

# Ubuntu Academy: Using Ubuntu Server

Practical Administration for Production & Cloud Environments

**Institut Teknologi Tangerang Selatan (ITTS)  
OnnoCenter  
2026**

# TABLE OF CONTENT

|                                                                     |           |
|---------------------------------------------------------------------|-----------|
| <b>TABLE OF CONTENT.....</b>                                        | <b>2</b>  |
| <b>PREFACE.....</b>                                                 | <b>19</b> |
| <b>PART I — FOUNDATIONS OF UBUNTU SERVER.....</b>                   | <b>20</b> |
| Chapter 1 — Understanding Ubuntu Server Architecture.....           | 20        |
| Introduction: Beyond “Just Another Linux”.....                      | 20        |
| <b>Ubuntu Desktop vs Ubuntu Server.....</b>                         | <b>20</b> |
| Ubuntu Desktop.....                                                 | 20        |
| Ubuntu Server.....                                                  | 21        |
| Why Minimal Matters.....                                            | 21        |
| <b>LTS Lifecycle &amp; Extended Security Maintenance (ESM).....</b> | <b>22</b> |
| What is LTS?.....                                                   | 22        |
| Extended Security Maintenance (ESM).....                            | 22        |
| <b>The Package Ecosystem: deb, snap, and Container Images.....</b>  | <b>22</b> |
| 1. deb Packages (APT).....                                          | 22        |
| 2. Snap Packages.....                                               | 23        |
| 3. Container Images.....                                            | 23        |
| <b>systemd as PID 1 — The Modern Init System.....</b>               | <b>24</b> |
| Why systemd Matters.....                                            | 24        |
| Conceptual Illustration.....                                        | 24        |
| <b>Ubuntu Server in Modern Infrastructure.....</b>                  | <b>25</b> |
| 1. Ubuntu Server in the Cloud.....                                  | 25        |
| 2. Ubuntu at the Edge.....                                          | 25        |
| 3. Ubuntu in Containers.....                                        | 26        |
| 4. Ubuntu as a Hypervisor Host.....                                 | 26        |
| <b>The Cloud-First and Immutable Direction.....</b>                 | <b>26</b> |
| <b>Why This Matters for You.....</b>                                | <b>27</b> |
| <b>Chapter Summary.....</b>                                         | <b>28</b> |
| <b>Reflection Challenge.....</b>                                    | <b>28</b> |
| Chapter 2 — Deploying Ubuntu Server (Exam Domain ~27%).....         | 29        |
| 2.1 Installation Fundamentals.....                                  | 29        |
| Introduction: Installation Is Architecture.....                     | 29        |
| The Subiquity Installer.....                                        | 29        |
| The Modern Ubuntu Server Installer.....                             | 29        |
| Why Subiquity Matters.....                                          | 30        |
| Installation Flow Overview.....                                     | 30        |
| <b>Guided vs Manual Partitioning.....</b>                           | <b>30</b> |
| Guided Partitioning.....                                            | 31        |
| Simplicity and Speed.....                                           | 31        |
| Why LVM Is Often Recommended.....                                   | 31        |
| Example Guided LVM Layout.....                                      | 31        |
| Manual Partitioning.....                                            | 32        |
| Precision and Control.....                                          | 32        |

|                                                       |           |
|-------------------------------------------------------|-----------|
| When Manual Partitioning Is Critical.....             | 32        |
| Practical Comparison.....                             | 33        |
| Best Practice Recommendation.....                     | 33        |
| <b>Secure Boot &amp; UEFI.....</b>                    | <b>33</b> |
| What Is UEFI?.....                                    | 34        |
| What Is a Secure Boot?.....                           | 34        |
| Illustration: Boot Chain Trust.....                   | 34        |
| When Should Secure Boot Be Used?.....                 | 35        |
| <b>GPT and Disk Partition Tables.....</b>             | <b>35</b> |
| <b>Installation Strategy: Production Mindset.....</b> | <b>35</b> |
| <b>Encryption Considerations.....</b>                 | <b>36</b> |
| <b>Practical Installation Scenario.....</b>           | <b>36</b> |
| Scenario 1 — Cloud Web Server.....                    | 36        |
| Scenario 2 — On-Prem Database Server.....             | 36        |
| <b>Installation as a Learning Experience.....</b>     | <b>37</b> |
| <b>Chapter Summary.....</b>                           | <b>37</b> |
| <b>Reflection Challenge.....</b>                      | <b>37</b> |
| 2.2 Advanced Deployment.....                          | 39        |
| Introduction: From Installation to Automation.....    | 39        |
| <b>Autoinstall YAML.....</b>                          | <b>39</b> |
| Cloud-Init Style Automated Installation.....          | 39        |
| Why Autoinstall Matters.....                          | 40        |
| Basic Autoinstall Structure.....                      | 40        |
| Autoinstall Deployment Methods.....                   | 41        |
| Illustration: Automated vs Manual Installation.....   | 41        |
| <b>PXE Boot Deployment.....</b>                       | <b>41</b> |
| Network-Based Installation at Scale.....              | 41        |
| How PXE Works.....                                    | 41        |
| Why PXE Is Powerful.....                              | 42        |
| When to Use PXE.....                                  | 42        |
| <b>Cloud Image Deployment.....</b>                    | <b>43</b> |
| Ubuntu in Public Cloud Environments.....              | 43        |
| What Is a Cloud Image?.....                           | 43        |
| Cloud-Init: The First Boot Engine.....                | 43        |
| Illustration: Cloud Deployment Model.....             | 44        |
| <b>ARM64 vs x86_64 Considerations.....</b>            | <b>44</b> |
| x86_64.....                                           | 44        |
| ARM64.....                                            | 45        |
| Architecture Comparison.....                          | 45        |
| Practical Decision Guidance.....                      | 45        |
| <b>Modern Deployment Mindset.....</b>                 | <b>46</b> |
| <b>Chapter Summary.....</b>                           | <b>46</b> |
| <b>Reflection Challenge.....</b>                      | <b>47</b> |
| 2.3 Post-Install Configuration.....                   | 48        |

|                                                             |           |
|-------------------------------------------------------------|-----------|
| Introduction: Installation Is Only the Beginning.....       | 48        |
| <b>Initial User &amp; SSH Setup.....</b>                    | <b>48</b> |
| Securing Remote Access from Day One.....                    | 48        |
| Understanding User Roles.....                               | 48        |
| Creating Additional Users.....                              | 49        |
| SSH Key-Based Authentication.....                           | 49        |
| Illustration: Secure SSH Model.....                         | 50        |
| <b>Updating the System Safely.....</b>                      | <b>50</b> |
| Stability Before Speed.....                                 | 50        |
| Basic Update Process.....                                   | 50        |
| Best Practice: Safe Updating Workflow.....                  | 50        |
| Unattended Security Updates.....                            | 51        |
| <b>Kernel Management.....</b>                               | <b>51</b> |
| Understanding the Core of the System.....                   | 51        |
| Checking Kernel Version.....                                | 51        |
| When Kernel Updates Occur.....                              | 51        |
| Safe Kernel Upgrade Strategy.....                           | 52        |
| Illustration: Kernel Fallback Model.....                    | 52        |
| <b>Rescue Mode &amp; Recovery Shell.....</b>                | <b>52</b> |
| When Things Go Wrong.....                                   | 52        |
| Accessing GRUB Recovery Mode.....                           | 52        |
| Recovery Shell Example.....                                 | 53        |
| Using Live ISO for Rescue.....                              | 53        |
| <b>Practical Scenario: Recovery in Production.....</b>      | <b>53</b> |
| <b>Post-Install Checklist for Production Readiness.....</b> | <b>54</b> |
| <b>Chapter Summary.....</b>                                 | <b>54</b> |
| <b>Reflection Challenge.....</b>                            | <b>54</b> |
| 2.4 Release Upgrade Strategy.....                           | 55        |
| Introduction: Upgrading Is a Strategic Decision.....        | 55        |
| <b>Understanding Ubuntu Release Types.....</b>              | <b>55</b> |
| <b>The do-release-upgrade Tool.....</b>                     | <b>56</b> |
| Controlled Distribution Upgrade.....                        | 56        |
| Checking Upgrade Availability.....                          | 56        |
| <b>Production-Safe Upgrade Planning.....</b>                | <b>57</b> |
| Think Before You Upgrade.....                               | 57        |
| Step 1: Assess the Environment.....                         | 57        |
| Step 2: Review Installed Packages.....                      | 57        |
| Step 3: Backup Strategy.....                                | 57        |
| Step 4: Staging Environment Testing.....                    | 58        |
| Step 5: Maintenance Window.....                             | 58        |
| <b>Snapshot &amp; Rollback Approach.....</b>                | <b>58</b> |
| Safety Net for Modern Infrastructure.....                   | 58        |
| LVM Snapshot Example.....                                   | 59        |
| Virtual Machine Snapshot Strategy.....                      | 59        |

|                                                                     |           |
|---------------------------------------------------------------------|-----------|
| Cloud Snapshot Strategy.....                                        | 59        |
| <b>Immutable Infrastructure Alternative.....</b>                    | <b>60</b> |
| <b>Handling Common Upgrade Issues.....</b>                          | <b>60</b> |
| <b>When Not to Upgrade.....</b>                                     | <b>60</b> |
| <b>Upgrade Lifecycle Philosophy.....</b>                            | <b>61</b> |
| <b>Real-World Scenario.....</b>                                     | <b>61</b> |
| <b>Chapter Summary.....</b>                                         | <b>61</b> |
| <b>Reflection Challenge.....</b>                                    | <b>62</b> |
| <b>PART II — SYSTEM CONFIGURATION &amp; PACKAGE MANAGEMENT.....</b> | <b>63</b> |
| Chapter 3 — APT, Snap & Repository Strategy.....                    | 63        |
| Introduction: Package Management Is System Governance.....          | 63        |
| <b>Understanding APT Fundamentals.....</b>                          | <b>63</b> |
| apt update.....                                                     | 64        |
| apt upgrade.....                                                    | 64        |
| apt full-upgrade.....                                               | 64        |
| Practical Comparison.....                                           | 64        |
| <b>Package Pinning.....</b>                                         | <b>65</b> |
| Controlling What Gets Installed.....                                | 65        |
| Example: Pin a Specific Version.....                                | 65        |
| <b>Snap Services on Server.....</b>                                 | <b>66</b> |
| Application Isolation with Snap.....                                | 66        |
| Managing Snap Services.....                                         | 66        |
| <b>Unattended-Upgrades.....</b>                                     | <b>67</b> |
| Security Automation Without Chaos.....                              | 67        |
| Production Recommendation.....                                      | 67        |
| <b>Repository Trust &amp; GPG Keys.....</b>                         | <b>68</b> |
| Verifying Software Integrity.....                                   | 68        |
| Why Repository Trust Matters.....                                   | 68        |
| <b>Best Practice: Staging vs Production Update.....</b>             | <b>69</b> |
| Staging Workflow.....                                               | 69        |
| Illustration: Safe Update Pipeline.....                             | 69        |
| Why Direct Production Updates Are Dangerous.....                    | 69        |
| <b>Managing Reboots After Updates.....</b>                          | <b>69</b> |
| <b>Holistic Repository Strategy.....</b>                            | <b>70</b> |
| <b>Real-World Scenario.....</b>                                     | <b>70</b> |
| <b>Chapter Summary.....</b>                                         | <b>71</b> |
| <b>Reflection Challenge.....</b>                                    | <b>71</b> |
| <b>PART III — SERVICE &amp; PROCESS MANAGEMENT.....</b>             | <b>72</b> |
| Chapter 4 — systemd Deep Dive (Modern Service Control).....         | 72        |
| 4.1 systemd Fundamentals.....                                       | 72        |
| Introduction: systemd as the Brain of Ubuntu Server.....            | 72        |
| <b>What Is systemd?.....</b>                                        | <b>73</b> |
| <b>Understanding Units.....</b>                                     | <b>73</b> |
| The Building Blocks of systemd.....                                 | 73        |

|                                                              |           |
|--------------------------------------------------------------|-----------|
| <b>Core Unit Types</b> .....                                 | <b>74</b> |
| 1. Service Units (.service).....                             | 74        |
| 2. Socket Units (.socket).....                               | 74        |
| 3. Timer Units (.timer).....                                 | 75        |
| 4. Target Units (.target).....                               | 75        |
| <b>Dependency Tree</b> .....                                 | <b>76</b> |
| The Power of Declarative Relationships.....                  | 76        |
| Visualizing Dependencies.....                                | 76        |
| Why Dependency Trees Matter.....                             | 77        |
| <b>Boot Targets</b> .....                                    | <b>77</b> |
| Defining System Startup Modes.....                           | 77        |
| Common Boot Targets.....                                     | 77        |
| Changing Default Target.....                                 | 78        |
| Illustration: Boot Flow with systemd.....                    | 78        |
| <b>Service Supervision &amp; Reliability</b> .....           | <b>78</b> |
| <b>Real-World Scenario</b> .....                             | <b>79</b> |
| <b>Debugging with systemd</b> .....                          | <b>79</b> |
| <b>Production Best Practices</b> .....                       | <b>79</b> |
| <b>Chapter Summary</b> .....                                 | <b>80</b> |
| <b>Reflection Challenge</b> .....                            | <b>80</b> |
| 4.2 Managing Services.....                                   | 81        |
| Introduction: Service Management Is Operational Control..... | 81        |
| <b>systemctl Fundamentals</b> .....                          | <b>81</b> |
| Starting, Stopping, Enabling, and Masking.....               | 81        |
| Starting and Stopping Services.....                          | 82        |
| Restarting and Reloading.....                                | 82        |
| <b>Enabling Services at Boot</b> .....                       | <b>83</b> |
| Checking Service Boot Status.....                            | 83        |
| <b>Masking Services</b> .....                                | <b>83</b> |
| Preventing Accidental Startup.....                           | 83        |
| <b>Override Files</b> .....                                  | <b>84</b> |
| Customizing Service Behavior Safely.....                     | 84        |
| Creating an Override File.....                               | 84        |
| Why Override Files Matter.....                               | 85        |
| <b>Analyzing Service Failures</b> .....                      | <b>85</b> |
| Debugging with Confidence.....                               | 85        |
| Checking Service Status.....                                 | 85        |
| Viewing Detailed Logs.....                                   | 86        |
| <b>Common Failure Scenarios</b> .....                        | <b>86</b> |
| 1. Port Already in Use.....                                  | 86        |
| 2. Configuration Error.....                                  | 86        |
| 3. Dependency Failure.....                                   | 87        |
| <b>Service Restart Policies</b> .....                        | <b>87</b> |
| <b>Practical Illustration: Service Failure Flow</b> .....    | <b>87</b> |

|                                                        |     |
|--------------------------------------------------------|-----|
| <b>Best Practices for Production</b> .....             | 88  |
| <b>Real-World Scenario</b> .....                       | 88  |
| <b>Chapter Summary</b> .....                           | 89  |
| <b>Reflection Challenge</b> .....                      | 89  |
| 4.3 Timers vs Cron.....                                | 90  |
| Introduction: Scheduling in a Modern Server World..... | 90  |
| <b>Traditional Cron</b> .....                          | 90  |
| The Classic Scheduler.....                             | 90  |
| Advantages of Cron.....                                | 91  |
| Limitations of Cron.....                               | 91  |
| <b>systemd Timers</b> .....                            | 92  |
| Modern Scheduling Integrated into the System.....      | 92  |
| Basic Timer Structure.....                             | 92  |
| <b>Key Differences: Cron vs systemd Timers</b> .....   | 93  |
| <b>Why systemd Timers Are Modern Scheduling</b> .....  | 93  |
| <b>Understanding Timer Triggers</b> .....              | 93  |
| 1. Calendar-Based Timers.....                          | 93  |
| 2. Monotonic Timers.....                               | 94  |
| <b>Migrating Cron Jobs to systemd Timers</b> .....     | 94  |
| Step 1: Identify Existing Cron Job.....                | 94  |
| Step 2: Create Service Unit.....                       | 94  |
| Step 3: Create Timer Unit.....                         | 94  |
| Step 4: Enable Timer.....                              | 95  |
| <b>Viewing Active Timers</b> .....                     | 95  |
| <b>Persistent Timers</b> .....                         | 95  |
| <b>Practical Illustration: Scheduling Model</b> .....  | 96  |
| <b>Real-World Production Example</b> .....             | 96  |
| <b>When to Use Cron vs systemd Timer</b> .....         | 97  |
| <b>Best Practices for Production</b> .....             | 97  |
| <b>Chapter Summary</b> .....                           | 97  |
| <b>Reflection Challenge</b> .....                      | 98  |
| 4.4 Logging with journalctl.....                       | 99  |
| Introduction: Logs Are the Memory of Your Server.....  | 99  |
| <b>The systemd Journal</b> .....                       | 99  |
| A Unified Logging System.....                          | 99  |
| <b>Basic Usage of journalctl</b> .....                 | 100 |
| View Logs for Current Boot.....                        | 100 |
| <b>Persistent Logging</b> .....                        | 101 |
| Ensuring Logs Survive Reboots.....                     | 101 |
| Why Persistent Logging Matters.....                    | 101 |
| <b>Filtering Logs</b> .....                            | 101 |
| Finding What Matters.....                              | 101 |
| Filter by Service.....                                 | 101 |
| Filter by Time.....                                    | 102 |

|                                                                    |            |
|--------------------------------------------------------------------|------------|
| Filter by Priority Level.....                                      | 102        |
| Filter by Boot Session.....                                        | 102        |
| <b>Structured Output &amp; Formatting.....</b>                     | <b>103</b> |
| <b>Troubleshooting Services with journalctl.....</b>               | <b>103</b> |
| Scenario 1 — Service Fails to Start.....                           | 103        |
| Scenario 2 — Port Conflict.....                                    | 103        |
| Scenario 3 — Permission Error.....                                 | 104        |
| <b>Combining systemctl and journalctl.....</b>                     | <b>104</b> |
| <b>Log Retention and Disk Usage.....</b>                           | <b>104</b> |
| <b>Integration with Traditional Logs.....</b>                      | <b>105</b> |
| <b>Real-World Production Example.....</b>                          | <b>105</b> |
| <b>Illustration: Troubleshooting Flow.....</b>                     | <b>105</b> |
| <b>Best Practices for Production Logging.....</b>                  | <b>106</b> |
| <b>Chapter Summary.....</b>                                        | <b>106</b> |
| <b>Reflection Challenge.....</b>                                   | <b>106</b> |
| <b>PART IV — NETWORKING FOR SERVERS.....</b>                       | <b>107</b> |
| Chapter 5 — Netplan & Network Architecture.....                    | 107        |
| Introduction: Networking Is the Nervous System of Your Server..... | 107        |
| <b>Understanding Netplan.....</b>                                  | <b>107</b> |
| Declarative Network Configuration.....                             | 107        |
| Basic Netplan YAML Structure.....                                  | 108        |
| <b>DHCP vs Static IP.....</b>                                      | <b>108</b> |
| Choosing the Right Addressing Strategy.....                        | 108        |
| DHCP (Dynamic Host Configuration Protocol).....                    | 108        |
| Static IP Configuration.....                                       | 109        |
| Practical Comparison.....                                          | 110        |
| <b>Bridges for Virtual Machines.....</b>                           | <b>110</b> |
| Networking for Hypervisors.....                                    | 110        |
| Example Bridge Configuration.....                                  | 110        |
| <b>Bonding for Redundancy.....</b>                                 | <b>111</b> |
| High Availability Network Interfaces.....                          | 111        |
| <b>DNS &amp; systemd-resolved.....</b>                             | <b>112</b> |
| Managing Name Resolution.....                                      | 112        |
| <b>Troubleshooting Networking.....</b>                             | <b>113</b> |
| Essential Diagnostic Tools.....                                    | 113        |
| ss — Socket Statistics.....                                        | 113        |
| ip route — Routing Table.....                                      | 113        |
| tcpdump — Packet Inspection.....                                   | 113        |
| curl — Application Layer Testing.....                              | 114        |
| <b>Practical Troubleshooting Flow.....</b>                         | <b>114</b> |
| <b>Real-World Scenario.....</b>                                    | <b>114</b> |
| <b>Chapter Summary.....</b>                                        | <b>115</b> |
| <b>Reflection Challenge.....</b>                                   | <b>115</b> |
| Chapter 6 — Firewall & Network Security.....                       | 116        |

|                                                             |            |
|-------------------------------------------------------------|------------|
| Introduction: Security Is Not Optional.....                 | 116        |
| <b>UFW Fundamentals.....</b>                                | <b>116</b> |
| A Simple Interface to Powerful Packet Filtering.....        | 116        |
| Default Policy Strategy.....                                | 117        |
| Allowing Essential Services.....                            | 117        |
| <b>Application Profiles.....</b>                            | <b>118</b> |
| Simplifying Service-Based Rules.....                        | 118        |
| <b>Basic Hardening.....</b>                                 | <b>118</b> |
| Strengthening the Server's Defense.....                     | 118        |
| 1. Restrict SSH Access.....                                 | 118        |
| 2. Rate Limiting SSH.....                                   | 119        |
| 3. Deny Specific IP Address.....                            | 119        |
| 4. Logging Firewall Activity.....                           | 119        |
| Understanding Port Exposure.....                            | 119        |
| <b>Production Firewall Layering Concept.....</b>            | <b>120</b> |
| Defense in Depth.....                                       | 120        |
| Layer 1 — Cloud Firewall / Security Group.....              | 120        |
| Layer 2 — Host Firewall (UFW).....                          | 120        |
| Layer 3 — Application-Level Security.....                   | 120        |
| Layer 4 — Internal Segmentation.....                        | 120        |
| Layered Illustration.....                                   | 120        |
| <b>Production Example Scenario.....</b>                     | <b>121</b> |
| <b>Advanced Considerations.....</b>                         | <b>121</b> |
| <b>Common Mistakes to Avoid.....</b>                        | <b>121</b> |
| <b>Testing Firewall Configuration.....</b>                  | <b>122</b> |
| <b>Chapter Summary.....</b>                                 | <b>122</b> |
| <b>Reflection Challenge.....</b>                            | <b>122</b> |
| <b>PART V — STORAGE &amp; FILESYSTEM MANAGEMENT.....</b>    | <b>123</b> |
| Chapter 7 — Disk, Filesystem & Mounting.....                | 123        |
| Introduction: Storage Is the Foundation of Reliability..... | 123        |
| Identifying Disks.....                                      | 123        |
| Understanding Your Storage Layout.....                      | 123        |
| Using lsblk.....                                            | 123        |
| Using blkid.....                                            | 124        |
| <b>Partitioning Disks.....</b>                              | <b>124</b> |
| Preparing Storage for Use.....                              | 124        |
| Using fdisk (Basic Example).....                            | 125        |
| Using parted (Modern GPT-Friendly Tool).....                | 125        |
| <b>Creating Filesystems.....</b>                            | <b>125</b> |
| Choosing the Right Filesystem Type.....                     | 125        |
| ext4 — General Purpose.....                                 | 126        |
| xfs — High Performance & Scalability.....                   | 126        |
| Practical Comparison.....                                   | 126        |
| <b>Mounting Filesystems.....</b>                            | <b>127</b> |

|                                                           |            |
|-----------------------------------------------------------|------------|
| Temporary vs Persistent Mount.....                        | 127        |
| Temporary Mount.....                                      | 127        |
| Persistent Mount (Using fstab).....                       | 127        |
| <b>fstab Deep Explanation.....</b>                        | <b>127</b> |
| Understanding Each Field.....                             | 127        |
| Why Use UUID Instead of /dev/sdb1?.....                   | 128        |
| Testing fstab Safely.....                                 | 128        |
| <b>Mount Options &amp; Security.....</b>                  | <b>128</b> |
| Hardening Storage Access.....                             | 128        |
| Common Secure Mount Options.....                          | 128        |
| Example: Secure Web Upload Directory.....                 | 129        |
| <b>Disk Usage Monitoring.....</b>                         | <b>129</b> |
| <b>Practical Illustration: Storage Flow.....</b>          | <b>129</b> |
| <b>Real-World Production Example.....</b>                 | <b>130</b> |
| <b>Common Mistakes to Avoid.....</b>                      | <b>130</b> |
| <b>Chapter Summary.....</b>                               | <b>131</b> |
| <b>Reflection Challenge.....</b>                          | <b>131</b> |
| <b>Chapter 8 — LVM &amp; RAID.....</b>                    | <b>132</b> |
| Introduction: Storage Must Be Flexible and Resilient..... | 132        |
| <b>Understanding LVM.....</b>                             | <b>132</b> |
| Storage Abstraction for Flexibility.....                  | 132        |
| <b>Core LVM Concepts.....</b>                             | <b>133</b> |
| 1. Physical Volume (PV).....                              | 133        |
| 2. Volume Group (VG).....                                 | 133        |
| 3. Logical Volume (LV).....                               | 133        |
| <b>Illustration: LVM Architecture.....</b>                | <b>134</b> |
| <b>Resizing Live Filesystems.....</b>                     | <b>134</b> |
| Growing Storage Without Downtime.....                     | 134        |
| Step 1 — Add New Physical Volume.....                     | 134        |
| Step 2 — Extend Logical Volume.....                       | 134        |
| Step 3 — Resize Filesystem.....                           | 134        |
| Result.....                                               | 135        |
| <b>LVM Snapshots.....</b>                                 | <b>135</b> |
| Backup & Testing Capability.....                          | 135        |
| <b>Software RAID with mdadm.....</b>                      | <b>135</b> |
| Building Redundancy at the OS Level.....                  | 135        |
| Common RAID Levels.....                                   | 136        |
| <b>Creating RAID 1 (Mirroring Example).....</b>           | <b>136</b> |
| <b>RAID + LVM Combined Strategy.....</b>                  | <b>136</b> |
| <b>Enterprise Storage Mindset.....</b>                    | <b>137</b> |
| Designing for Failure.....                                | 137        |
| Key Enterprise Principles.....                            | 137        |
| <b>Real-World Scenario.....</b>                           | <b>137</b> |
| <b>Common Mistakes to Avoid.....</b>                      | <b>138</b> |

|                                                              |     |
|--------------------------------------------------------------|-----|
| <b>Practical CLI Summary</b> .....                           | 138 |
| <b>Chapter Summary</b> .....                                 | 139 |
| <b>Reflection Challenge</b> .....                            | 139 |
| <b>PART VI — USER, PERMISSION &amp; ACCESS CONTROL</b> ..... | 140 |
| Chapter 9 — User & Group Administration.....                 | 140 |
| Introduction: Access Control Is Infrastructure Security..... | 140 |
| <b>Understanding Users in Linux</b> .....                    | 140 |
| <b>adduser vs useradd</b> .....                              | 141 |
| Understanding the Difference.....                            | 141 |
| adduser — High-Level, Interactive.....                       | 141 |
| useradd — Low-Level, Manual.....                             | 142 |
| Practical Comparison.....                                    | 142 |
| <b>UID: System vs Regular Users</b> .....                    | 143 |
| System Users.....                                            | 143 |
| <b>Understanding /etc/passwd and /etc/shadow</b> .....       | 143 |
| /etc/passwd.....                                             | 143 |
| /etc/shadow.....                                             | 144 |
| <b>Password Policy</b> .....                                 | 144 |
| Enforcing Strong Authentication.....                         | 144 |
| Enforcing Minimum Password Length.....                       | 145 |
| Enforcing Password Expiration.....                           | 145 |
| Locking User Account.....                                    | 145 |
| <b>Group Administration</b> .....                            | 145 |
| <b>sudoers Best Practice</b> .....                           | 146 |
| Controlled Administrative Privileges.....                    | 146 |
| Adding User to sudo Group.....                               | 146 |
| Editing sudoers Safely.....                                  | 146 |
| Restricting Specific Commands.....                           | 146 |
| <b>Illustration: Access Control Flow</b> .....               | 147 |
| <b>Real-World Production Example</b> .....                   | 147 |
| <b>Common Mistakes to Avoid</b> .....                        | 147 |
| <b>Monitoring User Activity</b> .....                        | 148 |
| <b>Chapter Summary</b> .....                                 | 148 |
| <b>Reflection Challenge</b> .....                            | 148 |
| Chapter 10 — Advanced Permission & ACL.....                  | 149 |
| Introduction: Permissions Define Trust.....                  | 149 |
| <b>Revisiting Linux Permission Model</b> .....               | 149 |
| <b>chmod Advanced</b> .....                                  | 150 |
| Symbolic and Recursive Control.....                          | 150 |
| Symbolic Mode.....                                           | 150 |
| Recursive Permission Changes.....                            | 150 |
| <b>setuid, setgid, and Sticky Bit</b> .....                  | 151 |
| Special Permission Bits.....                                 | 151 |
| setuid (Set User ID).....                                    | 151 |

|                                                          |            |
|----------------------------------------------------------|------------|
| setgid (Set Group ID).....                               | 151        |
| Sticky Bit.....                                          | 152        |
| <b>Understanding umask.....</b>                          | <b>152</b> |
| Default Permission Mask.....                             | 152        |
| Setting umask.....                                       | 153        |
| <b>Access Control Lists (ACL).....</b>                   | <b>153</b> |
| Fine-Grained Permission Control.....                     | 153        |
| Installing ACL Support.....                              | 153        |
| Setting ACL.....                                         | 154        |
| Removing ACL.....                                        | 154        |
| Default ACL for Directories.....                         | 154        |
| <b>Practical Illustration: Permission Hierarchy.....</b> | <b>154</b> |
| <b>Real-World Production Example.....</b>                | <b>155</b> |
| <b>Security Implications.....</b>                        | <b>155</b> |
| <b>Common Mistakes to Avoid.....</b>                     | <b>156</b> |
| <b>Monitoring Permissions.....</b>                       | <b>156</b> |
| <b>Chapter Summary.....</b>                              | <b>156</b> |
| <b>Reflection Challenge.....</b>                         | <b>157</b> |
| <b>PART VII — SECURING SERVER ACCESS.....</b>            | <b>158</b> |
| Chapter 11 — SSH Hardening.....                          | 158        |
| Introduction: SSH Is the Front Door of Your Server.....  | 158        |
| Key-Based Authentication.....                            | 158        |
| Eliminating Password-Based Risk.....                     | 158        |
| Generating SSH Keys (Client Side).....                   | 159        |
| Copying Public Key to Server.....                        | 159        |
| Enforcing Key-Only Login.....                            | 159        |
| Disabling Root Login.....                                | 160        |
| Preventing Direct Administrative Access.....             | 160        |
| Custom SSH Port.....                                     | 160        |
| Reducing Automated Noise.....                            | 160        |
| SSH Rate Limiting.....                                   | 161        |
| Slowing Down Attackers.....                              | 161        |
| Fail2Ban.....                                            | 161        |
| Automatic Intrusion Prevention.....                      | 161        |
| Basic SSH Jail Configuration.....                        | 161        |
| Firewall Integration.....                                | 162        |
| Coordinated Defense Strategy.....                        | 162        |
| Production SSH Hardening Checklist.....                  | 162        |
| Advanced Security Considerations.....                    | 163        |
| Practical Illustration: Hardened SSH Flow.....           | 163        |
| Real-World Scenario.....                                 | 164        |
| Monitoring SSH Activity.....                             | 164        |
| Common Mistakes to Avoid.....                            | 164        |
| Chapter Summary.....                                     | 165        |

|                                                                 |            |
|-----------------------------------------------------------------|------------|
| Reflection Challenge.....                                       | 165        |
| <b>PART VIII — CONFIGURING CORE SERVER SERVICES (~23%).....</b> | <b>166</b> |
| Chapter 12 — File Sharing Services.....                         | 166        |
| Introduction: Sharing Data Securely and Efficiently.....        | 166        |
| <b>NFS Setup.....</b>                                           | <b>166</b> |
| Native Linux File Sharing.....                                  | 166        |
| Installing NFS Server.....                                      | 167        |
| Configuring NFS Exports.....                                    | 167        |
| Mounting NFS on Client.....                                     | 167        |
| Illustration: NFS Architecture.....                             | 168        |
| <b>Samba Configuration.....</b>                                 | <b>168</b> |
| Cross-Platform File Sharing.....                                | 168        |
| Installing Samba.....                                           | 168        |
| Basic Samba Share Configuration.....                            | 168        |
| Creating Samba User.....                                        | 168        |
| Testing Samba Share.....                                        | 169        |
| <b>Simple Permission Integration.....</b>                       | <b>169</b> |
| Aligning Linux Permissions with Network Access.....             | 169        |
| Shared Group Strategy.....                                      | 169        |
| Illustration: Permission Flow.....                              | 170        |
| <b>NFS vs Samba Comparison.....</b>                             | <b>170</b> |
| <b>Security Considerations.....</b>                             | <b>170</b> |
| <b>Real-World Production Example.....</b>                       | <b>171</b> |
| <b>Monitoring and Troubleshooting.....</b>                      | <b>171</b> |
| <b>Best Practices for Production.....</b>                       | <b>172</b> |
| <b>Chapter Summary.....</b>                                     | <b>172</b> |
| <b>Reflection Challenge.....</b>                                | <b>172</b> |
| Chapter 13 — Web & Database Deployment.....                     | 173        |
| Introduction: From Infrastructure to Application Delivery.....  | 173        |
| <b>Web Servers: Apache vs Nginx.....</b>                        | <b>173</b> |
| Installing Apache.....                                          | 174        |
| Installing Nginx.....                                           | 174        |
| Apache vs Nginx Comparison.....                                 | 174        |
| <b>Deploying MySQL.....</b>                                     | <b>175</b> |
| Installing MySQL.....                                           | 175        |
| Accessing MySQL.....                                            | 175        |
| <b>Deploying PostgreSQL.....</b>                                | <b>176</b> |
| Installing PostgreSQL.....                                      | 176        |
| <b>Basic Reverse Proxy.....</b>                                 | <b>176</b> |
| Routing Traffic Professionally.....                             | 176        |
| Example: Proxy to Backend App.....                              | 177        |
| Reverse Proxy Flow Illustration.....                            | 177        |
| <b>Service Enablement via systemd.....</b>                      | <b>178</b> |
| <b>Production Deployment Workflow.....</b>                      | <b>178</b> |

|                                                                 |     |
|-----------------------------------------------------------------|-----|
| <b>Firewall Integration</b> .....                               | 179 |
| <b>Real-World Production Example</b> .....                      | 179 |
| <b>Common Mistakes to Avoid</b> .....                           | 179 |
| <b>Monitoring Deployment Health</b> .....                       | 180 |
| <b>Chapter Summary</b> .....                                    | 180 |
| <b>Reflection Challenge</b> .....                               | 180 |
| Chapter 14 — Backup & Automation.....                           | 181 |
| Introduction: If It Is Not Backed Up, It Does Not Exist.....    | 181 |
| <b>Backup Strategy Fundamentals</b> .....                       | 181 |
| <b>rsync Backup Strategy</b> .....                              | 182 |
| Efficient Incremental Synchronization.....                      | 182 |
| Basic rsync Command.....                                        | 182 |
| Remote Backup via SSH.....                                      | 182 |
| Deleting Removed Files (Mirror Mode).....                       | 182 |
| <b>Incremental Snapshot Strategy with rsync</b> .....           | 183 |
| <b>Snapshot Concept</b> .....                                   | 183 |
| Point-in-Time Protection.....                                   | 183 |
| LVM Snapshot Example.....                                       | 183 |
| Filesystem Snapshot (Conceptual Illustration).....              | 183 |
| <b>Cron vs systemd Timer</b> .....                              | 184 |
| Automating Backup Tasks.....                                    | 184 |
| Cron-Based Backup.....                                          | 184 |
| systemd Timer-Based Backup.....                                 | 184 |
| <b>Backup Verification</b> .....                                | 185 |
| <b>Basic Automation Mindset</b> .....                           | 185 |
| Example: Simple Backup Script.....                              | 185 |
| <b>3-2-1 Backup Rule</b> .....                                  | 186 |
| <b>Real-World Production Example</b> .....                      | 186 |
| <b>Automation + Monitoring</b> .....                            | 187 |
| <b>Common Mistakes to Avoid</b> .....                           | 187 |
| <b>Illustration: Recovery-Oriented Thinking</b> .....           | 187 |
| <b>Chapter Summary</b> .....                                    | 188 |
| <b>Reflection Challenge</b> .....                               | 188 |
| <b>PART IX — MONITORING &amp; TROUBLESHOOTING</b> .....         | 189 |
| Chapter 15 — Monitoring System Health.....                      | 189 |
| Introduction: If You Don't Measure It, You Can't Manage It..... | 189 |
| <b>top — Real-Time Process Monitoring</b> .....                 | 189 |
| Understanding Load Average.....                                 | 190 |
| CPU Breakdown in top.....                                       | 190 |
| <b>htop — Enhanced Process Monitoring</b> .....                 | 190 |
| <b>free — Memory Usage Overview</b> .....                       | 191 |
| <b>vmstat — Virtual Memory Statistics</b> .....                 | 191 |
| <b>iostat — Disk I/O Monitoring</b> .....                       | 192 |
| <b>journalctl Analysis</b> .....                                | 193 |

|                                                                            |            |
|----------------------------------------------------------------------------|------------|
| Detecting Hidden Problems.....                                             | 193        |
| <b>Basic Performance Interpretation.....</b>                               | <b>193</b> |
| Scenario 1 — High CPU Usage.....                                           | 193        |
| Scenario 2 — High Memory Usage.....                                        | 194        |
| Scenario 3 — High Disk I/O.....                                            | 194        |
| <b>Monitoring Illustration.....</b>                                        | <b>194</b> |
| <b>Proactive Monitoring Strategy.....</b>                                  | <b>195</b> |
| <b>Real-World Production Example.....</b>                                  | <b>195</b> |
| <b>Best Practices for Monitoring.....</b>                                  | <b>196</b> |
| <b>Chapter Summary.....</b>                                                | <b>196</b> |
| <b>Reflection Challenge.....</b>                                           | <b>196</b> |
| Chapter 16 — Practical Troubleshooting Methodology.....                    | 197        |
| Introduction: Troubleshooting Is a Discipline, Not a Guessing Game.....    | 197        |
| <b>The Layered Troubleshooting Model.....</b>                              | <b>197</b> |
| <b>Layer 1 — Hardware Investigation.....</b>                               | <b>198</b> |
| <b>Layer 2 — Kernel-Level Investigation.....</b>                           | <b>198</b> |
| <b>Layer 3 — Service-Level Troubleshooting.....</b>                        | <b>199</b> |
| <b>Layer 4 — Network-Level Troubleshooting.....</b>                        | <b>199</b> |
| <b>Log-First Debugging.....</b>                                            | <b>200</b> |
| <b>Boot Failure Investigation.....</b>                                     | <b>200</b> |
| Step 1 — Access Recovery Mode.....                                         | 200        |
| Step 2 — Inspect Logs.....                                                 | 201        |
| fstab Failure Example.....                                                 | 201        |
| <b>Service Dependency Failure.....</b>                                     | <b>201</b> |
| <b>Structured Troubleshooting Workflow.....</b>                            | <b>202</b> |
| <b>Real-World Scenario.....</b>                                            | <b>202</b> |
| <b>Boot Failure Case Study.....</b>                                        | <b>202</b> |
| <b>Troubleshooting Mindset.....</b>                                        | <b>203</b> |
| <b>Illustration: Troubleshooting Pyramid.....</b>                          | <b>203</b> |
| <b>Common Mistakes.....</b>                                                | <b>203</b> |
| <b>Chapter Summary.....</b>                                                | <b>204</b> |
| <b>Reflection Challenge.....</b>                                           | <b>204</b> |
| <b>PART X — MODERN SYSADMIN MINDSET.....</b>                               | <b>205</b> |
| Chapter 17 — Automation & Infrastructure Thinking.....                     | 205        |
| Introduction: From Manual Configuration to Infrastructure Engineering..... | 205        |
| <b>Avoiding Snowflake Servers.....</b>                                     | <b>205</b> |
| <b>Infrastructure as Code (IaC) Concept.....</b>                           | <b>206</b> |
| <b>Introduction to Ansible.....</b>                                        | <b>207</b> |
| Installing Ansible.....                                                    | 207        |
| Basic Inventory File.....                                                  | 207        |
| Simple Ansible Test.....                                                   | 207        |
| Simple Playbook Example.....                                               | 207        |
| <b>Reproducible Builds.....</b>                                            | <b>208</b> |
| <b>Golden Image Strategy.....</b>                                          | <b>209</b> |

|                                                              |            |
|--------------------------------------------------------------|------------|
| Building a Golden Image (Conceptual Steps).....              | 209        |
| <b>Automation Mindset.....</b>                               | <b>210</b> |
| <b>Real-World Scenario.....</b>                              | <b>210</b> |
| <b>Combining Golden Image + Ansible.....</b>                 | <b>211</b> |
| <b>Infrastructure Thinking Illustration.....</b>             | <b>211</b> |
| <b>Common Mistakes.....</b>                                  | <b>211</b> |
| <b>Chapter Summary.....</b>                                  | <b>212</b> |
| <b>Reflection Challenge.....</b>                             | <b>212</b> |
| Chapter 18 — Ubuntu Server in Cloud & Containers.....        | 213        |
| Introduction: Ubuntu Beyond the Physical Server.....         | 213        |
| <b>Cloud-Init: Automated First Boot Configuration.....</b>   | <b>213</b> |
| Basic Cloud-Init Example.....                                | 214        |
| Testing Cloud-Init Locally.....                              | 214        |
| <b>Ubuntu Minimal Cloud Image.....</b>                       | <b>214</b> |
| <b>Running Docker on Ubuntu Server.....</b>                  | <b>215</b> |
| Installing Docker.....                                       | 215        |
| Running a Simple Web Container.....                          | 215        |
| <b>Podman as Alternative.....</b>                            | <b>216</b> |
| <b>Container vs VM Illustration.....</b>                     | <b>216</b> |
| <b>Production Patterns in Cloud.....</b>                     | <b>216</b> |
| Pattern 1 — Immutable Infrastructure.....                    | 216        |
| Pattern 2 — Auto Scaling.....                                | 217        |
| Pattern 3 — Reverse Proxy + Containers.....                  | 217        |
| <b>Cloud Security Considerations.....</b>                    | <b>217</b> |
| <b>Combining Cloud-Init + Containers.....</b>                | <b>218</b> |
| <b>Real-World Deployment Scenario.....</b>                   | <b>218</b> |
| <b>Common Mistakes.....</b>                                  | <b>219</b> |
| <b>Monitoring Containers.....</b>                            | <b>219</b> |
| <b>Chapter Summary.....</b>                                  | <b>219</b> |
| <b>Reflection Challenge.....</b>                             | <b>220</b> |
| <b>PART XI — EXAM PREPARATION &amp; PRACTICAL LABS.....</b>  | <b>221</b> |
| Chapter 19 — Canonical Academy Exam Simulation.....          | 221        |
| Introduction: From Knowledge to Demonstrated Competence..... | 221        |
| <b>Understanding Performance-Based Exams.....</b>            | <b>221</b> |
| <b>Practice Scenario 1 — Secure SSH Deployment.....</b>      | <b>222</b> |
| <b>Practice Scenario 2 — Web Stack Deployment.....</b>       | <b>223</b> |
| <b>Practice Scenario 3 — Storage &amp; LVM.....</b>          | <b>224</b> |
| <b>Performance-Based Tasks.....</b>                          | <b>224</b> |
| <b>Timed Lab Simulation.....</b>                             | <b>225</b> |
| <b>Troubleshooting Simulation.....</b>                       | <b>225</b> |
| <b>Common Mistakes in Exam &amp; Real World.....</b>         | <b>226</b> |
| <b>Exam Preparation Strategy.....</b>                        | <b>226</b> |
| <b>Simulation Lab Exercise.....</b>                          | <b>226</b> |
| <b>Professional Mindset.....</b>                             | <b>227</b> |

|                                                          |     |
|----------------------------------------------------------|-----|
| <b>Illustration: Operational Confidence Loop</b> .....   | 227 |
| <b>Chapter Summary</b> .....                             | 227 |
| <b>Final Reflection</b> .....                            | 228 |
| Chapter 20 — Capstone Project.....                       | 229 |
| Introduction: From Learning to Building.....             | 229 |
| <b>Project Scenario</b> .....                            | 229 |
| <b>Step 1 — Configure Static IP</b> .....                | 229 |
| <b>Step 2 — Configure Firewall</b> .....                 | 230 |
| <b>Step 3 — SSH Hardening</b> .....                      | 231 |
| <b>Step 4 — Install LAMP Stack</b> .....                 | 231 |
| <b>Step 5 — Configure LVM Storage</b> .....              | 232 |
| <b>Step 6 — Backup Automation</b> .....                  | 233 |
| <b>Step 7 — Monitoring Baseline</b> .....                | 234 |
| <b>Final Validation Checklist</b> .....                  | 234 |
| <b>Capstone Architecture Overview</b> .....              | 235 |
| <b>Common Mistakes During Capstone</b> .....             | 235 |
| <b>Professional Reflection</b> .....                     | 235 |
| <b>Final Chapter Summary</b> .....                       | 236 |
| <b>Closing Reflection</b> .....                          | 236 |
| <b>Appendix A — Command Quick Reference</b> .....        | 237 |
| System Information.....                                  | 237 |
| Service Management (systemd).....                        | 237 |
| Networking.....                                          | 237 |
| Storage & Filesystem.....                                | 238 |
| User & Permissions.....                                  | 238 |
| Firewall & Security.....                                 | 238 |
| <b>Appendix B — systemd Unit Template Examples</b> ..... | 239 |
| Basic Service Template.....                              | 239 |
| Timer Example.....                                       | 239 |
| <b>Appendix C — Autoinstall YAML Templates</b> .....     | 240 |
| Minimal Autoinstall Template.....                        | 240 |
| Storage Section Example.....                             | 240 |
| <b>Appendix D — Netplan YAML Examples</b> .....          | 241 |
| Static IP Configuration.....                             | 241 |
| DHCP Configuration.....                                  | 241 |
| Bridge Example (For Virtualization).....                 | 241 |
| <b>Appendix E — Troubleshooting Flowcharts</b> .....     | 242 |
| Service Not Running.....                                 | 242 |
| Cannot Reach Server.....                                 | 242 |
| High System Load.....                                    | 242 |
| <b>Appendix F — Production Hardening Checklist</b> ..... | 243 |
| System Security.....                                     | 243 |
| Service Security.....                                    | 243 |
| Storage & Backup.....                                    | 243 |

|                                            |            |
|--------------------------------------------|------------|
| Monitoring.....                            | 243        |
| Automation.....                            | 243        |
| <b>Final Words for the Appendices.....</b> | <b>244</b> |
| <b>References.....</b>                     | <b>245</b> |
| <b>AUTHOR.....</b>                         | <b>246</b> |

# PREFACE

**Alhamdulillah** to **Allah SWT**, whose blessings have made the completion of this book possible. The author expresses sincere appreciation to colleagues, students, and the open-source community, especially the academic environment of **Institut Teknologi Tangerang Selatan (ITTS)** and the OnnoCenter ecosystem, whose support and collaboration inspired the development of this work.

Ubuntu Server has evolved significantly—from a traditional **on-premise Linux server** into a **cloud-native platform** powering modern infrastructure. Today it runs across data centers, cloud platforms, edge systems, and containerized workloads. The release of **Ubuntu 24.04 LTS** continues this evolution by providing long-term stability, security, and strong support for automation and cloud-based deployments.

This book is structured as a **hands-on, scenario-based, and exam-aligned guide** to practical Ubuntu Server administration. The labs can be performed using multiple environments: **VirtualBox or KVM, cloud virtual machines (AWS, Azure, GCP)**, or **bare-metal mini servers**.

The primary audience includes **Junior System Administrators, DevOps beginners, IT students**, and **Canonical Academy certification candidates** who want practical skills in modern server infrastructure.

The goal of this book is simple: to help readers move beyond basic commands and begin thinking like **modern system engineers—designing, automating, and operating reliable Ubuntu Server infrastructure**.

Jakarta, February 2026

**Author**

# PART I — FOUNDATIONS OF UBUNTU SERVER

## Chapter 1 — Understanding Ubuntu Server Architecture

### Introduction: Beyond “Just Another Linux”

Ubuntu Server is not simply Ubuntu Desktop without a graphical interface. It represents a deliberate architectural philosophy designed for **stability, automation, scalability, and long-term production reliability**.

In this chapter, we will explore not only the technical differences between Ubuntu Desktop and Ubuntu Server, but also the deeper principles that shape modern Ubuntu Server design:

- **Minimalism**
- **Immutable infrastructure direction**
- **Cloud-first architecture**
- **Service-oriented system control via systemd**

Understanding these principles early will shape how you think about servers—not as machines you “use,” but as systems you **design, deploy, maintain, and reproduce**.

### Ubuntu Desktop vs Ubuntu Server

At first glance, Ubuntu Desktop and Ubuntu Server share the same core:

- Same Linux kernel
- Same APT package management system
- Same repositories
- Same LTS lifecycle

The difference lies in **purpose and default design philosophy**.

#### Ubuntu Desktop

Ubuntu Desktop is optimized for:

- Interactive graphical use
- Productivity applications
- Multimedia
- Personal computing

It includes:

- GNOME desktop environment
- GUI tools
- Display manager
- Audio stack

- Consumer-oriented software

## Ubuntu Server

Ubuntu Server is optimized for:

- Headless operation (no graphical interface by default)
- Remote administration (SSH-first design)
- Service deployment
- Resource efficiency
- Automation and reproducibility

It includes:

- Minimal base system
- No desktop environment
- Server-focused packages
- Subiquity installer
- Cloud-init integration

## Why Minimal Matters

Imagine two systems:

- A laptop with 200 background services
- A production server with only 12 essential services

Which one is easier to secure, monitor, and debug?

**The fewer moving parts, the smaller the attack surface.**

Minimalism in Ubuntu Server means:

- Less memory usage
- Faster boot times
- Fewer security vulnerabilities
- Cleaner logs
- Easier troubleshooting

This philosophy is fundamental to modern infrastructure engineering.

## LTS Lifecycle & Extended Security Maintenance (ESM)

One of Ubuntu Server's strongest advantages in production environments is its **Long Term Support (LTS)** lifecycle.

What is LTS?

An LTS release provides:

- **5 years of standard security updates**
- Enterprise-grade stability
- API consistency
- Kernel support

For example:

- Ubuntu 24.04 LTS (Noble Numbat)
  - Standard support until 2029
  - ESM available beyond that

## Extended Security Maintenance (ESM)

After the initial 5 years:

- Canonical provides additional security patches
- Critical packages remain protected
- Organizations can extend lifecycle without forced upgrades

This matters in enterprise environments where:

- Compliance requires long support windows
- Upgrades must be scheduled carefully
- Stability is more important than novelty

**Production servers should prioritize predictability over rapid change.**

## The Package Ecosystem: deb, snap, and Container Images

Ubuntu Server operates in a hybrid packaging ecosystem.

Understanding this ecosystem is critical for modern administrators.

### 1. deb Packages (APT)

Traditional Debian packages:

- Installed via `apt install`
- Managed in `/usr`
- Integrated deeply into system libraries

Advantages:

- Mature
- Lightweight
- Integrated dependency resolution
- Ideal for core system services

Example:

```
apt install nginx
```

Think of deb packages as **native system components**.

## 2. Snap Packages

Snaps are:

- Self-contained application packages
- Sandboxed
- Transactional
- Automatically updatable

Advantages:

- Reduced dependency conflicts
- Isolation from host system
- Ideal for fast-moving applications

Example:

```
snap install nextcloud
```

Snaps behave more like **application containers without full container overhead**.

## 3. Container Images

Ubuntu is also distributed as:

- Docker images
- OCI images
- Cloud base images

These images are:

- Immutable
- Designed for orchestration
- Used in Kubernetes environments

Example:

```
docker pull ubuntu:24.04
```

In modern infrastructure, containers represent the *ephemeral server paradigm*.

## systemd as PID 1 — The Modern Init System

At the heart of Ubuntu Server is **systemd**, which acts as **PID 1**.

PID 1 is the first process started by the Linux kernel.

It controls:

- Service initialization
- Dependency ordering
- Logging
- Timers
- Targets (runlevels equivalent)

Modern Ubuntu no longer uses legacy init systems such as System V.

### Why systemd Matters

systemd introduces:

- Parallel service startup
- Dependency-aware service ordering
- Integrated logging via `journalctl`
- Service isolation
- Restart policies
- Resource control

Example:

```
systemctl status nginx
```

You are not just “starting a program.”

You are interacting with a **unit model architecture**.

### Conceptual Illustration

Think of systemd as an orchestra conductor.

- Each service = musician
- Dependencies = musical timing
- Targets = performance mode
- systemd = conductor ensuring harmony

Without coordination, services would start randomly.

With systemd, boot becomes predictable and traceable.

# Ubuntu Server in Modern Infrastructure

Ubuntu Server is no longer just a rack-mounted data center OS. It operates across multiple environments.

## 1. Ubuntu Server in the Cloud

Ubuntu dominates public cloud platforms:

- AWS
- Azure
- Google Cloud
- OpenStack

Why?

- Cloud-optimized images
- Cloud-init integration
- Lightweight footprint
- Strong Canonical support

Cloud-init allows:

- Automated user creation
- SSH key injection
- Network configuration
- Package installation at boot

Cloud instances are often **ephemeral**.

This aligns with Ubuntu's minimal and automated design.

## 2. Ubuntu at the Edge

Edge computing requires:

- Low latency
- Small hardware footprint
- Reliability in constrained environments

Ubuntu Server runs on:

- ARM devices
- Industrial gateways
- Edge AI systems

Minimal installations reduce:

- Power consumption
- Update size
- Failure surface

### 3. Ubuntu in Containers

Ubuntu provides:

- Official base container images
- Secure minimal runtime images
- OCI-compatible builds

Containerized Ubuntu is:

- Immutable
- Reproducible
- Orchestratable

This reflects the broader movement toward **immutable infrastructure**.

### 4. Ubuntu as a Hypervisor Host

Ubuntu Server is frequently used as:

- KVM host
- LXD/LXC host
- Virtualization platform

It can manage:

- Virtual machines
- Containers
- Nested infrastructure

In this scenario, Ubuntu is not the workload—it is the **platform layer**.

## The Cloud-First and Immutable Direction

Modern Ubuntu Server embraces a design direction that aligns with *infrastructure as code* principles.

Instead of:

- Manually configuring servers
- Modifying them over years
- Treating them as pets

We now:

- Deploy from templates
- Automate configuration
- Replace instead of repair
- Treat servers as cattle, not pets

Ubuntu supports this direction through:

- Autoinstall YAML
- Cloud-init
- Snap transactional updates
- Container images
- systemd service isolation

This architectural direction prepares administrators for:

- DevOps environments
- Cloud-native workloads
- Automated CI/CD pipelines

## Why This Matters for You

Understanding Ubuntu Server architecture is not about memorizing commands.

It is about understanding:

- Why minimalism improves security
- Why systemd centralizes control
- Why LTS ensures enterprise stability
- Why containers and cloud-init change deployment thinking
- Why automation is not optional anymore

If you approach Ubuntu Server as:

“Just Linux without a GUI”

you will miss its strategic power.

If you approach it as:

“A reproducible, automatable, cloud-native operating system platform”

you will begin thinking like a modern system engineer.

## Chapter Summary

In this chapter, we learned that:

- Ubuntu Server is **minimal by design**
- LTS releases provide **long-term production stability**
- The packaging ecosystem includes **deb, snap, and containers**
- systemd acts as **PID 1 and service orchestrator**
- Ubuntu operates across:
  - Cloud
  - Edge
  - Containers
  - Hypervisors
- Modern Ubuntu aligns with **immutable infrastructure principles**

## Reflection Challenge

Before moving to the next chapter, ask yourself:

- Could I explain why systemd is better than legacy init?
- Do I understand when to use deb vs snap vs container?
- Can I describe why LTS is critical in production?
- Do I see servers as static machines or reproducible units?

This mindset shift is the foundation of everything that follows.

# Chapter 2 — Deploying Ubuntu Server (Exam Domain ~27%)

## 2.1 Installation Fundamentals

### Introduction: Installation Is Architecture

Installing Ubuntu Server is not merely a technical step—it is the foundation of your system's reliability, security, scalability, and lifecycle management.

In modern infrastructure, **installation decisions affect performance, security posture, upgrade flexibility, and disaster recovery readiness**. A careless installation creates future problems. A well-designed installation creates long-term stability.

In this section, we will explore:

- The **Subiquity installer** (modern Ubuntu Server installer)
- Guided vs. manual partitioning strategies
- Secure Boot and UEFI considerations

By the end, you will not just “install Ubuntu.”

You will understand how to **design an installation intentionally**.

### The Subiquity Installer

#### The Modern Ubuntu Server Installer

Ubuntu Server 24.04 LTS uses **Subiquity**, Canonical's modern server installer. It replaces the legacy Debian-based text installer and aligns Ubuntu with modern deployment practices.

Subiquity is designed for:

- Simplicity
- Automation compatibility
- Cloud-init integration
- LVM and RAID support
- UEFI-aware installations
- Autoinstall YAML compatibility

## Why Subiquity Matters

Subiquity is not just an installer—it is part of Ubuntu’s cloud-first philosophy.

It enables:

- Interactive installation
- Fully automated installation (via *autoinstall*)
- Enterprise-ready disk configuration
- Secure Boot compatibility
- Headless-friendly operation

This installer is optimized for environments where:

- Servers are deployed repeatedly
- Infrastructure is automated
- Disk layouts must scale

## Installation Flow Overview

A typical Subiquity installation includes:

1. Language and keyboard configuration
2. Network detection and configuration
3. Proxy and mirror configuration
4. Storage configuration
5. User creation
6. SSH setup
7. Optional snaps
8. Final installation and reboot

Each step should be approached strategically.

## Guided vs Manual Partitioning

Disk layout decisions impact:

- Performance
- Security
- Backup strategy
- Upgrade flexibility
- Disaster recovery options

Let us examine both approaches.

## Guided Partitioning

### Simplicity and Speed

Guided partitioning is ideal when:

- Deploying lab systems
- Creating cloud instances
- Rapid provisioning is required
- Disk layout complexity is low

Subiquity offers options such as:

- Use entire disk
- Use LVM (recommended)
- Use encrypted LVM

### Why LVM Is Often Recommended

Using LVM during guided installation allows:

- Logical volume resizing
- Snapshot creation
- Flexible storage growth

Think of LVM as:

A flexible container system for your storage.

Instead of fixed partitions, you create expandable volumes.

### Example Guided LVM Layout

|       |                           |
|-------|---------------------------|
| /boot | (separate partition)      |
| /     | (logical volume)          |
| /var  | (optional logical volume) |
| /home | (optional logical volume) |
| swap  | (logical volume or file)  |

This layout is production-friendly while remaining simple.

## Manual Partitioning

### Precision and Control

Manual partitioning is ideal when:

- Deploying enterprise systems
- Using multiple disks
- Implementing RAID
- Designing high-availability servers
- Separating data workloads

Manual layout allows you to:

- Separate `/var` for logs
- Separate `/home`
- Separate `/srv` for service data
- Create RAID arrays
- Design performance-optimized storage

### When Manual Partitioning Is Critical

For example:

A database server should separate:

- `/`
- `/var/lib/mysql`
- `/var/log`

Why?

Because:

- Logs can grow unpredictably
- Database writes are heavy
- System stability must be isolated from data growth

If logs fill `/`, the entire system can crash.

Separation improves resilience.

## Practical Comparison

| Feature      | Guided         | Manual             |
|--------------|----------------|--------------------|
| Speed        | Fast           | Slower             |
| Complexity   | Low            | High               |
| Flexibility  | Moderate (LVM) | Maximum            |
| Best For     | Labs, Cloud    | Production servers |
| RAID Support | Limited        | Full control       |

## Best Practice Recommendation

For production:

- Use **LVM**
- Consider separating `/var`
- Use manual partitioning if:
  - Database workload
  - Large log systems
  - High storage I/O

For cloud instances:

- Guided LVM is usually sufficient

## Secure Boot & UEFI

Modern servers no longer rely on legacy BIOS.

They use:

- **UEFI firmware**
- GPT partition tables
- Secure Boot mechanisms

Understanding this is critical for modern deployments.

## What Is UEFI?

UEFI (*Unified Extensible Firmware Interface*) replaces traditional BIOS.

It provides:

- Faster boot
- Secure Boot support
- Larger disk support (GPT)
- Modern firmware interface

Ubuntu Server fully supports UEFI.

## What Is a Secure Boot?

Secure Boot ensures:

- Only signed bootloaders and kernels can execute
- Malware cannot hijack the boot chain
- Boot integrity is maintained

Ubuntu kernels are signed by Canonical.

When Secure Boot is enabled:

- The system verifies the bootloader
- The bootloader verifies the kernel
- The kernel verifies modules

This is called the **chain of trust**.

## Illustration: Boot Chain Trust

```
UEFI Firmware
  ↓
Shim (signed by Microsoft)
  ↓
GRUB (signed by Canonical)
  ↓
Linux Kernel (signed)
  ↓
systemd (PID 1)
```

If any component fails verification, boot stops.

This prevents rootkits from inserting malicious boot code.

## When Should Secure Boot Be Used?

Enable Secure Boot when:

- Running production servers
- Operating in enterprise environments
- Compliance is required
- Physical access risk exists

Disable Secure Boot only when:

- Using custom unsigned kernels
- Running specialized lab environments

## GPT and Disk Partition Tables

UEFI systems typically use:

- GPT (GUID Partition Table)

Advantages of GPT:

- Supports disks larger than 2TB
- More partitions
- Redundant partition tables
- More robust than MBR

Ubuntu Server automatically configures:

- EFI System Partition (ESP)
- Proper GPT layout

## Installation Strategy: Production Mindset

Before clicking “Install,” ask:

- Is this system:
  - Cloud-based?
  - Database-heavy?
  - Log-intensive?
  - Virtualization host?
- Do I need:
  - LVM?
  - RAID?
  - Encryption?
- Is Secure Boot required?
- Is UEFI enabled?

Installation is the first architectural decision.

## Encryption Considerations

Subiquity allows:

- Full disk encryption (LUKS)
- Encrypted LVM

Encryption protects:

- Data at rest
- Physical disk theft risk
- Compliance requirements

Trade-off:

- Slight performance overhead
- Requires passphrase at boot (unless automated)

## Practical Installation Scenario

### Scenario 1 — Cloud Web Server

Recommended:

- Guided LVM
- UEFI
- Secure Boot enabled
- SSH server installed
- No separate `/home`

Why?

Because:

- Ephemeral instance
- Stateless workload
- Easily redeployable

### Scenario 2 — On-Prem Database Server

Recommended:

- Manual partitioning
- Separate `/var/lib/mysql`
- RAID 1 or RAID 10
- LVM for flexibility

- Secure Boot enabled
- Possibly encrypted disks

Why?

Because:

- Critical persistent data
- Heavy disk usage
- Must avoid root filesystem saturation

## Installation as a Learning Experience

Installation teaches:

- How Linux boots
- How partitions work
- How storage hierarchy functions
- How security is enforced from firmware upward

Treat installation as:

Your first systems engineering exercise.

## Chapter Summary

In this section, we learned:

- Subiquity is Ubuntu's modern server installer
- Guided partitioning is fast and simple
- Manual partitioning provides precision
- LVM adds flexibility
- UEFI replaces legacy BIOS
- Secure Boot ensures boot chain integrity
- GPT supports modern storage standards

Most importantly:

**Installation is architecture.**

A server that begins with clarity will remain stable under pressure.

## Reflection Challenge

Before proceeding:

- Can you explain why LVM is preferred in production?
- Can you describe how Secure Boot protects a system?

- Do you know when to choose manual partitioning?
- Can you draw the boot chain trust model from memory?

If you can answer these confidently, you are beginning to think like a production system administrator.

## 2.2 Advanced Deployment

### Introduction: From Installation to Automation

In the previous section, we explored interactive installation using Subiquity. That approach works well for labs, small environments, or initial testing. However, modern infrastructure rarely relies on manual installation.

Today's production environments demand:

- Repeatability
- Automation
- Scalability
- Consistency

If you manually install ten servers, you will end up with ten slightly different systems. If you automate deployment, you get ten identical, predictable systems.

This section introduces the advanced deployment techniques that define modern Ubuntu Server administration:

- **Autoinstall (YAML-based automation)**
- **PXE network boot deployment**
- **Cloud image provisioning**
- **Architecture considerations (ARM64 vs x86\_64)**

By the end of this chapter, you will understand how Ubuntu Server transitions from a single machine OS to a scalable infrastructure platform.

### Autoinstall YAML

#### Cloud-Init Style Automated Installation

Ubuntu Server 24.04 uses **Subiquity with Autoinstall**, a powerful automation method that enables fully unattended installations using a YAML configuration file.

Autoinstall integrates with *cloud-init*, Ubuntu's initialization system used widely in cloud environments.

Instead of answering questions during installation, you provide answers in advance.

## Why Autoinstall Matters

Autoinstall enables:

- Fully automated provisioning
- Identical deployments across environments
- Infrastructure-as-Code workflows
- Version-controlled installation configurations
- Reduced human error

Think of Autoinstall as:

A blueprint for your server.

Instead of manually building each system, you deploy from a predefined architectural plan.

## Basic Autoinstall Structure

An autoinstall file typically looks like this:

```
#cloud-config
autoinstall:
  version: 1
  identity:
    hostname: webserver01
    username: admin
    password: "$6$hashedpassword"
  ssh:
    install-server: true
    authorized-keys:
      - ssh-rsa AAAAB3Nza...
  storage:
    layout:
      name: lvm
  packages:
    - nginx
    - curl
```

This configuration defines:

- Hostname
- Default user
- SSH configuration
- Storage layout
- Pre-installed packages

With this file, Ubuntu installs itself exactly as defined.

## Autoinstall Deployment Methods

Autoinstall can be delivered via:

- Local ISO with embedded YAML
- HTTP server
- PXE environment
- Cloud-init metadata service

This flexibility makes Ubuntu suitable for:

- Bare metal provisioning
- Virtual machine templates
- Cloud image scaling

## Illustration: Automated vs Manual Installation

Manual installation:

Admin → Installer → Click → Configure → Repeat → Repeat → Repeat

Autoinstall:

YAML Blueprint → Deploy → 100 Identical Servers

Automation transforms system administration from repetitive tasks into architectural design.

## PXE Boot Deployment

### Network-Based Installation at Scale

PXE (*Preboot Execution Environment*) allows servers to boot and install an operating system directly from the network.

This method is widely used in:

- Data centers
- Enterprise environments
- Large-scale provisioning
- Lab automation

### How PXE Works

PXE deployment requires:

- DHCP server
- TFTP server
- HTTP or NFS server (for installation files)

### Boot flow:

```
Server Power On
↓
UEFI Network Boot
↓
DHCP assigns IP
↓
TFTP loads bootloader
↓
Kernel & initrd downloaded
↓
Autoinstall configuration retrieved
↓
Ubuntu installs automatically
```

The entire installation process happens without physical media.

### Why PXE Is Powerful

PXE enables:

- Mass deployment
- Zero-touch installation
- Remote provisioning
- Hardware lab automation

Imagine powering on 50 servers in a rack and having all of them install Ubuntu automatically. That is the power of PXE combined with Autoinstall.

### When to Use PXE

PXE is ideal for:

- On-premise enterprise deployments
- Educational labs
- Server farms
- Bare metal cloud providers

It may not be necessary for:

- Single cloud instances
- Small home lab environments

## Cloud Image Deployment

### Ubuntu in Public Cloud Environments

Ubuntu is one of the most widely used Linux distributions in public cloud platforms:

- AWS
- Azure
- Google Cloud
- OpenStack

In cloud environments, you typically do not install Ubuntu interactively. Instead, you deploy **prebuilt cloud images**.

### What Is a Cloud Image?

A cloud image is:

- A minimal Ubuntu image
- Preconfigured for cloud-init
- Optimized for virtualized environments
- Lightweight and fast to boot

When you launch an instance in AWS or Azure:

- The image already contains Ubuntu
- cloud-init reads metadata
- SSH keys are injected automatically
- Networking is configured
- Optional packages can be installed

This is *infrastructure-on-demand*.

### Cloud-Init: The First Boot Engine

Cloud-init is responsible for:

- User creation
- SSH key injection
- Network configuration
- Package installation
- Script execution

Example cloud-init user-data:

```
#cloud-config
users:
  - name: devops
    groups: sudo
    shell: /bin/bash
```

```
ssh-authorized-keys:
  - ssh-rsa AAAAB3Nza...
packages:
  - docker.io
runcmd:
  - systemctl enable docker
```

This configuration runs automatically at first boot.

## Illustration: Cloud Deployment Model

### Traditional:

Buy Server → Install OS → Configure → Deploy Application

### Cloud model:

Launch Instance → Inject Config → Server Ready in Seconds

Speed and automation define modern server operations.

## ARM64 vs x86\_64 Considerations

Ubuntu Server supports multiple CPU architectures.

The two most common are:

- **x86\_64 (AMD64)**
- **ARM64 (AArch64)**

Understanding architecture differences is critical when designing deployments.

### x86\_64

Traditional server architecture:

- Intel Xeon
- AMD EPYC
- Most data center hardware

Advantages:

- Mature ecosystem
- Broad software compatibility
- Extensive enterprise support

Best suited for:

- Enterprise workloads
- Virtualization hosts

- Legacy application environments

## ARM64

Modern alternative architecture:

- AWS Graviton
- Raspberry Pi
- Edge computing devices
- Energy-efficient servers

Advantages:

- Lower power consumption
- High performance-per-watt
- Increasing cloud adoption

Challenges:

- Some legacy software not compiled for ARM
- Requires architecture-specific container images

## Architecture Comparison

| Feature                | x86_64    | ARM64              |
|------------------------|-----------|--------------------|
| Power Efficiency       | Moderate  | High               |
| Enterprise Maturity    | Very High | Growing            |
| Software Compatibility | Excellent | Good               |
| Edge Deployment        | Limited   | Excellent          |
| Cloud Adoption         | Very High | Rapidly Increasing |

## Practical Decision Guidance

Choose x86\_64 when:

- Running legacy enterprise software
- Hosting traditional virtualization
- Compatibility is critical

Choose ARM64 when:

- Deploying edge systems
- Optimizing for energy efficiency
- Running cloud-native workloads

## Modern Deployment Mindset

Advanced deployment techniques reflect a deeper philosophy:

Servers should be:

- Reproducible
- Disposable
- Automated
- Consistent

Instead of:

- Manually configured systems
- Snowflake servers
- Irreplaceable machines

Modern Ubuntu Server supports:

- Immutable infrastructure
- Cloud-native deployment
- Infrastructure as Code
- Automated scaling

This is not just technology. It is a shift in thinking.

## Chapter Summary

In this section, we learned:

- Autoinstall enables fully automated installation using YAML
- PXE boot allows large-scale network deployment
- Cloud images provide instant infrastructure provisioning
- Cloud-init automates first-boot configuration
- ARM64 and x86\_64 architectures require deployment planning

Most importantly:

**Deployment is no longer a manual task. It is an architectural discipline.**

## Reflection Challenge

Ask yourself:

- Can I deploy 20 identical servers automatically?
- Do I understand how cloud-init configures a system?
- When should I choose ARM64 over x86\_64?
- Could I design a PXE environment?

If you can answer these confidently, you are beginning to operate at a production infrastructure level.

## 2.3 Post-Install Configuration

### Introduction: Installation Is Only the Beginning

Installing Ubuntu Server is like constructing a building's foundation. Post-installation configuration is where you secure the doors, reinforce the walls, and prepare the structure for long-term operation.

Many real-world security incidents and stability failures occur **after installation**, not during it. A system that is left with default settings, outdated packages, or weak SSH configuration becomes vulnerable quickly.

In this section, we focus on:

- **Initial user and SSH configuration**
- **Safe system updating strategies**
- **Kernel management fundamentals**
- **Rescue mode and recovery shell operations**

These tasks define the transition from a fresh installation to a **production-ready server**.

### Initial User & SSH Setup

#### Securing Remote Access from Day One

Ubuntu Server is designed to be administered remotely. In production environments, you rarely attach a monitor and keyboard. Instead, you manage the system via SSH.

That means your first responsibility is to ensure:

- Proper user separation
- Secure SSH configuration
- Controlled administrative privileges

#### Understanding User Roles

During installation, Ubuntu creates a primary user. However, in professional environments, you should think in terms of roles:

- Regular user (non-privileged)
- Administrative user (sudo access)
- Service accounts (application-specific)
- Disabled root login

A production server should **not** rely solely on one shared administrative account.

## Creating Additional Users

Example:

```
sudo adduser devops
sudo usermod -aG sudo devops
```

This creates a user and grants administrative privileges via the *sudo* group.

Why this matters:

- Accountability (who executed which command)
- Separation of duties
- Audit compliance

## SSH Key-Based Authentication

Passwords are vulnerable to brute-force attacks. Production servers should prioritize **SSH key-based authentication**.

Generate a key (on client machine):

```
ssh-keygen -t ed25519
```

Copy the public key:

```
ssh-copy-id devops@server-ip
```

Then edit SSH configuration:

```
sudo nano /etc/ssh/sshd_config
```

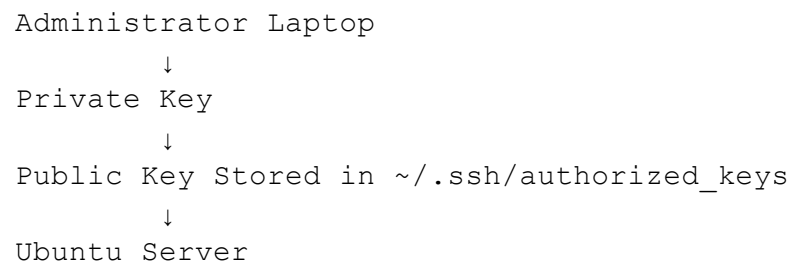
Recommended settings:

```
PermitRootLogin no
PasswordAuthentication no
PubkeyAuthentication yes
```

Restart SSH:

```
sudo systemctl restart ssh
```

### Illustration: Secure SSH Model



No password transmission occurs.

This simple setup significantly improves security posture.

### Updating the System Safely

#### Stability Before Speed

Immediately after installation, the system may not be fully up-to-date. Updating is essential—but it must be done thoughtfully.

Blindly running `apt upgrade` on production servers without planning can cause service interruptions.

#### Basic Update Process

```
sudo apt update
sudo apt upgrade
```

To include dependency changes:

```
sudo apt full-upgrade
```

#### Best Practice: Safe Updating Workflow

In production:

Review pending updates:

```
apt list --upgradable
```

Identify critical services.

Check changelogs:

```
apt changelog package-name
```

Perform updates during maintenance windows.

Restart affected services.

## Unattended Security Updates

Ubuntu supports automatic security updates via *unattended-upgrades*. Install and configure:

```
sudo apt install unattended-upgrades
sudo dpkg-reconfigure unattended-upgrades
```

This ensures:

- Security patches applied automatically
- Reduced exposure window
- Compliance alignment

However, automatic updates should still be monitored.

## Kernel Management

### Understanding the Core of the System

The Linux kernel is the heart of Ubuntu Server. It manages:

- Hardware interaction
- Memory
- CPU scheduling
- Filesystems
- Networking

Production systems must handle kernel updates carefully.

### Checking Kernel Version

```
uname -r
```

To list installed kernels:

```
dpkg --get-architecture | grep linux-image
```

### When Kernel Updates Occur

Kernel updates are typically released for:

- Security patches
- Hardware support
- Bug fixes

After kernel upgrade, reboot is required:

```
sudo reboot
```

## Safe Kernel Upgrade Strategy

In production:

- Avoid immediate reboot unless critical vulnerability
- Schedule reboot during maintenance window
- Confirm fallback kernel exists

Ubuntu retains older kernels. If something fails, you can select a previous kernel from GRUB.

Illustration: Kernel Fallback Model

```
GRUB Menu
  ↓
Kernel 6.x (current)
Kernel 6.x (previous)
Recovery Mode
```

This provides safety in case of incompatibility.

## Rescue Mode & Recovery Shell

When Things Go Wrong

Even well-managed systems can encounter:

- Boot failures
- Corrupted filesystem
- Misconfigured services
- Broken SSH configuration

Ubuntu provides multiple recovery methods.

Accessing GRUB Recovery Mode

During boot:

1. Hold Shift (BIOS) or press ESC (UEFI)
2. Select Advanced Options
3. Choose "Recovery Mode"

Recovery mode provides:

- Root shell access
- Filesystem check
- Network repair
- Package repair

## Recovery Shell Example

If SSH is misconfigured and you locked yourself out:

Boot into recovery mode

Mount root filesystem:

```
mount -o remount,rw /
```

Fix SSH configuration:

```
nano /etc/ssh/sshd_config
```

Restart SSH service

## Using Live ISO for Rescue

If the system fails to boot entirely:

- Boot from Ubuntu ISO
- Choose “Try Ubuntu”
- Mount system partitions manually
- Use **chroot** to repair system

Example:

```
sudo mount /dev/sda2 /mnt  
sudo chroot /mnt
```

This method allows deep recovery operations.

## Practical Scenario: Recovery in Production

Imagine:

- You update OpenSSH configuration.
- Restart SSH.
- Suddenly you cannot connect remotely.

What happens next?

If you have:

- Console access (IPMI / cloud console) → fix immediately.
- No console access → serious outage.

This is why:

- Always test SSH changes in a second session.
- Never close your original session until verified.
- Document recovery procedures.

## Post-Install Checklist for Production Readiness

After installation, verify:

- SSH key authentication enabled
- Root login disabled
- System fully updated
- Automatic security updates configured
- Time synchronization active
- Firewall enabled (UFW)
- Backup strategy defined

These steps transform a fresh installation into a secure operational server.

## Chapter Summary

In this section, we learned:

- Secure user creation and SSH key authentication
- Safe system update practices
- Kernel management and fallback strategy
- Recovery mode and rescue shell usage

Most importantly:

**A server is only secure and stable after proper post-install configuration.**

Installation creates the machine.

Post-install configuration defines its reliability.

## Reflection Challenge

Consider the following:

- Could you recover from a broken SSH configuration?
- Do you know how to roll back a kernel?
- Can you safely apply updates in production?
- Would your server survive a configuration mistake?

If you can answer confidently, you are progressing from installer to system engineer.

## 2.4 Release Upgrade Strategy

### Introduction: Upgrading Is a Strategic Decision

Upgrading an Ubuntu Server is not a routine command—it is a strategic operation.

A release upgrade changes:

- The base operating system version
- Core libraries
- The kernel
- Default configurations
- Security mechanisms

In development environments, upgrades may be frequent. In production environments, upgrades must be deliberate, tested, and reversible.

This section explores:

- The **do-release-upgrade** tool
- Production-safe upgrade planning
- Snapshot and rollback strategies

By the end, you will understand how to upgrade Ubuntu Server **without risking downtime or data loss**.

### Understanding Ubuntu Release Types

Before upgrading, you must understand Ubuntu's release model.

Ubuntu provides:

- **Interim releases** (every 6 months)
- **LTS releases** (every 2 years)

Production environments should almost always use **LTS versions**.

For example:

- Ubuntu 22.04 LTS → Ubuntu 24.04 LTS
- Supported for 5 years (with optional Extended Security Maintenance)

Upgrading from one LTS to the next is called an *in-place release upgrade*.

## The do-release-upgrade Tool

### Controlled Distribution Upgrade

Ubuntu provides the `do-release-upgrade` utility for performing release upgrades safely.

This tool:

- Checks system compatibility
- Disables third-party repositories
- Resolves package conflicts
- Performs staged upgrades
- Maintains logs for troubleshooting

Basic usage:

```
sudo do-release-upgrade
```

The system will:

1. Verify current release
2. Check for available upgrades
3. Download package lists
4. Prompt for confirmation
5. Upgrade core packages

### Checking Upgrade Availability

Before upgrading:

```
lsb_release -a
```

To check upgrade channel configuration:

```
sudo nano /etc/update-manager/release-upgrades
```

Ensure:

```
Prompt=lts
```

This ensures upgrades only move between LTS versions.

## Production-Safe Upgrade Planning

### Think Before You Upgrade

In production, you must never treat upgrades as routine maintenance. Instead, approach them as:

Controlled infrastructure events.

### Step 1: Assess the Environment

Before upgrading, ask:

- Is this server:
  - Hosting databases?
  - Running critical applications?
  - Acting as a hypervisor?
- Are custom repositories configured?
- Are kernel modules in use?
- Are third-party drivers installed?

Understanding your system's dependencies prevents unexpected failures.

### Step 2: Review Installed Packages

List manually installed packages:

```
apt-mark showmanual
```

Check for obsolete packages:

```
sudo apt autoremove
```

Clean unused dependencies before upgrading.

### Step 3: Backup Strategy

Before upgrading, ensure:

- Full system backup exists
- Application data is backed up
- Database dumps are verified
- Recovery procedure is documented

If you cannot restore, you should not upgrade.

#### Step 4: Staging Environment Testing

Professional upgrade planning requires:

- A staging server
- Same OS version
- Same packages
- Same configuration

Upgrade staging first.

Test:

- Application startup
- Database connectivity
- Service health
- Performance metrics

Only then proceed to production.

#### Step 5: Maintenance Window

Upgrades may require:

- Service restarts
- Kernel reboot
- Downtime

Schedule during:

- Low-traffic periods
- Approved maintenance windows

Communicate clearly with stakeholders.

#### Snapshot & Rollback Approach

##### Safety Net for Modern Infrastructure

Modern infrastructure rarely upgrades without rollback capability.

Depending on environment, you may use:

- LVM snapshots
- VM snapshots
- Cloud instance snapshots
- ZFS snapshots
- Storage-level backups

## LVM Snapshot Example

If using LVM:

```
sudo lvcreate -L 5G -s -n root_snapshot /dev/vg0/root
```

This creates a snapshot before upgrading.

If upgrade fails:

- Boot into recovery
- Revert to snapshot

Note: LVM snapshots are temporary and must be monitored for size.

## Virtual Machine Snapshot Strategy

If running Ubuntu inside:

- VMware
- KVM
- Proxmox
- Hyper-V

Take a full VM snapshot before upgrade.

Illustration:

```
VM Snapshot → Perform Upgrade →  
If Success → Delete Snapshot  
If Failure → Restore Snapshot
```

This method provides rapid rollback capability.

## Cloud Snapshot Strategy

In cloud platforms:

- Create an instance snapshot or image
- Store backup in different availability zone

Example (AWS conceptually):

```
Create AMI → Perform Upgrade →  
If Failure → Launch Old AMI
```

Cloud-native upgrades often favor **redeployment over repair**.

## Immutable Infrastructure Alternative

In cloud-native environments, in-place upgrades may not be preferred.

Instead:

1. Build a new instance with new Ubuntu version.
2. Deploy application via automation.
3. Switch traffic to new server.
4. Decommission old server.

This approach reduces upgrade risk significantly.

It aligns with *immutable infrastructure* philosophy:

Replace, don't repair.

## Handling Common Upgrade Issues

Even well-planned upgrades can encounter:

- Held packages
- Broken dependencies
- Deprecated libraries
- Configuration conflicts

Check held packages:

```
sudo apt-mark showhold
```

Check logs:

```
sudo less /var/log/dist-upgrade/main.log
```

Never ignore error messages. Logs provide clarity.

## When Not to Upgrade

Do not upgrade when:

- Vendor software does not support new release
- Critical hardware drivers are unsupported
- No tested backup exists
- No rollback method is available

Upgrading for novelty is not professionalism.

Upgrading for security, support, or lifecycle necessity is strategic.

## Upgrade Lifecycle Philosophy

Ubuntu's LTS model supports stability.

Typical production lifecycle:

Deploy LTS → Maintain 3-4 Years →  
Test Next LTS → Plan Upgrade → Execute Safely

Frequent upgrades are rarely required in enterprise environments.

Stability is often more valuable than novelty.

## Real-World Scenario

Imagine:

- You run Ubuntu 22.04 LTS.
- It hosts internal web applications.
- Support ends in two years.
- Security compliance requires supported OS.

Plan:

1. Build Ubuntu 24.04 test environment.
2. Migrate applications.
3. Test performance.
4. Take snapshots.
5. Upgrade during scheduled downtime.

This structured approach ensures reliability.

## Chapter Summary

In this section, we learned:

- **do-release-upgrade** is the official upgrade tool
- Production upgrades require planning, not improvisation
- Backups and snapshots are mandatory safeguards
- Staging environments reduce risk
- Immutable infrastructure may replace in-place upgrades

Most importantly:

**Upgrading is not a command—it is a strategy.**

A well-planned upgrade strengthens infrastructure.

A careless upgrade creates downtime.

## Reflection Challenge

Before upgrading any production server, ask:

- Do I have a tested rollback plan?
- Have I verified third-party software compatibility?
- Have I staged and tested the upgrade?
- Can I restore within acceptable downtime?

If the answer to any is “no,” pause and plan further.

Professional administrators do not fear upgrades.  
They design them.

# PART II — SYSTEM CONFIGURATION & PACKAGE MANAGEMENT

## Chapter 3 — APT, Snap & Repository Strategy

### Introduction: Package Management Is System Governance

If the Linux kernel is the heart of Ubuntu Server, then the package management system is its bloodstream.

Every application, library, security patch, and kernel update flows through:

- **APT (Advanced Package Tool)**
- **Snap package system**
- Configured software repositories

In production environments, package management is not simply about installing software. It is about:

- Stability
- Security
- Predictability
- Lifecycle control

A careless update can break services. A poorly configured repository can introduce untrusted software. A rushed upgrade can cause downtime.

This chapter builds a **professional package management mindset**.

### Understanding APT Fundamentals

APT manages traditional Debian-based packages ( `.deb` files). It works with repositories defined in:

```
/etc/apt/sources.list
/etc/apt/sources.list.d/
```

APT operations are typically divided into:

- `apt update`
- `apt upgrade`
- `apt full-upgrade`

Understanding the difference between these commands is critical.

## apt update

```
sudo apt update
```

This command:

- Downloads updated package lists
- Does **not** install anything
- Refreshes metadata about available versions

Think of **apt update** as:

Checking the catalog before buying.

It is safe and should be run before any upgrade.

## apt upgrade

```
sudo apt upgrade
```

This command:

- Installs available updates
- Does **not remove packages**
- Does not install new dependencies if removal is required

It is conservative and safer for production environments.

## apt full-upgrade

```
sudo apt full-upgrade
```

This command:

- Installs updates
- Allows dependency changes
- May remove obsolete packages
- May install new required packages

**full-upgrade** is more aggressive and should be reviewed carefully before execution in production.

## Practical Comparison

| Command          | Installs Updates | Removes Packages  | Installs New Dependencies |
|------------------|------------------|-------------------|---------------------------|
| apt update       | No               | No                | No                        |
| apt upgrade      | Yes              | No                | Limited                   |
| apt full-upgrade | Yes              | Yes (if required) | Yes                       |

In production:

- Use `apt upgrade` for routine patching.
- Use `full-upgrade` when upgrading core system components or kernel.

## Package Pinning

### Controlling What Gets Installed

In complex environments, you may have:

- Multiple repositories
- Backports
- Security repositories
- Custom internal mirrors

Package pinning allows you to:

- Control package version priority
- Prevent unintended upgrades
- Lock specific versions

Pinning configuration is stored in:

```
/etc/apt/preferences  
/etc/apt/preferences.d/
```

### Example: Pin a Specific Version

```
Package: nginx  
Pin: version 1.24.*  
Pin-Priority: 1001
```

This ensures that only version 1.24.x is installed, even if newer versions exist.

Why this matters:

- Production systems may rely on specific versions.
- Upgrading to a newer major version may break compatibility.

Package pinning is a form of **risk control**.

# Snap Services on Server

## Application Isolation with Snap

Snaps are self-contained application packages maintained by Canonical and upstream developers.

Install example:

```
sudo snap install nextcloud
```

Snap advantages:

- Application isolation
- Automatic updates
- Reduced dependency conflicts
- Sandboxing support

However, in production, automatic updates require monitoring.

## Managing Snap Services

Check snap services:

```
snap services
```

Control updates:

```
sudo snap refresh --hold
```

By default, snaps update automatically. In production, consider scheduling update windows.

Snaps are excellent for:

- Rapid deployment
- Self-contained applications
- Cloud-native services

But careful governance is required.

# Unattended-Upgrades

## Security Automation Without Chaos

Security updates must be applied quickly. However, manual updates may be inconsistent.

Ubuntu provides:

```
sudo apt install unattended-upgrades
```

Configuration file:

```
/etc/apt/apt.conf.d/50unattended-upgrades
```

This tool:

- Automatically installs security updates
- Focuses on critical patches
- Reduces exposure window

## Production Recommendation

Enable unattended-upgrades for:

- Security repositories only
- Not full distribution upgrades
- With logging enabled

Example configuration:

```
Unattended-Upgrade::Allowed-Origins {  
    "${distro_id}:${distro_codename}-security";  
};
```

This ensures only security patches are automatically installed.  
Automation improves security posture, but **visibility must be maintained**.

# Repository Trust & GPG Keys

## Verifying Software Integrity

APT uses cryptographic signatures to verify repository authenticity.

Every trusted repository must:

- Provide signed packages
- Be validated with GPG keys

Modern Ubuntu stores trusted keys in:

```
/etc/apt/trusted.gpg.d/
```

## Why Repository Trust Matters

Installing from unverified repositories can:

- Introduce malware
- Break system dependencies
- Violate compliance policies

Best practice:

- Use official Ubuntu repositories
- Use vendor-supported repositories only
- Avoid random third-party PPAs in production

Adding repository safely:

```
sudo add-apt-repository ppa:vendor/ppa
```

Always verify:

- Vendor legitimacy
- Maintenance status
- Security policy

Trust is foundational in production environments.

## Best Practice: Staging vs Production Update

Professional environments separate:

- Staging environment
- Production environment

### Staging Workflow

1. Apply updates in staging.
2. Test application behavior.
3. Monitor logs.
4. Validate performance.

Only after validation should updates be applied to production.

### Illustration: Safe Update Pipeline

```
Repository → Staging Server → Testing →  
Approval → Production Server
```

This process reduces risk dramatically.

### Why Direct Production Updates Are Dangerous

Updating directly in production can cause:

- Service restarts during business hours
- Unexpected dependency conflicts
- Kernel reboots at inconvenient times
- Broken application compatibility

Controlled rollout is professional discipline.

## Managing Reboots After Updates

Some updates require reboot.

Check if reboot is required:

```
sudo cat /var/run/reboot-required
```

In production:

- Schedule reboots
- Notify stakeholders
- Verify services post-restart

A reboot is not trivial in mission-critical systems.

## Holistic Repository Strategy

A mature repository strategy includes:

- Use LTS repositories
- Enable security repository
- Avoid mixing unstable channels
- Mirror repositories internally (enterprise environments)
- Maintain version documentation

Package governance is infrastructure governance.

## Real-World Scenario

Scenario:

A production server runs:

- Nginx
- PostgreSQL
- Custom application

An update introduces a new PostgreSQL major version.

Without pinning or staging:

- Application fails
- Downtime occurs

With proper strategy:

- PostgreSQL version pinned
- Update tested in staging
- Migration planned deliberately

The difference is not technical skill—it is strategic thinking.

## Chapter Summary

In this chapter, we explored:

- The difference between `apt update`, `upgrade`, and `full-upgrade`
- Package pinning to control version stability
- Snap services and update governance
- Unattended security automation
- Repository trust and GPG validation
- The importance of staging before production updates

Most importantly:

**Package management is not about installing software. It is about controlling change.**

In production environments, change must be:

- Planned
- Tested
- Approved
- Documented

Mastering this discipline separates administrators from system engineers.

## Reflection Challenge

Before applying updates in production, ask:

- Do I know what will change?
- Have I tested this in staging?
- Do I have rollback capability?
- Is repository trust verified?

If you answer yes confidently, you are managing infrastructure responsibly.

# PART III — SERVICE & PROCESS MANAGEMENT

## Chapter 4 — systemd Deep Dive (Modern Service Control)

### 4.1 systemd Fundamentals

#### Introduction: systemd as the Brain of Ubuntu Server

When Ubuntu Server boots, hundreds of components must start in the correct order:

- Filesystems must mount
- Networking must initialize
- Logging must activate
- Services must launch
- Dependencies must resolve

Coordinating all of this is **systemd**.

systemd is not merely a service manager. It is the **init system (PID 1)** — the first userspace process started by the Linux kernel. Everything that happens after boot is orchestrated by systemd.

Understanding systemd is essential because:

- It controls services.
- It defines boot behavior.
- It manages timers.
- It tracks dependencies.
- It integrates logging.
- It enforces isolation and restart policies.

If Ubuntu Server is a city, **systemd is the central control system managing traffic, power, and communication.**

## What Is systemd?

systemd replaces legacy *System V init* scripts and introduces a modern, declarative architecture.

Instead of sequential startup scripts, systemd uses:

- Unit files
- Dependency graphs
- Parallel startup
- Service supervision
- Event-based activation

This architecture improves:

- Boot speed
- Predictability
- Service reliability
- Observability

## Understanding Units

### The Building Blocks of systemd

In systemd, everything is a **unit**.

A unit is a configuration file that describes:

- What to start
- How to start it
- When to start it
- What it depends on

Unit files are stored in:

```
/lib/systemd/system/  
/etc/systemd/system/
```

Each unit type serves a different purpose.

## Core Unit Types

Let us examine the most important unit types in production environments.

### 1. Service Units (.service)

A *service unit* defines a background process.

Example:

```
nginx.service
```

This file contains instructions such as:

- Executable path
- Restart behavior
- Dependencies
- Environment variables

Common commands:

```
systemctl start nginx
systemctl stop nginx
systemctl restart nginx
systemctl status nginx
```

A service unit answers the question:

How should this application run?

### 2. Socket Units (.socket)

A *socket unit* enables **on-demand service activation**.

Instead of starting a service immediately at boot, systemd can:

- Listen on a network port
- Start the service only when traffic arrives

Example:

```
ssh.socket
```

This improves:

- Resource efficiency
- Startup performance
- Security (service not running unless needed)

Illustration:

```
Incoming Connection → systemd Socket → Service  
Starts Automatically
```

This is known as *socket activation*.

### 3. Timer Units (.timer)

Timers replace traditional *cron* jobs.

A timer unit schedules:

- Recurring tasks
- Delayed tasks
- Boot-based triggers

Example:

```
backup.timer
```

Timers are more flexible than cron because they:

- Integrate with dependency tracking
- Provide logging via journalctl
- Support calendar events
- Trigger after boot delays

Example command:

```
systemctl list-timers
```

Timers answer:

When should this action run?

### 4. Target Units (.target)

Targets group multiple units together.

They replace traditional Linux *runlevels*.

Examples:

- `multi-user.target`
- `graphical.target`
- `network-online.target`

A target defines a system state.

Example:

```
systemctl get-default
```

Output may be:

```
multi-user.target
```

This means the system boots into a multi-user, non-graphical state.

## Dependency Tree

### The Power of Declarative Relationships

One of systemd's most powerful features is dependency management.

Units can define relationships such as:

- Requires=
- Wants=
- After=
- Before=

This creates a **dependency graph**, not a simple startup sequence.

Example from a web server perspective:

```
nginx.service
    Requires=network.target
    After=network.target
```

This ensures:

- Network must be available
- Nginx starts only after networking is ready

### Visualizing Dependencies

You can view dependencies with:

```
systemctl list-dependencies nginx
```

Illustration:

```
network.target
  ↓
firewalld.service
  ↓
nginx.service
```

systemd ensures correct ordering automatically.

## Why Dependency Trees Matter

Without dependency control:

- Services start too early
- Failures cascade
- Debugging becomes difficult

With systemd:

- Failures are traceable
- Restart policies can be applied
- Service isolation is possible

systemd introduces **structured orchestration**, not chaos.

## Boot Targets

### Defining System Startup Modes

Traditional Linux used runlevels:

- 0 = Halt
- 1 = Single-user
- 3 = Multi-user
- 5 = Graphical

systemd replaces runlevels with targets.

### Common Boot Targets

| Target            | Purpose              |
|-------------------|----------------------|
| poweroff.target   | Shutdown             |
| rescue.target     | Single-user mode     |
| multi-user.target | Standard server mode |
| graphical.target  | Desktop mode         |

Ubuntu Server typically uses:

```
multi-user.target
```

This is appropriate for:

- Headless servers
- Production systems
- Cloud instances

### Changing Default Target

To change default boot target:

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

To temporarily switch:

```
sudo systemctl isolate rescue.target
```

### Illustration: Boot Flow with systemd

```
UEFI → GRUB → Linux Kernel →  
systemd (PID 1) →  
Initialize Targets →  
Start Services →  
System Ready
```

systemd acts as the central orchestrator.

### Service Supervision & Reliability

systemd also provides:

- Automatic restart policies
- Resource limits
- Service isolation
- Logging integration

Example restart policy:

```
Restart=always  
RestartSec=5
```

This ensures:

- If service crashes, it restarts automatically
- Downtime is minimized

This is critical for production resilience.

## Real-World Scenario

Imagine:

- A database service fails at 3 AM.
- systemd detects failure.
- Restart policy triggers.
- Service restarts automatically.
- Logs are recorded in journalctl.

No human intervention required.

This is why systemd is not just a startup manager—it is a **service supervisor**.

## Debugging with systemd

To check service status:

```
systemctl status nginx
```

To view logs:

```
journalctl -u nginx
```

systemd integrates process control and logging into a unified framework.

## Production Best Practices

When working with systemd in production:

- Avoid editing files in `/lib/systemd/system`
- Use override files in `/etc/systemd/system`
- Always reload after changes:

```
sudo systemctl daemon-reload
```

- Test changes before reboot

**systemd is powerful—but with power comes responsibility.**

## Chapter Summary

In this section, we learned:

- systemd is PID 1 and system orchestrator
- Units define how components behave
- Service, socket, timer, and target units serve different purposes
- Dependency trees ensure structured startup
- Boot targets replace legacy runlevels
- systemd supervises services and enhances reliability

Most importantly:

**systemd transforms Linux from a collection of scripts into a managed, orchestrated system.**

Mastering systemd means mastering Ubuntu Server behavior.

## Reflection Challenge

Ask yourself:

- Can I explain the difference between a service and a socket unit?
- Do I understand how dependency trees prevent startup chaos?
- Can I change the boot target safely?
- Could I debug a failing service using journalctl?

If you can answer confidently, you are thinking like a production systems engineer.

## 4.2 Managing Services

### Introduction: Service Management Is Operational Control

In Ubuntu Server, nearly everything runs as a service:

- Web servers
- Databases
- SSH
- Firewalls
- Logging systems
- Background agents

Managing services properly is one of the most important responsibilities of a system administrator.

If Chapter 4.1 introduced the architecture of `systemd`, this section focuses on **operational control** — how to manage services safely, override default behavior, and analyze failures when things go wrong.

In production environments, service management must be:

- Predictable
- Controlled
- Auditable
- Reversible

Let us explore how to achieve that.

### `systemctl` Fundamentals

#### Starting, Stopping, Enabling, and Masking

The primary interface for managing `systemd` services is:

```
systemctl
```

Understanding the difference between service actions is critical.

## Starting and Stopping Services

To start a service immediately:

```
sudo systemctl start nginx
```

To stop it:

```
sudo systemctl stop nginx
```

Important distinction:

- **start** affects the current session.
- It does **not** change boot behavior.

This means after reboot, the service may not start automatically unless enabled.

## Restarting and Reloading

Restarting:

```
sudo systemctl restart nginx
```

Reloading configuration without full restart (if supported):

```
sudo systemctl reload nginx
```

Use reload when:

- Configuration changes do not require full restart
- Minimizing downtime is important

In production, always check:

```
systemctl status nginx
```

Before and after restarting critical services.

## Enabling Services at Boot

To ensure a service starts automatically at boot:

```
sudo systemctl enable nginx
```

This creates a symbolic link in the appropriate target directory.

To disable automatic startup:

```
sudo systemctl disable nginx
```

Difference between **start** and **enable**:

| Command | Immediate Action | Affects Boot |
|---------|------------------|--------------|
| start   | Yes              | No           |
| enable  | No               | Yes          |
| restart | Yes              | No           |

## Checking Service Boot Status

```
systemctl is-enabled nginx
```

This confirms whether the service will start during boot.

In production, always verify service enablement after deployment.

## Masking Services

### Preventing Accidental Startup

Sometimes you must ensure that a service never starts.

For example:

- A conflicting service
- A legacy daemon
- A service replaced by another implementation

To prevent a service from starting, even manually:

```
sudo systemctl mask apache2
```

Masking links the service to:

```
/dev/null
```

This effectively disables it completely.

To unmask:

```
sudo systemctl unmask apache2
```

Masking is stronger than disabling

| Action  | Can Be Started Manually? | Starts at Boot? |
|---------|--------------------------|-----------------|
| disable | Yes                      | No              |
| mask    | No                       | No              |

Use masking when you want to **guarantee no accidental activation**.

## Override Files

### Customizing Service Behavior Safely

Editing service files directly in:

```
/lib/systemd/system/
```

is dangerous because updates may overwrite them.  
Instead, systemd supports **override files**.

### Creating an Override File

Use:

```
sudo systemctl edit nginx
```

This creates:

```
/etc/systemd/system/nginx.service.d/override.conf
```

This file modifies or extends service configuration.

Example override:

```
[Service]
Restart=always
RestartSec=5
```

This ensures:

- Nginx restarts automatically
- 5-second delay before restart

After editing:

```
sudo systemctl daemon-reload  
sudo systemctl restart nginx
```

Always reload systemd after modifying service definitions.

## Why Override Files Matter

Override files:

- Preserve vendor defaults
- Prevent configuration loss during upgrades
- Encourage clean configuration management
- Improve auditability

Professional administrators never modify original unit files directly.

## Analyzing Service Failures

### Debugging with Confidence

Even well-configured systems encounter failures.  
Understanding how to analyze failures is essential.

### Checking Service Status

```
systemctl status nginx
```

This command provides:

- Active state
- Exit codes
- Recent log entries
- Process ID
- Restart attempts

Example output summary:

```
Active: failed (Result: exit-code)
```

This indicates the service exited unexpectedly.

## Viewing Detailed Logs

systemd integrates logging with `journalctl`.

To view logs for a specific service:

```
journalctl -u nginx
```

To view logs from the last boot:

```
journalctl -b
```

To follow logs in real-time:

```
journalctl -u nginx -f
```

This is extremely useful during troubleshooting.

## Common Failure Scenarios

Let us explore typical failure causes.

### 1. Port Already in Use

If nginx fails to start:

```
bind() failed (98: Address already in use)
```

Solution:

```
sudo ss -tulnp | grep :80
```

Identify which process is using the port.

### 2. Configuration Error

If configuration syntax is incorrect:

```
sudo nginx -t
```

Always test configuration before restarting.

Professional habit:

Validate before restart.

### 3. Dependency Failure

If a service depends on another:

Example:

- Web server depends on database.
- Database fails.
- Web service also fails.

Check dependencies:

```
systemctl list-dependencies nginx
```

Fix underlying dependency first.

### Service Restart Policies

In production, services should not remain down silently.

systemd allows restart policies:

```
Restart=on-failure
Restart=always
RestartSec=3
```

These parameters define:

- When to restart
- How often to retry
- Delay between attempts

Proper restart configuration improves uptime automatically.

### Practical Illustration: Service Failure Flow

```
Application Crashes
↓
systemd Detects Failure
↓
Restart Policy Applied
↓
Service Restarts
↓
Logs Recorded in journalctl
```

This creates a **self-healing behavior**.

## Best Practices for Production

When managing services in production:

- Always check service status after configuration changes
- Never modify original unit files directly
- Use override files
- Validate configuration before restart
- Review logs regularly
- Implement restart policies thoughtfully
- Document customizations

Service management is not about memorizing commands.  
It is about **maintaining operational stability**.

## Real-World Scenario

Imagine:

You update a configuration file for a production web server.

You run:

```
sudo systemctl restart nginx
```

The service fails.

Because you:

- Tested configuration first?
- Kept another SSH session open?
- Checked logs immediately?

If yes — downtime avoided.

If no — outage created.

Professional service management reduces stress and prevents mistakes.

## Chapter Summary

In this section, we explored:

- Starting and stopping services
- Enabling and disabling at boot
- Masking services completely
- Using override files safely
- Analyzing failures using `systemctl` and `journalctl`
- Implementing restart policies

Most importantly:

**Service management is operational discipline.**

Mastering `systemctl` transforms you from someone who runs services into someone who controls infrastructure behavior.

## Reflection Challenge

Ask yourself:

- Can I explain the difference between `enable` and `start`?
- Do I know when to mask a service?
- Can I override service behavior safely?
- Could I debug a failing service under pressure?

If you answer confidently, you are building real production skills.

## 4.3 Timers vs Cron

### Introduction: Scheduling in a Modern Server World

Every production server performs recurring tasks:

- Backups
- Log rotation
- Cache cleanup
- Security scans
- Database maintenance
- System updates

For decades, Linux relied on **cron** for scheduling. It is simple, lightweight, and still widely used.

However, modern Ubuntu Server integrates scheduling directly into **systemd timers**, offering tighter integration, better logging, and improved dependency management.

This section explores:

- How traditional *cron* works
- How **systemd timers** function
- Why timers are considered modern scheduling
- How to migrate cron jobs to systemd timers

Understanding scheduling is not just about automation—it is about **predictable system behavior over time**.

### Traditional Cron

#### The Classic Scheduler

Cron operates using:

```
crontab -e
```

Cron entries look like this:

```
0 2 * * * /usr/local/bin/backup.sh
```

This means:

- At 02:00 AM
- Every day
- Execute backup.sh

Cron syntax:

Minute Hour Day Month Weekday Command

### Advantages of Cron

Cron is:

- Simple
- Lightweight
- Well-known
- Easy to configure

It works well for:

- Small systems
- Simple periodic tasks
- Legacy environments

### Limitations of Cron

However, cron has weaknesses:

- No built-in logging integration
- Limited dependency awareness
- No automatic restart policy
- No native awareness of system boot state
- Harder to debug

Cron jobs often fail silently unless output is redirected properly.

Example:

```
0 2 * * * /backup.sh >> /var/log/backup.log 2>&1
```

Without manual redirection, errors may go unnoticed.

## systemd Timers

### Modern Scheduling Integrated into the System

A **systemd timer** is a unit that schedules a corresponding service unit.

Unlike cron, timers are:

- Fully integrated with systemd
- Logged automatically in journalctl
- Dependency-aware
- Capable of delayed execution
- Boot-aware

Timers consist of:

- **.service** file (what runs)
- **.timer** file (when it runs)

### Basic Timer Structure

Example service file:

```
# /etc/systemd/system/backup.service
[Unit]
Description=Nightly Backup

[Service]
Type=oneshot
ExecStart=/usr/local/bin/backup.sh
```

Timer file:

```
# /etc/systemd/system/backup.timer
[Unit]
Description=Run Nightly Backup

[Timer]
OnCalendar=*-*-* 02:00:00
Persistent=true

[Install]
WantedBy=timers.target
```

Enable timer:

```
sudo systemctl enable backup.timer
sudo systemctl start backup.timer
```

## Key Differences: Cron vs systemd Timers

| Feature              | Cron     | systemd Timer            |
|----------------------|----------|--------------------------|
| Logging              | Manual   | Automatic via journalctl |
| Dependency Awareness | No       | Yes                      |
| Boot Awareness       | Limited  | Yes                      |
| Service Integration  | Separate | Unified                  |
| Restart Policies     | No       | Yes                      |
| Debugging            | Basic    | Detailed                 |

## Why systemd Timers Are Modern Scheduling

systemd timers are powerful because they:

- Understand system state
- Integrate with unit dependencies
- Automatically log execution
- Can delay execution until system is ready
- Support calendar syntax
- Support monotonic timers (time since boot)

Timers are more suitable for production environments.

## Understanding Timer Triggers

systemd supports two primary scheduling types:

### 1. Calendar-Based Timers

Example:

```
OnCalendar=Mon *-*-* 03:00:00
```

This runs every Monday at 3 AM.

You can test syntax:

```
systemd-analyze calendar "Mon *-*-* 03:00:00"
```

## 2. Monotonic Timers

These trigger relative to events like boot.

Example:

```
OnBootSec=10min
OnUnitActiveSec=1h
```

This means:

- Run 10 minutes after boot
- Then every hour after previous execution

This is extremely useful for:

- Delayed startup tasks
- Post-boot initialization
- Regular health checks

## Migrating Cron Jobs to systemd Timers

Migration is straightforward but requires structured thinking.

### Step 1: Identify Existing Cron Job

Example cron entry:

```
*/30 * * * * /usr/local/bin/cleanup.sh
```

This runs every 30 minutes.

### Step 2: Create Service Unit

```
[Unit]
Description=Cleanup Script

[Service]
Type=oneshot
ExecStart=/usr/local/bin/cleanup.sh
```

### Step 3: Create Timer Unit

```
[Unit]
Description=Run Cleanup Every 30 Minutes

[Timer]
OnCalendar=*:0/30
Persistent=true

[Install]
WantedBy=timers.target
```

#### Step 4: Enable Timer

```
sudo systemctl daemon-reload
sudo systemctl enable cleanup.timer
sudo systemctl start cleanup.timer
```

#### Viewing Active Timers

To list active timers:

```
systemctl list-timers
```

Example output:

| NEXT                    | LEFT    | LAST                    |
|-------------------------|---------|-------------------------|
| Mon 2026-01-10 02:00:00 | 5h left | Sun 2026-01-09 02:00:00 |

This provides visibility not available in traditional cron.

#### Persistent Timers

The option:

```
Persistent=true
```

Ensures that:

- If the system was offline at scheduled time
- The task runs immediately upon next boot

Cron does not handle this scenario automatically.

This is crucial for:

- Backup tasks
- Security scanning
- Maintenance operations

## Practical Illustration: Scheduling Model

### Cron model:

Time → Execute Command → Hope It Works

### systemd model:

Timer → Triggers Service →  
Service Controlled →  
Logged →  
Restart Policy Applied →  
Status Trackable

Timers integrate scheduling into infrastructure management.

## Real-World Production Example

### Imagine:

- A server reboots unexpectedly at 1:50 AM.
- Backup scheduled for 2:00 AM.
- System not fully ready at 2:00 AM.

### With cron:

- Backup may fail silently.

### With systemd timer:

- Boot delay configured.
- Timer triggers when system is ready.
- Logs recorded.
- Status visible via systemctl.

This improves reliability.

## When to Use Cron vs systemd Timer

Use cron when:

- Managing legacy systems
- Running simple personal scripts
- Minimal dependency control required

Use systemd timers when:

- Operating production servers
- Managing critical services
- Requiring detailed logging
- Integrating with systemd dependencies
- Needing boot awareness

Modern Ubuntu Server encourages timer-based scheduling.

## Best Practices for Production

- Use systemd timers for infrastructure tasks.
- Always test timer execution manually:

```
systemctl start backup.service
```

- Validate logs after execution.
- Use descriptive unit names.
- Document scheduling behavior.

Scheduling is not about automation alone.

It is about **predictable and observable automation**.

## Chapter Summary

In this section, we learned:

- Cron is simple but limited.
- systemd timers provide modern, integrated scheduling.
- Timers consist of service + timer units.
- Persistent timers handle missed executions.
- Migration from cron to timers improves reliability.

Most importantly:

**systemd timers transform scheduling from isolated commands into managed system behavior.**

## Reflection Challenge

Ask yourself:

- Can I create a timer from scratch?
- Do I understand OnCalendar syntax?
- Could I migrate a cron job safely?
- Do I know how to debug a failed timer?

If you can answer confidently, you are mastering modern Ubuntu scheduling practices.

## 4.4 Logging with journalctl

### Introduction: Logs Are the Memory of Your Server

A production server never speaks directly — it communicates through logs.

When something works, logs confirm it.

When something fails, logs explain it.

When something is compromised, logs reveal it.

In modern Ubuntu Server, logging is deeply integrated with **systemd** through the *systemd journal*. The primary tool for interacting with this journal is:

```
journalctl
```

Unlike traditional log files scattered across `/var/log`, the journal provides:

- Structured logging
- Indexed entries
- Service-level filtering
- Boot-based filtering
- Real-time monitoring

In this section, we explore:

- How journalctl works
- Persistent logging configuration
- Advanced filtering techniques
- Troubleshooting production services

Understanding logs is not optional.

**It is the difference between guessing and knowing.**

### The systemd Journal

#### A Unified Logging System

The *systemd journal* collects logs from:

- The kernel
- systemd services
- Applications
- Syslog
- Boot events

It stores them in a binary format for:

- Efficient indexing
- Structured metadata
- Reliable querying

Unlike traditional text logs, journal entries include metadata such as:

- Service name
- PID
- User
- Timestamp
- Boot ID
- Priority level

This makes analysis more powerful and precise.

### Basic Usage of journalctl

To view all logs:

```
journalctl
```

This can produce a large amount of output. In production, you rarely want everything.

### View Logs for Current Boot

```
journalctl -b
```

This shows logs since the last reboot.

To view logs from the previous boot:

```
journalctl -b -1
```

This is extremely useful when diagnosing:

- Boot failures
- Service crashes after reboot
- Kernel issues

## Persistent Logging

### Ensuring Logs Survive Reboots

By default, some systems may store logs in memory only.

To enable persistent logging:

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

Once enabled:

- Logs survive reboots
- Historical troubleshooting becomes possible
- Compliance requirements are easier to meet

You can verify journal storage mode:

```
journalctl --disk-usage
```

### Why Persistent Logging Matters

Without persistent logs:

- A crash wipes diagnostic history
- Security investigations become harder
- Root cause analysis is limited

In production systems, **persistent logging should always be enabled.**

## Filtering Logs

### Finding What Matters

Logs are only useful if you can filter them effectively.  
journalctl provides powerful filtering options.

#### Filter by Service

```
journalctl -u nginx
```

This displays logs only from the nginx service.  
Add real-time monitoring:

```
journalctl -u nginx -f
```

This acts like a live log stream.

### Filter by Time

```
journalctl --since "1 hour ago"
```

Or specific timestamps:

```
journalctl --since "2026-01-01 10:00:00"
```

This helps narrow down incidents.

### Filter by Priority Level

Log priorities:

| Level | Meaning   |
|-------|-----------|
| 0     | Emergency |
| 1     | Alert     |
| 2     | Critical  |
| 3     | Error     |
| 4     | Warning   |
| 5     | Notice    |
| 6     | Info      |
| 7     | Debug     |

To show errors only:

```
journalctl -p 3
```

To show warnings and above:

```
journalctl -p 4
```

In production, priority filtering speeds up troubleshooting dramatically.

### Filter by Boot Session

```
journalctl --list-boots
```

Then:

```
journalctl -b -2
```

This is essential when:

- Diagnosing intermittent boot issues
- Comparing behavior across reboots

## Structured Output & Formatting

`journalctl` supports output formats.

To display in short format:

```
journalctl -o short
```

To display JSON:

```
journalctl -o json
```

Structured output is useful for:

- Log aggregation systems
- Automation scripts
- Monitoring pipelines

## Troubleshooting Services with `journalctl`

Logs are the first place to look when services fail.

Let us examine practical scenarios.

### Scenario 1 — Service Fails to Start

Check service status:

```
systemctl status nginx
```

Then:

```
journalctl -u nginx
```

You may see:

```
nginx: configuration file test failed
```

Now you know the issue is configuration-related.

### Scenario 2 — Port Conflict

Log entry:

```
bind() failed (98: Address already in use)
```

Next step:

```
ss -tulnp | grep :80
```

Logs guide the troubleshooting direction.

### Scenario 3 — Permission Error

Example log:

```
Permission denied while accessing /var/www/html
```

Solution:

- Check file ownership
- Verify SELinux/AppArmor policies
- Adjust permissions

Logs do not fix the issue — they **reveal the cause**.

### Combining systemctl and journalctl

Professional troubleshooting workflow:

Check service state:

```
systemctl status service-name
```

Inspect detailed logs:

```
journalctl -u service-name
```

Filter for errors:

```
journalctl -u service-name -p 3
```

This structured approach prevents random guesswork.

### Log Retention and Disk Usage

Over time, journal logs consume disk space.

To check usage:

```
journalctl --disk-usage
```

To limit log size:

Edit:

```
sudo nano /etc/systemd/journald.conf
```

Set:

```
SystemMaxUse=500M
```

Then restart journald.

This ensures logs do not overwhelm storage.

## Integration with Traditional Logs

Ubuntu Server still maintains logs in:

```
/var/log/
```

Examples:

- syslog
- auth.log
- kern.log

journalctl often integrates with these logs, but understanding both systems is useful. Modern Ubuntu prioritizes journal-based analysis.

## Real-World Production Example

Imagine:

- A production API suddenly returns 500 errors.
- No visible crash.
- CPU usage normal.

Using journalctl:

```
journalctl -u api-service --since "10 minutes ago"
```

You discover:

```
Database connection timeout
```

Now the investigation shifts to database connectivity.  
Logs narrow your focus instantly.

## Illustration: Troubleshooting Flow

```
graph TD
    A[User Reports Problem] --> B[Check systemctl status]
    B --> C[Review journalctl logs]
    C --> D[Identify Error Pattern]
    D --> E[Fix Root Cause]
    E --> F[Verify Service Recovery]
```

Logging is the foundation of rational troubleshooting.

## Best Practices for Production Logging

- Enable persistent logging.
- Monitor disk usage regularly.
- Use filtering instead of scrolling blindly.
- Combine journalctl with monitoring tools.
- Review logs after every configuration change.
- Automate log monitoring for critical services.

Logging should be proactive, not reactive.

## Chapter Summary

In this section, we explored:

- The systemd journal architecture
- Enabling persistent logging
- Filtering logs by service, time, priority, and boot
- Troubleshooting real-world service failures
- Managing log retention safely

Most importantly:

**Logs are not noise — they are intelligence.**

A professional system administrator reads logs the way a pilot reads instruments.

## Reflection Challenge

Ask yourself:

- Can I retrieve logs from a previous boot?
- Do I know how to filter by error level?
- Could I diagnose a failed service quickly?
- Is persistent logging enabled on my server?

If the answer is yes, you are building strong operational awareness.

# PART IV — NETWORKING FOR SERVERS

## Chapter 5 — Netplan & Network Architecture

### Introduction: Networking Is the Nervous System of Your Server

A server without networking is like a brain without nerves — powerful, but isolated.

In modern production and cloud environments, networking determines:

- Accessibility
- Security
- Performance
- High availability
- Scalability

Ubuntu Server 24.04 LTS uses **Netplan** as its network configuration abstraction layer. Netplan simplifies network configuration through structured YAML files, while leveraging backend engines like *systemd-networkd* or *NetworkManager*.

This chapter will guide you through:

- Netplan YAML structure
- DHCP vs static IP configuration
- Bridges for virtualization
- Bonding for redundancy
- DNS integration with systemd-resolved
- Essential troubleshooting tools

Networking is not just configuration — it is architecture.

## Understanding Netplan

### Declarative Network Configuration

Netplan uses YAML configuration files stored in:

```
/etc/netplan/
```

Typical file:

```
01-netcfg.yaml
```

Netplan follows a **declarative approach**. Instead of issuing multiple manual commands, you describe the desired state.

## Basic Netplan YAML Structure

Example:

```
network:
  version: 2
  renderer: networkd
  ethernets:
    ens33:
      dhcp4: true
```

Key components:

- **network**: root block
- **version**: schema version
- **renderer**: backend engine
- **ethernets**: interface definitions
- **ens33**: interface name

After editing:

```
sudo netplan apply
```

To test safely:

```
sudo netplan try
```

This provides rollback if configuration fails.

## DHCP vs Static IP

### Choosing the Right Addressing Strategy

Choosing between DHCP and static addressing depends on your environment.

### DHCP (Dynamic Host Configuration Protocol)

Example:

```
network:
  version: 2
  renderer: networkd
  ethernets:
    ens33:
      dhcp4: true
```

Advantages:

- Automatic configuration
- Suitable for cloud environments
- Minimal setup
- IP managed centrally

Best used in:

- Cloud instances
- Lab environments
- Non-critical systems

## Static IP Configuration

Example:

```
network:
  version: 2
  renderer: networkd
  ethernets:
    ens33:
      dhcp4: false
      addresses:
        - 192.168.1.10/24
      routes:
        - to: default
          via: 192.168.1.1
      nameservers:
        addresses:
          - 8.8.8.8
          - 8.8.4.4
```

Static IPs are preferred for:

- Production servers
- Database systems
- Hypervisors
- Infrastructure services

Why?

Because predictability is critical.

## Practical Comparison

| Feature              | DHCP     | Static |
|----------------------|----------|--------|
| Setup Speed          | Fast     | Manual |
| Predictability       | Moderate | High   |
| Cloud-Friendly       | Yes      | Yes    |
| Production Stability | Limited  | Strong |

In production, core services should use static IP or DHCP reservation.

## Bridges for Virtual Machines

### Networking for Hypervisors

If Ubuntu Server acts as a virtualization host (KVM, LXD), you need **bridging**.

A bridge allows:

- Virtual machines to appear on the same network as the host
- External access without NAT
- Transparent connectivity

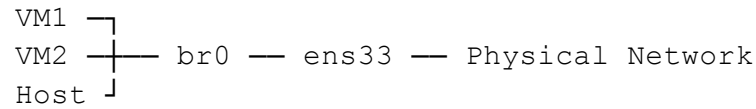
### Example Bridge Configuration

```
network:
  version: 2
  renderer: networkd
  ethernets:
    ens33:
      dhcp4: false
  bridges:
    br0:
      interfaces:
        - ens33
      addresses:
        - 192.168.1.20/24
      routes:
        - to: default
          via: 192.168.1.1
      nameservers:
        addresses:
          - 8.8.8.8
```

In this configuration:

- `ens33` becomes part of bridge `br0`
- Virtual machines attach to `br0`

Illustration:



Bridging enables seamless VM networking.

## Bonding for Redundancy

### High Availability Network Interfaces

Bonding combines multiple physical interfaces into one logical interface.

Benefits:

- Failover capability
- Increased bandwidth
- Network redundancy

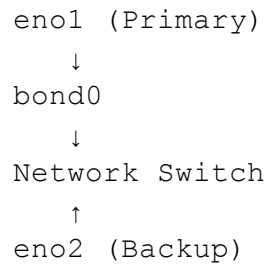
Example:

```
network:
  version: 2
  renderer: networkd
  ethernets:
    eno1: {}
    eno2: {}
  bonds:
    bond0:
      interfaces:
        - eno1
        - eno2
      parameters:
        mode: active-backup
      addresses:
        - 192.168.1.30/24
      routes:
        - to: default
          via: 192.168.1.1
```

`active-backup` mode ensures:

- One interface active
- Automatic failover if primary fails

Illustration:



Bonding is critical in:

- Data centers
- Enterprise servers
- Mission-critical systems

## DNS & systemd-resolved

### Managing Name Resolution

Ubuntu integrates DNS management via:

- systemd-resolved
- /etc/resolv.conf (managed dynamically)

To check DNS status:

```
resolvectl status
```

systemd-resolved:

- Caches DNS responses
- Supports DNS over TLS
- Manages multiple DNS servers

Example DNS configuration in Netplan:

```
nameservers:
  addresses:
    - 1.1.1.1
    - 8.8.8.8
```

Reliable DNS configuration ensures:

- Application connectivity
- API resolution
- Cloud metadata access

## Troubleshooting Networking

### Essential Diagnostic Tools

Networking problems are inevitable. The difference between panic and precision lies in knowing the right tools.

#### ss — Socket Statistics

```
ss -tulnp
```

Shows:

- Listening ports
- Active connections
- Associated processes

Use it to detect:

- Port conflicts
- Service binding issues

#### ip route — Routing Table

```
ip route
```

Displays:

- Default gateway
- Subnet routes
- Static routes

Incorrect routing causes connectivity failures.

#### tcpdump — Packet Inspection

```
sudo tcpdump -i ens33 port 80
```

Captures live traffic.

Useful for:

- Diagnosing firewall issues
- Verifying inbound requests
- Analyzing packet flow

tcpdump reveals what the network is actually doing.

## curl — Application Layer Testing

```
curl http://localhost
```

Tests:

- HTTP services
- API endpoints
- DNS resolution

curl confirms whether applications respond correctly.

## Practical Troubleshooting Flow

```
User reports service unreachable
↓
Check interface: ip addr
↓
Check routing: ip route
↓
Check listening port: ss -tulnp
↓
Check packets: tcpdump
↓
Test service: curl
```

Structured troubleshooting prevents random guessing.

## Real-World Scenario

Imagine:

- Web server is running.
- Users cannot access it.

Investigation:

1. `ss -tulnp` → Service listening on port 8080, not 80.
2. `ip route` → Default route missing.
3. `tcpdump` → No incoming traffic observed.

Logs and network tools reveal the issue.

Without systematic troubleshooting, you waste time. With proper tools, diagnosis becomes scientific.

## Chapter Summary

In this chapter, we explored:

- Netplan YAML structure
- DHCP vs static IP configuration
- Bridge networking for virtualization
- Bonding for redundancy
- DNS management via systemd-resolved
- Troubleshooting tools: ss, ip route, tcpdump, curl

Most importantly:

**Networking is not just configuration — it is architecture.**

Mastering network structure and diagnostics elevates you from installer to infrastructure engineer.

## Reflection Challenge

Ask yourself:

- Can I design a bridge for virtualization?
- Do I understand bonding modes?
- Could I troubleshoot routing issues confidently?
- Can I diagnose why a web service is unreachable?

If the answer is yes, you are developing real production networking competence.

.

# Chapter 6 — Firewall & Network Security

## Introduction: Security Is Not Optional

A freshly installed Ubuntu Server connected to the internet is immediately exposed to:

- Port scans
- Brute-force attempts
- Botnet probing
- Automated vulnerability exploitation

Security is not something you add later. It is part of the architecture.

Ubuntu provides a practical and production-friendly firewall management tool:

### UFW — Uncomplicated Firewall

Under the hood, UFW configures *nftables*/*iptables*, but it simplifies administration with clear, readable commands.

This chapter explores:

- UFW fundamentals
- Application profiles
- Basic hardening practices
- Production firewall layering concepts

Firewall configuration is not about blocking everything.  
It is about **controlling trust boundaries**.

## UFW Fundamentals

### A Simple Interface to Powerful Packet Filtering

UFW is designed to simplify firewall rule management.

Install (if not already present):

```
sudo apt install ufw
```

Check status:

```
sudo ufw status
```

By default, UFW is inactive.

## Default Policy Strategy

Best practice for production servers:

- Deny incoming connections
- Allow outgoing connections

Set default policies:

```
sudo ufw default deny incoming
sudo ufw default allow outgoing
```

This creates a **default deny stance**, a fundamental security principle.

Illustration:

```
Internet → Server
          ↓
Firewall checks rules
          ↓
Only allowed ports accepted
Everything else dropped
```

## Allowing Essential Services

Allow SSH (critical before enabling firewall):

```
sudo ufw allow OpenSSH
```

Or manually:

```
sudo ufw allow 22/tcp
```

Enable firewall:

```
sudo ufw enable
```

Verify rules:

```
sudo ufw status verbose
```

## Application Profiles

### Simplifying Service-Based Rules

Ubuntu ships with predefined application profiles located in:

```
/etc/ufw/applications.d/
```

List available profiles:

```
sudo ufw app list
```

Example output:

```
Available applications:
  OpenSSH
  Nginx Full
  Nginx HTTP
  Nginx HTTPS
```

Allowing Nginx HTTPS:

```
sudo ufw allow "Nginx HTTPS"
```

This avoids manual port configuration mistakes.

Application profiles improve:

- Readability
- Consistency
- Error reduction

## Basic Hardening

### Strengthening the Server's Defense

Firewall configuration is only one part of network hardening.  
Let us combine UFW with practical production security steps.

#### 1. Restrict SSH Access

Instead of allowing SSH globally:

```
sudo ufw allow from 192.168.1.0/24 to any port 22
```

This limits SSH access to the internal network only.

Even stronger:

- Disable password authentication
- Use SSH key-based authentication
- Change SSH port (optional, but not primary security)

## 2. Rate Limiting SSH

To prevent brute-force attempts:

```
sudo ufw limit ssh
```

This automatically blocks excessive connection attempts.

## 3. Deny Specific IP Address

```
sudo ufw deny from 203.0.113.45
```

Useful when detecting suspicious behavior.

## 4. Logging Firewall Activity

Enable logging:

```
sudo ufw logging on
```

Logs appear in:

```
/var/log/ufw.log
```

Security monitoring depends on visibility.

## Understanding Port Exposure

Check listening services:

```
ss -tulnp
```

Compare output with firewall rules.

Example:

```
LISTEN 0 128 0.0.0.0:3306
```

If MySQL listens publicly and firewall allows 3306, your database is exposed.

Always ensure:

- Only necessary ports are open
- Database services restricted to internal network

# Production Firewall Layering Concept

## Defense in Depth

In production environments, firewalling is layered.  
Do not rely on one single control.

### Layer 1 — Cloud Firewall / Security Group

Example:

- AWS Security Group
- Azure Network Security Group

Controls traffic before it reaches server.

### Layer 2 — Host Firewall (UFW)

Controls traffic at OS level.

### Layer 3 — Application-Level Security

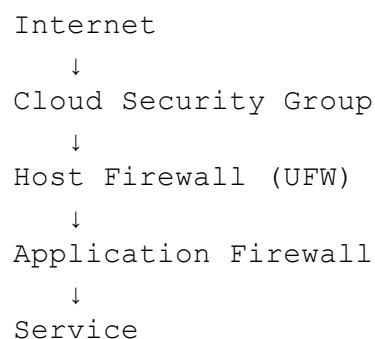
Example:

- Web application firewall (WAF)
- Reverse proxy filtering
- Rate limiting

### Layer 4 — Internal Segmentation

- VLAN separation
- Private subnets
- Zero-trust architecture

## Layered Illustration



If one layer fails, others remain active.

This is called:

**Defense in depth**

## Production Example Scenario

Scenario:

You deploy a web application.

Required services:

- Port 80 (HTTP)
- Port 443 (HTTPS)
- Port 22 (SSH, restricted)

UFW configuration:

```
sudo ufw default deny incoming
sudo ufw allow "Nginx HTTPS"
sudo ufw allow from 192.168.1.0/24 to any port 22
sudo ufw enable
```

Database (MySQL) listens on 3306 but not exposed publicly.

This ensures:

- Public web access allowed
- SSH internal only
- Database isolated

## Advanced Considerations

In high-security production environments:

- Disable IPv6 if unused
- Monitor firewall logs regularly
- Integrate logs with SIEM
- Use intrusion detection systems (IDS)
- Automate security checks

Firewall is not a one-time setup. It is ongoing governance.

## Common Mistakes to Avoid

- Enabling UFW before allowing SSH (lockout risk)
- Opening unnecessary ports
- Ignoring IPv6 rules
- Forgetting to verify rules after service changes
- Allowing database ports publicly

Security mistakes are often simple — but consequences are severe.

## Testing Firewall Configuration

After configuration:

Check rules:

```
sudo ufw status numbered
```

Test connectivity from external machine:

```
curl http://server-ip
```

Attempt blocked port:

```
telnet server-ip 3306
```

Verify expected behavior.

## Chapter Summary

In this chapter, we explored:

- UFW fundamentals
- Default deny strategy
- Application profiles
- SSH rate limiting
- Logging firewall activity
- Layered firewall architecture

Most importantly:

**Security is architectural, not reactive.**

A firewall is not just a rule list.

It is a boundary between trust and exposure.

## Reflection Challenge

Ask yourself:

- Do I know which ports are open on my server?
- Is SSH properly restricted?
- Have I implemented layered firewall protection?
- Could I detect suspicious network activity?

If the answer is yes, you are building a production-ready security mindset.

# PART V — STORAGE & FILESYSTEM MANAGEMENT

## Chapter 7 — Disk, Filesystem & Mounting

### Introduction: Storage Is the Foundation of Reliability

Applications may crash. Networks may fluctuate. Services may restart.

But if storage fails, everything fails.

Understanding disks, filesystems, and mounting strategy is one of the most critical skills in production system administration.

In Ubuntu Server, storage management involves:

- Detecting disks
- Partitioning properly
- Selecting appropriate filesystem
- Configuring persistent mounts
- Applying secure mount options

This chapter takes a **practical production-focused approach**, helping you move beyond theory into confident real-world administration.

### Identifying Disks

#### Understanding Your Storage Layout

Before making changes, always inspect existing disks.

### Using lsblk

```
lsblk
```

Example output:

| NAME   | MAJ:MIN | RM | SIZE | RO | TYPE | MOUNTPOINT |
|--------|---------|----|------|----|------|------------|
| sda    | 8:0     | 0  | 100G | 0  | disk |            |
| └─sda1 | 8:1     | 0  | 1G   | 0  | part | /boot      |
| └─sda2 | 8:2     | 0  | 99G  | 0  | part | /          |
| sdb    | 8:16    | 0  | 200G | 0  | disk |            |

This shows:

- Physical disks (`sda`, `sdb`)
- Partitions (`sda1`, `sda2`)
- Mount points

`lsblk` gives a hierarchical overview.

Using `blkid`

```
sudo blkid
```

Example:

```
/dev/sda2: UUID="a1b2c3d4" TYPE="ext4"  
/dev/sdb1: UUID="x9y8z7w6" TYPE="xfs"
```

This reveals:

- Filesystem type
- UUID
- Label

UUID is critical for stable mounting.

## Partitioning Disks

### Preparing Storage for Use

Before formatting, disks must be partitioned.

Two common tools:

- `fdisk` (MBR/GPT interactive)
- `parted` (advanced, modern)

## Using fdisk (Basic Example)

```
sudo fdisk /dev/sdb
```

Common commands inside fdisk:

| Command | Function         |
|---------|------------------|
| n       | New partition    |
| d       | Delete partition |
| p       | Print table      |
| w       | Write changes    |

Simple workflow:

1. Create new partition (n)
2. Accept defaults
3. Save (w)

## Using parted (Modern GPT-Friendly Tool)

```
sudo parted /dev/sdb
```

Example commands:

```
mklabel gpt
mkpart primary ext4 0% 100%
quit
```

parted supports:

- Large disks (>2TB)
- GPT partition tables
- Precise size allocation

In production, GPT is recommended.

## Creating Filesystems

### Choosing the Right Filesystem Type

After partitioning, create a filesystem.

## ext4 — General Purpose

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

### Advantages:

- Stable
- Mature
- Balanced performance
- Default for Ubuntu

### Best for:

- Root partitions
- General workloads

## xfs — High Performance & Scalability

```
sudo mkfs.xfs /dev/sdb1
```

### Advantages:

- Excellent for large files
- Strong scalability
- High-performance workloads

### Commonly used for:

- Database storage
- Large storage volumes
- Enterprise systems

## Practical Comparison

| Feature             | ext4              | xfs             |
|---------------------|-------------------|-----------------|
| Stability           | Excellent         | Excellent       |
| Large File Handling | Good              | Excellent       |
| Resize Support      | Yes (online grow) | Yes (grow only) |
| Enterprise Use      | Common            | Very Common     |

### In production:

- ext4 for general-purpose systems
- xfs for heavy I/O workloads

# Mounting Filesystems

## Temporary vs Persistent Mount

### Temporary Mount

```
sudo mount /dev/sdb1 /mnt/data
```

This lasts until reboot.

### Persistent Mount (Using fstab)

To mount automatically at boot, configure:

```
sudo nano /etc/fstab
```

Example entry:

```
UUID=a1b2c3d4 /data ext4 defaults 0 2
```

## fstab Deep Explanation

### Understanding Each Field

Format:

```
<device> <mountpoint> <type> <options> <dump> <pass>
```

Example:

```
UUID=a1b2c3d4 /data ext4 defaults,noatime 0 2
```

Field breakdown:

- UUID → stable device identifier
- /data → mount location
- ext4 → filesystem type
- defaults,noatime → mount options
- 0 → dump (backup flag)
- 2 → fsck order

## Why Use UUID Instead of /dev/sdb1?

Device names can change after reboot.

UUID ensures:

- Stability
- Predictability
- Safe mounting

Always prefer UUID in production.

## Testing fstab Safely

After editing:

```
sudo mount -a
```

If no errors appear, configuration is correct.  
Never reboot without testing fstab.

## Mount Options & Security

### Hardening Storage Access

Mount options influence:

- Performance
- Security
- Reliability

### Common Secure Mount Options

Example:

```
UUID=xxxx /data ext4 defaults,noexec,nosuid,nodev 0  
2
```

Explanation:

- noexec → Prevent execution of binaries
- nosuid → Disable setuid programs
- nodev → Ignore device files

These are powerful security controls.

## Example: Secure Web Upload Directory

If `/var/www/uploads` should not execute files:

```
UUID=xxxx /var/www/uploads ext4 defaults,noexec,nosuid,nodev 0 2
```

Even if an attacker uploads a malicious script, execution is blocked.

## Disk Usage Monitoring

Check disk usage:

```
df -h
```

Check inode usage:

```
df -i
```

Inodes matter especially for:

- Mail servers
- Log-heavy systems
- Small file workloads

## Practical Illustration: Storage Flow

```
Physical Disk → Partition → Filesystem → Mountpoint →  
Application
```

Each layer must be configured correctly.

A mistake in any layer may result in:

- Boot failure
- Data corruption
- Security exposure

## Real-World Production Example

Scenario:

You add a new 1TB disk for database storage.

Steps:

Verify disk:

```
lsblk
```

Partition using GPT.

Format with xfs:

```
mkfs.xfs /dev/sdb1
```

Create mount directory:

```
mkdir /dbdata
```

Add UUID to fstab.

Test with `mount -a`.

Result:

- Database storage isolated
- Predictable performance
- Persistent configuration

Professional storage planning prevents future chaos.

## Common Mistakes to Avoid

- Using device names instead of UUID
- Forgetting to test fstab
- Mixing filesystem types without planning
- Mounting without security options
- Not monitoring disk usage

Storage errors often appear only during reboot — which can be catastrophic in production.

## Chapter Summary

In this chapter, we explored:

- Detecting disks using lsblk and blkid
- Partitioning with fdisk and parted
- Filesystem selection (ext4 vs xfs)
- Deep understanding of fstab
- Secure mount options
- Production-safe mounting practices

Most importantly:

**Storage is not just space — it is structured reliability.**

When you master disk and filesystem management, you build infrastructure that survives growth, failure, and scale.

## Reflection Challenge

Ask yourself:

- Can I safely add a new disk to a production server?
- Do I understand fstab fields fully?
- Could I secure a mountpoint against code execution?
- Can I diagnose disk-related boot failure?

If yes, you are advancing toward real production-grade Linux administration.

# Chapter 8 — LVM & RAID

## Introduction: Storage Must Be Flexible and Resilient

In small environments, a single disk and a single partition may be enough.

In production environments, that approach is fragile.

Modern infrastructure demands:

- Flexible storage growth
- Online resizing
- Redundancy
- Failure tolerance
- Structured storage planning

Two critical technologies make this possible in Ubuntu Server:

- **LVM (Logical Volume Manager)**
- **Software RAID (mdadm)**

LVM provides flexibility.

RAID provides redundancy.

Together, they form the backbone of enterprise storage architecture.

## Understanding LVM

### Storage Abstraction for Flexibility

Traditional storage structure:

`Disk → Partition → Filesystem`

LVM adds an abstraction layer:

`Disk → Physical Volume → Volume Group → Logical Volume →  
Filesystem`

This abstraction allows:

- Dynamic resizing
- Disk aggregation
- Flexible allocation
- Snapshot creation

LVM transforms storage from static blocks into a **manageable pool**.

# Core LVM Concepts

## 1. Physical Volume (PV)

A Physical Volume is:

- A disk or partition initialized for LVM use

Create a PV:

```
sudo pvcreate /dev/sdb1
```

Check PVs:

```
sudo pvs
```

## 2. Volume Group (VG)

A Volume Group combines multiple physical volumes into a single storage pool.

Create VG:

```
sudo vgcreate vgdata /dev/sdb1
```

Check VGs:

```
sudo vgs
```

Think of VG as:

A storage pool from which logical volumes are carved.

## 3. Logical Volume (LV)

Logical Volumes are virtual partitions created from a VG.

Create LV:

```
sudo lvcreate -L 50G -n lvdata vgdata
```

Check LVs:

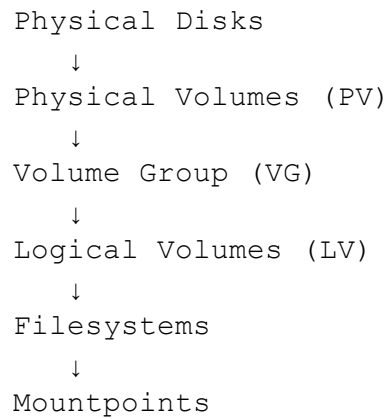
```
sudo lvs
```

After creating LV, format it:

```
sudo mkfs.ext4 /dev/vgdata/lvdata
```

Then mount normally.

## Illustration: LVM Architecture



This layered model gives administrators tremendous flexibility.

## Resizing Live Filesystems

### Growing Storage Without Downtime

One of the greatest advantages of LVM is live resizing.

Imagine:

- Application storage reaches 95%
- Downtime is unacceptable

With LVM:

1. Add new disk
2. Add it to VG
3. Extend LV
4. Resize filesystem

All without reboot.

### Step 1 — Add New Physical Volume

```
sudo pvcreate /dev/sdc1
sudo vgextend vgdata /dev/sdc1
```

### Step 2 — Extend Logical Volume

```
sudo lvextend -L +20G /dev/vgdata/lvdata
```

### Step 3 — Resize Filesystem

For ext4:

```
sudo resize2fs /dev/vgdata/lvdata
```

For xfs:

```
sudo xfs_growfs /mountpoint
```

## Result

Storage grows live.

Illustration:

```
Before: 50GB
After: 70GB
No reboot required
```

This capability is essential for:

- Database servers
- Application servers
- Growing workloads

## LVM Snapshots

### Backup & Testing Capability

Create snapshot:

```
sudo lvcreate -L 5G -s -n snapshot1 /dev/vgdata/lvdata
```

Snapshots allow:

- Backup without downtime
- Testing risky changes
- Temporary rollback

Snapshots are not permanent backups, but they are powerful operational tools.

## Software RAID with mdadm

### Building Redundancy at the OS Level

While LVM provides flexibility, it does not protect against disk failure.

For redundancy, use RAID.

Ubuntu uses **mdadm** for software RAID.

## Common RAID Levels

| RAID Level | Purpose           | Characteristics               |
|------------|-------------------|-------------------------------|
| RAID 0     | Performance       | No redundancy                 |
| RAID 1     | Mirroring         | Full redundancy               |
| RAID 5     | Parity            | Balanced                      |
| RAID 10    | Mirror + Striping | High performance & redundancy |

In production, RAID 1 and RAID 10 are common.

## Creating RAID 1 (Mirroring Example)

Assume two disks:

```
sudo mdadm --create --verbose /dev/md0 \  
  --level=1 \  
  --raid-devices=2 \  
  /dev/sdb1 /dev/sdc1
```

Check RAID status:

```
cat /proc/mdstat
```

After creation:

```
sudo mkfs.ext4 /dev/md0
```

Mount normally.

## RAID + LVM Combined Strategy

Professional setups often combine RAID and LVM:

```
Physical Disks  
  ↓  
RAID (md0)  
  ↓  
LVM Physical Volume  
  ↓  
Volume Group  
  ↓  
Logical Volumes
```

This provides:

- Redundancy (RAID)
- Flexibility (LVM)

## Enterprise Storage Mindset

### Designing for Failure

In enterprise environments:

- Disks will fail
- Storage needs will grow
- Applications will scale
- Downtime costs money

Storage design must anticipate change and failure.

### Key Enterprise Principles

1. Never rely on a single disk.
2. Always plan capacity growth.
3. Monitor disk health:

```
sudo smartctl -a /dev/sdb
```

4. Separate application data from OS.
5. Document storage layout.

## Real-World Scenario

Scenario:

You deploy a production database server.

Initial setup:

- 2 disks in RAID 1
- LVM on top
- Separate logical volumes:
  - `/`
  - `/var`
  - `/dbdata`

Later:

- Database grows beyond expected size.

Solution:

- Add new disks.
- Extend RAID or add new PV.
- Extend VG.
- Extend LV.
- Grow filesystem live.

No reinstall. No panic.

That is enterprise storage discipline.

## Common Mistakes to Avoid

- Mixing RAID levels without understanding.

Forgetting to save mdadm config:

```
sudo mdadm --detail --scan | sudo tee -a  
/etc/mdadm/mdadm.conf
```

- Creating a large single LV for everything.
- Ignoring disk health monitoring.
- Resizing without verifying backup.

Storage operations require caution and verification.

## Practical CLI Summary

Check LVM:

```
pvs  
vgs  
lvs
```

Extend LV:

```
lvextend -L +10G /dev/vgdata/lvdata  
resize2fs /dev/vgdata/lvdata
```

Check RAID:

```
cat /proc/mdstat
```

## Chapter Summary

In this chapter, we explored:

- LVM architecture (PV, VG, LV)
- Live resizing filesystems
- RAID fundamentals with mdadm
- Combining RAID and LVM
- Enterprise storage mindset

Most importantly:

**Storage must be flexible and resilient.**

LVM gives flexibility.

RAID gives redundancy.

Together, they provide operational confidence.

## Reflection Challenge

Ask yourself:

- Can I design a storage layout for a database server?
- Do I understand how to grow a filesystem live?
- Can I configure RAID 1 safely?
- Could I recover from a disk failure calmly?

If yes, you are building enterprise-grade storage competence.

# PART VI — USER, PERMISSION & ACCESS CONTROL

## Chapter 9 — User & Group Administration

### Introduction: Access Control Is Infrastructure Security

A production server is not just software and hardware — it is people.

- Developers deploy applications.
- Operators manage services.
- Automation accounts run background jobs.
- Attackers attempt unauthorized access.

User and group administration defines **who can do what** on your system.

In Ubuntu Server, proper user management ensures:

- Accountability
- Security
- Least privilege enforcement
- Compliance readiness

This chapter explores:

- `adduser` vs `useradd`
- System UID vs regular user accounts
- Understanding `/etc/passwd` and `/etc/shadow`
- Password policies
- Sudoers best practices

Mastering user administration means mastering trust boundaries.

### Understanding Users in Linux

In Linux, everything runs under a user identity:

- Applications
- Services
- Background processes
- Scheduled jobs

Every user has:

- Username
- UID (User ID)

- Group membership
- Home directory
- Login shell

Check current user:

```
whoami
```

Check user identity:

```
id
```

## adduser vs useradd

### Understanding the Difference

Ubuntu provides two commands to create users:

- **adduser** (recommended for interactive use)
- **useradd** (low-level utility)

### adduser — High-Level, Interactive

Example:

```
sudo adduser devops
```

You will be prompted for:

- Password
- Full name
- Optional details

**adduser** automatically:

- Creates home directory
- Sets permissions
- Configures default shell
- Adds user to standard groups

It is user-friendly and safe for daily administration.

## useradd — Low-Level, Manual

Example:

```
sudo useradd -m -s /bin/bash appuser
```

This requires explicit options:

- `-m` → create home directory
- `-s` → define shell

If you forget `-m`, home directory will not be created.

`useradd` is often used in:

- Automation scripts
- Configuration management tools
- Minimal deployments

### Practical Comparison

| Feature                 | adduser | useradd      |
|-------------------------|---------|--------------|
| Interactive             | Yes     | No           |
| Creates Home by Default | Yes     | No           |
| Recommended for Admin   | Yes     | Advanced Use |
| Script Friendly         | Limited | Yes          |

In production, use:

- `adduser` for human users
- `useradd` for automation scripts

## UID: System vs Regular Users

Each user has a numeric UID.

Check `/etc/passwd`:

```
cat /etc/passwd
```

Example:

```
root:x:0:0:root:/root:/bin/bash
devops:x:1001:1001:Dev Ops:/home/devops:/bin/bash
```

Key insight:

- $\text{UID} < 1000 \rightarrow$  System accounts
- $\text{UID} \geq 1000 \rightarrow$  Regular users (Ubuntu default)

## System Users

System accounts:

- Run services (e.g., `www-data`, `mysql`)
- Usually have no login shell
- Often have `/usr/sbin/nologin`

Example:

```
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
```

System users protect services from running as root.

## Understanding `/etc/passwd` and `/etc/shadow`

These two files define user authentication.

`/etc/passwd`

Contains:

- Username
- UID
- GID
- Home directory
- Login shell

Example:

```
devops:x:1001:1001:Dev Ops:/home/devops:/bin/bash
```

Fields explained:

```
username:password:UID:GID:comment:home:shell
```

The **x** means password is stored in `/etc/shadow`.

`/etc/shadow`

Viewable only by root:

```
sudo cat /etc/shadow
```

Contains:

- Encrypted password hash
- Password aging info
- Expiration dates

Example:

```
devops:$6$randomhash:19400:0:99999:7:::
```

This ensures:

- Password hashes protected
- Regular users cannot read password hashes

Security begins here.

## Password Policy

### Enforcing Strong Authentication

Ubuntu supports password policies via:

- PAM (Pluggable Authentication Modules)
- `/etc/login.defs`
- `/etc/security/pwquality.conf`

## Enforcing Minimum Password Length

Edit:

```
sudo nano /etc/security/pwquality.conf
```

Example:

```
minlen = 12
```

## Enforcing Password Expiration

Check user aging:

```
sudo chage -l devops
```

Set expiration policy:

```
sudo chage -M 90 devops
```

This forces password change every 90 days.

## Locking User Account

If suspicious activity detected:

```
sudo passwd -l devops
```

Unlock:

```
sudo passwd -u devops
```

## Group Administration

Groups allow shared access.

Create group:

```
sudo groupadd developers
```

Add user to group:

```
sudo usermod -aG developers devops
```

Check group membership:

```
groups devops
```

Groups simplify permission management.

# sudoers Best Practice

## Controlled Administrative Privileges

In Ubuntu, administrative access is granted via **sudo**.

Users in group:

```
sudo
```

Have administrative privileges.

## Adding User to sudo Group

```
sudo usermod -aG sudo devops
```

## Editing sudoers Safely

Never edit **/etc/sudoers** directly.

Use:

```
sudo visudo
```

Or create custom file:

```
sudo nano /etc/sudoers.d/devops
```

Example:

```
devops ALL=(ALL) ALL
```

## Restricting Specific Commands

Instead of full root access:

```
devops ALL=(ALL) /usr/bin/systemctl restart nginx
```

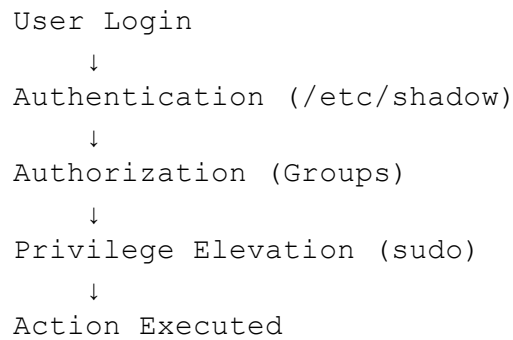
This grants:

- Limited privilege
- Reduced attack surface

Principle applied:

**Least privilege**

## Illustration: Access Control Flow



Each step defines trust boundaries.

## Real-World Production Example

Scenario:

A developer needs to:

- Restart application service
- View logs

Solution:

- Create user account
- Add to group
- Grant limited sudo for systemctl and journalctl

Result:

- Developer can manage service
- Cannot modify firewall
- Cannot access database

Access properly scoped.

## Common Mistakes to Avoid

- Using root directly for daily work
- Sharing one admin account
- Granting full sudo unnecessarily
- Ignoring password expiration
- Forgetting to remove old accounts

User management errors are security risks.

## Monitoring User Activity

Check login history:

```
last
```

Check failed login attempts:

```
sudo journalctl -u ssh
```

Audit sudo usage:

```
sudo journalctl | grep sudo
```

Visibility improves accountability.

## Chapter Summary

In this chapter, we explored:

- adduser vs useradd
- System UID vs regular users
- /etc/passwd and /etc/shadow structure
- Password policy enforcement
- Group administration
- Sudoers best practices

Most importantly:

**User management defines trust and accountability.**

A production server is secure not only because of firewalls and encryption — but because access is properly controlled.

## Reflection Challenge

Ask yourself:

- Do I understand the difference between system users and regular users?
- Can I enforce password expiration?
- Could I restrict sudo privileges to specific commands?
- Is root login disabled on my server?

If yes, you are thinking like a responsible production administrator.

# Chapter 10 — Advanced Permission & ACL

## Introduction: Permissions Define Trust

In Linux, everything is a file — and every file has permissions.

If a firewall defines network trust, file permissions define **local trust boundaries**.

Misconfigured permissions can cause:

- Data leakage
- Privilege escalation
- Service failures
- Security breaches

This chapter goes beyond basic `chmod 755` and explores:

- Advanced `chmod` usage
- `setuid` and `setgid` behavior
- Sticky bit protection
- Access Control Lists (ACL)
- `umask` mechanics

Understanding advanced permissions transforms you from a casual administrator into a **security-aware system engineer**.

## Revisiting Linux Permission Model

Standard permission structure:

```
-rwxr-xr--
```

Breakdown:

Owner Group Others

Each category has:

- `r` → read
- `w` → write
- `x` → execute

Numeric form:

| Permission | Value |
|------------|-------|
| r          | 4     |
| w          | 2     |
| x          | 1     |

Example:

```
chmod 750 script.sh
```

Means:

- Owner → rwx
- Group → r-x
- Others → none

## chmod Advanced

### Symbolic and Recursive Control

#### Symbolic Mode

Instead of numeric values:

```
chmod g+w file.txt  
chmod o-r file.txt  
chmod u+x script.sh
```

This improves readability in scripts.

#### Recursive Permission Changes

```
chmod -R 750 /var/www
```

Be careful — recursive changes may break services if applied blindly.

Professional tip:

Always verify before recursive operations:

```
ls -lR /var/www
```

## setuid, setgid, and Sticky Bit

### Special Permission Bits

Linux supports special permission bits beyond basic rwx.

#### setuid (Set User ID)

When set on executable:

- Program runs with file owner's privileges
- Not the user executing it

Example:

```
chmod u+s /usr/bin/passwd
```

Check with:

```
ls -l /usr/bin/passwd
```

Output example:

```
-rwsr-xr-x
```

Notice the **s**.

This allows normal users to change passwords even though `/etc/shadow` is root-owned.

Security note:

Improper setuid binaries are high-risk attack vectors.

#### setgid (Set Group ID)

Applied to:

- Executable files → run with group privileges
- Directories → new files inherit group ownership

Example (directory use case):

```
chmod g+s /shared/project
```

Now:

- All new files inside inherit project group

Illustration:

Directory: project (group=developers, setgid enabled)

↓

User creates file

↓

File group = developers

This is useful for team collaboration environments.

## Sticky Bit

Commonly used on shared directories like `/tmp`.

Example:

```
chmod +t /shared
```

Output:

```
drwxrwxrwt
```

Sticky bit ensures:

- Users can delete only their own files
- Even if directory is world-writable

Without sticky bit, users could delete each other's files.

## Understanding umask

### Default Permission Mask

When creating files, default permissions are:

- Files → 666
- Directories → 777

umask subtracts permissions.

Check current umask:

```
umask
```

Example output:

```
0022
```

Calculation:

```
666 - 022 = 644 (file)
777 - 022 = 755 (directory)
```

## Setting umask

Temporary:

```
umask 0027
```

Persistent (for user):

Edit:

```
~/.bashrc
```

For system-wide:

```
/etc/profile
```

In production:

- Use restrictive umask for security-sensitive systems
- Example: 027 or 077

## Access Control Lists (ACL)

### Fine-Grained Permission Control

Standard permissions allow:

- One owner
- One group
- Others

ACL extends this to:

- Multiple users
- Multiple groups
- Specific access entries

### Installing ACL Support

```
sudo apt install acl
```

Ensure filesystem supports ACL:

```
mount | grep acl
```

## Setting ACL

Example:

```
setfacl -m u:devuser:rwX file.txt
```

Check ACL:

```
getfacl file.txt
```

Example output:

```
user::rw-
user:devuser:rwX
group::r--
mask::rwX
other::---
```

## Removing ACL

```
setfacl -x u:devuser file.txt
```

## Default ACL for Directories

Set default ACL:

```
setfacl -d -m g:developers:rwX /shared
```

Now new files inherit ACL.

## Practical Illustration: Permission Hierarchy

```
Filesystem
  ↓
Ownership (user, group)
  ↓
Standard Permissions (rwX)
  ↓
Special Bits (setuid, setgid, sticky)
  ↓
ACL (fine-grained control)
```

Each layer adds control and complexity.

## Real-World Production Example

Scenario:

- Shared application directory
- Multiple developers
- Need group collaboration
- Prevent unauthorized deletion

Solution:

Create group:

```
groupadd devteam
```

Assign directory group:

```
chown -R :devteam /srv/app
```

Enable setgid:

```
chmod g+s /srv/app
```

Enable sticky bit (if shared writable):

```
chmod +t /srv/app
```

Result:

- Proper group inheritance
- File deletion protection
- Controlled collaboration

## Security Implications

Misconfigured permissions can lead to:

- Privilege escalation
- Sensitive data exposure
- Service malfunction
- Application compromise

Example dangerous scenario:

```
chmod 777 /etc
```

Never apply overly permissive rules.

Principle:

**Least privilege always**

## Common Mistakes to Avoid

- Overusing 777 permissions
- Forgetting setgid on collaborative directories
- Ignoring ACL complexity
- Not reviewing umask settings
- Setting setuid on unnecessary executables

Security mistakes in permissions often remain unnoticed until exploited.

## Monitoring Permissions

Audit world-writable files:

```
find / -type f -perm -002
```

Find setuid files:

```
find / -perm -4000
```

Review sensitive directories regularly.

## Chapter Summary

In this chapter, we explored:

- Advanced chmod usage
- setuid and setgid behavior
- Sticky bit for shared directories
- ACL fine-grained permission control
- umask behavior and impact

Most importantly:

**Permissions define trust inside the system.**

A secure production server requires disciplined permission management.

## Reflection Challenge

Ask yourself:

- Can I explain setgid directory inheritance?
- Do I understand how umask affects file creation?
- Could I configure ACL for a shared project folder?
- Can I identify risky permission configurations?

If yes, you are strengthening your security mindset.

# PART VII — SECURING SERVER ACCESS

## Chapter 11 — SSH Hardening

### Introduction: SSH Is the Front Door of Your Server

On most Ubuntu Servers, **SSH (Secure Shell)** is the primary access method. It is the gateway through which administrators deploy applications, troubleshoot systems, and manage infrastructure.

But it is also the most attacked service on the internet.

Every exposed SSH port receives:

- Brute-force attempts
- Credential stuffing attacks
- Automated scanning
- Exploit probing

If SSH is poorly configured, your entire server is at risk.

This chapter explores production-grade SSH hardening techniques:

- Key-based authentication
- Disabling root login
- Custom port configuration
- Rate limiting
- Fail2Ban integration
- Firewall coordination

Hardening SSH is not paranoia. It is a **professional responsibility**.

### Key-Based Authentication

#### Eliminating Password-Based Risk

Passwords are vulnerable to:

- Brute-force attacks
- Phishing
- Weak password reuse

The most important SSH security step is:

**Disable password authentication and use key-based authentication only.**

## Generating SSH Keys (Client Side)

On your local machine:

```
ssh-keygen -t ed25519 -C "admin@production"
```

This creates:

- Private key → `~/.ssh/id_ed25519`
- Public key → `~/.ssh/id_ed25519.pub`

## Copying Public Key to Server

```
ssh-copy-id devops@server-ip
```

Or manually append to:

```
~/.ssh/authorized_keys
```

## Enforcing Key-Only Login

Edit SSH configuration:

```
sudo nano /etc/ssh/sshd_config
```

Set:

```
PasswordAuthentication no  
PubkeyAuthentication yes
```

Restart SSH:

```
sudo systemctl restart ssh
```

Now:

- Only users with valid private keys can log in.
- Password guessing becomes ineffective.

## Disabling Root Login

### Preventing Direct Administrative Access

Allowing root login over SSH is extremely risky.

To disable:

```
sudo nano /etc/ssh/sshd_config
```

Set:

```
PermitRootLogin no
```

Restart:

```
sudo systemctl restart ssh
```

Instead:

- Use normal user
- Elevate privileges via **sudo**

Principle applied:

**Never expose root directly.**

## Custom SSH Port

### Reducing Automated Noise

By default, SSH listens on port 22.

Changing port does not replace security, but it reduces automated scanning noise.

Edit configuration:

```
sudo nano /etc/ssh/sshd_config
```

Set:

```
Port 2222
```

Restart:

```
sudo systemctl restart ssh
```

Important:

- Update firewall rules before restarting SSH
- Keep existing session open until tested

Update UFW:

```
sudo ufw allow 2222/tcp
```

Then remove port 22:

```
sudo ufw delete allow 22/tcp
```

## SSH Rate Limiting

### Slowing Down Attackers

Ubuntu's UFW supports rate limiting:

```
sudo ufw limit 2222/tcp
```

This limits repeated connection attempts.  
It reduces brute-force speed dramatically.

## Fail2Ban

### Automatic Intrusion Prevention

Fail2Ban monitors logs and bans IPs that show malicious behavior.

Install:

```
sudo apt install fail2ban
```

Enable:

```
sudo systemctl enable fail2ban  
sudo systemctl start fail2ban
```

### Basic SSH Jail Configuration

Create local config:

```
sudo nano /etc/fail2ban/jail.local
```

Example:

```
[sshd]  
enabled = true  
port = 2222  
logpath = /var/log/auth.log  
maxretry = 5  
bantime = 3600
```

This means:

- 5 failed attempts
- IP banned for 1 hour

Check status:

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

Fail2Ban integrates with a firewall to block attackers dynamically.

## Firewall Integration

### Coordinated Defense Strategy

Firewall and SSH configuration must align.

Example UFW configuration:

```
sudo ufw default deny incoming
sudo ufw allow from 192.168.1.0/24 to any port 2222
sudo ufw enable
```

This restricts SSH access to internal network only.

In cloud environments:

- Combine UFW with cloud security groups
- Only expose SSH to trusted IP ranges

## Production SSH Hardening Checklist

A production-ready SSH setup should include:

- Key-based authentication only
- Root login disabled
- Custom port (optional)
- Firewall restrictions
- Fail2Ban active
- Logging monitored

Verify with:

```
sudo systemctl status ssh
sudo ufw status
sudo fail2ban-client status
```

## Advanced Security Considerations

For high-security environments:

**Disable unused authentication methods:**

```
ChallengeResponseAuthentication no
```

**Restrict user access:**

```
AllowUsers devops admin
```

**Restrict SSH to specific IP:**

```
Match Address 192.168.1.*
```

Example restricted config:

```
AllowUsers admin@192.168.1.*
```

## Practical Illustration: Hardened SSH Flow

Client with Private Key

↓

Firewall Checks IP

↓

SSH Service on Custom Port

↓

Key Authentication Verified

↓

User Login

↓

sudo Elevation

Each layer adds protection.

## Real-World Scenario

Scenario:

A production server exposed on the internet.

Before hardening:

- Thousands of daily login attempts.
- Repeated password guessing.
- Log flooding.

After hardening:

- Password login disabled.
- Fail2Ban blocks attackers.
- Firewall restricts IP range.
- Attack attempts reduced significantly.

Security improves dramatically with minimal effort.

## Monitoring SSH Activity

Check failed attempts:

```
sudo journalctl -u ssh
```

Check active sessions:

```
who
```

Check last logins:

```
last
```

Security is continuous observation.

## Common Mistakes to Avoid

- Disabling password login before testing key access
- Changing port without updating firewall
- Forgetting to restart SSH service
- Leaving root login enabled
- Ignoring SSH logs

Always test SSH configuration before closing the active session.

## Chapter Summary

In this chapter, we explored:

- Key-based authentication
- Disabling root login
- Custom SSH port configuration
- Rate limiting with UFW
- Fail2Ban automatic banning
- Firewall integration

Most importantly:

**SSH security defines your server's first line of defense.**

Hardening SSH is not optional — it is foundational to production security.

## Reflection Challenge

Ask yourself:

- Is password authentication disabled?
- Is root login blocked?
- Are failed attempts monitored?
- Is SSH access restricted to trusted networks?

If yes, your server's front door is significantly more secure.

# PART VIII — CONFIGURING CORE SERVER SERVICES (~23%)

## Chapter 12 — File Sharing Services

### Introduction: Sharing Data Securely and Efficiently

In production environments, servers rarely operate in isolation. Data must be shared between:

- Application servers
- Backup systems
- Virtualization hosts
- Developer workstations
- Windows and Linux clients

Ubuntu Server supports two primary file sharing technologies:

- **NFS (Network File System)** — optimized for Linux/Unix environments
- **Samba (SMB/CIFS)** — ideal for Windows interoperability

Understanding when and how to deploy each service is essential for building scalable, secure file-sharing infrastructure.

This chapter covers:

- NFS setup and configuration
- Samba configuration
- Simple permission integration strategies

File sharing is not just about access — it is about **controlled collaboration**.

## NFS Setup

### Native Linux File Sharing

NFS is commonly used in:

- Linux server clusters
- Container platforms
- Virtualization storage
- HPC environments

It is lightweight, efficient, and integrates naturally with Linux permissions.

## Installing NFS Server

```
sudo apt update
sudo apt install nfs-kernel-server
```

Check service:

```
sudo systemctl status nfs-kernel-server
```

## Configuring NFS Exports

Edit:

```
sudo nano /etc/exports
```

Example configuration:

```
/srv/shared 192.168.1.0/24(rw,sync,no_subtree_check)
```

Explanation:

- `/srv/shared` → directory to share
- `192.168.1.0/24` → allowed network
- `rw` → read/write
- `sync` → write changes immediately
- `no_subtree_check` → performance improvement

Apply changes:

```
sudo exportfs -ra
```

Check exports:

```
sudo exportfs -v
```

## Mounting NFS on Client

On another Linux machine:

```
sudo mount 192.168.1.10:/srv/shared /mnt
```

To persist:

```
sudo nano /etc/fstab
```

Add:

```
192.168.1.10:/srv/shared /mnt nfs defaults 0 0
```

## Illustration: NFS Architecture

Linux Client → NFS Protocol → Ubuntu Server → Shared Directory

NFS works seamlessly with Linux UID/GID permissions.

## Samba Configuration

### Cross-Platform File Sharing

Samba allows Ubuntu Server to share files with:

- Windows clients
- macOS systems
- Mixed enterprise environments

### Installing Samba

```
sudo apt install samba
```

Check service:

```
sudo systemctl status smbd
```

### Basic Samba Share Configuration

Edit:

```
sudo nano /etc/samba/smb.conf
```

Add at bottom:

```
[Shared]
  path = /srv/shared
  browseable = yes
  read only = no
  guest ok = no
```

Restart Samba:

```
sudo systemctl restart smbd
```

### Creating Samba User

Samba requires user authentication:

```
sudo smbpasswd -a devops
```

Set Samba password.

## Testing Samba Share

From Linux:

```
smbclient -L localhost -U devops
```

From Windows:

```
\\192.168.1.10\Shared
```

## Simple Permission Integration

### Aligning Linux Permissions with Network Access

File sharing security depends on:

- Linux file ownership
- Group permissions
- Samba or NFS configuration

### Shared Group Strategy

Create group:

```
sudo groupadd sharedgroup
```

Assign directory group:

```
sudo chown -R :sharedgroup /srv/shared
```

Set group permissions:

```
sudo chmod -R 770 /srv/shared
```

Add users:

```
sudo usermod -aG sharedgroup devops
```

Enable setgid on directory:

```
sudo chmod g+s /srv/shared
```

Now:

- All new files inherit group
- Only group members have access

## Illustration: Permission Flow

User → Group Membership → Directory Ownership → Access Granted

File sharing must align with the Linux permission model.

## NFS vs Samba Comparison

| Feature         | NFS        | Samba              |
|-----------------|------------|--------------------|
| Best for        | Linux-only | Mixed environments |
| Protocol        | NFS        | SMB/CIFS           |
| Performance     | High       | Moderate           |
| Windows Support | Limited    | Native             |
| UID/GID Control | Native     | Requires mapping   |

Choose NFS for Linux clusters.

Choose Samba for Windows integration.

## Security Considerations

Never:

- Share `/` root directory
- Allow `*` network access in production
- Use `rw` without authentication
- Disable firewall rules unintentionally

Restrict NFS to trusted subnets:

```
sudo ufw allow from 192.168.1.0/24 to any port nfs
```

Restrict Samba:

```
sudo ufw allow 445/tcp
```

Firewall integration is essential.

## Real-World Production Example

Scenario:

- The development team requires a shared directory.
- Linux developers use NFS.
- Windows designers use Samba.

Solution:

- Same `/srv/projects` directory.
- Controlled group permissions.
- NFS export for internal subnet.
- Samba share with authentication.
- Firewall restricts access.

Result:

- Unified storage.
- Controlled collaboration.
- Secure access boundaries.

## Monitoring and Troubleshooting

Check NFS status:

```
sudo showmount -e localhost
```

Check Samba status:

```
sudo smbstatus
```

Test connectivity:

```
ping server-ip
```

Check open ports:

```
ss -tulnp
```

File sharing issues usually stem from:

- Permission mismatch
- Firewall restrictions
- Incorrect export configuration

## Best Practices for Production

- Use dedicated shared directories.
- Separate application data from shared data.
- Monitor disk usage.
- Log Samba access.
- Limit NFS exports to internal networks.
- Combine with ACL for advanced control.

## Chapter Summary

In this chapter, we explored:

- NFS installation and export configuration
- Samba setup for Windows integration
- Linux permission alignment
- Firewall integration
- Production best practices

Most importantly:

**File sharing is about controlled collaboration, not open exposure.**

A well-designed file-sharing service enhances productivity while maintaining security.

## Reflection Challenge

Ask yourself:

- Can I deploy both NFS and Samba correctly?
- Do I understand permission alignment?
- Can I restrict access by network?
- Could I troubleshoot a permission denial quickly?

If yes, you are building collaborative infrastructure skills.

# Chapter 13 — Web & Database Deployment

## Introduction: From Infrastructure to Application Delivery

Up to this point, we have built the foundation:

- Secure SSH access
- Proper firewall configuration
- Structured storage
- Controlled user management

Now we move into the heart of production infrastructure:

### **Deploying web servers and databases.**

In modern environments, Ubuntu Server frequently hosts:

- Web applications
- APIs
- Internal dashboards
- Microservices
- Database backends

This chapter covers:

- Apache and Nginx deployment
- MySQL and PostgreSQL setup
- Basic reverse proxy configuration
- Service management via systemd

Deploying applications is not just about installation — it is about **architecture, reliability, and scalability**.

## Web Servers: Apache vs Nginx

Ubuntu supports two dominant web servers:

- **Apache** — flexible, mature, modular
- **Nginx** — lightweight, high-performance, reverse proxy-friendly

## Installing Apache

```
sudo apt update
sudo apt install apache2
```

Check service status:

```
sudo systemctl status apache2
```

Enable at boot:

```
sudo systemctl enable apache2
```

Test in browser:

```
curl http://localhost
```

Default document root:

```
/var/www/html
```

## Installing Nginx

```
sudo apt install nginx
```

Check status:

```
sudo systemctl status nginx
```

Enable:

```
sudo systemctl enable nginx
```

Test:

```
curl http://localhost
```

## Apache vs Nginx Comparison

| Feature       | Apache        | Nginx                |
|---------------|---------------|----------------------|
| Configuration | .conf modular | Lightweight config   |
| Performance   | Good          | Excellent under load |
| Reverse Proxy | Yes           | Native strength      |
| Memory Usage  | Moderate      | Low                  |

In modern production environments:

- Nginx often acts as reverse proxy
- Apache used for compatibility-heavy workloads

## Deploying MySQL

MySQL is widely used for:

- WordPress
- Web applications
- Internal business systems

### Installing MySQL

```
sudo apt install mysql-server
```

Secure installation:

```
sudo mysql_secure_installation
```

This configures:

- Root password
- Remove test database
- Disable remote root login

### Accessing MySQL

```
sudo mysql
```

Create database:

```
CREATE DATABASE appdb;  
CREATE USER 'appuser'@'localhost' IDENTIFIED BY  
'StrongPassword';  
GRANT ALL PRIVILEGES ON appdb.* TO  
'appuser'@'localhost';  
FLUSH PRIVILEGES;
```

Check service:

```
sudo systemctl status mysql
```

## Deploying PostgreSQL

PostgreSQL is popular for:

- Enterprise systems
- Analytics
- Modern cloud-native applications

### Installing PostgreSQL

```
sudo apt install postgresql
```

Switch to postgres user:

```
sudo -i -u postgres
```

Create database and user:

```
createuser appuser  
createdb appdb
```

Access database:

```
psql appdb
```

Check service:

```
sudo systemctl status postgresql
```

## Basic Reverse Proxy

### Routing Traffic Professionally

Reverse proxy allows:

- Load balancing
- SSL termination
- Centralized routing
- Application isolation

Nginx is commonly used as a reverse proxy.

## Example: Proxy to Backend App

Assume backend running on port 5000.

Edit Nginx config:

```
sudo nano /etc/nginx/sites-available/app
```

Example:

```
server {  
    listen 80;  
    server_name example.com;  
  
    location / {  
        proxy_pass http://127.0.0.1:5000;  
        proxy_set_header Host $host;  
        proxy_set_header X-Real-IP $remote_addr;  
    }  
}
```

Enable site:

```
sudo ln -s /etc/nginx/sites-available/app  
/etc/nginx/sites-enabled/  
sudo nginx -t  
sudo systemctl restart nginx
```

## Reverse Proxy Flow Illustration

```
Client → Nginx (Port 80/443)  
    ↓  
    Proxy to Backend (Port 5000)  
    ↓  
    Application Response
```

This architecture improves:

- Security
- Flexibility
- Performance

## Service Enablement via systemd

All deployed services must integrate with systemd.

Check running services:

```
systemctl list-units --type=service
```

Enable at boot:

```
sudo systemctl enable nginx
sudo systemctl enable mysql
```

Restart service after config changes:

```
sudo systemctl restart nginx
```

View logs:

```
journalctl -u nginx
journalctl -u mysql
```

systemd ensures:

- Automatic restart
- Dependency handling
- Structured logging

## Production Deployment Workflow

Professional deployment sequence:

```
Install Web Server
    ↓
Install Database
    ↓
Configure Database Access
    ↓
Deploy Application Code
    ↓
Configure Reverse Proxy
    ↓
Secure Firewall Rules
    ↓
Enable Services at Boot
```

Never skip testing.

## Firewall Integration

Allow HTTP/HTTPS:

```
sudo ufw allow "Nginx Full"
```

Restrict database access:

```
sudo ufw deny 3306
```

Databases should not be exposed publicly.

## Real-World Production Example

Scenario:

You deploy:

- Backend API (Node.js on port 5000)
- Nginx reverse proxy
- PostgreSQL database

Architecture:

```
Internet
  ↓
Nginx (443 SSL)
  ↓
Node.js Backend (5000)
  ↓
PostgreSQL (localhost)
```

Database not exposed externally.

Reverse proxy handles TLS and routing.

Firewall restricts unnecessary ports.

This is a **secure production pattern**.

## Common Mistakes to Avoid

- Exposing database ports publicly
- Running web server as root
- Forgetting to enable services
- Skipping nginx -t before restart
- Ignoring firewall integration

Professional deployment requires verification at every step.

## Monitoring Deployment Health

Check listening ports:

```
ss -tulnp
```

Check disk usage:

```
df -h
```

Check logs:

```
journalctl -u nginx -f
```

Application stability depends on constant observation.

## Chapter Summary

In this chapter, we explored:

- Installing Apache and Nginx
- Deploying MySQL and PostgreSQL
- Configuring basic reverse proxy
- Enabling services via systemd
- Integrating firewall and service security

Most importantly:

**Web and database deployment transforms infrastructure into service delivery.**

When done properly, Ubuntu Server becomes a reliable production platform capable of serving thousands — or millions — of users.

## Reflection Challenge

Ask yourself:

- Can I deploy a secure web stack?
- Do I understand reverse proxy architecture?
- Is my database isolated properly?
- Are all services enabled and monitored?

If yes, you are transitioning from system administrator to infrastructure engineer.

# Chapter 14 — Backup & Automation

## Introduction: If It Is Not Backed Up, It Does Not Exist

In production environments, failure is not a possibility — it is an inevitability.

Disks fail.  
Humans make mistakes.  
Ransomware happens.  
Applications corrupt data.

The only real question is:

### **How fast can you recover?**

Backup and automation are not optional features. They are core operational disciplines.

This chapter explores:

- rsync-based backup strategies
- Snapshot concepts (LVM and filesystem level)
- Cron vs systemd timers for scheduling
- Building an automation mindset

A professional administrator does not just configure systems — they design for recovery.

## Backup Strategy Fundamentals

Before writing a single backup command, understand the goal:

- Protect critical data
- Minimize downtime
- Enable fast restoration
- Avoid silent failure

A good backup strategy answers:

- What to back up?
- Where to store backups?
- How often to back up?
- How to verify restore capability?

Backup is not about copying files.  
It is about **guaranteed recoverability**.

# rsync Backup Strategy

## Efficient Incremental Synchronization

**rsync** is one of the most powerful tools in Linux for backup.

It allows:

- Incremental backups
- Remote synchronization
- Efficient file comparison
- Permission preservation

## Basic rsync Command

```
rsync -av /data/ /backup/data/
```

Options:

- **-a** → archive (preserve permissions, ownership, timestamps)
- **-v** → verbose

## Remote Backup via SSH

```
rsync -av -e ssh /data/ user@backup-server:/backup/data/
```

This provides:

- Encrypted transfer
- Off-site backup capability

## Deleting Removed Files (Mirror Mode)

```
rsync -av --delete /data/ /backup/data/
```

This ensures backup matches the source exactly.

Be cautious — deleted source files will be deleted in backup.

## Incremental Snapshot Strategy with rsync

A common production pattern:

- Daily incremental backup
- Weekly full backup
- Monthly archive

Example using hard links:

```
rsync -a --delete --link-dest=/backup/previous /data/  
/backup/current
```

This creates space-efficient snapshots.

## Snapshot Concept

### Point-in-Time Protection

Backups protect against long-term loss.

Snapshots protect against immediate failure.

### LVM Snapshot Example

Create snapshot before major update:

```
lvcreate -L 5G -s -n data_snapshot /dev/vgdata/lvdata
```

This allows:

- Quick rollback
- Safe upgrade testing
- Temporary recovery point

Snapshots are short-term protection tools — not permanent backups.

### Filesystem Snapshot (Conceptual Illustration)

```
Filesystem  
↓  
Snapshot Created  
↓  
Application Modifies Data  
↓  
Original State Preserved in Snapshot
```

Snapshots are extremely useful before:

- System upgrades
- Application migrations
- Major configuration changes

## Cron vs systemd Timer

### Automating Backup Tasks

Automation ensures backups run consistently.

### Cron-Based Backup

Edit crontab:

```
crontab -e
```

Example:

```
0 2 * * * /usr/bin/rsync -a /data/ /backup/data/
```

Runs daily at 2 AM.

Simple but limited logging integration.

### systemd Timer-Based Backup

Service file:

```
[Unit]
Description=Daily Backup
[Service]
Type=oneshot
ExecStart=/usr/bin/rsync -a /data/ /backup/data/
```

Timer file:

```
[Timer]
OnCalendar=*-*-* 02:00:00
Persistent=true
```

Enable:

```
systemctl enable backup.timer
systemctl start backup.timer
```

Benefits:

- Integrated logging
- Boot awareness
- Missed run recovery

Modern Ubuntu favors **systemd timers** for production automation.

## Backup Verification

A backup without verification is a false sense of security.

Test restore:

```
rsync -av /backup/data/ /restore-test/
```

Verify integrity:

```
diff -r /data /restore-test
```

Periodic restore testing is essential.

## Basic Automation Mindset

Automation is not about complexity.

It is about consistency.

Key principles:

- Script repetitive tasks
- Log outputs
- Monitor success/failure
- Notify on errors
- Document procedures

### Example: Simple Backup Script

```
#!/bin/bash
LOGFILE="/var/log/backup.log"
rsync -a --delete /data/ /backup/data/ >> $LOGFILE 2>&1
echo "Backup completed at $(date)" >> $LOGFILE
```

Make executable:

```
chmod +x backup.sh
```

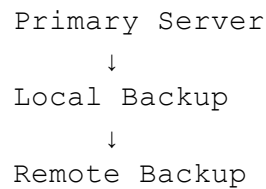
Then schedule with a systemd timer.

### 3-2-1 Backup Rule

Professional guideline:

- 3 copies of data
- 2 different media types
- 1 off-site copy

Illustration:



Redundancy reduces risk.

### Real-World Production Example

Scenario:

- Web server hosts application files
- PostgreSQL database stores user data

Strategy:

1. Nightly rsync for web files
2. Daily database dump:

```
pg_dump appdb > /backup/db/appdb.sql
```

3. Weekly full archive compressed:

```
tar -czf weekly-backup.tar.gz /backup
```

4. Off-site rsync to remote server

Result:

- Multiple recovery points
- Database integrity preserved
- Off-site protection

## Automation + Monitoring

Combine backup with monitoring:

- Log failures
- Check exit codes
- Use mail notification

Example:

```
if [ $? -ne 0 ]; then
    echo "Backup failed!" | mail -s "Backup Alert"
    admin@example.com
fi
```

Automation without monitoring is incomplete.

## Common Mistakes to Avoid

- Backing up to same disk
- Ignoring database dumps
- Not testing restore
- Running backup manually only
- Not logging backup status

Backup must be:

- Automated
- Verified
- Monitored

## Illustration: Recovery-Oriented Thinking

```
Failure Happens
    ↓
Restore from Snapshot (Short-Term)
    ↓
Restore from Backup (Long-Term)
    ↓
Service Resumes
```

Design systems assuming failure will happen.

## Chapter Summary

In this chapter, we explored:

- rsync incremental backup strategies
- LVM snapshot concepts
- Cron vs systemd timers
- Automation mindset
- Verification and monitoring practices

Most importantly:

**Backup is not about copying files — it is about guaranteeing recovery.**

Automation ensures consistency.

Verification ensures reliability.

Planning ensures resilience.

## Reflection Challenge

Ask yourself:

- Do I have automated backups?
- Have I tested restoration?
- Are backups stored off-site?
- Could I recover from disk failure calmly?

If yes, you are building operational resilience.

# PART IX — MONITORING & TROUBLESHOOTING

## Chapter 15 — Monitoring System Health

### Introduction: If You Don't Measure It, You Can't Manage It

A production server may appear “fine” — until suddenly it isn't.

Applications slow down.  
CPU spikes.  
Memory fills up.  
Disk I/O stalls.  
Users complain.

System monitoring is the art of **seeing problems before they become outages**.

In this chapter, we explore practical, built-in Linux tools for monitoring system health:

- `top` and `htop`
- `free`
- `vmstat`
- `iostat`
- `journalctl` analysis
- Basic performance interpretation

Monitoring is not about staring at numbers.

It is about **understanding behavior patterns**.

### `top` — Real-Time Process Monitoring

The `top` command provides a live view of:

- CPU usage
- Memory usage
- Running processes
- Load averages

Run:

```
top
```

Typical top display includes:

- Load average
- CPU percentage breakdown
- Memory usage
- Process list

## Understanding Load Average

Example:

```
load average: 0.52, 0.48, 0.45
```

These represent:

- 1-minute average
- 5-minute average
- 15-minute average

Interpretation rule:

If load average exceeds number of CPU cores, the system is under pressure.

Check CPU cores:

```
nproc
```

## CPU Breakdown in top

```
%Cpu(s): 10.0 us, 2.0 sy, 0.0 ni, 85.0 id, 3.0 wa
```

Meaning:

- us → user processes
- sy → system/kernel processes
- id → idle
- wa → I/O wait

High I/O wait (**wa**) indicates disk bottleneck.

## htop — Enhanced Process Monitoring

Install:

```
sudo apt install htop
```

Run:

```
htop
```

Advantages over top:

- Color-coded interface
- Easier process filtering
- Tree view
- Interactive process management

htop is user-friendly and excellent for live troubleshooting.

## free — Memory Usage Overview

To check memory usage:

```
free -h
```

Example:

|      | total | used | free | shared | buff/cache | available |
|------|-------|------|------|--------|------------|-----------|
| Mem: | 8Gi   | 3Gi  | 1Gi  | 200Mi  | 4Gi        | 4Gi       |

Important concept:

Linux uses available memory for caching.  
High “used” memory does not always mean a problem.

Focus on:

- Available memory
- Swap usage

Check swap:

```
swapon --show
```

High swap usage may indicate memory pressure.

## vmstat — Virtual Memory Statistics

**vmstat** provides deeper insight into:

- Memory
- Swap
- CPU
- I/O
- Context switching

Run:

```
vmstat 1
```

This refreshes every second.

Key columns:

| Column | Meaning           |
|--------|-------------------|
| r      | Running processes |
| b      | Blocked processes |
| si     | Swap in           |
| so     | Swap out          |
| wa     | I/O wait          |

High **si** or **so** means active swapping — often performance degradation.

## iostat — Disk I/O Monitoring

Install:

```
sudo apt install sysstat
```

Run:

```
iostat -xz 1
```

Important fields:

- %util → Disk utilization
- await → Average wait time
- r/s and w/s → Reads and writes per second

If **%util** approaches 100%, disk is saturated.

Disk bottlenecks are common in:

- Database servers
- Log-heavy applications
- Backup operations

# journalctl Analysis

## Detecting Hidden Problems

System logs often reveal performance issues.

View recent logs:

```
journalctl -b
```

Filter errors:

```
journalctl -p 3
```

Check service-specific logs:

```
journalctl -u nginx
```

Look for patterns:

- Out-of-memory killer messages
- Disk errors
- Service restarts
- Kernel warnings

Example OOM message:

```
Out of memory: Kill process 1234 (mysqld)
```

This indicates memory exhaustion.

## Basic Performance Interpretation

Understanding metrics requires context.

### Scenario 1 — High CPU Usage

Symptoms:

- Load average rising
- CPU %us high
- System slow

Check:

```
top
```

Find a high CPU process.

Action:

- Optimize application
- Scale vertically
- Limit process usage

## Scenario 2 — High Memory Usage

Symptoms:

- Swap usage increasing
- vmstat shows swap activity

Check:

```
free -h
```

Action:

- Add RAM
- Optimize application
- Tune database buffers

## Scenario 3 — High Disk I/O

Symptoms:

- High I/O wait
- iostat shows %util near 100%

Action:

- Move logs to separate disk
- Optimize database indexing
- Upgrade storage (SSD/NVMe)

## Monitoring Illustration

```
CPU → top / htop  
Memory → free / vmstat  
Disk → iostat  
Logs → journalctl
```

Each tool monitors a different layer of system health.

## Proactive Monitoring Strategy

Professional administrators:

- Monitor daily trends
- Log metrics over time
- Identify anomalies early
- Automate alerts

Even simple scripts help:

```
uptime >> /var/log/health.log  
free -h >> /var/log/health.log
```

Advanced setups may integrate:

- Prometheus
- Grafana
- Netdata

But foundational CLI tools remain essential.

## Real-World Production Example

Scenario:

Users report slow response.

Investigation:

1. `top` → CPU low.
2. `free -h` → Memory fine.
3. `iostat -xz 1` → Disk 100% utilization.
4. `journalctl` → Excessive logging detected.

Root cause:

- Application writing too many logs.

Solution:

- Log rotation adjustment.
- Separate log partition.

Monitoring enabled quick diagnosis.

## Best Practices for Monitoring

- Check system health daily.
- Review logs after major updates.
- Watch load averages over time.
- Monitor swap activity.
- Identify disk bottlenecks early.
- Document normal baseline performance.

Baseline understanding is critical.

## Chapter Summary

In this chapter, we explored:

- `top` and `htop` for process monitoring
- `free` for memory insight
- `vmstat` for system statistics
- `iostat` for disk performance
- `journalctl` for log-based analysis
- Interpreting system behavior

Most importantly:

**Monitoring is awareness. Awareness prevents outages.**

A healthy server is not one that never fails —  
it is one whose problems are detected early.

## Reflection Challenge

Ask yourself:

- Can I diagnose CPU saturation?
- Do I understand memory pressure indicators?
- Can I detect disk bottlenecks?
- Can I interpret load average correctly?

If yes, you are developing operational intuition.

# Chapter 16 — Practical Troubleshooting Methodology

## Introduction: Troubleshooting Is a Discipline, Not a Guessing Game

Every production administrator will face incidents:

- Server not reachable
- Service suddenly stopped
- System boot failure
- Database refusing connections
- Performance degradation

The difference between panic and professionalism is **methodology**.

Troubleshooting is not random trial-and-error.  
It is a structured, layered investigation process.

This chapter introduces:

- Layered troubleshooting approach
- Log-first debugging mindset
- Boot failure investigation
- Service dependency failure analysis

The goal is to think like a systems engineer, not just a command executor.

## The Layered Troubleshooting Model

Professional troubleshooting follows a logical hierarchy:

```
Hardware
  ↓
Kernel
  ↓
Service
  ↓
Network
  ↓
Application
```

Always start from the lowest layer possible.  
Do not restart services blindly without understanding the root cause.

## Layer 1 — Hardware Investigation

Even in cloud environments, hardware abstraction does not eliminate hardware issues.

Check disk health:

```
sudo smartctl -a /dev/sda
```

Check disk usage:

```
df -h
```

Check I/O errors:

```
dmesg | grep -i error
```

Symptoms of hardware issues:

- Input/output errors
- Disk not mounted
- Filesystem corruption
- Sudden system freezes

If the hardware layer is unstable, everything above it will fail.

## Layer 2 — Kernel-Level Investigation

Kernel problems affect:

- Memory allocation
- Device drivers
- Process scheduling

Check kernel logs:

```
dmesg | less
```

Check OOM (Out-of-Memory) killer events:

```
journalctl | grep -i oom
```

Example:

```
Out of memory: Kill process 1243 (postgres)
```

This indicates memory exhaustion, not database corruption.

Check running kernel:

```
uname -r
```

Kernel mismatches or failed upgrades can cause boot or driver issues.

## Layer 3 — Service-Level Troubleshooting

Most production issues occur at the service layer.

Check service status:

```
sudo systemctl status nginx
```

Common indicators:

- Active (running)
- Failed
- Restarting

Check logs:

```
journalctl -u nginx -n 50
```

Always ask:

- Did the configuration change?
- Was system updated?
- Is the service listening on the correct port?

Check open ports:

```
ss -tulnp
```

## Layer 4 — Network-Level Troubleshooting

If service runs but unreachable:

Check connectivity:

```
ping server-ip
```

Check route:

```
ip route
```

Check firewall:

```
sudo ufw status
```

Test locally:

```
curl http://localhost
```

If local works but remote fails → firewall or network issue.

Illustration:

```
Client → Network → Firewall → Service
```

Break the chain step by step.

## Log-First Debugging

Modern Ubuntu systems rely heavily on **systemd** logging.  
Always inspect logs before restarting anything.

View system logs:

```
journalctl -xe
```

View boot logs:

```
journalctl -b
```

Filter errors:

```
journalctl -p 3
```

Logs provide:

- Exact error messages
- Timestamp correlation
- Dependency failures

Restarting without reading logs hides root causes.

## Boot Failure Investigation

A server that fails to boot requires calm analysis.

### Step 1 — Access Recovery Mode

From GRUB:

- Select Advanced Options
- Choose recovery mode

## Step 2 — Inspect Logs

```
journalctl -b -1
```

This checks the previous boot.

Common boot failures:

- Filesystem mount errors
- fstab misconfiguration
- Missing kernel modules
- Failed service dependency

### fstab Failure Example

If `/etc/fstab` is misconfigured:

Boot may drop to the emergency shell.

Fix by:

```
nano /etc/fstab
```

Correct mount entry.

Then reboot.

## Service Dependency Failure

`systemd` manages service dependencies.

Check dependencies:

```
systemctl list-dependencies nginx
```

If dependent service fails:

```
systemctl status dependent-service
```

Example scenario:

- Web server depends on `network-online.target`
- Network misconfigured
- Web server fails at boot

Use:

```
systemctl edit nginx
```

Override improper dependency if necessary.

# Structured Troubleshooting Workflow

Professional method:

1. Define the problem clearly.
2. Identify layer.
3. Gather evidence.
4. Test hypothesis.
5. Apply minimal fix.
6. Verify stability.
7. Document root cause.

## Real-World Scenario

Scenario:

Website down.

Investigation:

1. ping → reachable.
2. curl localhost → works.
3. ss → nginx listening.
4. ufw → port 80 blocked.

Root cause:

Firewall misconfiguration.

Fix:

```
sudo ufw allow 80/tcp
```

Service restored.

No reboot required.

## Boot Failure Case Study

After updating, the server fails to boot.

Steps:

1. Enter recovery mode.
2. Check logs:

```
journalctl -b -1
```

Find:

- Filesystem UUID mismatch.

Fix:

```
blkid
```

Update `/etc/fstab`.

Reboot successfully.

## Troubleshooting Mindset

Key principles:

- Never assume.
- Always verify.
- Change one variable at a time.
- Document findings.
- Avoid unnecessary reboots.
- Think in layers.

Troubleshooting is a logical deduction.

## Illustration: Troubleshooting Pyramid

```
Application Error
    ↓
Service Check
    ↓
Kernel Log
    ↓
Hardware Verification
```

Start low, move upward.

## Common Mistakes

- Restarting services repeatedly without logs
- Blaming network before checking local service
- Ignoring disk space
- Editing config without backup
- Not reading exact error message

Error messages are clues, not noise.

## Chapter Summary

In this chapter, we explored:

- Layered troubleshooting model
- Hardware and kernel checks
- Service-level investigation
- Network debugging
- Log-first methodology
- Boot failure recovery
- Service dependency analysis

Most importantly:

**Troubleshooting is systematic reasoning, not random experimentation.**

Professional administrators solve problems calmly because they follow structure.

## Reflection Challenge

Ask yourself:

- Can I isolate issues by layer?
- Do I check logs before restarting?
- Can I diagnose boot failures?
- Do I understand service dependencies?

If yes, you are building true operational confidence.

# PART X — MODERN SYSADMIN MINDSET

## Chapter 17 — Automation & Infrastructure Thinking

### Introduction: From Manual Configuration to Infrastructure Engineering

At the beginning of your journey, configuring a server manually feels powerful:

- Install packages
- Edit configuration files
- Restart services
- Tune settings

But in production environments, manual configuration quickly becomes dangerous.

One day, you configure Server A.  
The next week, you configure Server B slightly differently.  
Three months later, no one remembers what was changed.

This is how **snowflake servers** are born.

A snowflake server is unique, undocumented, and impossible to reproduce.

Professional infrastructure avoids snowflakes.  
It embraces automation, reproducibility, and structured thinking.

This chapter explores:

- Avoiding snowflake servers
- Infrastructure as Code (IaC)
- Introduction to Ansible
- Reproducible builds
- Golden image strategy

The goal is to think like an **infrastructure architect**, not just a system operator.

### Avoiding Snowflake Servers

A snowflake server has these characteristics:

- Manual configuration changes
- No version control
- No documentation
- Different from other servers
- Hard to replicate
- High operational risk

Illustration:

```
Server A: manually configured
Server B: slightly different
Server C: unknown changes
```

When one fails, recovery becomes guesswork.

Professional mindset:

**If you cannot rebuild it, you do not control it.**

## Infrastructure as Code (IaC) Concept

Infrastructure as Code means:

- Server configuration defined in text files
- Version controlled (e.g., Git)
- Repeatable deployments
- Automated provisioning

Instead of:

Manually installing nginx and editing config

You define:

```
Install nginx
Copy config
Start service
```

And apply automatically.

Illustration:

```
Code → Automation Tool → Identical Servers
```

This approach provides:

- Consistency
- Auditability
- Scalability
- Reduced human error

## Introduction to Ansible

Ansible is a lightweight automation tool widely used in Linux environments.

Key advantages:

- Agentless (uses SSH)
- Simple YAML-based playbooks
- Easy to learn
- Powerful at scale

## Installing Ansible

On control node:

```
sudo apt install ansible
```

Check version:

```
ansible --version
```

## Basic Inventory File

Create inventory:

```
nano inventory.ini
```

Example:

```
[webservers]
192.168.1.10
192.168.1.11
```

## Simple Ansible Test

```
ansible webservers -i inventory.ini -m ping
```

If successful, servers respond with "pong".

## Simple Playbook Example

Create playbook:

```
nano install-nginx.yml
```

Example:

```
- hosts: webservers
  become: yes
  tasks:
    - name: Install nginx
```

```
apt:
  name: nginx
  state: present

- name: Ensure nginx is running
  systemd:
    name: nginx
    state: started
    enabled: yes
```

Run:

```
ansible-playbook -i inventory.ini install-nginx.yml
```

Result:

- Both servers configured identically.

No snowflakes.

## Reproducible Builds

A reproducible build means:

- Same configuration
- Same packages
- Same versions
- Same structure
- Any time, any server

Steps toward reproducibility:

- Version-control configuration files
- Automate installation
- Avoid manual edits in production
- Use configuration templates

Example of version control mindset:

```
nginx.conf → Git repository
Playbooks → Git repository
Infrastructure definitions → Git repository
```

Reproducibility creates confidence.

## Golden Image Strategy

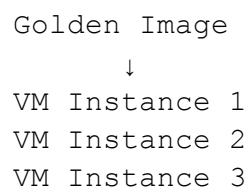
A golden image is a pre-configured base system image used to deploy new servers.

In cloud environments:

- Build base Ubuntu image
- Install baseline tools
- Apply security hardening
- Save as template

Then launch new servers from this template.

Illustration:



All start from the same trusted baseline.

## Building a Golden Image (Conceptual Steps)

1. Install Ubuntu Server.
2. Apply security hardening.
3. Install baseline packages.
4. Configure monitoring.
5. Clean temporary files.
6. Snapshot image.

Check installed packages:

```
dpkg -l
```

Remove unnecessary services:

```
sudo apt purge unused-package
```

A golden image should be:

- Minimal
- Secure
- Documented
- Tested

## Automation Mindset

Automation is not about complexity.  
It is about consistency and scalability.

Professional infrastructure thinking includes:

- Automate repetitive tasks.
- Avoid manual production edits.
- Test changes in staging.
- Keep configuration in version control.
- Design systems to be rebuilt easily.

## Real-World Scenario

Scenario:

The company needs to deploy 20 web servers.

Without automation:

- 20 manual installations
- 20 opportunities for error
- Inconsistent configurations

With Ansible:

- Single playbook
- Apply to all servers
- Identical configuration
- Faster deployment

Result:

- Reduced risk
- Faster scaling
- Easier maintenance

## Combining Golden Image + Ansible

Best practice:

1. Use golden image for baseline OS.
2. Use Ansible for application configuration.
3. Use version control for infrastructure definitions.

This layered automation approach provides:

- Fast provisioning
- Flexible configuration
- Reproducibility
- Traceability

## Infrastructure Thinking Illustration

Manual Server → Snowflake Risk  
Automation → Consistency  
Golden Image → Fast Provisioning  
IaC → Reproducibility

Infrastructure becomes predictable instead of fragile.

## Common Mistakes

- Editing production servers manually
- Not documenting configuration
- Storing scripts locally only
- Ignoring version control
- Skipping testing

Infrastructure without discipline becomes unstable.

## Chapter Summary

In this chapter, we explored:

- The danger of snowflake servers
- Infrastructure as Code principles
- Introduction to Ansible
- Reproducible builds
- Golden image strategy

Most importantly:

**Professional infrastructure is designed to be rebuilt, not manually repaired.**

Automation reduces human error.

Reproducibility increases reliability.

Infrastructure thinking enables scalability.

## Reflection Challenge

Ask yourself:

- Could I rebuild my production server from scratch?
- Is my configuration documented in code?
- Do I avoid manual changes?
- Can I deploy multiple servers consistently?

If yes, you are transitioning from administrator to infrastructure engineer.

# Chapter 18 — Ubuntu Server in Cloud & Containers

## Introduction: Ubuntu Beyond the Physical Server

Modern Ubuntu Server is no longer limited to:

- Bare-metal machines
- Traditional data centers
- Static infrastructure

Today, Ubuntu is widely used in:

- Public cloud environments (AWS, Azure, GCP)
- Private clouds (OpenStack, Proxmox)
- Container platforms
- Edge deployments

Understanding Ubuntu in cloud and container environments is essential for modern infrastructure engineers.

This chapter explores:

- Cloud-init deeper dive
- Running Docker and Podman
- Ubuntu minimal cloud images
- Production deployment patterns

Cloud-native thinking requires automation, reproducibility, and scalability.

## Cloud-Init: Automated First Boot Configuration

Cloud-init is a powerful initialization system used in cloud environments.

It automatically configures:

- Users
- SSH keys
- Hostname
- Network
- Packages
- Scripts at first boot

Instead of manually configuring every VM, cloud-init allows you to define configuration in YAML format.

## Basic Cloud-Init Example

Create a user-data file:

```
#cloud-config
users:
  - name: devops
    sudo: ALL=(ALL) NOPASSWD:ALL
    ssh_authorized_keys:
      - ssh-ed25519 AAAA...

packages:
  - nginx
  - curl

runcmd:
  - systemctl enable nginx
  - systemctl start nginx
```

This file:

- Creates user
- Install packages
- Starts services

## Testing Cloud-Init Locally

In cloud VM, verify:

```
cloud-init status
```

Check logs:

```
sudo cat /var/log/cloud-init.log
```

Cloud-init eliminates manual post-install configuration.

## Ubuntu Minimal Cloud Image

Cloud environments typically use minimal images:

- Reduced package footprint
- Faster boot time
- Lower attack surface
- Optimized for automation

Ubuntu cloud images are:

- Pre-built
- Cloud-init enabled
- LTS optimized

Minimal system inspection:

```
dpkg -l
```

You will notice fewer packages compared to full server installation.

Production principle:

**Install only what you need.**

## Running Docker on Ubuntu Server

Containers allow application isolation without full virtualization.  
Docker remains one of the most widely used container engines.

### Installing Docker

```
sudo apt update  
sudo apt install docker.io
```

Enable service:

```
sudo systemctl enable docker  
sudo systemctl start docker
```

Test installation:

```
sudo docker run hello-world
```

### Running a Simple Web Container

```
sudo docker run -d -p 8080:80 nginx
```

Test:

```
url http://localhost:8080
```

Containerized applications allow rapid deployment.

## Podman as Alternative

Podman is a daemonless container engine.

Advantages:

- No central daemon
- Rootless container support
- Better security isolation

Install Podman:

```
sudo apt install podman
```

Run container:

```
podman run -d -p 8080:80 nginx
```

Podman aligns well with modern security models.

## Container vs VM Illustration

VM:

Hardware → Hypervisor → OS → App

Container:

Hardware → OS → Container Engine → App

Containers are lightweight and faster to scale.

## Production Patterns in Cloud

Modern Ubuntu cloud deployments follow common patterns:

### Pattern 1 — Immutable Infrastructure

Instead of modifying running servers:

- Build new image
- Deploy new instance
- Replace old instance

Illustration:

Old Server → New Version Built → Replace → Old Destroyed

No manual patching.

## Pattern 2 — Auto Scaling

Cloud servers can scale automatically based on:

- CPU usage
- Traffic volume
- Queue size

Example:

```
Traffic Spike
      ↓
Cloud Auto Scale
      ↓
New Ubuntu Instances
```

Ubuntu cloud images are optimized for rapid scaling.

## Pattern 3 — Reverse Proxy + Containers

Architecture:

```
Internet
      ↓
Load Balancer
      ↓
Nginx Reverse Proxy
      ↓
Docker Containers
```

Advantages:

- Application isolation
- Easier updates
- Controlled routing

## Cloud Security Considerations

In cloud environments:

- Disable unnecessary services
- Restrict security groups
- Use SSH key authentication
- Monitor instance logs
- Use minimal images

Check active services:

```
systemctl list-units --type=service
```

Review open ports:

```
ss -tulnp
```

## Combining Cloud-Init + Containers

Best practice:

1. Use minimal Ubuntu cloud image.
2. Cloud-init installs Docker.
3. Container runs application.
4. No manual configuration.

This provides:

- Rapid provisioning
- Consistency
- Reproducibility

## Real-World Deployment Scenario

Scenario:

Deploy scalable web API in cloud.

Steps:

1. Create minimal Ubuntu cloud VM.
2. Use cloud-init to:
  - Create admin user
  - Install Docker
  - Pull application container
3. Configure firewall.
4. Attach load balancer.

Result:

- Automated deployment
- Horizontal scalability
- Reduced configuration drift

## Common Mistakes

- Using full desktop image in cloud
- Exposing Docker daemon publicly
- Running containers as root unnecessarily
- Manual changes inside container
- Ignoring cloud security group rules

Cloud infrastructure requires discipline.

## Monitoring Containers

List containers:

```
docker ps
```

Check logs:

```
docker logs container_id
```

Inspect resource usage:

```
docker stats
```

Monitoring ensures container health.

## Chapter Summary

In this chapter, we explored:

- Cloud-init automation
- Ubuntu minimal cloud images
- Docker and Podman container engines
- Immutable infrastructure concept
- Production cloud deployment patterns

Most importantly:

**Ubuntu Server thrives in cloud environments when automation and minimalism are applied.**

Cloud-native thinking emphasizes:

- Automation
- Scalability
- Reproducibility
- Security

## Reflection Challenge

Ask yourself:

- Can I deploy Ubuntu automatically using cloud-init?
- Do I understand container deployment?
- Could I design immutable infrastructure?
- Is my cloud instance minimal and secure?

If yes, you are thinking like a modern cloud engineer.

# PART XI — EXAM PREPARATION & PRACTICAL LABS

## Chapter 19 — Canonical Academy Exam Simulation

### Introduction: From Knowledge to Demonstrated Competence

Reading about Ubuntu Server is one thing.  
Operating it confidently under pressure is another.

Professional certification — and real-world operations — are not about memorizing commands. They test:

- Logical thinking
- Troubleshooting ability
- Time management
- Precision under pressure
- Understanding system relationships

This chapter simulates a **performance-based exam environment**, similar to practical certification assessments.

You will face:

- Scenario-driven tasks
- Hands-on configuration challenges
- Timed lab simulations
- Common pitfalls to avoid

The objective is simple:

**Demonstrate that you can configure, secure, and troubleshoot Ubuntu Server independently.**

### Understanding Performance-Based Exams

Unlike multiple-choice tests, practical exams require you to:

- Configure services
- Edit configuration files
- Troubleshoot broken systems
- Verify results

You are evaluated on:

- Functionality
- Stability
- Accuracy
- Security

Illustration:

Scenario → Analyze → Configure → Test → Verify

Every step matters.

## Practice Scenario 1 — Secure SSH Deployment

### Scenario:

You are given a fresh Ubuntu Server instance.

Requirements:

- Create user `adminops`
- Disable root login
- Enforce key-based authentication
- Change SSH port to 2222
- Enable firewall

### Expected CLI Actions:

Create user:

```
sudo adduser adminops
sudo usermod -aG sudo adminops
```

Edit SSH config:

```
sudo nano /etc/ssh/sshd_config
```

Set:

```
PermitRootLogin no
PasswordAuthentication no
Port 2222
```

Restart SSH:

```
sudo systemctl restart ssh
```

Enable firewall:

```
sudo ufw allow 2222/tcp
```

```
sudo ufw enable
```

Verify:

```
sudo systemctl status ssh
sudo ufw status
```

Assessment criteria:

- Can you log in via SSH using key?
- Is root login disabled?
- Is port 22 closed?

## Practice Scenario 2 — Web Stack Deployment

Scenario:

Deploy Nginx with reverse proxy to backend service on port 5000.

Install Nginx:

```
sudo apt install nginx
```

Create config:

```
sudo nano /etc/nginx/sites-available/app
```

Example:

```
server {
    listen 80;
    location / {
        proxy_pass http://127.0.0.1:5000;
    }
}
```

Enable:

```
sudo ln -s /etc/nginx/sites-available/app
/etc/nginx/sites-enabled/
sudo nginx -t
sudo systemctl restart nginx
```

Verification:

```
curl http://localhost
```

## Practice Scenario 3 — Storage & LVM

Scenario:

Add new disk `/dev/sdb`, create LVM volume, mount at `/data`.

Create physical volume:

```
sudo pvcreate /dev/sdb
```

Create volume group:

```
sudo vgcreate vgdata /dev/sdb
```

Create logical volume:

```
sudo lvcreate -L 20G -n lvdata vgdata
```

Format and mount:

```
sudo mkfs.ext4 /dev/vgdata/lvdata
sudo mkdir /data
sudo mount /dev/vgdata/lvdata /data
```

Persist in `/etc/fstab`.

Verification:

```
df -h
```

## Performance-Based Tasks

In exam conditions, you may be asked to:

- Diagnose service failure
- Fix broken fstab entry
- Configure systemd timer
- Restrict sudo privileges
- Create ACL for shared directory

Example ACL task:

```
setfacl -m u:devuser:rwX /shared
```

Verification:

```
getfacl /shared
```

Exams test understanding, not memorization.

## Timed Lab Simulation

In real exam settings:

- Time is limited (2–3 hours)
- Multiple tasks required
- Systems may contain intentional misconfigurations

Simulation approach:

1. Read all tasks first.
2. Estimate difficulty.
3. Start with confident tasks.
4. Test after each configuration.
5. Avoid unnecessary reboots.

Time discipline is essential.

## Troubleshooting Simulation

Scenario:

Service fails to start.

Step-by-step:

```
sudo systemctl status service-name
journalctl -u service-name
```

Look for:

- Permission issues
- Port conflicts
- Syntax errors

Check listening ports:

```
ss -tulnp
```

Correct and verify.

## Common Mistakes in Exam & Real World

- Not testing configuration before moving on
- Forgetting to enable service
- Ignoring firewall rules
- Editing wrong file
- Typing errors in YAML
- Panic under time pressure

Most failures occur due to lack of verification.

Always test:

```
systemctl status  
ss -tulnp  
curl localhost
```

## Exam Preparation Strategy

Professional preparation includes:

- Practicing on fresh VM repeatedly
- Rebuilding server from scratch
- Simulating failures intentionally
- Timing yourself during tasks
- Documenting commands used

Repetition builds muscle memory.

## Simulation Lab Exercise

Try this full simulation:

1. Install Ubuntu Server minimal.
2. Configure static IP with Netplan.
3. Secure SSH.
4. Deploy Nginx + reverse proxy.
5. Configure firewall.
6. Create a user with limited sudo.
7. Setup automated backup with systemd timer.
8. Monitor system using top and journalctl.

Time yourself.

This builds confidence.

## Professional Mindset

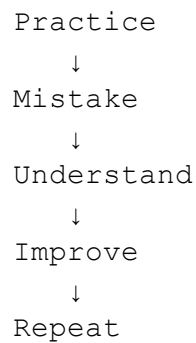
Certification exams measure competence, but real-world production requires composure.

Key principles:

- Stay calm.
- Break problem into layers.
- Verify each change.
- Avoid guesswork.
- Think before acting.

Infrastructure confidence comes from disciplined practice.

## Illustration: Operational Confidence Loop



Growth comes from iteration.

## Chapter Summary

In this chapter, we explored:

- Practice-based exam scenarios
- Performance-oriented tasks
- Timed lab simulation techniques
- Common configuration mistakes
- Professional exam mindset

Most importantly:

**Competence is demonstrated through action, not theory.**

When you can deploy, secure, troubleshoot, and verify under time pressure, you are ready — not just for certification, but for production responsibility.

## Final Reflection

Ask yourself:

- Can I rebuild a secure Ubuntu server from scratch?
- Can I diagnose service failures confidently?
- Can I deploy a complete web stack?
- Can I manage time during configuration tasks?

If yes, you are not just studying Ubuntu Server —  
you are mastering operational excellence.

# Chapter 20 — Capstone Project

## Introduction: From Learning to Building

Throughout this book, you have explored:

- Installation and deployment
- Security hardening
- Storage management
- Service configuration
- Monitoring and troubleshooting
- Automation and cloud integration

Now it is time to integrate everything into one unified challenge:

**Build a production-ready Ubuntu Server from scratch.**

This capstone project simulates a real-world deployment scenario. By completing it, you demonstrate operational readiness.

The goal is not perfection — it is structured, disciplined implementation.

## Project Scenario

You are tasked with deploying a production server for a small organization. The server must:

- Use static IP configuration
- Implement firewall rules
- Harden SSH
- Deploy a LAMP stack
- Use LVM-based storage
- Automate backups
- Establish monitoring baseline

You will build a complete production-ready system.

## Step 1 — Configure Static IP

Edit Netplan:

```
sudo nano /etc/netplan/00-installer-config.yaml
```

### Example configuration:

```
network:
  version: 2
  ethernets:
    ens33:
      dhcp4: no
      addresses:
        - 192.168.1.50/24
      gateway4: 192.168.1.1
      nameservers:
        addresses:
          - 8.8.8.8
          - 1.1.1.1
```

### Apply configuration:

```
sudo netplan apply
```

### Verify:

```
ip a
ping 8.8.8.8
```

### Illustration:

Server → Static IP → Gateway → Internet

A production server should not rely on DHCP unpredictability.

## Step 2 — Configure Firewall

### Enable UFW:

```
sudo ufw default deny incoming
sudo ufw default allow outgoing
```

### Allow essential services:

```
sudo ufw allow 80/tcp
sudo ufw allow 443/tcp
sudo ufw allow 2222/tcp
```

### Enable firewall:

```
sudo ufw enable
```

### Verify:

```
sudo ufw status
```

Firewall is your first defensive barrier.

## Step 3 — SSH Hardening

Edit SSH configuration:

```
sudo nano /etc/ssh/sshd_config
```

Set:

```
PermitRootLogin no
PasswordAuthentication no
Port 2222
```

Restart SSH:

```
sudo systemctl restart ssh
```

Verify service:

```
sudo systemctl status ssh
```

Key principle:

**Never expose password-based SSH in production.**

## Step 4 — Install LAMP Stack

LAMP includes:

- Linux
- Apache
- MySQL
- PHP

Install Apache:

```
sudo apt install apache2
```

Install MySQL:

```
sudo apt install mysql-server
sudo mysql_secure_installation
```

Install PHP:

```
sudo apt install php libapache2-mod-php php-mysql
```

Verify Apache:

```
curl http://localhost
```

Enable Apache at boot:

```
sudo systemctl enable apache2
```

Illustration:

```
Client → Apache → PHP → MySQL
```

Test full stack functionality.

## Step 5 — Configure LVM Storage

Add new disk `/dev/sdb`.

Create physical volume:

```
sudo pvcreate /dev/sdb
```

Create volume group:

```
sudo vgcreate vgdata /dev/sdb
```

Create logical volume:

```
sudo lvcreate -L 20G -n lvweb vgdata
```

Format:

```
sudo mkfs.ext4 /dev/vgdata/lvweb
```

Mount:

```
sudo mkdir /var/www-data
sudo mount /dev/vgdata/lvweb /var/www-data
```

Persist in `/etc/fstab`.

Verify:

```
df -h
```

LVM ensures flexible storage growth.

## Step 6 — Backup Automation

Create backup directory:

```
sudo mkdir -p /backup/web
```

Create backup script:

```
sudo nano /usr/local/bin/backup.sh
```

Example:

```
#!/bin/bash
rsync -a /var/www-data/ /backup/web/
mysqldump --all-databases > /backup/db.sql
```

Make executable:

```
sudo chmod +x /usr/local/bin/backup.sh
```

Create systemd service:

```
[Unit]
Description=Daily Backup

[Service]
Type=oneshot
ExecStart=/usr/local/bin/backup.sh
```

Create timer:

```
[Timer]
OnCalendar=daily
Persistent=true
```

Enable timer:

```
sudo systemctl enable backup.timer
sudo systemctl start backup.timer
```

Verify:

```
systemctl list-timers
```

Backup must be automated, not manual.

## Step 7 — Monitoring Baseline

Check CPU:

```
top
```

Check memory:

```
free -h
```

Check disk I/O:

```
iostat -xz 1
```

Check logs:

```
journalctl -p 3
```

Document baseline metrics:

- Idle CPU %
- Memory usage
- Disk utilization
- Load average

Monitoring baseline allows anomaly detection later.

## Final Validation Checklist

Before declaring production readiness:

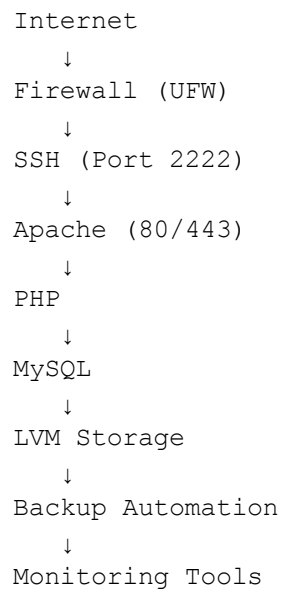
- Static IP confirmed
- Firewall enabled and tested
- SSH hardened and tested
- Apache running
- MySQL secured
- LVM mounted correctly
- Backup timer active
- Logs clean
- No unnecessary services running

Verify open ports:

```
ss -tulnp
```

Only expected services should be exposed.

## Capstone Architecture Overview



This is a simplified production blueprint.

## Common Mistakes During Capstone

- Forgetting to enable services at boot
- Not testing firewall rules
- Misconfiguring fstab
- Skipping MySQL secure installation
- Not verifying backup integrity
- Ignoring monitoring baseline

Verification is as important as configuration.

## Professional Reflection

Completing this capstone demonstrates:

- Structured thinking
- Security awareness
- Storage discipline
- Automation capability
- Monitoring competence

This is the foundation of production engineering.

## Final Chapter Summary

In this capstone project, you:

- Configured static networking
- Implemented firewall security
- Hardened SSH
- Deployed full LAMP stack
- Built flexible LVM storage
- Automated backups
- Established monitoring baseline

You have now built a **production-ready Ubuntu Server** from the ground up.

## Closing Reflection

Ask yourself:

- Can I rebuild this entire setup from scratch?
- Can I troubleshoot each layer independently?
- Do I understand why each configuration matters?
- Can I scale this design to multiple servers?

If the answer is yes, you have moved beyond beginner administration.

You are now thinking like a production engineer.

# Appendix A — Command Quick Reference

This section provides a fast-access reference for essential Ubuntu Server administration commands. These are the commands you should know confidently during production operations or certification exams.

## System Information

```
uname -a
lsb_release -a
hostnamectl
uptime
```

Purpose:

- Identify kernel version
- Check Ubuntu release
- Verify system uptime
- Confirm hostname configuration

## Service Management (systemd)

```
systemctl status nginx
systemctl start nginx
systemctl stop nginx
systemctl restart nginx
systemctl enable nginx
systemctl disable nginx
```

Logs:

```
journalctl -u nginx
journalctl -xe
```

**Always check status before restarting a service.**

## Networking

```
ip a
ip route
ss -tulnp
ping 8.8.8.8
curl http://localhost
```

Netplan apply:

```
sudo netplan apply
```

These commands validate connectivity and port listening.

## Storage & Filesystem

```
lsblk  
blkid  
df -h  
mount
```

### LVM:

```
pvs  
vgs  
lvs  
lvextend  
resize2fs
```

### Disk monitoring:

```
iostat -xz 1
```

## User & Permissions

```
adduser  
usermod -aG sudo username  
chmod  
chown  
getfacl  
setfacl
```

## Firewall & Security

```
ufw status  
ufw allow 80/tcp  
fail2ban-client status
```

### Quick Reference Philosophy:

If you can type these commands confidently without hesitation, you are operationally ready.

# Appendix B — systemd Unit Template Examples

systemd is the heart of modern Ubuntu.

Understanding unit files allows you to control service behavior precisely.

## Basic Service Template

```
[Unit]
Description=My Custom App
After=network.target

[Service]
Type=simple
ExecStart=/usr/local/bin/app.sh
Restart=on-failure
User=appuser

[Install]
WantedBy=multi-user.target
```

Enable:

```
sudo systemctl enable app.service
sudo systemctl start app.service
```

## Timer Example

Service file:

```
[Unit]
Description=Backup Job

[Service]
Type=oneshot
ExecStart=/usr/local/bin/backup.sh
```

Timer file:

```
[Timer]
OnCalendar=daily
Persistent=true
```

Check timers:

```
systemctl list-timers
```

Timers are preferred over cron in modern production environments.

# Appendix C — Autoinstall YAML Templates

Ubuntu's *Subiquity Autoinstall* uses cloud-init style YAML.

## Minimal Autoinstall Template

```
#cloud-config
autoinstall:
  version: 1
  identity:
    hostname: prod-server
    username: admin
    password: "$6$hashedpassword"
  ssh:
    install-server: true
    authorized-keys:
      - ssh-ed25519 AAAA...
  packages:
    - nginx
    - curl
```

This enables:

- Automated OS installation
- Preconfigured SSH
- Preinstalled packages

## Storage Section Example

```
storage:
  layout:
    name: lvm
```

Autoinstall ensures:

**Repeatable server deployment without manual interaction.**

# Appendix D — Netplan YAML Examples

Netplan defines modern Ubuntu networking.

## Static IP Configuration

```
network:
  nameservers:
    version: 2
  ethernets:
    ens33:
      dhcp4: no
      addresses:
        - 192.168.1.50/24
      gateway4: 192.168.1.1
      addresses:
        - 8.8.8.8
        - 1.1.1.1
```

Apply:

```
sudo netplan apply
```

## DHCP Configuration

```
network:
  version: 2
  ethernets:
    ens33:
      dhcp4: true
```

## Bridge Example (For Virtualization)

```
bridges:
  br0:
    interfaces: [ens33]
    addresses: [192.168.1.100/24]
    gateway4: 192.168.1.1
```

Netplan is declarative — YAML must be properly indented.

# Appendix E — Troubleshooting Flowcharts

Structured troubleshooting prevents panic.

## Service Not Running

```
Check systemctl status
↓
Check journalctl logs
↓
Check configuration syntax
↓
Check firewall
↓
Verify port listening
```

## Cannot Reach Server

```
Ping
↓
Check IP config
↓
Check gateway
↓
Check firewall
↓
Check routing table
```

## High System Load

```
top → CPU?
free → Memory?
iostat → Disk?
journalctl → Errors?
```

Always troubleshoot layer by layer.

# Appendix F — Production Hardening Checklist

Before deploying any Ubuntu Server to production, verify the following:

## System Security

- Root login disabled
- Password authentication disabled
- SSH on custom port (optional)
- Firewall enabled
- Fail2Ban active

## Service Security

- Only required services enabled
- Database not exposed publicly
- Reverse proxy configured
- TLS enabled (if applicable)

## Storage & Backup

- LVM configured
- Disk usage monitored
- Automated backup enabled
- Backup restore tested

## Monitoring

- Baseline CPU recorded
- Baseline memory usage recorded
- Baseline disk I/O recorded
- Log review routine established

## Automation

- Configuration documented
- No snowflake servers
- Infrastructure scripts version controlled

### **Final Principle:**

Production readiness is not about a single command.  
It is about consistent discipline across security, storage, services, and automation.

# Final Words for the Appendices

These appendices are your operational toolkit.

When facing:

- System failure
- Performance degradation
- Security incident
- Deployment pressure

Return here.

A professional administrator:

- Works methodically
- Documents carefully
- Automates responsibly
- Secures proactively

If you can apply everything in this book — from installation to automation — you are prepared to operate Ubuntu Server in real production environments.

# References

Both, D. (2025). *systemd for Linux sysadmins: All you need to know about the systemd suite for Linux users*. Apress. <https://doi.org/10.1007/979-8-8688-1328-3>

Gotangco, J., & Berton, L. (2025). *Red Hat Enterprise Linux 9 for sysadmins: A technical guide for building secure production systems using RHEL 9 administration*. BPB Publications.

Hess, K. (2023). *Practical Linux system administration: A guide to installation, configuration, and management*. O'Reilly Media.

Hitchcock, K. (2022). *Linux system administration for the 2020s: The modern sysadmin leaving behind the culture of build and maintain*. Apress. <https://doi.org/10.1007/978-1-4842-7984-7>

Hitchcock, K. (2023). *The enterprise Linux administrator: Journey to a new Linux career*. Apress. <https://doi.org/10.1007/978-1-4842-8801-6>

van Vugt, S. (2009). *Pro Ubuntu Server administration*. Apress.

Canonical. (2026). *Using Ubuntu Server: Practical administration for production environments (Canonical Academy SysAdmin Track module outline)*. Canonical Academy.

## AUTHOR



**Onno W. Purbo** currently serves as Rector at Institut Teknologi Tangerang Selatan (ITTS). He earned his Ph.D. in Electrical Engineering from the University of Waterloo, Canada. He is a copyleft advocate, educator, and ICT evangelist. He has published more than 50 books, including free ICT e-books for schools in 2008. Some of his latest books include *"Internet-TCP/IP: Konsep Dan Implementasi"* (2018); *"Sistem Operasi, Konsep Dan Membuat Linux OpenWRT Dan ROM Android"* (2019); *"IPv6 Untuk Mendukung Operasi Jaringan Dan Domain Name System"* (2019); *"Kubernetes untuk Pemula"* (2024); and *"Membuat Operator Seluler 5G Sendiri"* (2024).

He led the first Internet connection at Institut Teknologi Bandung from 1993–2000 and used it to establish Indonesia's first educational Internet network. He played a key role in liberating WiFi frequencies, introducing RT/RW-net, the Wajanbolic antenna, as well as OpenBTS-based cellular networks and private 5G networks. He also led the first telephone network over the Internet, VoIP Merdeka, which later became known as VoIP Rakyat based on SIP and using the area codes +62520 and +62521.

He is currently active in promoting e-Learning and operates an e-Learning server at <http://lms.onnocenter.or.id/moodle/> serving more than 66,000 students.