Optional, Limited Value)

Port 2222

Changing the default port reduces the volume of automated, opportunistic scanning traffic (most mass scanners target port 22 specifically), which can meaningfully reduce log noise and marginal attack surface. However, this should be understood clearly as **security through obscurity, not a substitute for genuine hardening**—a determined attacker or a scanner configured to check all ports will find a service on any port. If used, ensure firewall rules (Chapter 7) are updated to match, and remember to specify the port explicitly in future SSH connections and `~/.ssh/config` entries.

Limiting Authentication Attempts and Connection Rate

MaxAuthTries 3

MaxSessions 4

LoginGraceTime 30

`MaxAuthTries` limits authentication attempts per connection before disconnecting; `LoginGraceTime` limits how long an unauthenticated connection may remain open, reducing the effectiveness of slow, resource-consuming connection attempts.

Using fail2ban for Automated Intrusion Prevention

`fail2ban` monitors authentication logs and automatically firewalls (temporarily bans) source addresses exhibiting repeated failed authentication attempts:

bash

```
sudo apt install fail2ban
```

ini

```
# /etc/fail2ban/jail.local
```

```
[sshd]
```

```
enabled = true
```

```
port = ssh
```

```
filter = sshd
```

```
logpath = %(sshd_log)s
```

```
maxretry = 4
```

```
findtime = 600
```

```
bantime = 3600
```

maxretry = 4 within findtime = 600 (10 minutes) triggers a ban lasting bantime = 3600 (1 hour). Apply and verify:

bash

```
sudo systemctl enable --now fail2ban
```

```
sudo fail2ban-client status sshd
```

fail2ban-client status sshd shows currently banned IPs and recent activity, useful both for confirming the jail is functioning and for routine security review.

Restricting SSH by Source Address (Where Feasible)

Where administrative access originates from known, stable source addresses (an office network, a VPN, a bastion host), restricting SSH access at the firewall level to only those sources represents one of the most effective hardening measures available an attacker cannot brute-force or exploit a service they cannot reach at all. This is covered in firewall configuration detail in Chapter 7, but is worth flagging here as a complementary, often more impactful measure than in-daemon hardening alone.

Post-Hardening Checklist

After applying hardening changes, systematically verify:

bash

```
sudo sshd -t
```

```
sudo systemctl status ssh
```

- Key-based login for the administrative account works, in a fresh session.

- Password authentication is correctly rejected.
- Direct root SSH login is correctly rejected.
- fail2ban (if installed) shows the `sshd` jail active.
- Firewall rules (once configured in Chapter 7) correctly permit the intended access while blocking everything else.

4.5 Secure Remote Administration Workflows

Secure File Transfer

SSH's authenticated, encrypted channel underlies several secure file transfer tools that administrators use routinely:

bash

scp - simple file/directory copy over SSH

scp local-file.txt adminuser@debian-server:/home/adminuser/

scp -r local-directory/ adminuser@debian-server:/opt/myapp/

rsync - efficient, incremental transfer, preferred for larger or repeated transfers

rsync -avz --progress local-directory/ adminuser@debian-server:/opt/myapp/

rsync is generally preferable to scp for anything beyond trivial single-file transfers: it transfers only changed data on subsequent runs (-a for archive mode, preserving permissions and timestamps; -z for compression), and its --dry-run flag allows previewing exactly what would be transferred before committing:

bash

rsync -avz --dry-run local-directory/ adminuser@debian-server:/opt/myapp/

Remote Command Execution

SSH allows executing a single remote command without an interactive session, useful for scripting and automation (developed fully in Chapter 9):

bash

ssh adminuser@debian-server "sudo systemctl status nginx"

ssh adminuser@debian-server "df -h" > /tmp/remote-disk-usage.txt

SSH Agent Forwarding and Its Risks

SSH agent forwarding allows a private key held on a local workstation to be used for further authentication from the remote server (for example, to git

clone a repository over SSH from the server itself, using the administrator's local key rather than placing a separate key on the server):

```
bash
```

```
ssh -A adminuser@debian-server
```

Security caution: Agent forwarding grants any process able to access the forwarded agent socket on the remote server (including, notably, root on that server, or another user if permissions are misconfigured) the ability to use your private key for authentication *while the forwarded session is active* though not to extract or copy the key itself. Use agent forwarding selectively, only to servers you trust completely, and prefer more scoped alternatives (a dedicated deploy key installed directly on the server, for instance) for routine automated workflows.

Persistent and Multiplexed Sessions

For administrators frequently connecting to the same server, SSH connection multiplexing significantly speeds up repeated connections by reusing a single underlying TCP connection:

```
# ~/.ssh/config
Host debian-server
    HostName 192.0.2.10
    User adminuser
    IdentityFile ~/.ssh/id_ed25519_debian_server
    ControlMaster auto
    ControlPath ~/.ssh/sockets/%r@%h-%p
    ControlPersist 600
```

ControlPersist 600 keeps the underlying connection alive for 10 minutes after the last session closes, making subsequent connections within that window nearly instantaneous.

Using tmux or screen for Resilient Sessions

Long-running administrative tasks (large file transfers, lengthy builds, extended monitoring sessions) are vulnerable to being interrupted by a dropped SSH connection. Terminal multiplexers solve this by running the session on the server independently of the SSH connection itself:

```
bash
```

```
sudo apt install tmux
```

```
tmux new -s maintenance
# ... perform work; if disconnected, reconnect and run:
tmux attach -t maintenance
```

This pattern — starting long-running or important interactive work inside `tmux` (or the older but similar `screen`) — is a habit worth adopting broadly, since it converts what would otherwise be a connectivity-dependent, fragile operation into one that survives network interruptions cleanly.

Building an Administrative Access Runbook

As a closing practice for this chapter, document — for every server under your administration — the specific access details future-you (or a colleague) will need under pressure: the SSH port in use, the administrative account name, where the corresponding private key is kept, whether `fail2ban` or source-IP restrictions are in effect, and the out-of-band/console access method available if SSH access is ever lost entirely. This is a direct, concrete application of the Introduction's documentation principle, and is exactly the kind of information whose absence turns a minor access hiccup into a genuine incident.

With secure, hardened remote access firmly established, the book turns to the services Debian servers most commonly exist to run: web and application services, covered next.

Chapter 5 : Web and Application Server Administration

5.1 Deploying a Web Server on Debian

Web servers are among the most common workloads Debian servers host, and their administration touches nearly every discipline covered so far: package management, service configuration, networking, and critically the security posture of anything exposed to public traffic.

Choosing Between Nginx and Apache

Debian's repositories offer mature packages for both Nginx and Apache HTTP Server (`apache2`), and both are entirely capable, production-grade choices. The practical differences that typically drive the decision:

Consideration	Nginx	Apache (<code>apache2</code>)
Architecture	Event-driven, asynchronous	Traditionally process/thread-based (though event MPM narrows this gap)
Static content performance	Excellent, very low memory per connection	Good, generally higher memory per connection
Reverse proxy role	Extremely common, first-class configuration	Fully capable via <code>mod_proxy</code> , less commonly the primary use case
<code>.htaccess</code> -style per-directory config	Not supported (by design)	Supported, useful for shared/multi-tenant hosting
Configuration style	Declarative, block-based	Directive-based, historically more granular per-directory control
Common role in modern stacks	Reverse proxy in front of application servers	Traditional full-stack web serving, especially with embedded module-based languages

Neither is objectively "better" Nginx is generally favored for high-concurrency, reverse-proxy-centric, and static-content-heavy deployments, while Apache remains a strong choice where its per-directory configuration flexibility or specific module ecosystem is genuinely needed. This chapter covers both, since real-world Debian infrastructure runs both extensively, sometimes even together (Nginx as a reverse proxy in front of Apache).

Installing Nginx

bash

```
sudo apt update
sudo apt install nginx
sudo systemctl enable --now nginx
sudo systemctl status nginx
```

Verify from the server itself and confirm the default page is served:

bash

```
curl -I http://localhost/
```

Installing Apache

bash

```
sudo apt install apache2
sudo systemctl enable --now apache2
sudo systemctl status apache2
curl -I http://localhost/
```

Understanding the Package Layout

Nginx's configuration lives primarily under `/etc/nginx/`, with `nginx.conf` as the main file and `sites-available` / `sites-enabled` directories (linked via symlinks) managing individual site configurations:

bash

```
ls /etc/nginx/sites-available/
ls /etc/nginx/sites-enabled/
```

Apache follows a broadly similar convention, with `apache2.conf` as the main file and `sites-available` / `sites-enabled`, alongside `mods-available` / `mods-enabled` for module management:

bash

```
ls /etc/apache2/sites-available/
sudo a2enmod rewrite      # enable a module
sudo a2dismod status      # disable a module
sudo a2ensite mysite      # enable a site config
sudo a2dissite 000-default # disable the default site
```

Apache's `a2en*` / `a2dis*` helper commands manage the symlinks between the `-available` and `-enabled` directories automatically, which is a meaningfully more convenient workflow than Nginx's typically manual symlink management for site enablement.

5.2 Configuring Nginx and Apache HTTP Server

A Realistic Nginx Server Block

```
bash
sudo nano /etc/nginx/sites-available/example.com
nginx
server {
    listen 80;
    listen [::]:80;
    server_name example.com www.example.com;

    root /var/www/example.com/html;
    index index.html index.htm;

    access_log /var/log/nginx/example.com.access.log;
    error_log /var/log/nginx/example.com.error.log;

    location / {
        try_files $uri $uri/ =404;
    }

    location ~*\.(jpg|jpeg|png|gif|css|js|ico|svg)$ {
        expires 30d;
        add_header Cache-Control "public, immutable";
    }

    client_max_body_size 10M;
}
```

Key directives explained: `server_name` matches incoming `Host` headers to this block, allowing a single server to host multiple sites via name-based virtual hosting (covered fully in 5.3); `try_files` attempts to serve the requested URI as a file, then a directory, then returns `404` if neither exists, a standard and safe default; the regex `location` block applies aggressive caching headers to static assets, a meaningful performance practice for real sites; `client_max_body_size` caps upload size, a sensible default protection against unexpectedly large request bodies.

Enable and test:

```
bash
sudo mkdir -p /var/www/example.com/html
echo "<h1>Example site</h1>" | sudo tee /var/www/example.com/html/index.html
sudo ln -s /etc/nginx/sites-available/example.com /etc/nginx/sites-enabled/
sudo nginx -t
sudo systemctl reload nginx
```

`nginx -t` tests configuration syntax and reports errors without affecting the running server. Run this before every reload, without exception, mirroring the discipline established for SSH configuration in Chapter 4.

An Equivalent Apache Virtual Host

bash

```
sudo nano /etc/apache2/sites-available/example.com.conf
```

apache

```
<VirtualHost *:80>
    ServerName example.com
    ServerAlias www.example.com
    DocumentRoot /var/www/example.com/html

    ErrorLog ${APACHE_LOG_DIR}/example.com.error.log
    CustomLog ${APACHE_LOG_DIR}/example.com.access.log combined

    <Directory /var/www/example.com/html>
        Options -Indexes +FollowSymLinks
        AllowOverride None
        Require all granted
    </Directory>
</VirtualHost>
```

`Options -Indexes` disables automatic directory listing (a common, easily overlooked information-disclosure risk); `AllowOverride None` disables `.htaccess` processing for this directory, improving both performance and predictability unless per-directory override is specifically needed; `Require all granted` (Apache 2.4+ syntax) permits access from all clients at the HTTP layer, distinct from any firewall-level restriction.

bash

```
sudo a2ensite example.com.conf
sudo apachectl configtest
sudo systemctl reload apache2
```

`apachectl configtest` is Apache's equivalent of `nginx -t`—always validate before reloading.

Reverse Proxy Configuration

A very common role for Nginx specifically is as a reverse proxy in front of an application server (covered in section 5.4):

nginx

```

server {
    listen 80;
    server_name app.example.com;

    location / {
        proxy_pass http://127.0.0.1:8000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_read_timeout 60s;
    }
}

```

The `proxy_set_header` directives forward important context the backend application needs but would otherwise lose behind a proxy the original client IP (`X-Real-IP` / `X-Forwarded-For`) and the original protocol (`X-Forwarded-Proto` , essential once TLS termination happens at the proxy, covered next). An application unaware of these headers will incorrectly log the proxy's own IP as every client's address, and may make incorrect decisions about whether a request was originally HTTPS.

5.3 Virtual Hosts, TLS, and HTTPS

Name-Based Virtual Hosting

Both Nginx (`server_name` / `server` blocks) and Apache (`ServerName` / `VirtualHost`) support hosting multiple independent sites on a single IP address by matching the HTTP `Host` header, as shown in the examples above. This is the standard mechanism underlying most shared web infrastructure, and requires no special network configuration beyond correct DNS records pointing each hostname at the server's address.

Obtaining TLS Certificates with Let's Encrypt

Let's Encrypt, accessed via the `certbot` tool, provides free, automatically renewable TLS certificates and is the standard mechanism for securing Debian-hosted websites with HTTPS:

```

bash
sudo apt install certbot python3-certbot-nginx
# or, for Apache:
sudo apt install certbot python3-certbot-apache

```

For Nginx, with the site already configured and reachable via HTTP (required for domain validation):

```
bash
sudo certbot --nginx -d example.com -d www.example.com
```

Certbot automatically detects the relevant Nginx server block, obtains a certificate, and modifies the configuration to enable HTTPS, including an HTTP-to-HTTPS redirect (offered interactively during the process). For Apache, the equivalent:

```
bash
sudo certbot --apache -d example.com -d www.example.com
```

Verifying and Understanding the Resulting Configuration

Inspect the modified Nginx configuration to understand what Certbot added:

```
nginx
server {
    listen 443 ssl;
    server_name example.com www.example.com;

    ssl_certificate /etc/letsencrypt/live/example.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/example.com/privkey.pem;
    include /etc/letsencrypt/options-ssl-nginx.conf;
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem;

    # ... original location blocks ...
}

server {
    listen 80;
    server_name example.com www.example.com;
    return 301 https://$host$request_uri;
}
```

The second `server` block redirects all HTTP traffic to HTTPS, a standard and important practice ensuring the site is never served insecurely once a certificate is in place.

Automatic Renewal

Let's Encrypt certificates are valid for 90 days; Certbot installs a systemd timer (or cron job, on older configurations) to handle renewal automatically.

Verify it is present and functioning:

bash

```
systemctl list-timers | grep certbot  
sudo certbot renew --dry-run
```

`certbot renew --dry-run` simulates the renewal process without actually requesting a new certificate, a safe and recommended way to confirm the renewal mechanism is correctly configured before it needs to run for real.

Verifying TLS Configuration Quality

Beyond simply having a certificate, confirm the resulting TLS configuration is sound:

bash

```
curl -Iv https://example.com/ 2>&1 | grep -i "SSL|TLS"  
openssl s_client -connect example.com:443 -servername example.com < /dev/null 2>/dev/null |  
openssl x509 -noout -dates
```

The `openssl x509 -noout -dates` output confirms the certificate's validity window, useful for a quick manual sanity check independent of Certbot's own renewal tracking.

5.4 Deploying Applications and Managing Services

The General Pattern: Application Server Behind a Reverse Proxy

A very common and robust architecture for deploying a custom application (a Python/Django or Flask app, a Node.js service, or similar) on Debian combines: an application server process, managed as a systemd service (as introduced in section 2.3); listening on a local, non-public port; with Nginx as a reverse proxy in front of it, handling TLS termination, static file serving, and public exposure.

Example: Deploying a Python WSGI Application with Gunicorn

bash

```
sudo adduser --system --group --no-create-home --home /opt/myapp myappuser
sudo mkdir -p /opt/myapp
# ... deploy application code to /opt/myapp ...

cd /opt/myapp
sudo -u myappuser python3 -m venv venv
sudo -u myappuser ./venv/bin/pip install gunicorn django
```

Create a systemd service, applying the patterns established in section 2.3:

ini

```
# /etc/systemd/system/myapp.service
[Unit]
Description=Gunicorn instance for myapp
After=network.target

[Service]
Type=simple
User=myappuser
Group=myappuser
WorkingDirectory=/opt/myapp
Environment=DJANGO_SETTINGS_MODULE=myapp.settings.production
ExecStart=/opt/myapp/venv/bin/gunicorn --workers 3 --bind 127.0.0.1:8000 myapp.wsgi:application
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target

bash
sudo systemctl daemon-reload
```

```
sudo systemctl enable --now myapp.service
sudo systemctl status myapp.service
```

Pair this with the Nginx reverse proxy configuration from section 5.2, pointing `proxy_pass` at `http://127.0.0.1:8000`. This architecture application bound only to loopback, reverse proxy handling public exposure and TLS is a direct application of the least-exposure principle introduced in Chapter 3's discussion of service binding.

Managing Application Configuration and Secrets

Avoid hardcoding secrets (database passwords, API keys) directly in application code or world-readable configuration files. A simple, effective pattern is an environment file referenced by the systemd unit, with restricted permissions:

```
bash
sudo nano /opt/myapp/.env
DATABASE_URL=postgresql://myapp:strongpassword@127.0.0.1:5432/myapp_production
SECRET_KEY=a-long-random-value
bash
sudo chown myappuser:myappuser /opt/myapp/.env
sudo chmod 600 /opt/myapp/.env
ini
[Service]
EnvironmentFile=/opt/myapp/.env
```

Restricting this file to `600`, owned by the application's own service account, ensures only that account (and root) can read the secrets it contains a direct, practical application of least privilege at the file-permission level.

Health Checks and Process Supervision

Beyond systemd's own `Restart=on-failure` behavior, many applications benefit from an explicit health-check endpoint that monitoring (Chapter 10) can query:

```
bash
curl -sf http://127.0.0.1:8000/healthz && echo "OK" || echo "FAILING"
```

Building a simple, dependency-free health endpoint into any custom application (confirming it can respond to a basic request, and ideally that it

can reach its database) is a small investment that substantially improves both automated monitoring and manual troubleshooting.

5.5 Diagnosing Web Server and Application Problems

A Systematic Approach to Web Server Troubleshooting

When a web service is not behaving as expected, work through layers similar to the network diagnostic approach in Chapter 3:

1. Is the service running at all?

```
bash
sudo systemctl status nginx
sudo systemctl status myapp.service
```

2. Is it listening on the expected address and port?

```
bash
sudo ss -tulnp | grep -E ':80|:443|:8000'
```

3. Does the configuration have valid syntax?

```
bash
sudo nginx -t
sudo apachectl configtest
```

4. What do the access and error logs show?

```
bash
sudo tail -f /var/log/nginx/example.com.error.log
sudo tail -f /var/log/nginx/example.com.access.log
sudo journalctl -u myapp.service -f
```

5. Does a local request succeed, isolating the problem from external network/DNS issues?

```
bash
curl -v http://127.0.0.1/
curl -v http://127.0.0.1:8000/ # direct to the application, bypassing the proxy
```

Testing directly against the backend application (bypassing the reverse proxy) is a particularly valuable isolation technique: if the direct request to 127.0.0.1:8000 succeeds but the proxied request through Nginx on port 80/443 fails, the problem is isolated to the proxy configuration; if both fail, the problem lies in the application itself.

Common HTTP Error Codes and Their Typical Causes

Status	Meaning	Common Server-Side Cause
502 Bad Gateway	Proxy could not get a valid response from the backend	Backend application not running, crashed, or unreachable at the configured <code>proxy_pass</code> address
503 Service Unavailable	Server temporarily unable to handle the request	Backend overloaded, explicitly returning 503, or a maintenance mode configuration
504 Gateway Timeout	Proxy's request to the backend exceeded the configured timeout	Backend hung, slow database query, <code>proxy_read_timeout</code> set too low for legitimate slow operations
403 Forbidden	Server understood the request but refuses to authorize it	File permission issue, <code>Require / allow / deny</code> misconfiguration, missing index with directory listing disabled
404 Not Found	Requested resource does not exist at the resolved path	Incorrect <code>root / DocumentRoot</code> , misconfigured <code>try_files /rewrite</code> rules, genuinely missing file

A 502 specifically should immediately direct attention to the backend application's own status and logs (`systemctl status myapp.service` , `journalctl -u myapp.service`), rather than the proxy configuration the proxy is functioning correctly by reporting that it *could not reach* the backend; the actual fault lies there.

Diagnosing TLS/Certificate Problems

bash

```
openssl s_client -connect example.com:443 -servername example.com
echo | openssl s_client -connect example.com:443 2>/dev/null | openssl x509 -noout -issuer -subject -dates
sudo certbot certificates
```

`sudo certbot certificates` lists all certificates Certbot manages, along with their expiration dates a quick way to confirm a specific certificate's validity and identify any that may have failed to renew.

Load and Resource-Related Diagnosis

When a web service is running and reachable but performing poorly, correlate with system resource state (developed fully in Chapter 10):

bash

top

free -h

sudo ss -tan state established | wc -l # current established connection count

A high established-connection count alongside high memory or CPU consumption often points toward either genuinely increased legitimate load (a scaling conversation) or a problematic pattern such as connections not being closed properly by the application—a distinction worth investigating carefully before assuming more server capacity is the answer, echoing the performance-engineering discipline developed fully in Chapter 10.

With web and application services deployed, secured, and diagnosable, the book turns next to the data layer beneath them: storage architecture and database administration.

Chapter 6 : Storage, Databases, and Server Data Management

6.1 Server Storage Architecture and Planning

Chapter 1 introduced storage planning as part of initial installation. This chapter develops that foundation into full operational competence: managing disks and filesystems on a running system, deploying and administering database services, monitoring capacity over time, and building a coherent backup strategy – the data-protection discipline that underlies nearly every other chapter's assumption that recovery is possible.

Revisiting the Storage Hierarchy

A running Debian server's storage stack typically has several layers, each worth understanding distinctly when diagnosing or planning:

- **Physical or virtual block devices** – actual disks, or virtual disks presented by a hypervisor/cloud provider.
- **Partitions** – divisions of a block device (`/dev/sda1` , for instance).
- **LVM physical volumes, volume groups, and logical volumes** – an optional but commonly used abstraction layer atop partitions, providing flexibility.
- **Filesystems** – the structure (ext4, XFS, etc.) formatted onto a partition or logical volume.
- **Mount points** – where a filesystem is attached into the overall directory hierarchy.

```
bash
lsblk -f
```

The `-f` flag additionally shows filesystem type and UUID for each device – useful both for understanding the current layout and for cross-referencing against `/etc/fstab` , described next.

Understanding /etc/fstab

/etc/fstab defines which filesystems are mounted automatically at boot, and how:

bash

cat /etc/fstab

# <file system>	<mount point>	<type>	<options>	<dump>	<pass>
UUID=1234-5678-90ab-cdef	/	ext4	errors=remount-ro	0	1
UUID=abcd-ef01-2345-6789	/var/log	ext4	defaults	0	2
UUID=fedc-ba98-7654-3210	none	swap	sw	0	0

Referencing filesystems by UUID (rather than a device path like /dev/sda1) is strongly preferred, since device naming can shift between boots (particularly with multiple disks or after hardware changes), while a filesystem's UUID remains stable. Find a filesystem's UUID with:

bash

sudo blkid

Warning: An incorrect /etc/fstab entry can prevent a system from booting normally (dropping instead into an emergency recovery shell). Always test new mount configurations with mount -a before rebooting, and keep a backup of a known-working /etc/fstab :

bash

sudo cp /etc/fstab /etc/fstab.bak

sudo nano /etc/fstab

sudo mount -a # applies fstab entries without rebooting; reports errors immediately

6.2 Managing Disks, Filesystems, LVM, and Mounts

Adding and Partitioning a New Disk

When a server needs additional storage a new virtual disk attached to a VM, or an additional physical drive the process begins with identifying the new device:

bash

lsblk

sudo fdisk -l

Newly attached, unpartitioned disks appear in `lsblk` without partition children. Partition the new disk using `parted` (preferred over the older `fdisk` for GPT-based disks, which are standard on modern systems):

```
bash
sudo parted /dev/sdb --script mklabel gpt
sudo parted /dev/sdb --script mkpart primary 0% 100%
```

Warning: These commands modify partition tables and can destroy existing data if run against the wrong device. Always confirm the target device with `lsblk / fdisk -l` immediately before running any partitioning command, and never run these commands against a disk currently in use without explicit, deliberate intent.

Creating a Filesystem

```
bash
sudo mkfs.ext4 /dev/sdb1
```

Extending Storage with LVM

Where the initial installation used LVM (as recommended in Chapter 1), extending storage is significantly more flexible than working with raw partitions directly. To add a new physical disk to an existing volume group and extend a logical volume:

```
bash
sudo pvcreate /dev/sdb1
sudo vgextend vg_debian-server /dev/sdb1
sudo lvextend -L +20G /dev/vg_debian-server/lv_data
```

After extending the logical volume, the filesystem itself must also be grown to use the newly available space — extending the logical volume alone does not automatically resize the filesystem sitting on it:

```
bash
sudo resize2fs /dev/vg_debian-server/lv_data      # for ext4
# or, for XFS:
sudo xfs_growfs /data
```

Verify the result:

```
bash
```

```
df -h /data
sudo lvs
```

This LVM-based growth path – add physical storage, extend the volume group, extend the logical volume, grow the filesystem – is precisely the operational payoff of the LVM-based partitioning scheme recommended during installation in Chapter 1: it can typically be performed on a live, mounted filesystem with no downtime, which is essentially never true of resizing raw partitions.

Mounting and Unmounting

```
bash
sudo mkdir -p /data
sudo mount /dev/vg_debian-server/lv_data /data
sudo umount /data
```

Add a corresponding `/etc/fstab` entry (using the volume's UUID, obtained via `blkid`, or its LVM path) to ensure the mount persists across reboots, then verify with `mount -a` as shown earlier.

Monitoring Filesystem and Inode Usage

```
bash
df -h      # space usage by mounted filesystem
df -i      # inode usage – a distinct, sometimes overlooked resource
du -sh /var/log/* # space consumed by subdirectories, sorted for investigation
```

Inode exhaustion is a genuinely common and easily overlooked failure mode: a filesystem can report ample free space (`df -h`) while simultaneously being unable to create new files because it has run out of inodes (`df -i`), typically caused by an application creating an extremely large number of very small files. Checking both is a habit worth building specifically because the space-only check (`df -h`) can be misleadingly reassuring.

6.3 Deploying and Managing Database Services

Choosing Between PostgreSQL and MariaDB/MySQL

Both are fully capable, mature relational database systems well supported on Debian. PostgreSQL is generally favored for advanced SQL feature support, strict standards compliance, and complex data modeling; MariaDB

(Debian's default MySQL-compatible package) is favored for broad compatibility with existing MySQL-oriented application ecosystems and generally simpler initial administration for straightforward workloads. This section covers PostgreSQL in primary depth, with MariaDB's equivalent operations noted alongside.

Installing PostgreSQL

```
bash
sudo apt update
sudo apt install postgresql postgresql-contrib
sudo systemctl enable --now postgresql
sudo systemctl status postgresql
```

PostgreSQL creates a system user named `postgres`, which owns the database cluster and is used for initial administrative access:

```
bash
sudo -u postgres psql -c "SELECT version();"
```

Installing MariaDB

```
bash
sudo apt install mariadb-server
sudo systemctl enable --now mariadb
sudo mysql_secure_installation
```

`mysql_secure_installation` walks through an interactive hardening sequence setting a root password, removing anonymous users, disabling remote root login, and removing the default test database and should always be run immediately after installation.

Creating Databases and Users (PostgreSQL)

```
bash
sudo -u postgres psql
sql
CREATE DATABASE myapp_production;
CREATE USER myapp WITH PASSWORD 'strong-generated-password';
GRANT ALL PRIVILEGES ON DATABASE myapp_production TO myapp;
\q
```

Following least-privilege principles established throughout the book, prefer granting only the specific privileges an application actually needs rather

than defaulting to broad grants, particularly for applications that should not, for example, be able to drop tables:

```
sql
```

```
GRANT CONNECT ON DATABASE myapp_production TO myapp;  
GRANT USAGE ON SCHEMA public TO myapp;  
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO  
myapp;
```

Creating Databases and Users (MariaDB)

```
bash
```

```
sudo mysql
```

```
sql
```

```
CREATE DATABASE myapp_production;  
CREATE USER 'myapp'@'localhost' IDENTIFIED BY 'strong-generated-password';  
GRANT SELECT, INSERT, UPDATE, DELETE ON myapp_production.* TO 'myapp'@'localhost';  
FLUSH PRIVILEGES;
```

Note 'myapp'@'localhost' MariaDB's user identity includes the host from which connections are permitted, another built-in least-privilege mechanism worth using deliberately (restricting to `localhost` unless remote database access is genuinely required).

Network Access Considerations

By default, both PostgreSQL and MariaDB bind only to `localhost`, consistent with the least-exposure principle established in Chapter 3. Confirm this:

```
bash
```

```
sudo ss -tulnp | grep -E '5432|3306'
```

If remote database access is genuinely required (an application server on a separate host, for instance), this should be a deliberate, narrowly-scoped decision bind to a specific internal interface rather than all interfaces, and restrict access at the firewall layer (Chapter 7) to only the specific hosts that need it:

For PostgreSQL, edit `postgresql.conf` and `pg_hba.conf`:

```
bash
```

```
sudo nano /etc/postgresql/*/main/postgresql.conf  
listen_addresses = '10.0.0.5'
```

bash

```
sudo nano /etc/postgresql/*/main/pg_hba.conf
# TYPE DATABASE USER ADDRESS METHOD
host myapp_production myapp 10.0.0.10/32 scram-sha-256
```

This `pg_hba.conf` entry permits the `myapp` user to connect to `myapp_production` only from the specific address `10.0.0.10`, using SCRAM-SHA-256 password authentication a considerably more restrictive and auditable posture than a blanket allowance.

bash

```
sudo systemctl restart postgresql
```

Basic Performance and Configuration Tuning

Both database systems ship with conservative default resource settings suitable for small instances. For production workloads, review and tune at minimum:

PostgreSQL(`postgresql.conf`):

```
shared_buffers = 256MB      # typically ~25% of available RAM on a dedicated DB server
work_mem = 16MB
maintenance_work_mem = 128MB
max_connections = 100
```

MariaDB(`/etc/mysql/mariadb.conf.d/50-server.cnf`):

```
innodb_buffer_pool_size = 1G # typically 60-70% of available RAM on a dedicated DB server
max_connections = 150
```

Tuning these values meaningfully requires understanding actual workload characteristics, and should be approached incrementally with measurement (following the performance-engineering discipline developed in Chapter 10) rather than applied speculatively but the values above represent reasonable, deliberate starting points for a dedicated database server, well beyond the very conservative out-of-box defaults intended to run safely even on minimal hardware.

6.4 Storage Monitoring, Capacity Planning, and Maintenance

Ongoing Disk Usage Monitoring

bash

`df -h`

`du -sh /var/lib/postgresql/*/main/` # database storage consumption specifically

`ncdu /var` # interactive disk usage explorer; requires: `sudo apt install ncdu`

`ncdu` is a particularly valuable tool for investigating disk usage interactively it presents a navigable, sorted breakdown of directory sizes, making it far faster than repeated manual `du` invocations to identify what is actually consuming space during a capacity investigation.

Establishing Capacity Thresholds and Trends

Rather than reacting only when a filesystem is nearly full, establish a habit (formalized with actual alerting in Chapter 10) of reviewing usage trends over time is `/var/log` growing at a rate consistent with normal operation, or has something changed? Is database storage growing in line with expected business/application growth, or unexpectedly faster? A single point-in-time `df -h` reading tells you current state; comparing readings over days or weeks reveals trend, which is what actually drives capacity planning decisions (when to provision additional storage, when to investigate an anomaly).

Routine Storage Maintenance Tasks

bash

`sudo apt autoremove --purge` # remove unused package dependencies

`sudo journalctl --vacuum-time=30d` # enforce journal retention (also configurable persistently, per Chapter 2)

`sudo find /tmp -type f -atime +7 -delete` # clear stale temporary files older than 7 days

Warning: The `find ... -delete` pattern is destructive and should be tested first with `-print` in place of `-delete` to confirm exactly which files would be affected before committing to deletion:

bash

`sudo find /tmp -type f -atime +7 -print`

Database-Specific Maintenance

PostgreSQL requires periodic `VACUUM` operations to reclaim space from deleted/updated rows and maintain query planner statistics largely

automated via `autovacuum` (enabled by default), but worth understanding and monitoring:

sql

```
SELECT schemaname, relname, last_autovacuum, last_autoanalyze  
FROM pg_stat_user_tables  
ORDER BY last_autovacuum ASC NULLS FIRST;
```

This query surfaces tables that may not be receiving adequate `autovacuum` attention (relevant on very large or very high-churn tables), a useful periodic health check for database administrators managing PostgreSQL at any meaningful scale.

MariaDB/InnoDB handles most routine space reclamation automatically as well, though periodic `OPTIMIZE TABLE` can be useful for tables with substantial deletion/update churn:

sql

```
OPTIMIZE TABLE myapp_production.large_churny_table;
```

6.5 Backup Strategies for Server Data

The Central Principle: An Untested Backup Is Not a Verified Backup

As stated in this book's specification and worth restating directly: **a backup that has never been tested cannot automatically be assumed to be recoverable.** A backup process that runs successfully, producing files of a plausible size, provides false confidence if the actual restoration procedure has never been exercised. This section's guidance is built around that principle throughout.

Categories of Backup

Full backups capture the entire dataset at a point in time – simplest to restore from, but most expensive in storage and time to create.

Incremental backups capture only data changed since the last backup (full or incremental) – efficient to create, but restoration requires the full backup plus the entire chain of subsequent incrementals, increasing restoration complexity and time.

Configuration backups capturing `/etc/`, service configuration, systemd unit files, and similar are frequently underweighted relative to data backups, but are equally essential: recovering a database with no record of the application configuration, firewall rules, or service definitions that made use of it leaves a server only partially recovered.

File-Level Backups with rsync

For straightforward file and configuration backup to a remote location:

bash

```
rsync -avz --delete /etc/ backup-host:/backups/debian-server/etc/  
rsync -avz --delete /opt/myapp/ backup-host:/backups/debian-server/myapp/
```

`--delete` ensures the backup destination mirrors the source exactly, removing files at the destination that no longer exist at the source appropriate for a straightforward mirror-style backup, but understand its implication: if a file is accidentally deleted at the source and the backup then runs, it will also disappear from this destination. Where retaining deleted files' history matters, use a backup tool or approach that preserves multiple versions (see below), not a pure mirror.

Database Backups

PostgreSQL logical backup via `pg_dump` (per-database) or `pg_dumpall` (cluster-wide, including roles):

bash

```
sudo -u postgres pg_dump myapp_production | gzip > /backups/myapp_production_$(date  
+%Y%m%d).sql.gz  
sudo -u postgres pg_dumpall --globals-only > /backups/globals_$(date +%Y%m%d).sql
```

MariaDB logical backup via `mysqldump` :

bash

```
mysqldump -u root -p myapp_production | gzip > /backups/myapp_production_$(date  
+%Y%m%d).sql.gz
```

For larger databases where logical dump/restore time becomes impractical, physical backup tools (`pg_basebackup` for PostgreSQL, or `mariabackup` / `xtrabackup` for MariaDB/MySQL) provide faster, binary-level backup and restoration worth adopting once dataset size makes logical

backups' restoration time operationally unacceptable, a threshold that varies by environment but is worth watching for deliberately rather than discovering during an actual outage.

Automating Backups

A basic backup script tying together file and database backups, intended to be run via a systemd timer or cron (Chapter 9 covers scheduling in full depth):

```
bash
sudo nano /opt/scripts/backup.sh
bash
#!/bin/bash
set -euo pipefail

BACKUP_DIR="/backups/$(date +%Y%m%d)"
REMOTE_DEST="backup-host:/backups/debian-server/"
LOG_FILE="/var/log/backup.log"

echo "[$(date)] Starting backup" >> "$LOG_FILE"

mkdir -p "$BACKUP_DIR"

# Database backup
sudo -u postgres pg_dump myapp_production | gzip > "$BACKUP_DIR/myapp_production.sql.gz"

# Configuration backup
tar -czf "$BACKUP_DIR/etc-backup.tar.gz" /etc

# Application data backup
tar -czf "$BACKUP_DIR/myapp-data.tar.gz" /opt/myapp/data

# Sync to remote backup host
rsync -avz "$BACKUP_DIR" "$REMOTE_DEST"

# Retain local backups for 7 days only
find /backups -maxdepth 1 -mtime +7 -type d -exec rm -rf {} \;

echo "[$(date)] Backup completed successfully" >> "$LOG_FILE"
```

`set -euo pipefail` at the top is an important scripting discipline (developed further in Chapter 9): `-e` exits immediately on any command failure, `-u` treats unset variables as errors, and `-o pipefail` ensures a failure anywhere in a pipeline is correctly detected, rather than only the final command's exit status. Together, this prevents a backup script from silently continuing (and reporting apparent success) after an earlier step has actually failed.

Off-Site and 3-2-1 Backup Principles

A widely used, sound baseline is the **3-2-1 rule**: maintain at least **3** copies of data, on at least **2** different types of storage media, with at least **1** copy stored off-site (physically or geographically separate from the primary server). For a typical small-to-medium Debian deployment, this might translate concretely to: the live data itself (copy 1), a local backup on separate storage or a separate disk (copy 2), and a remote/off-site backup a separate server, a cloud object storage bucket, or a different data center/region entirely (copy 3, satisfying the off-site requirement). The specific tooling matters less than genuinely satisfying this structural principle a backup stored only on the same physical server it protects provides no protection at all against the single most catastrophic failure mode: total loss of that server.

Testing Restoration

Periodically and this is the step most commonly skipped actually perform a restoration, ideally into an isolated test environment, to confirm the backup is genuinely usable:

```
bash
```

```
# Restore a PostgreSQL dump into a freshly created test database
```

```
sudo -u postgres createdb myapp_restore_test
```

```
gunzip -c /backups/20260920/myapp_production.sql.gz | sudo -u postgres psql myapp_restore_test
```

```
bash
```

```
# Verify the restored data looks correct
```

```
sudo -u postgres psql myapp_restore_test -c "SELECT count(*) FROM users;"
```

Scheduling a recurring, calendared restoration test monthly or quarterly, depending on the data's criticality converts backup verification from a hoped-for good practice into an actual, enforced operational habit, and is one of the highest-value disciplines a server administrator can build into ongoing maintenance. This theme is picked up again, with fuller disaster-recovery planning context, in Chapter 11.

Chapter 7 : Server Security and Hardening

7.1 Building a Debian Server Security Baseline

Security on a production server is not a single configuration step completed once and then forgotten it is a layered, ongoing discipline built from many individually modest measures that, together, substantially reduce both the likelihood and the impact of a compromise. Previous chapters have already introduced several pieces of this discipline: least-privilege user administration (Chapter 2), loopback-only binding for internal services (Chapter 3), and SSH key-based authentication with password login disabled (Chapter 4). This chapter formalizes security as its own coherent practice, brings together a complete baseline, and extends it into firewalling, deeper account hardening, patch and vulnerability management, and intrusion detection.

The Layered Security Model

No single control is sufficient on its own. A firewall does not protect against a vulnerable application exposed through an allowed port. Strong SSH hardening does not protect against a weak database password reachable from an internal network. The practical implication is that security work proceeds across several independent layers simultaneously, each catching different categories of failure:

- **Network layer** firewalling, minimizing exposed services, network segmentation
- **Authentication layer** SSH key enforcement, account lockout policy, credential hygiene
- **Authorization layer** least-privilege sudo, file permissions, service account scoping
- **Patch layer** keeping the OS and installed software current against known vulnerabilities
- **Detection layer** logging, monitoring, and intrusion detection to notice what prevention missed
- **Recovery layer** backups and documented incident response, so a successful attack is a setback, not an existential event

This chapter is organized around these layers, building on the network and account foundations already established, and closing the loop with detection and monitoring because prevention, however careful, is never complete, and an administrator who cannot detect a compromise cannot respond to one.

Establishing a Baseline Inventory

Before hardening a system further, establish exactly what is currently running and reachable a security baseline against which future changes and any suspicious deviation can be measured:

bash

```
sudo ss -tulnp # all listening services
systemctl list-units --type=service --state=running
dpkg -l > /root/baseline-packages-$(date +%Y%m%d).txt
sudo ss -tulnp > /root/baseline-listening-$(date +%Y%m%d).txt
```

Store these baseline files somewhere durable (ideally as part of the configuration backups discussed in Chapter 6), and revisit them

periodically. A newly listening port or an unexpected running service, discovered by comparing current state against this baseline, is often the first observable sign of either a misconfiguration or a genuine compromise.

Minimizing Attack Surface

Every running service, every installed package, and every open port represents potential attack surface. A core security principle not unique to Debian, but consistently under-applied is to run only what is genuinely needed:

bash

```
# Identify packages that provide network-facing services but are not required
dpkg -l | grep -i -E "telnet|rsh|ftp|tftp"
```

Historic, insecure network services (Telnet, rsh, unencrypted FTP, TFTP) have no legitimate place on a modern production server and should be removed entirely if present:

bash

```
sudo apt purge telnetd rsh-server tftpd-hpa -y 2>/dev/null
```

More broadly, periodically audit installed packages for anything unused:

bash

```
dpkg -l | grep ^ii | awk '{print $2}' | sort > /tmp/installed-packages.txt
```

Reviewing this list against actual server purpose and removing packages that serve no function reduces both the patching burden (fewer packages means fewer potential vulnerabilities to track) and the practical attack surface available to an intruder who gains initial access.

File Permission and Ownership Baseline

Excessively permissive file permissions are a common, quiet source of risk. Review sensitive locations explicitly:

bash

```
find / -xdev -perm -4000 -type f 2>/dev/null # SUID binaries
find / -xdev -perm -2000 -type f 2>/dev/null # SGID binaries
find /etc -type f -perm -o+w 2>/dev/null # world-writable files under /etc
find / -xdev -nouser -o -nogroup 2>/dev/null # files with no valid owner/group
```

SUID/SGID binaries run with the privileges of the file's owner/group regardless of who executes them, and represent a well-understood privilege-escalation risk if unexpected ones are present (a legitimately unusual SUID binary appearing outside a normal package-managed set is a meaningful red flag). World-writable files under `/etc` are almost never appropriate and should be corrected immediately:

```
bash
```

```
sudo chmod o-w /etc/suspicious-file
```

A Consolidated Baseline Checklist

Drawing together this section with the practices already established in earlier chapters, a Debian 13.7 server's security baseline should satisfy:

- No direct root SSH login (Chapter 4)
- SSH password authentication disabled, key-based auth verified (Chapter 4)
- A dedicated, least-privilege administrative account with scoped `sudo` (Chapter 2)
- Service accounts with no interactive shell (Chapter 2, Chapter 4)
- Sensitive services (databases, internal APIs) bound to loopback or internal interfaces only (Chapters 3 and 6)
- No legacy insecure network services installed
- Package updates applied currently, with a defined update cadence (this chapter, section 7.4)
- A firewall with an explicit, minimal allow-list (this chapter, section 7.2)
- Centralized, persistent logging in place (Chapter 2, extended in this chapter)

The remainder of this chapter builds out the pieces of this checklist not yet covered in full depth: firewalling, deeper account and privilege hardening, systematic patch and vulnerability management, and intrusion detection.

7.2 Firewall Configuration and Network Access Control

A firewall enforces, at the network layer, the principle already applied at the service-binding layer in earlier chapters: only expose what genuinely needs to be reachable, from only where it genuinely needs to be reachable from. Debian 13.7 uses **nftables** as its modern, native packet-filtering framework, superseding the older `iptables` (though `iptables` remains available, largely as a compatibility layer translating to `nftables` underneath). This section covers `nftables` directly, with UFW (Uncomplicated Firewall) presented as a simpler, higher-level alternative appropriate for less complex rule sets.

Understanding nftables Architecture

`nftables` organizes rules into **tables**, containing **chains**, containing individual **rules**. Tables are associated with an address family (`inet` for combined IPv4/IPv6, `ip` for IPv4-only, `ip6` for IPv6-only); chains attach to specific network processing points (`input` , `output` , `forward`) and carry a default policy (`accept` or `drop`).

Verify `nftables` is installed and check current state:

```
bash
sudo apt install nftables
sudo systemctl enable --now nftables
sudo nft list ruleset
```

An empty ruleset (as noted in Chapter 1's baseline check) means no filtering is currently occurring the starting point before this section's configuration.

Designing a Baseline nftables Ruleset

The following ruleset establishes a deliberately restrictive baseline: default-drop on inbound traffic, explicit allowances for SSH, HTTP/HTTPS, and established/related connections, with loopback and ICMP handled appropriately.

```
bash
sudo nano /etc/nftables.conf
#!/usr/sbin/nft -f

flush ruleset

table inet filter {
    chain input {
        type filter hook input priority 0; policy drop;
```

```

# Allow loopback traffic
iif "lo" accept

# Allow established and related connections
ct state established,related accept

# Drop invalid packets outright
ct state invalid drop

# Allow ICMP (ping) and essential ICMPv6 for IPv6 functionality
ip protocol icmp icmp type { echo-request, echo-reply, destination-unreachable, time-
exceeded } accept
ip6 nexthdr icmpv6 icmpv6 type { echo-request, echo-reply, nd-neighbor-solicit, nd-neighbor-
advert, nd-router-solicit, nd-router-advert } accept

# Allow SSH (adjust port if using a non-default port per Chapter 4)
tcp dport 22 ct state new accept

# Allow HTTP and HTTPS
tcp dport { 80, 443 } ct state new accept

# Log and drop everything else reaching this point
log prefix "nft-input-drop: " limit rate 5/minute
counter drop
}

chain forward {
    type filter hook forward priority 0; policy drop;
}

chain output {
    type filter hook output priority 0; policy accept;
}
}

```

Explanation of key elements:

- **policy drop on the input chain** implements a default-deny posture: any traffic not explicitly matched by an `accept` rule is dropped. This is the single most important architectural decision in the ruleset – a default-accept firewall with scattered `drop` rules is fragile and easy to misconfigure into uselessness, while default-drop with explicit `accept` rules is auditable at a glance.
- **ct state established,related accept** allows return traffic for connections the server itself initiated, or legitimately related to an already-permitted connection, without which the server would be unable to complete even its own outbound requests (DNS lookups, package updates) since return traffic would be blocked.
- **ct state invalid drop** discards packets that don't correspond to any recognized connection state, a common characteristic of certain scanning and spoofing techniques.
- **The final log and counter drop** rule ensures unmatched traffic is both dropped and recorded (rate-limited to 5 log entries per minute to avoid log flooding during a scan or attack), providing visibility into what is being blocked – valuable both for security monitoring (section 7.5) and for diagnosing an overly restrictive rule during testing.
- **chain forward with policy drop** ensures this server does not act as an unintended router/relay between networks, appropriate for a standard single-purpose server; this would be adjusted for a server deliberately configured to forward traffic (a VPN gateway, for instance).
- **chain output with policy accept** takes a permissive stance on outbound traffic, appropriate for most general-purpose servers; more security-sensitive deployments sometimes restrict outbound traffic as well, a pattern briefly discussed later in this section.

Applying and Testing the Configuration

Warning: Applying a restrictive firewall configuration carries genuine risk of losing remote SSH access if a rule is wrong. Follow the safety discipline established in Chapter 3's networking guidance: confirm out-of-

band/console access before applying, and consider a scheduled safety-net reboot for the first application of a new ruleset.

bash

```
sudo nft -c -f /etc/nftables.conf # syntax check only, does not apply
```

If the syntax check passes, apply carefully, using the dead-man's-switch pattern introduced in Chapter 3:

bash

```
sudo shutdown -r +5 "Safety reboot after firewall change"
sudo nft -f /etc/nftables.conf
sudo nft list ruleset
```

From a **separate** terminal, immediately verify SSH access still works:

bash

```
ssh adminuser@debian-server "echo firewall test successful"
```

If access is confirmed working, cancel the scheduled safety reboot:

bash

```
sudo shutdown -c
```

If access fails, the scheduled reboot will restore the system to its prior (working) state automatically this is precisely the safety net's purpose, and is far preferable to being locked out of a remote system with no console access readily available.

Ensure the ruleset persists across reboots by confirming `nftables.service` loads `/etc/nftables.conf` at boot (the default Debian package configuration already does this via `/etc/nftables.conf` being sourced from the systemd unit) and is enabled:

bash

```
sudo systemctl enable nftables
sudo systemctl status nftables
```

Restricting SSH by Source Address

Where administrative access originates from known, stable networks, restrict SSH further at the firewall layer one of the highest-value hardening measures available, as noted in Chapter 4:

```
# Replace the general SSH rule with a source-restricted one
ip saddr 203.0.113.0/24 tcp dport 22 ct state new accept
```

This permits SSH connection attempts only from the 203.0.113.0/24 network, silently dropping (per the default policy) any SSH attempt from elsewhere—dramatically reducing the population of hosts that can even attempt authentication, regardless of how strong that authentication is.

Using UFW as a Simpler Alternative

For administrators managing a small number of servers with straightforward rule requirements, UFW provides a considerably simpler interface atop the same underlying netfilter/nftables infrastructure:

```
bash
sudo apt install ufw
sudo ufw default deny incoming
sudo ufw default allow outgoing
sudo ufw allow 22/tcp
sudo ufw allow 80/tcp
sudo ufw allow 443/tcp
sudo ufw enable
```

Warning: As with raw nftables, enabling UFW's default-deny incoming policy before confirming an SSH allow rule is in place risks immediate lockout. The sequence above deliberately adds the SSH rule *before* running `ufw enable`—never reverse this order.

```
bash
sudo ufw status verbose
```

UFW is well suited to simpler deployments; nftables directly offers considerably more expressive power (rate limiting, connection tracking nuance, complex matching) appropriate for more sophisticated network security requirements. This book's later chapters (8 and 12) use direct nftables syntax where container networking and multi-service projects require finer control.

Rate Limiting Against Brute-Force and Scanning

nftables can rate-limit connection attempts directly at the firewall layer, complementing `fail2ban` (Chapter 4) with a network-level control that applies before any authentication attempt is even processed:

```
tcp dport 22 ct state new limit rate 10/minute accept
tcp dport 22 ct state new drop
```

This permits at most 10 new SSH connection attempts per minute from the ruleset's perspective, dropping the rest a coarse but effective mitigation against high-volume automated scanning, deployed independently of and in addition to `fail2ban`'s more granular, per-source-IP banning behavior.

Verifying Firewall Behavior from Outside

Testing firewall rules from the server itself is insufficient, since loopback traffic bypasses the `input` chain's external-facing rules entirely. Test from a genuinely external vantage point:

```
bash
```

```
# From a separate machine
```

```
nc -zv debian-server.example.com 22
```

```
nc -zv debian-server.example.com 3306 # should fail/timeout if MySQL is correctly firewalled
```

```
nmap -Pn debian-server.example.com # requires: sudo apt install nmap, run from a separate host
```

`nmap` run against your own server from an external vantage point provides a comprehensive, independent view of exactly which ports are reachable a valuable verification step after any significant firewall change, and one that should be repeated periodically as part of routine security auditing (section 7.4).

7.3 SSH, Account, and Privilege Hardening

Chapter 4 established SSH hardening fundamentals key-based authentication, disabled root login, `fail2ban`. This section extends account and privilege hardening more broadly across the system.

Enforcing Password Policy for Any Remaining Password-Based Accounts

Even with SSH password authentication disabled, local account passwords remain relevant for console access and `sudo` authentication. Enforce a reasonable password aging and complexity policy via `/etc/login.defs` and PAM:

```
bash
```

```
sudo nano /etc/login.defs
```

```
PASS_MAX_DAYS 90
PASS_MIN_DAYS 1
PASS_WARN_AGE 14
```

These settings apply to accounts created after the change; apply to an existing account explicitly with `chage` :

```
bash
sudo chage -M 90 -m 1 -W 14 adminuser
sudo chage -l adminuser
```

For password complexity enforcement, install and configure `libpam-pwquality` :

```
bash
sudo apt install libpam-pwquality
sudo nano /etc/pam.d/common-password
password requisite pam_pwquality.so retry=3 minlen=14 difok=3 ucredit=-1 lcredit=-1 dcredit=-1
ocredit=-1
```

This enforces a minimum length of 14 characters, requiring at least one uppercase letter, one lowercase letter, one digit, and one special character, with three attempts permitted before rejection.

Locking Down su

Restrict use of `su` (switching to another user, particularly root) to members of a specific group, adding an additional control beyond `sudo` alone:

```
bash
sudo groupadd wheel-su 2>/dev/null
sudo usermod -aG wheel-su adminuser
bash
sudo nano /etc/pam.d/su
```

Uncomment or add:

```
auth required pam_wheel.so group=wheel-su
```

With this in place, only members of `wheel-su` can use `su` at all, regardless of whether they know the target account's password a meaningful additional barrier against lateral privilege escalation if a lower-privileged account is compromised.

Auditing sudo Usage Continuously

Building on Chapter 2's introduction of `sudoers.d` delegation, establish routine review of actual `sudo` usage:

bash

```
sudo journalctl _COMM=sudo --since "7 days ago" | grep COMMAND
sudo grep sudo /var/log/auth.log | tail -50
```

For environments requiring more rigorous audit trails, `sudo`'s own logging can be directed to a dedicated file via `Defaults logfile` in `sudoers`:

bash

```
sudo visudo
Defaults logfile="/var/log/sudo.log"
Defaults log_input, log_output
```

`log_input` / `log_output` additionally capture the actual terminal session content of privileged commands – a significant audit capability, though one that increases log volume and should be applied deliberately, particularly on systems handling especially sensitive operations.

Restricting Cron and At for Non-Administrative Users

By default, any user may schedule jobs via `cron` or `at`. Restrict this explicitly to intended accounts:

bash

```
echo "adminuser" | sudo tee /etc/cron.allow
echo "myappuser" | sudo tee -a /etc/cron.allow
sudo rm -f /etc/cron.deny

echo "adminuser" | sudo tee /etc/at.allow
sudo rm -f /etc/at.deny
```

With `/etc/cron.allow` present, only listed users may create cron jobs; all others are silently denied, preventing an unexpected account (including a compromised low-privilege service account) from establishing scheduled persistence.

Kernel-Level Hardening via sysctl

Several kernel network and security parameters, adjustable via `sysctl`, provide additional defense-in-depth:

bash

```
sudo nano /etc/sysctl.d/99-security-hardening.conf
# Disable IP forwarding unless this server is deliberately a router
net.ipv4.ip_forward = 0
net.ipv6.conf.all.forwarding = 0

# Ignore ICMP redirects (mitigates certain MITM techniques)
net.ipv4.conf.all.accept_redirects = 0
net.ipv4.conf.all.send_redirects = 0
net.ipv6.conf.all.accept_redirects = 0

# Enable SYN flood protection
net.ipv4.tcp_syncookies = 1

# Ignore broadcast pings (mitigates smurf-style amplification)
net.ipv4.icmp_echo_ignore_broadcasts = 1

# Log packets with impossible/spoofed source addresses
net.ipv4.conf.all.log_martians = 1

# Enable reverse path filtering (anti IP-spoofing)
net.ipv4.conf.all.rp_filter = 1

# Restrict kernel pointer exposure in /proc
kernel.kptr_restrict = 2

# Restrict dmesg access to privileged users
kernel.dmesg_restrict = 1
```

Apply and verify:

bash

```
sudo sysctl -p /etc/sysctl.d/99-security-hardening.conf
sysctl net.ipv4.tcp_syncookies
```

Each of these settings is a narrowly scoped mitigation against a specific, well-understood class of network-layer attack technique; none is individually dramatic, but together they meaningfully raise the difficulty of several common attack patterns, consistent with this chapter's defense-in-depth framing.

Restricting Physical/Console Recovery Access

Debian's default GRUB configuration typically allows unauthenticated access to a root shell via boot parameter editing at the console. For servers where physical or console security cannot be otherwise assured (a concern more relevant to colocated or shared physical infrastructure than most cloud/VPS deployments, but worth understanding), GRUB can be configured to require a password for boot parameter editing:

```
bash
sudo grub-mkpasswd-pbkdf2
```

This generates a PBKDF2 password hash; add it to a custom GRUB configuration snippet:

```
bash
sudo nano /etc/grub.d/40_custom
set superusers="admin"
password_pbkdf2 admin grub.pbkdf2.sha512.10000.<generated-hash>
bash
sudo update-grub
```

Warning: This restricts the ability to edit boot parameters and access single-user/rescue mode without the configured password—a genuine security improvement against physical-access attacks, but also a genuine risk of self-lockout if the password is lost, since it may then be difficult to recover the system through normal boot-parameter-based rescue techniques. Only apply this where physical/console access security is a real, identified concern for your deployment, and record the password with the same care given to any other administrative credential.

7.4 Security Updates, Auditing, and Vulnerability Management

The Security Update Cadence

Chapter 2 introduced `unattended-upgrades` for automating security patch application. This section develops the broader discipline of vulnerability management around that automation.

Debian's security team maintains the `trixie-security` archive (as established in Chapter 2) specifically for prompt, targeted security patches, decoupled

from the broader point-release update cadence. Understanding this distinction matters: a server correctly consuming `trixie-security` updates receives critical patches quickly, even between full point releases.

Confirm the security archive is correctly configured and actively contributing updates:

```
bash
apt-cache policy | grep security
sudo apt update
apt list --upgradable 2>/dev/null | grep -i security
```

Tracking Debian Security Advisories

For environments requiring more active vulnerability awareness than automated patching alone provides, Debian publishes formal Security Advisories (DSAs) for significant vulnerabilities. Administrators managing security-sensitive infrastructure should establish a routine of reviewing these either via the Debian Security mailing list, the security tracker, or periodic manual review particularly for any package running a network-exposed service on the system.

Vulnerability Scanning with debsecan

`debsecan` cross-references installed packages against known vulnerability data, providing a more targeted view than generic `apt list --upgradable` :

```
bash
sudo apt install debsecan
sudo debsecan --suite trixie
```

Output lists known vulnerabilities (by CVE identifier) affecting currently installed package versions, along with a severity/status indicator. This provides a useful periodic audit distinct from routine patching particularly valuable for confirming that a specific, publicly disclosed vulnerability has genuinely been addressed on this system, rather than simply trusting that "updates are being applied" translates to "known vulnerabilities are closed."

Auditing System Configuration with Lynis

Lynis is a widely used, comprehensive security auditing tool that examines system configuration broadly far beyond just package vulnerabilities covering authentication configuration, file permissions, firewall state, logging configuration, and dozens of other areas, producing a scored report with specific, actionable recommendations.

bash

```
sudo apt install lynis  
sudo lynis audit system
```

Lynis's output includes a hardening index score and a categorized list of warnings and suggestions. Review these systematically not every suggestion will be relevant to every deployment (Lynis is intentionally broad and somewhat generic), but the tool is valuable specifically for surfacing configuration gaps an administrator might not think to check manually, complementing the deliberate, targeted hardening steps covered explicitly throughout this chapter.

bash

```
sudo cat /var/log/lynis.log  
sudo cat /var/log/lynis-report.dat
```

Run Lynis periodically (monthly or quarterly is a reasonable cadence for most production servers) and track the hardening score and outstanding warnings over time, treating unexpected score regressions as a signal worth investigating similar in spirit to the baseline-comparison approach introduced in section 7.1.

Managing Secrets Properly

Secrets management deserves explicit treatment as part of vulnerability management, since leaked credentials are among the most common actual causes of real-world compromise, independent of any OS-level vulnerability.

Building on Chapter 5's environment-file pattern (restricted-permission `.env` files owned by a dedicated service account), extend this discipline consistently:

- Never commit secrets (database passwords, API keys, TLS private keys) to version control. Use `.gitignore` entries for any file containing secrets, and audit repositories periodically for accidentally committed credentials.
- Restrict secret-bearing files to `600` permissions, owned by only the account that needs them, as established in Chapter 5.
- Rotate credentials periodically, and immediately upon any suspected exposure (a departing team member, an accidentally shared log excerpt, a compromised workstation).
- Avoid passing secrets as command-line arguments where possible, since command-line arguments are visible to any local user via `ps aux` — prefer environment files or dedicated secrets-management tooling.

`bash`

```
# Verify no other local user can read process command lines containing secrets
ps aux | grep -i password
```

For environments with more sophisticated secrets management needs (multiple applications, frequent rotation, audit requirements), dedicated secrets-management tools exist beyond this chapter's scope, but the fundamental discipline — least privilege, restricted permissions, no plaintext exposure in logs or process lists, periodic rotation — applies regardless of tooling sophistication.

Establishing a Formal Patch Management Policy

Even a small, single-administrator operation benefits from a written (even if brief) patch management policy, answering: how quickly are security updates applied after release (automated via `unattended-upgrades`, as configured in Chapter 2, is a reasonable default for most updates); how are kernel updates, which require a reboot, scheduled (a defined maintenance window, rather than indefinitely deferred); and how are updates that could plausibly break compatibility (a major version bump to a critical service) tested before being applied to production. This policy need not be elaborate, but

having it written down again, the Introduction's documentation principle prevents patch management from silently lapsing into inconsistency as operational pressure and competing priorities inevitably arise.

7.5 Intrusion Detection, Logging, and Security Monitoring

Prevention, however careful, is never complete. This final section of the chapter builds the detection layer: the ability to notice when something has gone wrong, ideally before significant damage occurs, and to have the evidence necessary to understand what happened.

File Integrity Monitoring with AIDE

AIDE (Advanced Intrusion Detection Environment) builds a database of cryptographic checksums for specified files and directories, then detects and reports any subsequent modification—a valuable detection mechanism for unauthorized changes to system binaries, configuration files, or other sensitive locations that a compromised system might otherwise alter without an administrator's knowledge.

bash

```
sudo apt install aide aide-common  
sudo aideinit
```

`aideinit` builds the initial baseline database, which by default is placed as `/var/lib/aide/aide.db.new`. Activate it as the reference database:

bash

```
sudo mv /var/lib/aide/aide.db.new /var/lib/aide/aide.db
```

Run a check to detect any deviation from the baseline:

bash

```
sudo aide --check
```

Schedule regular checks via a systemd timer (developed fully in Chapter 9), and treat any unexpected reported change—particularly to files under `/etc`, `/bin`, `/sbin`, or `/usr/bin`—as worthy of investigation before assuming it is

benign. AIDE's own database should itself be protected (stored on read-only media or a separate, more restricted system where feasible) since a sufficiently privileged attacker could otherwise simply update the baseline database to hide their changes.

rkhunter for Rootkit Detection

rkhunter (Rootkit Hunter) scans for signatures and behaviors associated with known rootkits, backdoors, and local exploits:

bash

```
sudo apt install rkhunter
sudo rkhunter --update
sudo rkhunter --propupd
sudo rkhunter --check
```

--propupd establishes a baseline of expected file properties (similar in spirit to AIDE, though more narrowly focused on rootkit-specific indicators); --check performs the actual scan, reporting warnings for anything matching known malicious patterns or unexpected system state. Review output in /var/log/rkhunter.log, and as with any scanning tool expect some warnings to require investigation and judgment rather than being unambiguous confirmations of compromise; rkhunter, like most heuristic detection tools, produces occasional false positives that require the administrator's own assessment.

Centralizing Authentication and Security-Relevant Logs

Building on Chapter 2's journald coverage, ensure security-relevant log sources are receiving particular attention:

bash

```
sudo journalctl -u ssh --since "24 hours ago" | grep -i "fail|invalid|refused"
sudo journalctl _COMM=sudo --since "24 hours ago"
sudo grep "Failed password" /var/log/auth.log | tail -20
```

For servers where volume or retention requirements exceed what local storage comfortably supports, forwarding logs to a remote, centralized log collector is a meaningful security improvement beyond what this book's single-server scope covers directly—a compromised server's local logs can be altered or deleted by an attacker with sufficient privilege, while logs already forwarded to a separate system remain intact as evidence. Chapter 10 introduces the broader monitoring and log-aggregation concepts relevant to this pattern.

Monitoring for Unexpected Network Listeners

Periodically compare current listening services against the baseline established in section 7.1:

```
bash
sudo ss -tulnp > /tmp/current-listening.txt
diff /root/baseline-listening-*.txt /tmp/current-listening.txt
```

An unexpected new entry in this diff a port that shouldn't be listening, or a familiar port now owned by an unfamiliar process is one of the more reliable, simple indicators that something on the system has changed outside of deliberate administrative action, and merits immediate investigation using the process- and service-inspection tools from Chapter 2.

Monitoring for Unexpected Accounts and Privilege Changes

```
bash
sudo getent passwd | awk -F: '$3 >= 1000 {print $1, $3}'
sudo getent group sudo
```

Compare this output periodically against expected administrative accounts an unexpected new account, or an existing account unexpectedly added to the `sudo` group, is a serious finding that should trigger immediate investigation, not routine dismissal.

A Basic Security Monitoring Script

Bringing several of this section's checks together into a single, schedulable script (full automation and scheduling mechanics are covered in Chapter 9; this script anticipates that treatment):

```
bash
sudo nano /opt/scripts/security-check.sh
bash
#!/bin/bash
set -euo pipefail

REPORT="/var/log/security-check-$(date +%Y%m%d).log"

echo "=== Security Check: $(date) ===" > "$REPORT"
```

```

echo "--- Failed SSH logins (last 24h) ---" >> "$REPORT"
grep "Failed password" /var/log/auth.log 2>/dev/null | tail -20 >> "$REPORT" || echo "No log data
available" >> "$REPORT"

echo "--- Listening services ---" >> "$REPORT"
ss -tulnp >> "$REPORT"

echo "--- sudo group membership ---" >> "$REPORT"
getent group sudo >> "$REPORT"

echo "--- AIDE integrity check ---" >> "$REPORT"
aide --check >> "$REPORT" 2>&1 || echo "AIDE reported changes - review required" >>
"$REPORT"

echo "--- fail2ban status ---" >> "$REPORT"
fail2ban-client status sshd >> "$REPORT" 2>&1 || echo "fail2ban not active" >> "$REPORT"

echo "==== Check complete =====" >> "$REPORT"
bash
sudo chmod +x /opt/scripts/security-check.sh
sudo /opt/scripts/security-check.sh
cat /var/log/security-check-$(date +%Y%m%d).log

```

This script does not replace deliberate, periodic manual review or the deeper tools introduced above (Lynis, debsecan, rkhunter) it provides a fast, routine snapshot suitable for daily or frequent automated execution, flagging the categories of change most likely to indicate an emerging problem, and directing attention toward the deeper tools when something looks unusual.

Responding to a Suspected Compromise

Should monitoring or auditing reveal genuine evidence of compromise, the immediate priorities are: preserve evidence (avoid rebooting or making changes that would destroy volatile evidence such as running processes and network connections, unless continued exposure poses greater risk); isolate the system (disconnect from the network, or restrict via firewall, if continued operation risks further damage or lateral movement); and begin systematic investigation using the log and process analysis tools developed throughout this chapter and Chapter 2, rather than immediately reinstalling or restoring from backup before understanding how the compromise occurred restoring without understanding the root cause risks simply reintroducing the same vulnerability. A full incident response methodology is a substantial topic beyond this book's scope, but the foundational habits established throughout this chapter baseline awareness, logging, file

integrity monitoring, and disciplined account management are precisely what make a real incident survivable and investigable rather than an unresolvable mystery.

With a layered, verifiable security baseline in place network-restricted, account-hardened, patch-current, and monitored the server is ready to safely host the containerized workloads covered next.

Chapter 8 : Containers and Modern Debian Server Workloads

8.1 Understanding Containers and Containerized Applications

Containers have become a standard deployment mechanism for modern server workloads, and Debian 13.7 is a fully capable, well-supported host for container runtimes. Understanding containers correctly — what they are, and equally, what they are not — is essential before deploying them in production.

What a Container Actually Is

A container is not a lightweight virtual machine, despite superficial similarity in how it's often described. A container is a set of **isolated processes running directly on the host's kernel**, using Linux kernel features — namespaces (isolating process IDs, network interfaces, filesystem mounts, and other resources from the host's view) and cgroups (control groups, limiting and accounting for resource usage: CPU, memory, I/O) — to create the *appearance* of an isolated environment, without the overhead of virtualizing hardware or running a separate kernel.

This distinction has real operational consequences. Containers start in milliseconds rather than the seconds-to-minutes typical of a VM boot, because there is no kernel to boot — the container process starts directly. Containers share the host kernel, meaning a kernel-level vulnerability or misconfiguration can potentially affect container isolation in ways that don't apply to genuine hardware-virtualized VMs (a security consideration returned to in section 8.5). And containers are, by design, most naturally suited to running a single, well-defined process or service — the "one primary process per container" convention, distinct from a VM's ability to run an arbitrary multi-service operating system.

Why Containers Matter for Server Administration

Containers solve a specific, recurring operational problem: **dependency and environment consistency**. An application and its exact runtime dependencies—specific library versions, language runtime versions, configuration—are packaged together into an image, which then runs identically regardless of what else is installed on the host, or what version of that dependency the host's own package manager happens to provide. This directly addresses a category of problem this book has already touched on: the difficulty of running multiple applications with potentially conflicting dependency requirements on a single host, previously mitigated through separate service accounts and careful configuration (Chapter 5), but now addressable through genuine process and filesystem isolation.

Containers also enable a particular operational pattern increasingly common in production infrastructure: treating deployment as replacing a container with a new one (built from an updated image) rather than mutating a long-running process in place—directly supporting the idempotent, repeatable-actions principle established in the Introduction.

Where Containers Are and Are Not the Right Tool

Containers are well suited to: stateless or externally-stateful application services (web APIs, background workers) that benefit from rapid, consistent deployment; environments running multiple applications with genuinely different or conflicting dependency requirements; and workflows that benefit from image-based versioning and rollback.

Traditional systemd-managed services (as built throughout Chapters 2 and 5) remain entirely appropriate, and often preferable, for: services deeply integrated with the host OS (system-level daemons, certain monitoring agents); workloads where the added abstraction layer provides no meaningful benefit and simply adds operational complexity; and situations where an organization's operational maturity or team size doesn't yet justify the additional tooling (image building, registry management, container-specific monitoring) that containerized operations require to be done well. This book does not treat containers as a universal replacement for the systemd-service patterns built earlier—both are legitimate, and the right choice depends on the specific workload and organizational context.

8.2 Installing and Configuring Docker on Debian

Docker is the most widely deployed container runtime and management toolchain, and this section installs it correctly on Debian 13.7 using Docker's official repository providing more current container-specific features and more frequent updates than Debian's own bundled container packages, which is the reasoning behind this deliberate exception to generally preferring Debian's own archive (as discussed in Chapter 2's third-party repository guidance).

Installing Docker from Docker's Official Repository

Following the secure third-party repository pattern established in Chapter 2:

```
bash
```

```
sudo apt update
```

```
sudo apt install ca-certificates curl gnupg
```

```
sudo install -m 0755 -d /etc/apt/keyrings
```

```
curl -fsSL https://download.docker.com/linux/debian/gpg | sudo gpg --dearmor -o  
/etc/apt/keyrings/docker.gpg
```

```
sudo chmod a+r /etc/apt/keyrings/docker.gpg
```

```
echo \
```

```
"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg]
```

```
https://download.docker.com/linux/debian \
```

```
$(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
```

```
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

```
sudo apt update
```

```
sudo apt install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

Verify installation:

```
bash
```

```
sudo systemctl status docker
```

```
sudo docker run hello-world
```

The `hello-world` container pulls a minimal test image, runs it (printing a confirmation message), and exits a clean, low-risk way to confirm Docker's daemon, image pulling, and container execution are all functioning correctly.

Managing Docker Access Without Constant `sudo`

By default, only root (and members of the `docker` group) can interact with the Docker daemon. Add the administrative user to this group to avoid prefixing every command with `sudo` :

```
bash
sudo usermod -aG docker adminuser
```

Security consideration: Membership in the `docker` group is functionally equivalent to unrestricted root access on the host – a member of this group can trivially mount the host's root filesystem into a container and modify it with full privileges. Grant `docker` group membership with exactly the same seriousness as granting `sudo` access, following the same least-privilege discipline established in Chapter 2, not as a casual convenience.

Log out and back in (or start a new session) for the group change to take effect, then verify:

```
bash
docker run hello-world
```

Configuring Docker's Daemon

Docker's daemon configuration lives at `/etc/docker/daemon.json` . A production-oriented baseline:

```
bash
sudo nano /etc/docker/daemon.json
json
{
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "10m",
    "max-file": "3"
  },
  "live-restore": true,
  "default-address-pools": [
    { "base": "172.30.0.0/16", "size": 24 }
  ]
}
```

- `log-driver` / `log-opts` cap the size and rotation of individual container logs, directly addressing the same unbounded-log-growth risk discussed in Chapter 2's journald retention

configuration without this, a chatty container can otherwise consume unbounded disk space over time.

- **live-restore** allows running containers to keep running even if the Docker daemon itself is restarted (for an update, for instance), avoiding unnecessary service interruption during routine Docker maintenance.
- **default-address-pools** explicitly defines the IP range Docker uses for its internally created networks, avoiding potential collision with the server's own network ranges established in Chapter 3 worth setting deliberately rather than accepting Docker's default, particularly on servers with non-trivial internal networking.

bash

```
sudo systemctl restart docker
sudo docker info
```

`docker info` reports the daemon's current configuration and state, useful for confirming the `daemon.json` changes took effect correctly.

8.3 Managing Images, Containers, Networks, and Volumes

Working with Images

Images are the immutable, layered filesystem templates containers are instantiated from.

bash

```
docker pull nginx:1.27
docker images
docker image inspect nginx:1.27
docker rmi nginx:1.27
```

Always pin specific image versions (`nginx:1.27` , not the unqualified `nginx` , which implicitly means `nginx:latest`) in any deployment intended to be reproducible `latest` is a moving target that can silently introduce breaking changes on a routine image pull, directly undermining the idempotency principle this book has emphasized throughout.

Running Containers

bash

- **-d** runs the container detached (in the background).
- **--name** assigns a predictable, referenceable name, rather than Docker's randomly generated default.
- **-p 8080:80** publishes container port 80 to host port 8080 an important distinction from the loopback-binding discipline established in Chapter 3 and Chapter 5: by default, Docker's port publishing binds to all host interfaces (0.0.0.0), which can inadvertently expose a container publicly even when the equivalent traditional service would have been deliberately bound to loopback. Bind explicitly to loopback where public exposure isn't intended:

```
docker run -d \  
  --name web-test \  
  -p 8080:80 \  
  --restart unless-stopped \  
  nginx:1.27
```

bash

```
docker run -d --name internal-app -p 127.0.0.1:8080:80 --restart unless-stopped nginx:1.27
```

- **--restart unless-stopped** ensures the container restarts automatically after a crash or host reboot, unless it was deliberately stopped the container-level equivalent of the systemd `Restart=on-failure` pattern established in Chapter 2.

Inspect and manage running containers:

bash

```
docker ps  
docker ps -a          # include stopped containers  
docker logs web-test  
docker logs -f web-test  # follow, live tail the container equivalent of journalctl -f  
docker exec -it web-test /bin/bash # interactive shell inside a running container  
docker stop web-test  
docker start web-test  
docker restart web-test  
docker rm web-test
```

`docker exec -it` opens an interactive session inside the running container's namespace useful for inspection and debugging, though as a general practice, prefer rebuilding an image with a fix over making ad hoc changes inside a running container, since such changes are lost the moment the container is removed or recreated (this ephemerality is a deliberate, important property of the container model, not a limitation to work around).

Container Resource Limits

Consistent with the resource-scoping discipline established for systemd services in Chapter 2, constrain container resource consumption explicitly:

```
bash
docker run -d \
  --name web-test \
  --memory=512m \
  --memory-swap=512m \
  --cpus=1.0 \
  -p 127.0.0.1:8080:80 \
  --restart unless-stopped \
  nginx:1.27
```

`--memory=512m` caps the container's memory usage; `--memory-swap=512m` equal to `--memory` effectively disables swap for the container, forcing an out-of-memory condition to be handled explicitly rather than degrading performance through excessive swapping; `--cpus=1.0` limits CPU consumption to the equivalent of one full core. Without explicit limits, a single misbehaving container can consume enough host resources to degrade or crash every other service on the machine – a real and common operational risk on hosts running multiple containers.

Docker Networks

By default, Docker creates a `bridge` network that containers attach to, providing basic inter-container connectivity and NAT-based outbound access. For multi-container applications, custom networks provide better isolation and name-based service discovery:

```
bash
docker network create --driver bridge app-network
docker network ls
docker network inspect app-network
```

Containers attached to the same custom bridge network can reach each other by container name (Docker provides internal DNS resolution within a custom network), a capability the default `bridge` network does not offer – this distinction becomes directly relevant in section 8.4's multi-container example.

Docker Volumes

Container filesystems are ephemeral by default data written inside a container is lost when the container is removed. **Volumes** provide persistent storage that survives container removal and recreation, and are the correct mechanism for any data a containerized application needs to retain (database files, uploaded content, application state).

```
bash
```

```
docker volume create app-data
```

```
docker volume ls
```

```
docker volume inspect app-data
```

```
bash
```

```
docker run -d \
  --name db \
  -v app-data:/var/lib/postgresql/data \
  -e POSTGRES_PASSWORD=strongpassword \
  --restart unless-stopped \
  postgres:16
```

Docker-managed named volumes (as above) are generally preferable to **bind mounts** (directly mapping a host directory into the container) for application data specifically because Docker manages their location and lifecycle consistently, and they avoid certain host-filesystem-permission complications bind mounts can introduce. Bind mounts remain appropriate for specific cases mounting configuration files from the host into a container, for instance:

```
bash
```

```
docker run -d \
  --name web-test \
  -v /opt/nginx-config/nginx.conf:/etc/nginx/nginx.conf:ro \
  -p 127.0.0.1:8080:80 \
  --restart unless-stopped \
  nginx:1.27
```

The `:ro` suffix mounts the configuration file read-only inside the container an appropriate, deliberate restriction for a file the containerized application should never need to modify itself.

8.4 Running Multi-Container Applications

Real applications typically involve multiple cooperating containers – an application server, a database, perhaps a reverse proxy or cache – and **Docker Compose** provides a declarative way to define and manage such a stack as a single unit, directly supporting this book's recurring emphasis on repeatable, documented infrastructure over manually assembled, undocumented configuration.

A Realistic Compose Application

Consider a web application (following the same architecture pattern established in Chapter 5: application behind a reverse proxy, with a database backend) defined as a multi-container stack:

bash

```
mkdir -p /opt/myapp-stack
```

```
cd /opt/myapp-stack
```

```
nano docker-compose.yml
```

yaml

```
services:
```

```
  db:
```

```
    image: postgres:16
```

```
    restart: unless-stopped
```

```
    environment:
```

```
      POSTGRES_DB: myapp_production
```

```
      POSTGRES_USER: myapp
```

```
      POSTGRES_PASSWORD_FILE: /run/secrets/db_password
```

```
  volumes:
```

```
    - db-data:/var/lib/postgresql/data
```

```
  networks:
```

```
    - internal
```

```
  secrets:
```

```
    - db_password
```

```
  healthcheck:
```

```
    test: ["CMD-SHELL", "pg_isready -U myapp"]
```

```
    interval: 10s
```

```
    timeout: 5s
```

```
    retries: 5
```

```
  app:
```

```
    build: ./app
```

```
    restart: unless-stopped
```

```
    depends_on:
```

```
      db:
```

```
        condition: service_healthy
```

```
    environment:
```

```
      DATABASE_URL: postgresql://myapp@db:5432/myapp_production
```

```
  secrets:
```

- db_password

networks:

- internal

expose:

- "8000"

nginx:

image: nginx:1.27

restart: unless-stopped

depends_on:

- app

volumes:

- ./nginx.conf:/etc/nginx/conf.d/default.conf:ro

ports:

- "127.0.0.1:8080:80"

networks:

- internal

networks:

internal:

driver: bridge

volumes:

db-data:

secrets:

db_password:

file: ./secrets/db_password.txt

Key architectural details worth explaining directly:

- **depends_on with condition: service_healthy** ensures the `app` service does not start until the database's `healthcheck` genuinely reports `healthy` not merely that the container has started, but that PostgreSQL is actually ready to accept connections. This directly mirrors the `systemd Requires= / After=` dependency discipline from Chapter 2, applied at the container-orchestration layer.
- **secrets** provides a meaningfully better mechanism for the database password than a plain environment variable the value is mounted as a file (at `/run/secrets/db_password` inside the container) rather than being visible via `docker inspect` or a container's environment listing, addressing exactly the secrets-hygiene concern raised in Chapter 7.
- **The custom internal network** ensures `db`, `app`, and `nginx` can reach each other by service name (`db`, `app`) while the database and application ports are never published to the host at all only `nginx` exposes a host-reachable port, and even that only on loopback (`127.0.0.1:8080:80`), directly extending the least-exposure principle from Chapters 3, 5, and 6 into the container networking layer.
- **expose: "8000"** on the `app` service documents the internal port without publishing it to the host accessible to other containers on the same network (specifically `nginx`, via its own configuration referencing `app:8000`), but never reachable from outside the Docker host at all.

Create the referenced secret file with appropriately restricted permissions, exactly as established for `.env` files in Chapter 5:

```
bash
mkdir -p secrets
echo -n "strongpassword" > secrets/db_password.txt
chmod 600 secrets/db_password.txt
```

Managing the Compose Stack

```
bash
docker compose up -d
```

```
docker compose ps
docker compose logs -f app
docker compose stop
docker compose down      # stops and removes containers (volumes persist unless -v is also passed)
docker compose down -v   # also removes volumes  destructive, use deliberately
```

Warning: `docker compose down -v` removes named volumes along with their contents. Never run this against a stack holding production data without a verified, current backup this is precisely the kind of destructive operation Chapter 7's backup and recovery discipline exists to protect against.

Verify the full stack is functioning correctly, applying the same layered verification approach established in Chapter 5:

```
bash
docker compose ps
curl -I http://127.0.0.1:8080/
docker compose exec db psql -U myapp -d myapp_production -c "SELECT 1;"
```

8.5 Container Security, Maintenance, and Troubleshooting

Running Containers as Non-Root

By default, many container images run their primary process as root *within the container*. While this is somewhat contained by the namespace isolation described in 8.1, running as a non-root user inside the container provides meaningful additional defense-in-depth, particularly against certain container-escape vulnerability classes:

```
dockerfile
FROM python:3.12-slim
RUN useradd --create-home --shell /usr/sbin/nologin appuser
WORKDIR /app
COPY --chown=appuser:appuser . .
USER appuser
CMD ["python3", "app.py"]
```

This mirrors, inside the container build itself, the same service-account, no-login-shell discipline established for systemd-managed applications in Chapters 2 and 5 the principle of least privilege applies identically regardless of whether the process is running directly on the host or inside a container.

Keeping Images Current

Container images bundle their own dependencies, which means they require their own, independent patching discipline distinct from the host's own `apt` update cycle established in Chapter 2 – an outdated base image can carry known vulnerabilities even on a perfectly current Debian host.

```
bash
```

```
docker pull nginx:1.27
docker compose pull
docker compose up -d    # recreates containers using newly pulled images
```

Establishing a routine (ideally automated, per Chapter 9) of pulling updated images and recreating affected containers is as essential to a container-based deployment's security posture as `unattended-upgrades` is to the host itself.

Scanning Images for Known Vulnerabilities

```
bash
```

```
docker scout quickview nginx:1.27
```

(`docker scout` , bundled with current Docker CE installations, provides vulnerability scanning against known CVE databases for a given image – reviewing this before deploying a new or updated image, particularly one from a third-party or less-trusted source, extends the vulnerability-management discipline established in Chapter 7 directly into the container image layer.)

Restricting Container Capabilities

By default, Docker containers run with a reduced (but still fairly broad) set of Linux capabilities compared to genuine root. For services that need even less, restrict explicitly:

```
bash
```

```
docker run -d \
  --name web-test \
  --cap-drop=ALL \
  --cap-add=NET_BIND_SERVICE \
  -p 127.0.0.1:8080:80 \
  nginx:1.27
```

`--cap-drop=ALL` removes all Linux capabilities, then `--cap-add=NET_BIND_SERVICE` restores only the specific capability required (binding to a privileged port below 1024) another direct, concrete application of least privilege, this time at the kernel-capability layer within the container.

Container Log Management

Building on the `daemon.json` log rotation configured in section 8.2, review individual container logs consistent with the diagnostic patterns established in Chapters 2 and 5:

bash

```
docker logs --since 1h myapp
docker compose logs --tail=100 -f
```

For centralized log handling across many containers, Docker's logging drivers can forward output to `journald` directly, unifying container logs with the rest of the system's logging infrastructure established in Chapter 2:

json

```
{
  "log-driver": "journald"
}
```

bash

```
sudo systemctl restart docker
sudo journalctl CONTAINER_NAME=web-test -f
```

Troubleshooting Common Container Problems

Symptom	Likely Cause	Diagnostic Approach
Container exits immediately after starting	Primary process crashed or exited normally (no long-running foreground process)	<code>docker logs <container></code> , check the image's CMD / ENTRYPOINT
"Port already in use" error on <code>docker run</code>	Host port conflict with another service or container	<code>sudo ss -tulnp grep <port></code>
Container can't reach another container by name	Containers on different/default networks, no custom network configured	<code>docker network inspect</code> , confirm both containers share a custom network
Application inside container can't reach the internet	Host firewall (Chapter 7) blocking Docker's own iptables/nftables chains, or DNS misconfiguration	<code>docker exec <container> ping 9.9.9.9</code> , <code>docker exec <container> cat /etc/resolv.conf</code>
Container using excessive memory/CPU	No resource limits configured, or a genuine application issue	<code>docker stats</code> , review <code>--memory</code> / <code>--cpus</code> configuration
Changes to files inside container don't persist	Data written to ephemeral container filesystem rather than a mounted volume	Confirm intended persistent paths are mapped via <code>-v</code> / <code>volumes</code> :

`docker stats` deserves particular mention as a real-time, per-container resource monitoring tool, directly analogous to `top` / `htop` at the host level (covered fully in Chapter 10):

```
bash
docker stats
```

An Important Note on Firewall Interaction

Docker manipulates the host's netfilter rules directly to implement port publishing and container networking, which can interact in non-obvious ways with the nftables configuration established in Chapter 7 notably, Docker's own rules can sometimes take precedence in ways that cause a published container port to be reachable even when the host's own firewall ruleset appears to restrict it. After deploying containerized services, always verify actual external reachability directly (as established in section 7.2), rather than assuming the host firewall's ruleset alone fully describes what is reachable once Docker is involved:

```
bash
sudo nft list ruleset | grep -A 5 DOCKER
```

Reviewing Docker's own injected chains within the `nftables` ruleset, and testing actual external connectivity with `nmap` or `nc` from a separate host as established in section 7.2, is the reliable way to confirm a container's actual network exposure matches what was intended – an important verification step specifically because container networking is one of the more common sources of an unintended public exposure on an otherwise carefully firewalled Debian server.

With containerized workloads deployable, secured, and diagnosable alongside the traditional systemd-managed services from earlier chapters, the book turns next to tying both deployment models together through systematic automation.

Chapter 9 : Automation and Configuration Management

9.1 Automating Server Administration with Bash

Every chapter so far has demonstrated individual commands and procedures performed manually. Production server administration, however, depends on converting repeated manual procedures into reliable, tested automation directly serving the idempotent, repeatable-actions principle established in the Introduction, and reducing the human-error risk inherent in manually repeating multi-step procedures under time pressure.

Why Bash, and Where Its Limits Are

Bash scripting remains the standard automation language for Debian server administration because it operates natively wherever a shell is available, requires no additional runtime installation, and directly wraps the same commands an administrator would type interactively making scripts easy to develop incrementally from manual command sequences already validated by hand. Bash is not, however, well suited to complex data manipulation, sophisticated error handling, or large-scale application logic for genuinely complex automation needs, Python is often a better choice. This chapter focuses on Bash for the class of tasks it excels at: sequencing existing commands reliably, with proper error handling, logging, and scheduling.

Script Structure and Safety Defaults

Every production script in this book follows a consistent safety baseline, extending the pattern introduced in Chapter 6's backup script:

```
bash
#!/bin/bash
set -euo pipefail
IFS=$'\n\t'
```

- `set -e` exit immediately if any command returns a non-zero (failure) exit status, preventing a script from blindly continuing after an earlier step has already failed.
- `set -u` treat any reference to an unset variable as an error, catching typos in variable names before they cause silent, incorrect behavior.
- `set -o pipefail` ensures a pipeline's exit status reflects the first failing command in the chain, not merely the last one (without this, `false | true` reports success, masking the actual failure).
- `IFS=$'\n\t'` restricts word-splitting to newlines and tabs rather than Bash's default (which includes spaces), avoiding a common class of subtle bugs when handling filenames or values containing spaces.

Variables, Conditionals, and Loops in Practice

Rather than treating these as abstract syntax, this section demonstrates them within a realistic operational script – a disk-space monitoring check building directly on the `df`/storage concepts from Chapter 6:

`bash`

`#!/bin/bash`

`set -euo pipefail`

`THRESHOLD=85`

`ALERT_EMAIL="admin@example.com"`

`HOSTNAME=$(hostname -f)`

```
check_filesystem() {
    local mount_point="$1"
    local usage
    usage=$(df --output=pcent "$mount_point" | tail -1 | tr -d ' %')

    if [[ "$usage" -ge "$THRESHOLD" ]]; then
        echo "WARNING: $mount_point on $HOSTNAME is at ${usage}% usage"
        return 1
    else
        echo "OK: $mount_point at ${usage}%"
        return 0
    fi
}
```

`FAILURES=0`

```
for fs in / /var /var/log /home; do
    if ! check_filesystem "$fs"; then
```

```

        FAILURES=$((FAILURES + 1))
    fi
done

if [[ "$FAILURES" -gt 0 ]]; then
    echo "Disk space check found $FAILURES filesystem(s) above ${THRESHOLD}% threshold" |
    \
        mail -s "Disk space warning on $HOSTNAME" "$ALERT_EMAIL"
    exit 1
fi

echo "All filesystems within normal thresholds"
exit 0

```

This script demonstrates several practical conventions worth naming explicitly: a **function**(`check_filesystem`) encapsulating a single, reusable check, with a `local` variable scoped to that function rather than polluting the global script namespace; a **for loop** iterating over a defined list of mount points, directly reusable if additional filesystems need checking later without duplicating logic; explicit **exit codes**(0 for success, 1 for failure) allowing the script's result to be checked programmatically by whatever invokes it essential for the scheduling and alerting integration covered in sections 9.2 and 9.3; and a **conditional**(`if ! check_filesystem`) using the function's own return status directly, rather than parsing its text output, a more robust pattern than string-matching a command's output to determine success or failure.

Argument Handling for Reusable Scripts

Scripts intended for repeated, varied use should accept parameters rather than hardcoding values:

```

bash
#!/bin/bash
set -euo pipefail

usage() {
    echo "Usage: $0 -s <service_name> [-t <timeout_seconds>]"
    exit 1
}

TIMEOUT=30

while getopts "s:t:h" opt; do
    case "$opt" in
        s) SERVICE="$OPTARG" ;;

```

```

        t) TIMEOUT="$OPTARG" ;;
        h) usage ;;
        *) usage ;;
    esac
done

if [[ -z "${SERVICE:-}" ]]; then
    echo "Error: service name is required" >&2
    usage
fi

echo "Checking service: $SERVICE (timeout: ${TIMEOUT}s)"

if timeout "$TIMEOUT" systemctl is-active --quiet "$SERVICE"; then
    echo "Service $SERVICE is active"
    exit 0
else
    echo "Service $SERVICE is not active or check timed out" >&2
    exit 1
fi

```

getopts provides standard, POSIX-conventional argument parsing (-s myapp -t 10); the usage function, called both on explicit -h and on invalid input, follows the standard Unix convention of self-documenting command-line tools, valuable when a script is later invoked by someone other than its original author or by the original author, months later, having forgotten the exact invocation syntax.

Error Handling and Logging Conventions

For scripts intended to run unattended (via cron or systemd timers, covered next), robust logging is essential a script that fails silently during an unattended 3 AM run provides no opportunity for timely investigation.

```

bash
#!/bin/bash
set -euo pipefail

LOG_FILE="/var/log/maintenance-task.log"

log() {
    echo "[$(date '+%Y-%m-%d %H:%M:%S')] $*" | tee -a "$LOG_FILE"
}

trap 'log "ERROR: Script failed at line $LINENO"' ERR

log "Starting maintenance task"

# ... actual task logic here ...

```

```
log "Maintenance task completed successfully"
```

The `trap 'log "..."' ERR` line registers a handler that executes automatically whenever a command fails (consistent with `set -e`'s exit-on-error behavior) capturing exactly which line triggered the failure into the log file, providing far more actionable diagnostic information than a script that simply stops without explanation. This pattern, combined with `set -euo pipefail`, forms the reliable error-handling baseline this book uses for every subsequent automation script.

9.2 Scheduled Administration with cron and systemd Timers

Chapter 2 introduced systemd timers briefly, alongside the more traditional cron. This section develops both mechanisms fully, since real Debian infrastructure commonly uses both, and understanding when to prefer one over the other is a genuinely useful operational skill.

cron Fundamentals

System-wide scheduled tasks are configured in `/etc/crontab` or, more commonly and more cleanly, as individual files under `/etc/cron.d/`:

```
bash
```

```
sudo nano /etc/cron.d/security-check
# Run security check daily at 2:15 AM as root
15 2 * * * root /opt/scripts/security-check.sh >> /var/log/cron-security-check.log 2>&1
```

The five schedule fields represent minute, hour, day-of-month, month, and day-of-week respectively (in that order), each accepting `*` for "every," specific values, ranges, or step values (`*/15` for "every 15 units," for instance). A crontab entry under `/etc/cron.d/` explicitly names the executing user (here, `root`) as the sixth field distinct from a user's personal crontab (edited via `crontab -e`), which omits the user field since it is implicit.

Per-user scheduling:

```
bash
```

```
crontab -e      # edit the invoking user's own crontab
crontab -l      # list current entries
sudo crontab -u adminuser -l
```

Important cron Operational Details

Always redirect output explicitly. cron by default emails any command output to the executing user's local mail (which, on most server configurations, goes nowhere useful unless mail delivery is specifically configured) explicit redirection to a log file, as shown above, ensures output is actually retrievable:

```
15 2 * * * root /opt/scripts/security-check.sh >> /var/log/cron-security-check.log 2>&1
```

cron's environment is minimal, lacking the full `PATH` and environment variables present in an interactive shell session. Scripts intended for cron execution should use absolute paths for all commands and files, and explicitly source any required environment rather than assuming interactive-shell defaults:

```
bash
#!/bin/bash
export PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

This is a genuinely common source of "works when I run it manually, fails under cron" confusion, worth explicitly guarding against in any script intended for scheduled execution.

systemd Timers as cron's Modern Alternative

As introduced in Chapter 2, systemd timers offer tighter integration with the rest of the systemd ecosystem dependency handling, resource limits (applicable to the timer's triggered service unit exactly as to any other service), and unified logging via `journalctl` rather than cron's more fragmented output handling.

```
bash
sudo nano /etc/systemd/system/disk-check.service
ini
[Unit]
Description=Disk space threshold check

[Service]
Type=oneshot
ExecStart=/opt/scripts/disk-check.sh
bash
sudo nano /etc/systemd/system/disk-check.timer
```

ini

[Unit]

Description=Run disk space check twice daily

[Timer]

OnCalendar=*-*-* 06,18:00:00

Persistent=true

RandomizedDelaySec=300

[Install]

WantedBy=timers.target

OnCalendar=*-*-* 06,18:00:00 schedules execution at 06:00 and 18:00 daily, using systemd's calendar time syntax (more expressive and more readable than cron's five-field format for many common schedules); RandomizedDelaySec=300 introduces up to a 5-minute random delay, useful for avoiding synchronized load spikes when the same timer pattern is deployed across many servers simultaneously a consideration with no direct cron equivalent.

bash

sudo systemctl daemon-reload

sudo systemctl enable --now disk-check.timer

systemctl list-timers

journalctl -u disk-check.service --since today

journalctl -u disk-check.service provides unified, filterable log access to every execution of this scheduled task directly comparable to any other systemd service's logging, and generally more convenient for troubleshooting than parsing cron's redirected log files.

Choosing Between cron and systemd Timers

Consideration	cron	systemd Timers
Simplicity for basic schedules	Simple, widely familiar syntax	Slightly more verbose (two files per task)
Logging	Requires explicit output redirection	Native, unified via journalctl
Dependency handling	None	Full systemd dependency model (After= , Requires=)
Resource limits	None native	Full systemd resource control (MemoryMax= , etc.)
Missed-run handling	No native catch-up	Persistent=true catches up missed runs
Portability across systems	Nearly universal across Unix-likes	systemd-specific

For straightforward, simple scheduled tasks, `cron` remains a perfectly reasonable, low-overhead choice. For tasks benefiting from dependency awareness, resource limiting, or tighter log integration `systemd` timers are increasingly the norm for anything operationally significant `systemd` timers are generally the better-engineered choice on a `systemd`-based system like Debian 13.7, and are used preferentially for the more substantial automation examples in the remainder of this chapter.

9.3 Automating Updates, Backups, and Maintenance

This section builds complete, production-oriented automation for three recurring operational needs, applying the scripting discipline from 9.1 and the scheduling mechanisms from 9.2.

Automated Update Verification and Reporting

While `unattended-upgrades` (Chapter 2) handles the actual application of security updates, a complementary script providing explicit visibility into what was updated closes an important gap `unattended` automation that provides no ongoing visibility is difficult to trust over time.

```
bash
sudo nano /opt/scripts/update-report.sh
bash
#!/bin/bash
set -euo pipefail

LOG_FILE="/var/log/update-report.log"
REPORT_EMAIL="admin@example.com"
HOSTNAME=$(hostname -f)

log() {
    echo "[$(date '+%Y-%m-%d %H:%M:%S')] $*" | tee -a "$LOG_FILE"
}

trap 'log "ERROR: update-report failed at line $LINENO"' ERR

log "Checking recently applied updates"

RECENT_UPDATES=$(grep " upgrade " /var/log/dpkg.log 2>/dev/null | grep "$(date +%Y-%m-%d)" || echo "No updates applied today")

log "Updates applied today:"
```

```

log "$RECENT_UPDATES"

PENDING=$(apt list --upgradable 2>/dev/null | grep -v "^Listing" | wc -l)

if [[ "$PENDING" -gt 0 ]]; then
    log "WARNING: $PENDING package(s) still pending update"
    {
        echo "Host: $HOSTNAME"
        echo "Pending updates: $PENDING"
        echo ""
        apt list --upgradable 2>/dev/null | grep -v "^Listing"
    } | mail -s "Pending updates on $HOSTNAME" "$REPORT_EMAIL"
else
    log "System fully up to date"
fi

```

Automated, Verified Backups

Extending Chapter 6's backup script with the error-handling and logging discipline established in this chapter, and adding an important addition: automated verification that the backup actually succeeded, rather than merely assuming it did.

```

bash
sudo nano /opt/scripts/automated-backup.sh
bash
#!/bin/bash
set -euo pipefail

BACKUP_ROOT="/backups"
BACKUP_DIR="$BACKUP_ROOT/$(date +%Y%m%d)"
REMOTE_DEST="backup-host:/backups/debian-server/"
LOG_FILE="/var/log/automated-backup.log"
ALERT_EMAIL="admin@example.com"
HOSTNAME=$(hostname -f)

log() {
    echo "[$(date '+%Y-%m-%d %H:%M:%S')] $*" | tee -a "$LOG_FILE"
}

alert_failure() {
    log "BACKUP FAILURE: $1"
    echo "Backup failed on $HOSTNAME: $1" | mail -s "BACKUP FAILURE on $HOSTNAME"
"$ALERT_EMAIL"
    exit 1
}

trap 'alert_failure "Unexpected error at line $LINENO"' ERR

log "Starting automated backup"

```

```

mkdir -p "$BACKUP_DIR"

log "Backing up PostgreSQL database"
if ! sudo -u postgres pg_dump myapp_production | gzip >
"$BACKUP_DIR/myapp_production.sql.gz"; then
    alert_failure "PostgreSQL dump failed"
fi

if [[ ! -s "$BACKUP_DIR/myapp_production.sql.gz" ]]; then
    alert_failure "Database backup file is empty"
fi

log "Backing up configuration"
tar -czf "$BACKUP_DIR/etc-backup.tar.gz" /etc

log "Backing up application data"
tar -czf "$BACKUP_DIR/myapp-data.tar.gz" /opt/myapp/data

log "Verifying archive integrity"
for archive in "$BACKUP_DIR"/*.tar.gz; do
    if ! tar -tzf "$archive" > /dev/null 2>&1; then
        alert_failure "Archive integrity check failed for $archive"
    fi
done

log "Syncing to remote backup host"
if ! rsync -avz "$BACKUP_DIR" "$REMOTE_DEST"; then
    alert_failure "rsync to remote backup host failed"
fi

log "Removing local backups older than 7 days"
find "$BACKUP_ROOT" -maxdepth 1 -mtime +7 -type d -exec rm -rf {} \;

log "Backup completed successfully"
echo "Backup completed successfully on $HOSTNAME at $(date)" | \
    mail -s "Backup success on $HOSTNAME" "$ALERT_EMAIL"

```

The additions beyond Chapter 6's version are significant and deliberate:

`alert_failure` ensures any failure is both logged and actively surfaced via email, rather than only discoverable through a later log review that might not happen promptly; the empty-file check (`-s`) catches the specific, easily-missed failure mode of a backup command that exits successfully but produces no actual data (a corrupted PostgreSQL connection returning immediately, for instance); and the archive integrity check (`tar -tzf`) confirms the resulting archives are genuinely readable, not merely present directly serving Chapter 6's core principle that an unverified backup cannot be assumed recoverable.

Schedule via a systemd timer, following the pattern from 9.2:

```
bash
sudo nano /etc/systemd/system/automated-backup.timer
ini
[Timer]
OnCalendar=*-*-* 01:30:00
Persistent=true
```

Automated Maintenance Tasks

A consolidated routine maintenance script, tying together several housekeeping tasks introduced across earlier chapters:

```
bash
sudo nano /opt/scripts/routine-maintenance.sh
bash
#!/bin/bash
set -euo pipefail

LOG_FILE="/var/log/routine-maintenance.log"

log() {
    echo "[$(date '+%Y-%m-%d %H:%M:%S')] $*" | tee -a "$LOG_FILE"
}

trap 'log "ERROR: maintenance failed at line $LINENO"' ERR

log "Starting routine maintenance"

log "Cleaning unused packages"
apt autoremove --purge -y >> "$LOG_FILE" 2>&1

log "Cleaning apt cache"
apt clean

log "Enforcing journal retention"
journalctl --vacuum-time=30d >> "$LOG_FILE" 2>&1

log "Cleaning stale temporary files"
find /tmp -type f -atime +7 -delete

log "Running file integrity check"
aide --check >> "$LOG_FILE" 2>&1 || log "AIDE reported changes - review required"

log "Routine maintenance completed"
```

9.4 Introduction to Configuration Management Concepts

While this book's automation to this point has focused on Bash scripting for discrete operational tasks, larger or more dynamic infrastructure typically benefits from **configuration management** tools—systems designed specifically to declare and enforce a server's desired configuration state, rather than scripting individual imperative steps.

Imperative Scripting vs. Declarative Configuration Management

Every script in this chapter so far is **imperative**: it describes a specific sequence of steps to perform. This works well and is entirely appropriate for many operational tasks, but has a real limitation at scale: a script describes what to *do*, not what the *end state* should be, and running it against a server already in a different, unexpected state can produce inconsistent or unintended results.

Declarative configuration management tools (Ansible, Puppet, Chef, and others) instead describe the *desired end state*—"this package should be installed," "this file should contain this content," "this service should be running and enabled"—and the tool itself determines what changes, if any, are needed to reach that state, safely doing nothing if the system is already correctly configured. This property, called **idempotency**, means the same configuration can be applied repeatedly and safely, converging any starting state toward the same defined target—a stronger, more systematic version of the `set -e` safety and careful scripting discipline this chapter has built manually into individual Bash scripts.

A Brief, Practical Introduction to Ansible

Ansible is a widely used, agentless configuration management tool it requires no persistent software installed on managed servers beyond Python and SSH access, both of which a Debian server already has by default, making it a low-friction entry point into configuration management for administrators already comfortable with the SSH-based workflows this book has built throughout.

Install Ansible on a **control machine** (a workstation or dedicated management server not typically installed on the managed servers themselves):

```
bash
sudo apt install ansible
```

Define an inventory of managed hosts:

```
bash
nano ~/ansible/inventory.ini
ini
[debian_servers]
debian-server ansible_host=192.0.2.10 ansible_user=adminuser
```

A simple playbook applying several practices already established manually throughout this book demonstrating how prior chapters' individual procedures translate into declarative, repeatable configuration:

```
bash
nano ~/ansible/baseline.yml
yaml
---
- name: Apply Debian server security baseline
  hosts: debian_servers
  become: true

  tasks:
    - name: Ensure system packages are up to date
      apt:
        update_cache: true
        upgrade: dist

    - name: Ensure fail2ban is installed
      apt:
```

```

    name: fail2ban
    state: present

- name: Ensure fail2ban is running and enabled
  systemd:
    name: fail2ban
    state: started
    enabled: true

- name: Ensure SSH password authentication is disabled
  lineinfile:
    path: /etc/ssh/sshd_config
    regexp: '^#?PasswordAuthentication'
    line: 'PasswordAuthentication no'
    notify: restart ssh

- name: Ensure unattended-upgrades is installed
  apt:
    name: unattended-upgrades
    state: present

handlers:
- name: restart ssh
  systemd:
    name: ssh
    state: restarted

```

Run the playbook against the defined inventory:

bash

```
ansible-playbook -i ~/ansible/inventory.ini ~/ansible/baseline.yml --check
```

The `--check` flag performs a **dry run**, reporting what changes *would* be made without actually applying them directly analogous to the `nginx -t / sshd -t` verify-before-apply discipline established throughout this book, and an essential safety habit before applying any configuration management change to production infrastructure. Once reviewed, apply for real:

bash

```
ansible-playbook -i ~/ansible/inventory.ini ~/ansible/baseline.yml
```

This is necessarily a brief introduction. Ansible, and configuration management as a discipline, is a substantial topic in its own right, beyond this book's scope to cover exhaustively. The purpose here is to establish the conceptual bridge: everything this book has taught as manual procedure and Bash automation can be expressed, more systematically and more safely at scale, as declarative configuration and recognizing when an environment

has grown to the point where that investment is worthwhile is itself an important piece of operational judgment.

9.5 Building Repeatable Server Provisioning Workflows

From Manual Installation to Reproducible Provisioning

Chapter 1 walked through installing and configuring a Debian 13.7 server manually—a necessary foundation for understanding what's actually happening at each step. Production operations, however, increasingly benefit from being able to reproduce that same baseline configuration reliably, whether for disaster recovery (rebuilding a lost server quickly, directly supporting Chapter 11's recovery objectives), horizontal scaling (provisioning additional, identically configured servers), or simply eliminating configuration drift between servers that were each configured slightly differently by hand over time.

Combining This Chapter's Tools into a Provisioning Workflow

A practical, achievable provisioning workflow for a small-to-medium Debian deployment combines several tools already introduced: a base Debian 13.7 image or installation (Chapter 1); a cloud-init configuration (where the deployment platform supports it—most VPS and cloud providers do) or a bootstrap script handling immediate post-installation setup; and an Ansible playbook (or equivalent) applying the full baseline configuration—users, SSH hardening, firewall rules, monitoring agents, and any standard software—consistently across every server.

A representative cloud-init configuration, supplying much of Chapter 1's initial setup automatically at first boot on platforms that support it:

```
yaml
#cloud-config
hostname: debian-server
manage_etc_hosts: true

users:
  - name: adminuser
    groups: sudo
    shell: /bin/bash
```

```
sudo: ['ALL=(ALL) NOPASSWD:ALL']
ssh_authorized_keys:
  - ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAI... adminuser@workstation

ssh_pwauth: false

package_update: true
package_upgrade: true

packages:
  - fail2ban
  - nftables
  - unattended-upgrades

runcmd:
  - systemctl enable --now fail2ban
  - systemctl enable --now nftables
```

This single configuration, supplied to a supporting provider at instance creation time, automatically establishes a hostname, a properly configured administrative user with SSH key authentication (and no password authentication) already in place, and baseline security packages installed and enabled directly reproducing a substantial portion of Chapters 1, 2, 4, and 7's manual work, consistently, at the moment of provisioning, before the server is ever manually touched.

Documenting the Provisioning Process Itself

Whatever combination of tools a given environment uses manual procedure, Bash scripts, cloud-init, Ansible playbooks, or some combination the provisioning process itself should be version-controlled and documented, not scattered across ad hoc notes or, worse, existing only in a single administrator's memory. A Git repository containing the organization's Ansible playbooks, cloud-init templates, and standard scripts with commit history documenting when and why each piece of configuration changed is itself a direct, practical realization of the Introduction's "documentation as an operational asset" principle, converted from an abstract value into concrete, durable infrastructure.

The Payoff: Disaster Recovery Through Reproducibility

The clearest demonstration of this chapter's value becomes apparent in a genuine disaster scenario the subject of Chapter 11's recovery procedures. A server that can be provisioned entirely from documented, version-

controlled configuration can be rebuilt from scratch, correctly, in the time it takes to provision new infrastructure and run existing automation against it a dramatically different, and dramatically less stressful, recovery experience than attempting to manually recall and reproduce a uniquely, undocumented, hand-configured server's exact state under the time pressure of an active outage. This is, ultimately, the point of everything this chapter has built: not automation as an end in itself, but automation as the mechanism that makes a server's configuration genuinely durable, auditable, and recoverable the operational foundation the book's remaining chapters on monitoring, backup, and recovery depend on directly.

Chapter 10 : Monitoring, Logging, and Performance Engineering

10.1 Server Monitoring Fundamentals

Monitoring is the discipline of continuously observing a server's behavior to detect problems early, understand normal operating ranges, and gather the evidence needed for both routine capacity planning and incident investigation. Earlier chapters have introduced individual diagnostic commands reactively, in response to specific problems (Chapter 3's network diagnostics, Chapter 5's web server troubleshooting). This chapter builds monitoring as a proactive, ongoing practice rather than a purely reactive one.

Why Monitoring Is Distinct From Troubleshooting

Troubleshooting (developed fully in Chapter 11) is what an administrator does *after* a problem has been noticed. Monitoring is what makes that noticing possible in the first place and, done well, surfaces problems before they become outages at all. A server whose disk is filling up gradually, whose memory usage is climbing steadily across days, or whose response times are slowly degrading under growing load, presents no dramatic failure event to react to only a trend, visible solely to an administrator who is actually looking, ideally through tooling that makes the trend visible without requiring constant manual checking.

The Three Pillars: Metrics, Logs, and Alerts

Effective monitoring practice rests on three complementary categories of information:

Metrics numeric measurements sampled over time (CPU utilization, memory usage, disk I/O, request latency), valuable specifically because they reveal trends and allow threshold-based alerting.

Logs the detailed, often textual record of specific events (already covered extensively in Chapter 2's journald treatment and Chapter 7's security logging), valuable for understanding exactly *what happened* once a metric or alert indicates *something* happened.

Alerts notifications triggered when a metric crosses a defined threshold or a specific log pattern occurs, converting passive data collection into active notification, without which an administrator would need to continuously watch dashboards manually to notice a problem impractical for any real operation.

This chapter builds practical, Debian-native tooling across all three pillars, without assuming a large-scale, dedicated monitoring platform appropriate to the scale of the single or small-cluster deployments this book has focused on throughout, while noting where more sophisticated tooling becomes appropriate as infrastructure scales.

Establishing Baselines Before Alerting Thresholds

A recurring theme worth stating explicitly here: meaningful alert thresholds require knowing what "normal" looks like for a given server and workload first. A database server under genuinely heavy, legitimate load might show CPU utilization patterns that would be alarming on a lightly loaded web server. Before configuring alerting thresholds (section 10.2 onward), spend time observing actual normal operating ranges directly paralleling the performance-baseline discipline formalized later in section 10.4.

10.2 CPU, Memory, Disk, and Network Monitoring

CPU Monitoring

bash
[top](#)

`top`'s default view refreshes continuously, showing per-process CPU consumption alongside system-wide summary figures. The **load average** figures shown at the top (three numbers representing 1, 5, and 15-minute averages) deserve particular understanding: they represent the average

number of processes either running or waiting for CPU time a load average consistently above the number of available CPU cores indicates genuine CPU contention, while a load average below core count generally indicates adequate headroom. Check core count for context:

```
bash
nproc
```

For a more configurable, scriptable view:

```
bash
htop
```

htop (install via `sudo apt install htop`) provides an interactive, color-coded, more navigable alternative to `top`, including per-core CPU breakdown and easier process management (killing, renicing) directly from the interface generally the preferred tool for interactive investigation, while `top`'s simpler, more universally available output remains useful for quick checks or scripting.

For non-interactive, scriptable CPU sampling:

```
bash
mpstat 1 5          # requires: sudo apt install sysstat; 5 samples, 1 second apart
vmstat 1 5
```

vmstat's output includes CPU columns (`us` user time, `sy` system time, `id` idle time, `wa` I/O wait time) the `wa` column deserves particular attention: high I/O wait indicates the CPU is frequently idle specifically because it's waiting on disk I/O to complete, a meaningfully different bottleneck (addressed by storage improvements) than genuine CPU saturation (addressed by reducing CPU-bound work or adding CPU capacity).

Memory Monitoring

```
bash
free -h
```

Understanding `free`'s output correctly is essential and commonly misunderstood: the `available` column (not `free`) represents memory genuinely available for new processes, since Linux deliberately uses otherwise-idle memory for filesystem caching (the `buff/cache` column) cache memory that

is immediately reclaimable the moment an application actually needs it. A server showing low free memory but high available memory is operating entirely normally, not experiencing memory pressure a distinction worth internalizing thoroughly, since misreading this output is a very common source of unwarranted concern.

Genuine memory pressure is better identified through swap activity and specific pressure indicators:

```
bash
vmstat 1 5
```

Sustained non-zero values in `vmstat`'s `si` / `so` (swap in/swap out) columns indicate the system is actively under enough memory pressure to be swapping a meaningful performance concern, since swap I/O is dramatically slower than RAM access, unlike merely low free /high buff-cache, which is not a concern at all.

Per-process memory consumption:

```
bash
ps aux --sort=-%mem | head -15
```

Disk I/O Monitoring

Building on Chapter 6's space-utilization monitoring (`df`, `du`), disk *performance* (as opposed to space consumption) requires different tooling:

```
bash
iostat -x 1 5      # requires: sudo apt install sysstat
```

Key columns in `iostat -x` output: `%util` (percentage of time the device was busy servicing requests sustained values near 100% indicate the disk is a genuine bottleneck); `await` (average time, in milliseconds, for I/O requests to complete rising `await` values under load indicate the storage subsystem struggling to keep up); and `r/s` / `w/s` (reads and writes per second), useful for understanding whether a workload is read-heavy, write-heavy, or balanced.

For identifying which specific process is generating disk I/O:

```
bash
sudo iotop          # requires: sudo apt install iotop
```

`iostat` provides a `top`-like interface specifically for disk I/O, showing per-process read/write throughput the most direct tool for answering "what is actually hammering the disk right now" during an active performance investigation.

Network Monitoring

Building on Chapter 3's connectivity-focused networking tools, ongoing network *performance* monitoring uses complementary commands:

bash

```
sudo ss -s          # summary socket statistics
sudo ss -tan state established | wc -l    # current established connection count, trended over time
```

For throughput monitoring:

bash

```
sudo apt install nload
sudo nload enp1s0
```

nload provides a live, graphical (text-mode) view of inbound/outbound throughput on a given interface useful for quickly confirming whether a performance issue correlates with genuinely high network throughput, or whether network capacity is not actually the limiting factor.

For detailed per-connection or per-process network usage:

bash

```
sudo apt install nethogs
sudo nethogs enp1s0
```

nethogs attributes network bandwidth consumption to specific processes, directly analogous to `iotop`'s process-level attribution for disk I/O valuable when investigating unexpectedly high network usage and needing to identify its actual source.

A Consolidated Resource-Check Script

Bringing this section's tools together into a single, schedulable snapshot script, extending the automation patterns from Chapter 9:

bash

```
sudo nano /opt/scripts/resource-snapshot.sh
```

bash

```
#!/bin/bash
```

```
set -euo pipefail
```

```
REPORT="/var/log/resource-snapshot-$(date +%Y%m%d-%H%M).log"
```

```

{
    echo "=== Resource Snapshot: $(date) ==="
    echo ""
    echo "--- Load Average / CPU ---"
    uptime
    echo ""
    echo "--- Memory ---"
    free -h
    echo ""
    echo "--- Disk Space ---"
    df -h
    echo ""
    echo "--- Disk I/O ---"
    iostat -x 1 2 | tail -n +7
    echo ""
    echo "--- Top 10 processes by CPU ---"
    ps aux --sort=-%cpu | head -11
    echo ""
    echo "--- Top 10 processes by Memory ---"
    ps aux --sort=-%mem | head -11
    echo ""
    echo "--- Network connections summary ---"
    ss -s
} > "$REPORT"

echo "Snapshot written to $REPORT"

```

Scheduled hourly via a systemd timer (following Chapter 9's pattern), this script builds a historical record of resource state, directly useful both for identifying trends over time and for providing immediate context when investigating a specific incident after the fact "what did resource usage look like in the hour leading up to this problem" is a question this script's accumulated history can answer directly.

10.3 Centralized and Persistent Log Management

Extending journald's Role Beyond a Single Server

Chapter 2 established journald as the primary logging mechanism for a single Debian server, with persistent storage and retention policy. As infrastructure grows beyond a single server, or as log retention and tamper-resistance requirements increase (directly relevant to Chapter 7's security-investigation concerns), centralizing logs to a dedicated, separate system becomes operationally valuable.

Forwarding Logs with rsyslog

While `journald` handles local structured logging well, `rsyslog` (commonly present alongside `journald` on Debian, as noted in Chapter 2) provides mature, widely supported remote log forwarding:

```
bash
sudo apt install rsyslog
sudo nano /etc/rsyslog.d/60-remote-forward.conf
*. * @@logs.example.com:514
```

The `@@` prefix specifies TCP (a single `@` specifies UDP). TCP is generally preferred for reliability, since UDP-based forwarding can silently drop messages under network stress, exactly when logs may matter most). Restart to apply:

```
bash
sudo systemctl restart rsyslog
```

On the receiving log server side, `rsyslog` must be explicitly configured to accept incoming connections:

```
module(load="imtcp")
input(type="imtcp" port="514")

$template RemoteLogs,"/var/log/remote/%HOSTNAME%/%PROGRAMNAME%.log"
*. * ?RemoteLogs
& stop
```

This configuration accepts incoming TCP syslog traffic and writes it into a per-host, per-program directory structure providing exactly the separated-from-the-source-system durability discussed in Chapter 7: a compromised or failed source server's local logs may be lost or tampered with, but logs already forwarded and stored on a separate system remain intact evidence.

Forwarding journald Data Specifically

For environments wanting to preserve `journald`'s structured metadata (rather than flattening to traditional syslog text) across systems, `systemd-journal-remote` provides native journal-to-journal forwarding:

```
bash
sudo apt install systemd-journal-remote
```

On the receiving server:

```
bash
sudo systemctl enable --now systemd-journal-remote.socket
```

On each source server:

```
bash
sudo nano /etc/systemd/journal-upload.conf
ini
[Upload]
URL=https://logs.example.com:19532
bash
sudo systemctl enable --now systemd-journal-upload.service
```

This preserves journald's rich structured fields (unit name, PID, priority, and the many other fields journald captures natively) across the centralized log store, generally providing a richer investigative experience than flattened syslog forwarding, at the cost of being a systemd-specific mechanism rather than the more universally supported syslog protocol.

Log Retention and Storage Planning for a Centralized Log Host

A dedicated log server accumulating logs from multiple source systems requires its own storage planning, directly applying Chapter 6's storage-planning discipline: isolate log storage onto dedicated, appropriately sized storage (following the same reasoning as isolating `/var/log` locally in Chapter 1); establish retention policy appropriate to both operational usefulness and any compliance requirements; and ensure the log host itself is backed up (Chapter 6, Chapter 11) a log server holding the only copy of security-relevant historical data is itself a single point of failure worth protecting deliberately.

10.4 Performance Analysis and Bottleneck Identification

The Baseline-Measure-Change-Verify Methodology

This book's specification establishes a specific, disciplined performance methodology, and this section applies it concretely: establish a baseline,

measure the actual problem, identify the specific bottleneck, change the appropriate component, measure again, and verify genuine improvement. This structure deliberately resists the common, less disciplined alternative making speculative changes based on intuition and hoping performance improves which frequently wastes effort on changes that don't address the actual bottleneck, and which provides no reliable way to confirm whether a change genuinely helped.

Step 1: Establish a Baseline

Before any performance investigation begins, and ideally well before any problem occurs, establish what normal resource utilization and response characteristics look like for the workload in question, using the tools from section 10.2:

```
bash
```

```
# Example: baseline web application response time under normal conditions
for i in {1..20}; do
    curl -o /dev/null -s -w "%{time_total}\n" https://example.com/
done | awk '{sum+=$1; count++;} END {print "Average response time:", sum/count, "seconds"}'
```

Recording this figure (and the corresponding resource-snapshot output from section 10.2) during known-normal operation provides the concrete comparison point every subsequent investigation depends on.

Step 2: Measure the Actual Problem

When a performance issue is reported or suspected, measure it concretely rather than relying on impression:

```
bash
```

```
for i in {1..20}; do
    curl -o /dev/null -s -w "%{time_total}\n" https://example.com/
done | awk '{sum+=$1; count++;} END {print "Average response time:", sum/count, "seconds"}'
```

Comparing this measurement directly against the established baseline confirms whether a genuine regression exists, and quantifies its magnitude essential both for prioritizing the investigation appropriately and for later confirming that a fix actually worked.

Step 3: Identify the Specific Bottleneck

Work systematically through the resource categories from section 10.2, rather than guessing:

```
bash
uptime                # CPU/load pressure
free -h; vmstat 1 5    # memory pressure, swap activity
iostat -x 1 5          # disk I/O saturation
sudo ss -tan state established | wc -l  # connection volume
```

For a web/application-specific investigation, correlate with application-level logs (Chapter 5) and, if the application is containerized, with `docker stats` (Chapter 8):

```
bash
sudo journalctl -u myapp.service --since "10 minutes ago" | grep -i "slow\\|timeout\\|error"
```

A concrete example of this diagnostic sequence in practice: if `iostat -x` shows %util near 100% with elevated `await` specifically on the database's storage volume, while CPU and memory show ample headroom, the bottleneck is clearly disk I/O on the database directing the investigation and remediation specifically there (perhaps toward the database tuning discussed in Chapter 6, faster storage, or query optimization) rather than toward CPU or memory changes that would not address the actual constraint.

Step 4: Change the Appropriate Component

With the bottleneck specifically identified, apply a targeted change whether that's the database configuration tuning introduced in Chapter 6, an application-level fix (an unoptimized query, a missing index, an inefficient loop), a resource limit adjustment (Chapter 8's container resource limits, or a systemd service's own limits), or, where genuinely warranted, additional infrastructure capacity.

Step 5 and 6: Measure Again and Verify

Repeat the exact same measurement from step 2 after the change:

```
bash
```

```
for i in {1..20}; do
  curl -o /dev/null -s -w "%{time_total}\n" https://example.com/
done | awk '{sum+=$1; count++;} END {print "Average response time:", sum/count, "seconds"}'
```

Compare directly against both the problem measurement (confirming genuine improvement) and the original baseline (confirming the system has returned to, or improved beyond, normal operating characteristics) this final verification step is what distinguishes disciplined performance engineering from speculative tweaking, and is not optional: a change applied without confirmed measurement is, at best, an unverified hypothesis, not a resolved problem.

Common Bottleneck Patterns and Their Resolution Direction

Bottleneck Signature	Likely Constraint	Typical Resolution Direction
High CPU, low I/O wait, load > core count	Genuine CPU saturation	Optimize CPU-bound code, add CPU capacity, distribute load
Low CPU, high I/O wait	Disk I/O constrained	Faster storage, query/index optimization, caching
High swap activity (si / so)	Memory pressure	Add RAM, reduce memory consumption, tune application memory limits
High %util on a specific disk, elevated await	Storage subsystem saturated	Faster storage tier, workload distribution across disks, I/O-reducing caching
High established connection count, degraded response time	Application-layer concurrency limit	Increase worker/connection pool sizing, investigate slow queries holding connections
Elevated network %util , no CPU/disk pressure	Network bandwidth constrained	Bandwidth upgrade, compression, CDN/caching for static content

10.5 Capacity Planning and Server Optimization

From Reactive Performance Fixes to Proactive Capacity Planning

Where section 10.4 addresses performance problems as they arise, capacity planning uses the same monitoring infrastructure proactively anticipating resource exhaustion before it becomes a performance problem or outage at all, directly extending the trend-awareness theme introduced in section 10.1.

Trending Resource Utilization Over Time

The resource-snapshot script from section 10.2, run regularly and retained over weeks or months, provides the raw material for trend analysis. A

simple approach for extracting a specific trend from accumulated snapshots:

```
bash
```

```
grep -A1 "Memory" /var/log/resource-snapshot-*.log | grep "Mem:" | awk '{print $3}'
```

For more rigorous, visual trend analysis, dedicated time-series monitoring tools (Prometheus with Node Exporter being a common, widely adopted combination on Debian infrastructure) provide considerably more capability than manually parsing accumulated log snapshots appropriate to introduce once an environment's scale and criticality justify the additional operational complexity, a judgment call similar to the configuration-management adoption discussion in Chapter 9.

A Brief Introduction to Prometheus Node Exporter

For environments ready to move beyond snapshot-based monitoring, `node_exporter` exposes detailed system metrics in a format a Prometheus server can scrape and retain as genuine time-series data:

```
bash
```

```
cd /tmp
```

```
curl -LO https://github.com/prometheus/node_exporter/releases/download/v1.8.2/node_exporter-1.8.2.linux-amd64.tar.gz
```

```
tar xzf node_exporter-1.8.2.linux-amd64.tar.gz
```

```
sudo mv node_exporter-1.8.2.linux-amd64/node_exporter /usr/local/bin/
```

```
bash
```

```
sudo useradd --system --no-create-home --shell /usr/sbin/nologin node_exporter
```

```
ini
```

```
# /etc/systemd/system/node_exporter.service
```

```
[Unit]
```

```
Description=Prometheus Node Exporter
```

```
After=network.target
```

```
[Service]
```

```
User=node_exporter
```

```
Group=node_exporter
```

```
Type=simple
```

```
ExecStart=/usr/local/bin/node_exporter
```

```
[Install]
```

```
WantedBy=multi-user.target
```

```
bash
```

```
sudo systemctl daemon-reload
```

```
sudo systemctl enable --now node_exporter
```

```
curl http://127.0.0.1:9100/metrics | head -20
```

This exposes a detailed metrics endpoint (bound, per the least-exposure principle from Chapters 3 and 6, to loopback by default – a Prometheus server elsewhere would need explicit, deliberate network access configured, following the same firewall discipline established in Chapter 7) that a central Prometheus instance scrapes periodically, building genuine historical time-series data far more suitable for trend analysis, alerting threshold tuning, and capacity forecasting than manually parsed log snapshots. A full Prometheus/Grafana deployment is a substantial topic beyond this book's scope, but understanding this as the natural next step beyond the snapshot-based monitoring built in this chapter is valuable orientation for readers whose infrastructure will eventually outgrow simpler tooling.

Forecasting Capacity Needs

With reliable trend data in hand – whether from simple accumulated snapshots or a proper time-series system – capacity planning becomes a matter of extrapolation: if disk usage has grown consistently by a certain percentage per month, a straightforward projection indicates roughly when a given filesystem will reach its threshold (directly informing the LVM expansion procedures from Chapter 6, ideally executed well before space is actually exhausted); similarly for memory, CPU, and connection-count trends relative to current capacity.

Server Optimization Beyond Bottleneck Remediation

Beyond reactive bottleneck fixes, several general optimization practices apply broadly across most Debian server workloads:

Right-sizing resource allocations – for VMs, containers (Chapter 8), and systemd resource limits alike, allocations should reflect actual, measured usage patterns rather than conservative overestimates or, worse, unconstrained defaults that permit one workload to starve others.

Caching where appropriate – whether at the application layer, the web server layer (Nginx's caching directives, building on Chapter 5), or the database layer (query result caching), reducing redundant work is frequently a more effective optimization than simply adding raw resource capacity.

Periodic review of the performance baseline itself as an application or workload evolves, the "normal" baseline established in section 10.4 should be periodically re-established, since a baseline that is months or years stale may no longer accurately represent genuinely normal operation, undermining the very comparison this chapter's entire methodology depends on.

With comprehensive monitoring, logging, and a disciplined performance methodology in place, the server is now genuinely observable the essential precondition for the backup, recovery, and troubleshooting practices the book turns to next.

Chapter 11 : Backup, Recovery, and Troubleshooting

11.1 Designing a Reliable Debian Server Backup Strategy

Chapter 6 introduced backup fundamentals file and database backup mechanics, the 3-2-1 principle, and the central discipline of testing restoration. This chapter develops backup strategy into a complete, production-oriented practice, and extends it into full system recovery and a systematic troubleshooting methodology that draws on every diagnostic technique established across the preceding ten chapters.

Defining Recovery Objectives Before Choosing Tools

Effective backup strategy begins not with tool selection, but with two organizational questions that should be answered explicitly, in writing, before backup mechanics are designed:

Recovery Point Objective (RPO) how much data loss is acceptable in the event of a failure, expressed as a time duration. An RPO of 24 hours means losing up to a day's worth of data is acceptable; an RPO of 15 minutes demands vastly more frequent backup or replication than daily `pg_dump` runs could provide.

Recovery Time Objective (RTO) how quickly the system must be restored to operation after a failure. An RTO of a few hours permits a manual, careful restoration process; an RTO of minutes demands automated failover or standby infrastructure far beyond simple backup-and-restore.

These objectives, once defined, directly determine backup frequency, retention, and the specific mechanisms required a server with an RPO of 15 minutes cannot rely on nightly `pg_dump` alone, and needs streaming replication or frequent incremental backup instead; a server with a generous RTO of a day or more can tolerate a slower, more manual restoration

process without needing standby infrastructure. Most small-to-medium deployments this book addresses have RPO/RTO requirements comfortably served by the disciplined daily backup automation built in Chapter 9 but stating these objectives explicitly, rather than leaving them implicit and undefined, is what allows an administrator to confirm the backup strategy actually meets genuine business or operational requirements, rather than simply matching whatever happened to be convenient to implement.

The Complete Backup Inventory

A genuinely complete backup strategy accounts for every category of data and configuration a server's recovery depends on, extending Chapter 6's file/database/configuration categories into a full inventory:

- **Application data** database contents, user-uploaded files, any persistent application state
- **Configuration** `/etc/`, systemd unit files, Nginx/Apache virtual hosts, firewall rules, application `.env` files
- **Secrets** TLS private keys, SSH host keys, database credentials (backed up with particular care, given the sensitivity discussed in Chapter 7)
- **Infrastructure-as-code** Ansible playbooks, cloud-init templates, Docker Compose files (Chapter 9), ideally already durably stored in version control independent of the server itself
- **System state documentation** the baseline inventories from Chapter 7, package lists, and any manually maintained runbooks

Notably, infrastructure-as-code and documentation are the categories most often neglected in backup planning, precisely because they don't live on the server being backed up but a recovery that restores data and configuration onto a freshly reprovisioned server *requires* that provisioning material to exist and be accessible, making its own durability (via version control, ideally hosted independently of the infrastructure it describes) just as essential as the server's own backup.

Backup Retention Strategy

Extending Chapter 9's automated 7-day local retention, a more complete retention strategy typically layers multiple time horizons:

bash

```
# Retention tiers example, extending the automated-backup.sh pattern from Chapter 9
# Daily backups: retained 7 days
# Weekly backups (e.g., every Sunday): retained 8 weeks
# Monthly backups (e.g., 1st of month): retained 12 months
```

A practical implementation, extended from Chapter 9's script:

bash

```
sudo nano /opt/scripts/tiered-backup-retention.sh
```

bash

```
#!/bin/bash
```

```
set -euo pipefail
```

```
BACKUP_ROOT="/backups"
```

```
DAY_OF_WEEK=$(date +%u)    # 1 = Monday ... 7 = Sunday
```

```
DAY_OF_MONTH=$(date +%d)
```

```
# Daily retention: 7 days
```

```
find "$BACKUP_ROOT/daily" -maxdepth 1 -mtime +7 -type d -exec rm -rf {} \; 2>/dev/null || true
```

```
# Weekly retention: promote Sunday's backup, keep 8 weeks
```

```
if [[ "$DAY_OF_WEEK" == "7" ]]; then
```

```
    cp -al "$BACKUP_ROOT/daily/$(date +%Y%m%d)" "$BACKUP_ROOT/weekly/$(date +%Y%m%d)"
```

```
    find "$BACKUP_ROOT/weekly" -maxdepth 1 -mtime +56 -type d -exec rm -rf {} \; 2>/dev/null || true
```

```
fi
```

```
# Monthly retention: promote 1st-of-month backup, keep 12 months
```

```
if [[ "$DAY_OF_MONTH" == "01" ]]; then
```

```
    cp -al "$BACKUP_ROOT/daily/$(date +%Y%m%d)" "$BACKUP_ROOT/monthly/$(date +%Y%m%d)"
```

```
    find "$BACKUP_ROOT/monthly" -maxdepth 1 -mtime +366 -type d -exec rm -rf {} \; 2>/dev/null || true
```

```
fi
```

`cp -al` creates hard links rather than duplicating file content promoting a daily backup to weekly/monthly retention tiers without consuming additional disk space for identical data, an efficient pattern for tiered retention on a single backup volume. This tiered approach provides considerably more recovery flexibility than a single flat retention window the ability to recover a file or database state from three months ago, not merely from within the last week, at a modest storage cost.

11.2 File, Configuration, and Database Backups

Consolidating This Book's Backup Techniques

This section draws together the file, configuration, and database backup mechanics established individually in Chapter 6 and automated in Chapter 9, adding depth on selective restoration—recovering a single file or table without a full system restore, often the actual, more common recovery need in practice.

Selective File Restoration

Given `rsync`-based backups (Chapter 6), restoring a single file or directory is straightforward:

```
bash
```

```
rsync -avz backup-host:/backups/debian-server/etc/nginx/sites-available/example.com \
/tmp/restored-example.com.conf
```

Reviewing the restored file before reinstating it into active configuration (rather than blindly overwriting the current, possibly-since-modified version) is a worthwhile habit:

```
bash
```

```
diff /tmp/restored-example.com.conf /etc/nginx/sites-available/example.com
```

Selective Database Restoration

Restoring a single table from a full `pg_dump` archive, rather than the entire database, is a common and valuable capability:

```
bash
```

```
gunzip -c /backups/20260920/myapp_production.sql.gz > /tmp/full-restore.sql
```

For a plain-text SQL dump (as produced by the `pg_dump` commands used throughout this book), extracting a single table's data requires either `grep`-based extraction of the relevant `COPY` block (workable but fragile) or, preferably, using `pg_dump`'s own table-filtering capability at backup time to produce more selectively restorable output, or performing the extraction from a restored copy of the full database:

```
bash
```

```

sudo -u postgres createdb temp_restore_db
gunzip -c /backups/20260920/myapp_production.sql.gz | sudo -u postgres psql temp_restore_db
sudo -u postgres pg_dump --table=users temp_restore_db > /tmp/users_table_only.sql
sudo -u postgres psql myapp_production < /tmp/users_table_only.sql
sudo -u postgres dropdb temp_restore_db

```

This pattern restore the full backup into an isolated, temporary database, extract only the specifically needed data, apply it to the actual target, then discard the temporary copy is a safe, general-purpose technique for selective recovery that avoids ever risking the production database directly during the extraction process itself.

Configuration Restoration with Verification

Restoring configuration files should always follow the same validate-before-apply discipline established throughout this book (Chapter 5's `nginx -t`, Chapter 4's `sshd -t`):

```

bash
sudo cp /etc/nginx/sites-available/example.com /etc/nginx/sites-available/example.com.pre-restore
sudo cp /tmp/restored-example.com.conf /etc/nginx/sites-available/example.com
sudo nginx -t

```

If `nginx -t` reports an error, the pre-restore backup (`example.com.pre-restore`) provides an immediate rollback path restoring from backup should never bypass this book's established verify-before-apply discipline simply because the source is "known good" backup data; the target system's current state and the backup's compatibility with it still deserve confirmation.

Verifying Backup Completeness Proactively

Rather than only discovering a backup gap during an actual recovery attempt, periodically audit backup completeness directly:

```

bash
sudo nano /opt/scripts/backup-audit.sh
bash
#!/bin/bash
set -euo pipefail

BACKUP_DIR="/backups/daily/$(date +%Y%m%d)"
EXPECTED_FILES=("myapp_production.sql.gz" "etc-backup.tar.gz" "myapp-data.tar.gz")

```

```

ALERT_EMAIL="admin@example.com"
HOSTNAME=$(hostname -f)

MISSING=()

for f in "${EXPECTED_FILES[@]"; do
    if [[ ! -s "$BACKUP_DIR/$f" ]]; then
        MISSING+=("$f")
    fi
done

if [[ "${#MISSING[@]}" -gt 0 ]]; then
    echo "Missing or empty backup files: ${MISSING[*]}" | \
        mail -s "BACKUP AUDIT FAILURE on $HOSTNAME" "$ALERT_EMAIL"
    exit 1
fi

echo "Backup audit passed: all expected files present and non-empty"

```

This audit script, scheduled to run shortly after the nightly backup job completes (Chapter 9), provides an independent verification layer distinct from the backup script's own internal checks—a valuable defense against the specific failure mode of the backup script itself silently failing to run at all (a cron/timer misconfiguration, for instance), which an audit script checking for the *presence* of expected output would catch even if the backup process's own internal alerting somehow failed to fire.

11.3 System Recovery and Disaster Recovery Procedures

Distinguishing Recovery Scenarios

Different failure scenarios demand different recovery procedures, and recognizing which scenario is actually occurring is the first, essential step:

Data corruption or loss with the server otherwise intact the server boots and is reachable, but specific data (a database, a configuration file, an application directory) has been lost or corrupted. Recovery involves selective restoration (section 11.2) without needing to rebuild the server itself.

Service-level failure with the server otherwise intact a specific service has failed and needs its own recovery, addressed through the

troubleshooting methodology in section 11.5 rather than backup restoration at all.

Complete server loss the underlying VM, VPS instance, or physical hardware is unrecoverable, requiring full rebuild from provisioning automation (Chapter 9) plus complete data restoration.

Boot failure with data intact the server fails to boot, but its underlying storage remains accessible (via rescue mode, a live boot environment, or by attaching the disk to another system), requiring boot-specific troubleshooting rather than data recovery.

Full Server Rebuild Procedure

For complete server loss, the recovery procedure directly exercises the reproducibility work established in Chapter 9:

1. **Provision replacement infrastructure** a new VM, VPS instance, or physical server, following the same process as Chapter 1's original deployment.
2. **Apply baseline configuration** via the Ansible playbook, cloud-init configuration, or documented manual procedure established in Chapter 9, bringing the new server to the same security and service baseline as the original.
3. **Restore data** database restoration (section 11.2), application data restoration, and configuration restoration from the most recent verified backup.
4. **Verify service functionality** working systematically through each service's own verification procedure (Chapter 5's web server checks, Chapter 6's database connectivity checks), rather than assuming restoration alone constitutes success.
5. **Update DNS/routing** if the replacement server has a different IP address than the original, update DNS records (Chapter 3) and any firewall allow-lists elsewhere that reference the old address explicitly.
6. **Conduct a post-recovery review** documenting what caused the original loss, how long recovery took against the RTO established in section 11.1, and what, if anything, should change in the backup or provisioning process as a result.

Boot Failure Recovery

When a server fails to boot, diagnosis and recovery typically require console access (physical, virtual, or the out-of-band management established in Chapter 1) since, by definition, SSH is unavailable until boot succeeds.

Booting into rescue/single-user mode: at the GRUB menu (accessible via console), select the kernel entry, press `e` to edit boot parameters, and append `single` or `systemd.unit=rescue.target` to the kernel line, then boot with the modified parameters (typically `Ctrl+X` or `F10`) providing a minimal shell for investigation without starting the full range of normally-booted services.

Common boot failure causes and their investigation:

`bash`

`# From a rescue shell, check for a fstab-related boot failure`

`cat /etc/fstab`

`mount -a`

`# Check for filesystem corruption`

`fsck -n /dev/vg_debian-server/lv_root` `# -n: check only, do not repair`

Warning: Running `fsck` on a mounted filesystem, or without the `-n` (check-only) flag on a filesystem you have not first confirmed genuinely needs repair, risks further data damage. Always attempt a check-only pass first, understand what it reports, and only proceed to an actual repair (`fsck` without `-n`, on an unmounted filesystem) deliberately, ideally after backing up what remains accessible.

Recovering from a botched kernel or GRUB update: at the GRUB boot menu, select a previous, known-working kernel version from the boot menu's listed options (Debian retains prior kernel versions by default specifically to support this recovery path) as an immediate recovery step, then investigate and address the underlying update issue with the system successfully booted.

Disaster Recovery Planning and Documentation

Extending this chapter's procedures into a genuinely actionable disaster recovery plan means documenting, in advance and independent of the

server itself (since the server may be exactly what's unavailable during an actual disaster), the specific, concrete steps for this server's recovery: where backups are stored and how to access them, where provisioning automation (Ansible playbooks, cloud-init templates) is version-controlled and how to access that repository, the specific RTO/RPO objectives established in section 11.1, and named responsibility for executing recovery. A disaster recovery plan that exists only as implicit administrator knowledge is not a plan at all – it is exactly the kind of undocumented, irreplaceable "pet" infrastructure the Introduction warned against, and its absence is discovered at precisely the worst possible moment.

11.4 Troubleshooting Boot, Network, Storage, and Service Failures

This section applies the systematic troubleshooting model formalized in 11.5 to the specific failure categories most common in Debian server operation, consolidating and extending diagnostic techniques introduced throughout the book.

Boot Failures

Beyond the rescue-mode procedures in 11.3, common boot-time failure signatures and their diagnosis:

bash

From a successfully booted system, review the previous boot's logs

`journalctl -b -1 -p err`

`journalctl -b -1 --no-pager | tail -100`

`journalctl -b -1` shows logs from the previous boot – invaluable for understanding what happened immediately before an unexpected reboot or crash, reviewed from a subsequent successful boot.

Network Failures

Chapter 3 established a complete layered network diagnostic methodology (link, IP, routing, connectivity, DNS, port-level). In a troubleshooting context specifically, the key addition is correlating network symptoms with recent changes:

bash

```
# Check for recent network configuration changes
ls -la /etc/network/interfaces.d/ /etc/systemd/network/ 2>/dev/null
sudo journalctl -u networking -u systemd-networkd --since "1 hour ago"
```

Combined with Chapter 7's firewall troubleshooting (verifying `nft list ruleset` against expected rules, and checking for unexpected Docker-injected chains per Chapter 8), this covers the great majority of network-layer failure investigation.

Storage Failures

Beyond space exhaustion (addressed thoroughly in Chapter 6), storage-layer failures include device errors and filesystem issues:

bash

```
sudo journalctl -k | grep -i -E "error|fail" | grep -i -E "sd[a-z]|nvme"
sudo smartctl -a /dev/sda # requires: sudo apt install smartmontools
```

`smartctl` reads a disk's SMART (Self-Monitoring, Analysis, and Reporting Technology) data, often providing early warning of impending physical disk failure well before it becomes a full outage worth checking whenever storage-related errors appear in kernel logs, and worth monitoring proactively (extending section 10.2's monitoring discipline) on servers with physical, rather than cloud-provider-abstracted, storage.

Service Failures

Chapter 2 established the core service-troubleshooting sequence (`systemctl status` , `journalctl -u`); this remains the correct starting point for any service failure:

bash

```
sudo systemctl status myapp.service
sudo journalctl -u myapp.service --since "30 minutes ago"
sudo journalctl -u myapp.service -p err
```

Where a service depends on another (database connectivity, for instance, per Chapter 2's dependency directives), confirm the dependency's own health before assuming the fault lies in the dependent service itself:

bash

```
sudo systemctl status postgresql
```

```
sudo -u postgres psql -c "SELECT 1;"
```

11.5 Building a Structured Linux Troubleshooting Methodology

The Problem → Symptoms → Evidence → Investigation → Diagnosis → Remediation → Verification Model

This book's specification establishes a specific, disciplined troubleshooting model, and this closing section makes it explicit and complete, drawing together the diagnostic instincts built implicitly throughout every preceding chapter.

Problem a clear, specific statement of what is actually wrong, distinct from symptoms. "The website is down" is a symptom description; "HTTP requests to example.com are returning 502 errors" is closer to an actual problem statement, and "the Unicorn application process behind Nginx is not running" is closer still, once investigation has progressed.

Symptoms the observable, reported effects: error messages, unexpected behavior, alerts fired, user reports. Symptoms are the starting point of an investigation, not its conclusion resist the temptation to jump directly from a symptom to a fix without passing through evidence and diagnosis.

Evidence the concrete, factual data gathered through the diagnostic tools this book has built throughout: `journalctl` output, `systemctl status`, resource utilization figures, network diagnostic results, log file contents. Evidence gathering should happen *before* making changes to the system a change made before evidence is gathered risks destroying the very evidence needed to understand what actually happened, and risks "fixing" a coincidental symptom while leaving the actual root cause unaddressed.

Investigation the systematic process of working through possible causes, informed by the evidence, typically proceeding from broad to specific (following the layered approach established in Chapter 3's network diagnostics and echoed throughout this chapter's failure-category coverage) confirming or ruling out each layer before assuming a fault lies deeper.

Diagnosis a specific, evidence-supported conclusion about the actual root cause, distinct from a guess or a plausible-sounding hypothesis not yet confirmed by evidence.

Remediation the actual fix, applied deliberately, following this book's established safety disciplines: back up or snapshot before a risky change, validate configuration syntax before applying (`nginx -t` , `sshd -t` , `nft -c -f`), and apply during an appropriate window with rollback capability in mind.

Verification explicit confirmation that the remediation actually resolved the original problem, using the same measurement or check that identified the symptom in the first place directly mirroring the baseline/measure/change/verify performance methodology from Chapter 10, applied to troubleshooting generally rather than performance specifically.

A Worked Example

Applying this model to a realistic scenario, tying together tools and reasoning from across the book:

Problem statement: Users report the application at `app.example.com` is intermittently returning 502 errors.

Symptoms: Intermittent 502 Bad Gateway responses, reported by users over roughly the past hour, no corresponding alert yet configured (a gap this investigation should note for remediation per Chapter 10's monitoring practice).

Evidence gathering (per Chapter 5's diagnostic sequence and Chapter 10's monitoring tools):

```
bash
sudo tail -100 /var/log/nginx/app.example.com.error.log
sudo journalctl -u myapp.service --since "1 hour ago" -p err
sudo systemctl status myapp.service
free -h
iostat -x 1 3
```

Suppose this evidence reveals: the Nginx error log shows repeated "upstream prematurely closed connection" entries; `journalctl -u myapp.service`

shows the application process has restarted several times in the past hour (visible via systemd's own restart logging, per Chapter 2's `Restart=on-failure` behavior); and `free -h` shows memory usage consistently near capacity, with `vmstat` showing active swap activity.

Investigation: The pattern intermittent 502s correlating with application restarts, correlating with memory pressure points toward the application being killed (likely by the kernel's OOM killer under memory pressure) and restarted by systemd's `Restart=on-failure` policy (Chapter 2), with each restart causing a brief window of 502s while Nginx's upstream is unavailable. Confirm this specific hypothesis:

```
bash
sudo journalctl -k | grep -i "out of memory\|oom"
```

Diagnosis: The application is being killed by the kernel OOM killer due to memory pressure, triggering systemd's automatic restart (masking the underlying problem as brief, recurring outages rather than a single clear failure) the memory-limit configuration established in Chapter 2's systemd service hardening (`MemoryMax=`, if configured) or the container memory limits from Chapter 8 (if the application were containerized) are either absent or set too high relative to actual available system memory, and either the application has a memory leak, or genuine memory demand has grown beyond current server capacity (directly connecting to Chapter 10's capacity planning discipline).

Remediation: Depending on further investigation into whether this reflects a genuine leak (requiring an application-level fix) or legitimate growth (requiring either a memory limit adjustment paired with additional server memory, or query/caching optimization to reduce demand) the specific fix follows from the diagnosis, not from the initial symptom.

Verification: After remediation, confirm both the immediate symptom (502 errors no longer occurring) and the underlying condition (memory pressure resolved, via the same `free -h` / `vmstat` checks used during evidence gathering) closing the loop completely, rather than assuming the absence of immediate complaints constitutes verification.

Building Troubleshooting Habits That Compound

The specific commands and failure categories in this chapter, and throughout this book, will inevitably need updating as software evolves but the methodology itself, and the underlying discipline of gathering evidence before acting, verifying rather than assuming, and maintaining the documentation and backup practices that make recovery possible at all, is durable, transferable, and precisely what separates an administrator who merely operates Debian servers from one who can be trusted to operate them reliably under real, production pressure. This is the practical foundation the book's final chapter now applies to complete, integrated production projects.

Chapter 12 : Production Debian Server Projects

12.1 Building a Secure Web Server

This chapter integrates the entire book's disciplines into complete, realistic projects each drawing on multiple prior chapters rather than isolated demonstration. This first project builds a fully hardened, production-ready Debian 13.7 web server from scratch, following the sequence an experienced administrator would actually use.

Project Requirements

A public-facing web server for `example.com`, serving a static marketing site with an accompanying blog, requiring: HTTPS with automatic certificate renewal, a hardened security baseline, firewall protection, and monitoring sufficient to detect problems proactively.

Architecture

Single Debian 13.7 server (hostname `debian-server`, consistent with this book's running example), Nginx serving static content directly, Let's Encrypt-issued TLS, nftables firewall, fail2ban intrusion prevention, and journald-based logging with the security monitoring script from Chapter 7.

Implementation Sequence

Step 1 Base installation and initial configuration (Chapter 1): Debian 13.7 installed with LVM-based partitioning isolating `/var/log`, static IP configured, hostname set, time synchronized, all packages current.

Step 2 Administrative access and account hardening (Chapters 2, 4): dedicated `adminuser` account created with scoped `sudo`; SSH key authentication configured and verified; password authentication and root login disabled; `fail2ban` installed and configured for the `sshd` jail.

bash

```
sudo adduser adminuser
sudo usermod -aG sudo adminuser
ssh-copy-id -i ~/.ssh/id_ed25519_debian_server.pub adminuser@debian-server
# Verify key auth works in a fresh session before proceeding
sudo sed -i 's/^#?PasswordAuthentication.*/PasswordAuthentication no/' /etc/ssh/sshd_config
sudo sed -i 's/^#?PermitRootLogin.*/PermitRootLogin no/' /etc/ssh/sshd_config
sudo sshd -t && sudo systemctl reload ssh
```

Step 3 Firewall baseline (Chapter 7): default-drop nftables ruleset permitting only SSH, HTTP, and HTTPS, applied using the safety-reboot discipline established in section 7.2.

```
bash
sudo apt install nftables
sudo nft -c -f /etc/nftables.conf
sudo shutdown -r +5 "Firewall safety reboot"
sudo nft -f /etc/nftables.conf
# Verify SSH access from a separate session, then:
sudo shutdown -c
```

Step 4 Web server deployment (Chapter 5): Nginx installed, site configured with the static-asset caching pattern established previously.

```
bash
sudo apt install nginx
sudo mkdir -p /var/www/example.com/html
sudo ln -s /etc/nginx/sites-available/example.com /etc/nginx/sites-enabled/
sudo nginx -t && sudo systemctl reload nginx
```

Step 5 TLS via Let's Encrypt (Chapter 5):

```
bash
sudo apt install certbot python3-certbot-nginx
sudo certbot --nginx -d example.com -d www.example.com
sudo certbot renew --dry-run
```

Step 6 Security auditing and monitoring (Chapter 7, Chapter 10): Lynis baseline audit, AIDE integrity baseline, and the security-check script scheduled via systemd timer.

```
bash
sudo apt install lynis aide aide-common
sudo lynis audit system
sudo aideinit && sudo mv /var/lib/aide/aide.db.new /var/lib/aide/aide.db
```

Testing and Verification

```
bash
curl -Iv https://example.com/
sudo nmap -Pn example.com          # from a separate host
sudo fail2ban-client status sshd
sudo systemctl status nginx nftables fail2ban
```

Confirm HTTPS redirect, correct certificate chain, only the intended ports reachable externally, and all baseline services active directly applying the verification discipline established throughout the book rather than assuming success from configuration alone.

12.2 Deploying a Database-Backed Application Server

Project Requirements

An internal application server hosting a Django-based application with a PostgreSQL backend, accessible only via an internal reverse proxy, with the application and database strictly isolated from public exposure.

Architecture

Gunicorn application server (Chapter 5) bound to loopback, PostgreSQL (Chapter 6) bound to loopback with scoped user privileges, Nginx reverse proxy (Chapter 5) handling public TLS termination, secrets managed via a restricted-permission environment file (Chapter 5, Chapter 7).

Implementation Sequence

Step 1 Database deployment (Chapter 6):

```
bash
sudo apt install postgresql
sudo -u postgres psql -c "CREATE DATABASE myapp_production;"
sudo -u postgres psql -c "CREATE USER myapp WITH PASSWORD 'generated-strong-password';"
sudo -u postgres psql -c "GRANT CONNECT ON DATABASE myapp_production TO myapp;"
```

Step 2 Application deployment as a managed service (Chapters 2, 5):

```
bash
sudo adduser --system --group --no-create-home --home /opt/myapp myappuser
sudo mkdir -p /opt/myapp
sudo -u myappuser python3 -m venv /opt/myapp/venv
```

```

sudo -u myappuser /opt/myapp/venv/bin/pip install gunicorn django psycopg2-binary
bash
sudo nano /opt/myapp/.env
DATABASE_URL=postgresql://myapp:generated-strong-
password@127.0.0.1:5432/myapp_production
SECRET_KEY=generated-secret-value
bash
sudo chown myappuser:myappuser /opt/myapp/.env
sudo chmod 600 /opt/myapp/.env
ini
# /etc/systemd/system/myapp.service
[Unit]
Description=Gunicorn instance for myapp
After=network.target postgresql.service
Requires=postgresql.service

[Service]
Type=simple
User=myappuser
Group=myappuser
WorkingDirectory=/opt/myapp
EnvironmentFile=/opt/myapp/.env
ExecStart=/opt/myapp/venv/bin/gunicorn --workers 3 --bind 127.0.0.1:8000 myapp.wsgi:application
Restart=on-failure
RestartSec=5
NoNewPrivileges=true
PrivateTmp=true

[Install]
WantedBy=multi-user.target
bash
sudo systemctl daemon-reload
sudo systemctl enable --now myapp.service

```

Step 3 Reverse proxy and TLS (Chapter 5): Nginx configured with proxy_pass http://127.0.0.1:8000 , secured via Certbot exactly as in 12.1.

Step 4 Firewall and network verification (Chapters 3, 7):

```

bash
sudo ss -tulnp | grep -E '5432|8000'    # confirm both bound to loopback only
sudo nft list ruleset                  # confirm only 22/80/443 permitted externally

```

Backup Integration

Applying Chapter 9's automated backup script directly to this deployment, scheduled nightly via systemd timer, with the audit script from Chapter 11

confirming completeness each morning.

12.3 Building a Containerized Debian Application Environment

Project Requirements

The same application architecture as 12.2, reimplemented as a containerized multi-service stack, demonstrating the containerized alternative established in Chapter 8 for an equivalent workload.

Implementation

Directly applying Chapter 8's multi-container Compose pattern:

```
bash
```

```
sudo apt install docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

```
sudo usermod -aG docker adminuser
```

```
bash
```

```
mkdir -p /opt/myapp-stack/secrets
```

```
echo -n "generated-strong-password" > /opt/myapp-stack/secrets/db_password.txt
```

```
chmod 600 /opt/myapp-stack/secrets/db_password.txt
```

Using the `docker-compose.yml` structure established in Chapter 8, with `db`, `app`, and `nginx` services on an isolated internal network, only `nginx` published to loopback:

```
bash
```

```
cd /opt/myapp-stack
```

```
docker compose up -d
```

```
docker compose ps
```

Firewall Integration for the Containerized Stack

Applying Chapter 8's guidance on Docker's nftables interaction, verify actual external exposure explicitly rather than assuming the host firewall configuration alone describes reachability:

```
bash
```

```
sudo nft list ruleset | grep -A 5 DOCKER
```

```
sudo nmap -Pn debian-server.example.com # from an external host
```

Container-Specific Monitoring and Maintenance

```
bash
docker stats
docker compose logs --tail=100 -f
docker compose pull && docker compose up -d    # applying image updates per Chapter 8
```

12.4 Implementing Monitoring, Backups, and Automated Maintenance

This section integrates Chapters 9, 10, and 11's disciplines across whichever of the preceding projects (12.1, 12.2, or 12.3) is in use, establishing the complete operational layer a production deployment requires beyond initial build.

Consolidated Automation Schedule

```
bash
systemctl list-timers
```

A complete production server's scheduled automation, drawing together this book's automation chapters:

Task	Schedule	Script/Mechanism	Source Chapter
Security updates	Automatic, ongoing	unattended-upgrades	Chapter 2, 7
Full backup	Daily, 01:30	automated-backup.sh	Chapter 9
Backup audit	Daily, 03:00	backup-audit.sh	Chapter 11
Tiered retention	Daily, 03:15	tiered-backup-retention.sh	Chapter 11
Security check	Daily, 02:15	security-check.sh	Chapter 7
Resource snapshot	Hourly	resource-snapshot.sh	Chapter 10
Routine maintenance	Weekly	routine-maintenance.sh	Chapter 9
AIDE integrity check	Daily	aide --check	Chapter 7
Certificate renewal check	Twice daily	certbot renew (installed timer)	Chapter 5

Each entry implemented as a systemd timer, following the consistent pattern established throughout Chapter 9.

End-to-End Monitoring Verification

```
bash
journalctl -u automated-backup.service --since "24 hours ago"
journalctl -u security-check.service --since "24 hours ago"
```

```
cat /var/log/resource-snapshot-*.log | tail -50
```

Restoration Testing as a Scheduled Practice

Applying Chapter 6 and Chapter 11's core principle directly: schedule an actual, periodic restoration test, not merely backup creation:

```
bash
sudo nano /opt/scripts/quarterly-restore-test.sh
bash
#!/bin/bash
set -euo pipefail

LOG_FILE="/var/log/restore-test.log"
LATEST_BACKUP=$(ls -lt /backups/daily/*/myapp_production.sql.gz | head -1)

echo "[$(date)] Testing restore from $LATEST_BACKUP" >> "$LOG_FILE"

sudo -u postgres createdb restore_test_db
gunzip -c "$LATEST_BACKUP" | sudo -u postgres psql restore_test_db >> "$LOG_FILE" 2>&1

ROW_COUNT=$(sudo -u postgres psql -t -c "SELECT count(*) FROM users;" restore_test_db)
echo "[$(date)] Restored user count: $ROW_COUNT" >> "$LOG_FILE"

sudo -u postgres dropdb restore_test_db
echo "[$(date)] Restore test completed successfully" >> "$LOG_FILE"
```

Scheduled quarterly via systemd timer, this converts the book's repeatedly emphasized "test your backups" principle from aspiration into enforced, automated practice.

12.5 Performing a Complete Production Server Review

The Purpose of a Structured Review

Beyond individual monitoring and automation, a periodic, comprehensive review—walking the entire server against this book's cumulative practices—catches drift and gaps that piecemeal, task-specific checks might miss. This closing section provides that structured review as a complete, reusable checklist, organized by the book's own chapter progression.

Installation and System Baseline (Chapters 1–2)

```
bash
```

```
uname -r; lsb_release -a
df -h; sudo vgs && sudo lvs
sudo apt list --upgradable
systemctl list-units --type=service --state=failed
```

Confirm: system current on updates, no failed services, storage within healthy thresholds, LVM structure matches documented design.

Networking and Remote Access (Chapters 3–4)

bash

```
ip addr show; ip route show
sudo ss -tulnp
sudo sshd -t
sudo fail2ban-client status sshd
```

Confirm: static addressing correct, only intended services listening, SSH configuration valid, fail2ban active with reasonable ban activity (not zero, which might indicate the jail isn't actually engaging; not excessive, which might indicate a targeted, ongoing attack worth deeper investigation).

Web, Application, and Database Layers (Chapters 5–6)

bash

```
sudo nginx -t
curl -Iv https://example.com/
sudo certbot certificates
sudo -u postgres psql -c "\l"
sudo ss -tulnp | grep -E '5432|8000'
```

Confirm: TLS certificates valid with meaningful remaining validity, application and database services healthy and correctly bound, no unexpected public exposure of internal services.

Security Posture (Chapter 7)

bash

```
sudo nft list ruleset
sudo lynis audit system --quick
sudo debsecan --suite trixie
sudo aide --check
getent group sudo
```

Confirm: firewall ruleset matches documented, minimal intent; no unresolved known vulnerabilities in installed packages; no unexpected file

integrity changes; `sudo` group membership matches expected administrative accounts exactly.

Containers, if Applicable (Chapter 8)

```
bash
docker compose ps
docker images
docker system df
```

Confirm: all containers healthy and current on images; no accumulated unused images/volumes consuming unnecessary storage.

Automation and Configuration Management (Chapter 9)

```
bash
systemctl list-timers
journalctl -u automated-backup.service --since "7 days ago" | grep -i error
```

Confirm: every scheduled task from the table in section 12.4 has executed successfully within its expected window, with no unaddressed errors in recent logs.

Monitoring and Performance (Chapter 10)

```
bash
uptime
free -h
iostat -x 1 3
```

Confirm current resource utilization against the established baseline (section 10.4), investigating any meaningful deviation using this chapter's own methodology before it becomes an active incident.

Backup and Recovery Readiness (Chapter 11)

```
bash
cat /var/log/backup-audit.log 2>/dev/null | tail -20
cat /var/log/restore-test.log 2>/dev/null | tail -20
ls -la /backups/daily/ /backups/weekly/ /backups/monthly/
```

Confirm: backups completing and passing audit consistently; most recent restoration test succeeded within the expected quarterly window; retention tiers populated as designed.

Documentation Currency

Finally and consistent with the Introduction's foundational emphasis confirm that the runbook, disaster recovery plan (section 11.3), and infrastructure-as-code repository (Chapter 9) accurately reflect the server's actual current state, updating any documentation that has drifted from reality during the course of ordinary operation. A review that confirms the server's technical health while leaving its documentation stale has only completed half its purpose.

This structured review, performed on a regular cadence (monthly is a reasonable baseline for most production deployments; more frequently for higher-stakes infrastructure), is the practical synthesis of everything this book has built not a new skill, but the disciplined, systematic application of every prior chapter's individual practices, together, as a single coherent operational habit.

Conclusion : Operating Reliable Debian Servers

From Initial Deployment to Production Operations

This book began with a single question: what separates knowing Linux from being able to operate Debian server infrastructure that other people and systems genuinely depend on? Twelve chapters later, the answer has taken concrete shape, not as a single technique but as an accumulated, integrated practice. A server installed with deliberate partitioning and a validated baseline (Chapter 1) becomes a properly administered system through disciplined user, package, and service management (Chapter 2). That system becomes reachable and safely so through carefully designed networking (Chapter 3) and hardened remote access (Chapter 4). It becomes useful by hosting web and application services (Chapter 5) and the data layer beneath them (Chapter 6) each deployed with the same recurring

discipline: configure deliberately, explain the important directives, validate before applying, verify after applying, and understand what can go wrong.

None of these chapters stood alone. The loopback-binding discipline introduced for databases in Chapter 6 reappeared identically for containerized services in Chapter 8. The validate-before-reload pattern established for SSH in Chapter 4 reappeared for Nginx in Chapter 5, for nftables in Chapter 7, and for Ansible playbooks in Chapter 9. This repetition was deliberate, not accidental—it reflects how these patterns actually function in real operational practice: not as chapter-specific techniques, but as a small number of durable habits, applied consistently across every layer of a server's operation.

Maintaining Security and Reliability

Security, as Chapter 7 established explicitly, is never a single completed configuration—it is a layered, ongoing practice: minimized attack surface, firewalled and access-restricted networking, hardened accounts and privilege boundaries, current patches verified against actual vulnerability data, and detection mechanisms watching for what prevention inevitably misses. Reliability follows the same layered logic: a server that starts its services correctly at boot (Chapter 2), recovers automatically from transient failures (systemd's `Restart=` directives, echoed at the container level in Chapter 8), and is continuously observed against a known baseline (Chapter 10) is a server whose failures are caught early and understood quickly, rather than discovered only when a user complains.

The throughline connecting security and reliability across this entire book is verification. Nearly every procedure presented—from Chapter 1's post-installation validation checklist through Chapter 12's complete production review—insisted on confirming that a change actually achieved its intended effect, rather than assuming success from the absence of an error message. This is not a stylistic preference; it is the practical difference between infrastructure an administrator can trust under pressure and infrastructure that merely appears to work until the moment it doesn't.

Automating Repetitive Administration

Chapter 9's automation work, and its extensions throughout Chapters 10, 11, and 12, exist to make everything else in this book sustainable at scale and over time. A single server can be maintained manually by a careful administrator; a growing environment, or a single administrator's own limited time and attention, cannot indefinitely sustain purely manual operation. The shell scripting discipline established early in Chapter 9 — `set -euo pipefail`, explicit logging, deliberate error handling — and its extension into scheduled execution via `cron` and `systemd` timers, converted this book's individual procedures into durable, unattended, self-verifying operations: backups that alert on their own failure, security checks that run without being remembered, and maintenance that happens on schedule rather than whenever an administrator finds time.

The brief introduction to configuration management in section 9.4, and to reproducible provisioning in section 9.5, pointed toward where this discipline naturally leads as infrastructure grows: from scripted automation of individual tasks toward declarative, version-controlled descriptions of a server's entire desired state — the foundation that made Chapter 11's disaster recovery procedures genuinely tractable, rather than dependent on a single administrator's irreplaceable memory of exactly how a server was configured.

Developing Long-Term Server Management Practices

The specific commands, package names, and configuration syntax in this book are accurate for Debian 13.7 as of this writing, and will, in the ordinary course of software evolution, eventually need updating. The practices underlying them will not. Least privilege, defense in depth, verify before trusting, document what cannot be easily reconstructed, test your backups before you need them, gather evidence before changing a broken system — these are not Debian-specific truths, and they will remain true long after this book's specific `nginx.conf` syntax has been superseded by whatever comes next. Readers who internalize the methodology — Chapter 11's Problem-Symptoms-Evidence-Investigation-Diagnosis-Remediation-Verification model, Chapter 10's baseline-measure-change-verify performance discipline, Chapter 7's layered security thinking — carry

something considerably more durable than command syntax: a way of approaching infrastructure that transfers across tool changes, version upgrades, and even entirely different operating systems.

Continuing Your Debian and Linux Administration Journey

This book has deliberately stayed within the scope of a single server, or a small number of closely related servers—the realistic starting point for the great majority of Debian deployments, and the level at which the book's core disciplines are best learned directly and concretely. Real infrastructure, as it grows, extends naturally beyond this scope: configuration management maturing from the brief Ansible introduction in Chapter 9 into fully managed fleets; monitoring maturing from the snapshot-based tooling in Chapter 10 into genuine time-series platforms like Prometheus and Grafana; container orchestration extending beyond Chapter 8's Docker Compose into Kubernetes for environments genuinely requiring that additional complexity; and high-availability architectures addressing the availability requirements briefly noted but not built out in this book's single-server focus.

None of that growth is possible on an unstable foundation, and that foundation—a properly installed, deliberately configured, secured, monitored, automated, and recoverable Debian 13.7 server—is precisely what this book has built, chapter by chapter, from the first `apt update` in Chapter 1 to the complete production review in Chapter 12. The reader who has worked through this book, built its scripts, applied its configurations, and internalized its verification habits is no longer simply someone who knows Linux commands. They are someone equipped to operate real Debian server infrastructure—deliberately, securely, and reliably—and ready to build on that foundation as their own infrastructure, and their own responsibilities, continue to grow.
