Fedora LINUX

for Developers and Power Users

Configure Your Fedora 40 Workstation, Use Podman, Flatpak, and DNF, and Build a Fast and Secure Development Environment



Robert Austin

Fedora Linux for Developers and Power Users

Configure Your Fedora 40 Workstation, Use Podman, Flatpak, and DNF, and Build a Fast and Secure Development Environment

By
Robert Austin

Copyright © Robert Austin

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means—including photocopying, recording, or other electronic or mechanical methods—without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

This book is intended for educational purposes only. The techniques, tools, and concepts described herein are designed to enhance the knowledge and skills of penetration testers, ethical hackers, and security professionals working within the bounds of the law. Misuse of this content for unauthorized or malicious activities is strictly prohibited. The author and publisher disclaim any responsibility for misuse or damages resulting from the application of the material in this publication.

Table of Contents

Chapter 1: Getting Started with Fedora for Development

1.1 Introduction to Fedora Linux for Power Users

1.2 Choosing the Right Fedora Spin or Edition

1.3 Installing Fedora Workstation or Server

1.4 First-Time Setup: Hostname, Users, and SSH

1.5 Updating Your System and Enabling RPM Fusion

Chapter 2: Power User Tools and Customization

2.1 Terminal Multiplexers: tmux and Tilix

2.2 Customizing the Shell with Zsh and Oh-My-Zsh

2.3 Productivity with GNOME Extensions and Keyboard Shortcuts

2.4 File Management with Ranger, fzf, and Bat

2.5 Using Systemd for Custom Service Management

Chapter 3: Package Management and Software Sources

3.1 DNF Advanced Usage and Plugin System

3.2 Managing Software Repositories and Modular Streams

3.3 Building and Rebuilding RPM Packages

3.4 Automating Installs with Ansible and Kickstart

3.5 Flatpak vs Snap on Fedora: Integration and Use

Chapter 4: Setting Up a Developer-Friendly Environment

4.1 Installing Programming Languages and SDKs

4.2 Managing Python Environments with Pipx, venv, and Poetry

4.4 Using Dev Containers with Visual Studio Code

4.5 Troubleshooting Compilation and Dependency Issues

Chapter 5: Version Control and DevOps Tools

5.1 Git Power Features and Configuration

5.2 Using GitHub CLI and Forge Integration

5.3 CI/CD with GitHub Actions on Fedora

5.4 Local Git Server with Gitea

5.5 Secrets Management with SOPS and GPG

Chapter 6: Working with Containers Using Podman

6.1 Understanding Podman and Docker Compatibility

6.2 Building and Running Containers Without Root

6.4 Using Systemd to Manage Podman Containers

6.5 Debugging and Securing Containers

Chapter 7: Developing with Virtual Machines and Toolboxes

7.1 Creating VMs with GNOME Boxes and Virt-Manager

7.2 Using Fedora Toolbx for Isolated Dev Environments
7.3 Bridging Podman, Toolbox, and IDEs
7.4 Optimizing VM Performance for Development
7.5 CLI Tools for Managing VMs and Toolboxes
Chapter 8: Networking, Security, and Firewalls
8.1 Managing Network Interfaces with nmcli
8.2 Configuring a Developer-Friendly Firewall with firewalld
8.3 Using SELinux in Targeted Mode for Security and Flexibility
8.4 Troubleshooting Port Conflicts and Service Bindings
8.5 SSH Hardening and Secure Remote Access
Chapter 9: Performance, Power, and Battery Tweaks
9.1 Tuning System Performance with tuned and powertop
9.2 Optimizing I/O and CPU for Build Tasks
9.3 Managing Logs and Journals with journalctl
9.4 Resource Monitoring with btop and dstat
9.5 Benchmarking and Profiling Applications
Chapter 10: Backup, Recovery, and Maintenance
10.1 Backing Up with Borg, Restic, and rsync
10.2 Managing Snapshots with Btrfs on Fedora
10.3 Automating Backups with systemd Timers
10.4 Performing System Rollbacks and Recovery
10.5 Keeping Fedora Clean and Lean Over Time

Chapter 1: Getting Started with Fedora for Development

A well-prepared Fedora system provides developers and power users with a fast, secure, and fully customizable environment. This chapter walks you through selecting the right Fedora edition, performing a clean installation, and preparing the system for efficient, developer-focused use. By the end of this chapter, your Fedora environment will be updated, secure, and ready for development.

1.1 Introduction to Fedora Linux for Power Users

Fedora Linux is engineered for users who want early access to modern open-source technologies without compromising system integrity or security. Its release cycle ensures access to up-to-date development toolchains, desktop environments, and system-level innovations, making it an excellent choice for developers, DevOps engineers, and technical professionals.

What sets Fedora apart is its strong commitment to upstream-first development. This means the tools and features you use in Fedora often appear first in the broader Linux ecosystem. It integrates tightly with systemd, Wayland, SELinux, firewalld, and Podman—giving you a sandboxed, policy-controlled environment out of the box.

For developers and power users, Fedora provides:

- Native support for containers using **Podman**, a rootless Docker alternative.
- A reliable package manager, **DNF**, that supports plugin extensions, rollback, and modularity.
- An immutable variant, **Fedora Silverblue**, tailored for container-based workflows and rollback-safe development.
- Access to multiple editions and spins optimized for various use cases—GNOME for Workstation, headless builds for Server, or minimal setups for VMs and containers.

The default installation of Fedora Workstation includes a clean GNOME desktop environment, useful system tools, and a solid foundation for custom environments. For example, unlike distributions bloated with preinstalled packages, Fedora encourages user-driven configuration, which aligns well with development best practices.

Fedora's security posture is also advanced by default. SELinux is enabled in enforcing mode, the firewall is active upon installation, and packages are signed. These settings reduce your exposure to common attack vectors while still being flexible for local development tasks.

For users coming from Ubuntu or Debian, the shift to Fedora involves adapting to DNF-based workflows, the RPM packaging system, and the separation of free vs non-free repositories. These changes introduce more precise control over system state and package behavior.

To verify your Fedora version at any point, use:

bash

```
cat /etc/fedora-release
```

For system architecture or kernel info:

```
bash
uname -srmo
```

These commands confirm you are working with the intended Fedora release and architecture—important when building custom binaries or using specific language runtimes.

Whether you're building apps, deploying containers, managing infrastructure, or simply want a more controlled desktop, Fedora provides a stable, secure, and highly adaptable environment that caters to modern workflows.

1.2 Choosing the Right Fedora Spin or Edition

Selecting the appropriate Fedora variant is critical to aligning your system with your workflow. Fedora offers several editions, each optimized for different user profiles—from traditional desktop development to container-based environments or minimalist headless servers.

For most developers targeting local GUI-based workflows, **Fedora Workstation** is the default choice. It features the GNOME desktop, DNF package manager, SELinux enforced security, and native support for containers using Podman. It also includes developer tools like GNOME Boxes for virtualization, Flatpak integration for sandboxed applications, and GNOME Software for simplified GUI-based package management.

If you prefer a system that uses containers as the foundation for all software and development, **Fedora Silverblue** offers an immutable architecture. The base OS is read-only and updates atomically, making it ideal for developers working with Flatpak, Toolbox, and containerized development environments. It's a compelling option for those building reproducible workstations or requiring rollback features.

For infrastructure and server-side development, **Fedora Server** provides a minimal install with Cockpit preinstalled for remote web-based administration. This edition is tailored for headless operation and ideal for CI/CD nodes, container registries, or database backends. It provides more granular control over installed services and roles using `rolekit` and systemd.

Alternatively, **Fedora Spins** cater to users who prefer a non-GNOME desktop. The **KDE Plasma Spin**, for instance, is favored by developers who want greater control over their environment with a more traditional interface. Spins are lightweight and can run efficiently on older hardware or minimal VM instances.

To explore all editions, visit:

```
bash
xdg-open https://getfedora.org
```

This command opens the Fedora download page in your default browser, allowing you to visually compare editions and download ISOs.

If you're uncertain which edition fits your use case, download Fedora Workstation first and

customize it from there. Additional editions or spins can be evaluated using virtualization tools like GNOME Boxes or `virt-manager`. For example, to install a Fedora Server image in a VM:

bash
```
sudo dnf install @virtualization
virt-install \
  --name fedora-server \
  --ram 4096 \
  --vcpus 2 \
  --disk size=20 \
  --os-variant fedora40 \
  --cdrom /path/to/Fedora-Server.iso \
  --graphics spice
```

This sets up a virtualized Fedora Server for isolated experimentation, ensuring your primary development machine remains stable.

Choosing the right edition upfront prevents unnecessary configuration overhead later. Each flavor is tuned for a different type of user, so aligning the base system with your long-term goals—from container-based workflows to traditional full-stack development—saves time and improves reliability.

1.3 Installing Fedora Workstation or Server

Installing Fedora starts with preparing a clean, bootable medium and making smart decisions during setup to ensure system performance, stability, and future flexibility. Whether you're targeting the desktop-focused Workstation or the minimal Server edition, the installation process is straightforward and highly customizable.

Start by downloading the ISO for the edition you need. For Fedora Workstation, choose the live ISO with GNOME. For Fedora Server, use the minimal ISO to keep the footprint lean and tailored for backend services. Write the ISO to a USB stick using Fedora Media Writer, `dd`, or Balena Etcher. If you're using Fedora already:

bash
```
sudo dnf install mediawriter -y
```

This installs the official utility to safely write Fedora images to USB.

After booting into the USB, launch the **Anaconda installer** and follow the guided setup. During disk partitioning, select **Btrfs** as the default file system. Btrfs supports snapshots, compression, and rollback-friendly features—useful for developers testing unstable software or kernel modules. For development workloads, allocate at least 60 GB of disk space and 8 GB of RAM to ensure space for containers, SDKs, IDEs, and virtual machines.

Create a standard user account during installation and add it to the **wheel** group for administrative privileges. If you're setting up a remote server, skip the graphical interface and enable SSH access by default.

Once installed and rebooted, confirm that the network is active and working. Then perform an

immediate update to ensure all system packages are current:

bash
```
sudo dnf upgrade --refresh
```

The `--refresh` flag ensures that metadata is fully synced with the latest repository state. This is important after a fresh install, especially if the ISO image was released several weeks prior.

If you're installing Fedora Server, make sure the SSH daemon is active so you can access it remotely:

bash
```
sudo systemctl enable --now sshd
```

This command both starts the SSH service and enables it to launch at every boot. You can then connect using:

bash
```
ssh your_user@server_ip_address
```

For headless setups, this step is essential for remote provisioning and software deployment.

To verify which edition you've installed:

bash
```
cat /etc/os-release
```

This outputs a summary including whether it's a Workstation or Server variant.

Whether you're preparing a developer laptop or configuring a backend server, making smart installation choices—like selecting Btrfs, allocating enough disk, and immediately updating the system—sets the stage for a reliable and efficient Fedora-based environment.

1.4 First-Time Setup: Hostname, Users, and SSH

A clean Fedora installation should be followed by a structured system setup to ensure proper identification, secure user access, and remote connectivity. These early steps form the foundation for a smooth development experience, especially when working with remote repositories, containers, or virtual machines.

To assign your machine a meaningful identity on the network, set a permanent hostname. This helps during SSH connections, DNS configuration, and when managing multiple systems:

bash
```
sudo hostnamectl set-hostname devbox.local
```

This command updates the system hostname to `devbox.local` . You can verify the change with:

bash
```
hostnamectl
```

After setting the hostname, update your `/etc/hosts` file to reflect the new name, especially if you’re using local name resolution or hosting services:

bash
```
127.0.0.1   localhost devbox.local
```

Next, create a primary non-root user with administrative privileges. This user should belong to the `wheel` group, which is allowed to use `sudo` :

bash
```
sudo useradd -m -G wheel -s /bin/bash emmie
sudo passwd emmie
```

The `-m` flag ensures a home directory is created, `-G wheel` adds the user to the admin group, and `-s /bin/bash` sets the default shell to Bash.

To confirm that the user has sudo rights, try switching users and running a simple privileged command:

bash
```
su - emmie
sudo dnf check-update
```

If `sudo` prompts for your password and performs the check, the setup is correct.

For secure remote access, configure SSH with key-based authentication. On your local machine, generate a key pair:

bash
```
ssh-keygen -t ed25519 -C "fedora-dev"
```

This creates a public and private key in `~/.ssh/` . Next, copy your public key to the Fedora system:

bash
```
ssh-copy-id emmie@192.168.1.10
```

Replace the IP with the Fedora machine’s address. This command appends your public key to the `~/.ssh/authorized_keys` file on the target system.

To enhance security, disable password-based login and force SSH to use keys only. Open the SSH daemon config:

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

Edit or add the following lines:

nginx
```
PasswordAuthentication no
PermitRootLogin no
```

Save and restart the service:

bash

```
sudo systemctl restart sshd
```

At this point, only users with a valid SSH key can log in, significantly reducing the risk of brute-force attacks.

If you're planning to connect frequently via SSH, define a host block in your `~/.ssh/config` file:

```
nginx
Host devbox
    HostName 192.168.1.10
    User emmie
    IdentityFile ~/.ssh/id_ed25519
```

This allows you to connect simply with:

```
bash
ssh devbox
```

Setting your hostname, managing users securely, and configuring SSH for key-based authentication are foundational steps for a secure and manageable Fedora development system. These actions prepare your environment for remote workflows, collaboration, and future automation without compromising security.

1.5 Updating Your System and Enabling RPM Fusion

After installation, the first priority is to bring your Fedora system up to date. This ensures you’re running the latest kernel, tools, and security patches. Fedora follows a fast release cycle, so updates often include critical improvements for stability and development support.

Begin by synchronizing your system with the latest package metadata and applying all available updates:

```
bash
sudo dnf upgrade --refresh
```

The `--refresh` flag forces DNF to discard any cached repository metadata and pull fresh information. This is particularly useful right after installation, as the ISO may be outdated compared to what’s available online. Once the upgrade completes, a reboot is recommended if the kernel was updated:

```
bash
sudo reboot
```

To check the current kernel version before and after reboot:

```
bash
uname -r
```

A system up to date with the latest packages reduces compatibility issues when installing software development kits, container runtimes, or drivers.

Fedora prioritizes free and open-source software by default, but many commonly used developer

and multimedia tools are hosted in third-party repositories. RPM Fusion bridges this gap by offering both **free** and **nonfree** software that adheres to Fedora's policies but isn't included in the official repositories.

To enable RPM Fusion, install both repositories using the following command:

bash
```
sudo dnf install \
  https://download1.rpmfusion.org/free/fedora/rpmfusion-free-release-$(rpm -E %fedora).noarch.rpm \
  https://download1.rpmfusion.org/nonfree/fedora/rpmfusion-nonfree-release-$(rpm -E %fedora).noarch.rpm
```

The `$(rpm -E %fedora)` part dynamically inserts your Fedora version to ensure you're pulling the correct repository files.

Once installed, verify that the repositories have been added:

bash
```
dnf repolist
```

Look for entries labeled `rpmfusion-free` and `rpmfusion-nonfree`. These repositories now give you access to packages like proprietary NVIDIA drivers, additional media codecs, and tools that are essential in many development scenarios.

For developers working with multimedia apps or testing video/audio features, install the core codec group:

bash
```
sudo dnf groupupdate multimedia --setop="install_weak_deps=False" --exclude=PackageKit-gstreamer-plugin
sudo dnf groupupdate sound-and-video
```

This command installs only the essential multimedia packages and avoids unnecessary plugin installations, keeping the system lean.

RPM Fusion also enables access to useful developer tools like VirtualBox, Steam, and GStreamer plugins that are otherwise unavailable through the default Fedora repos.

Ensuring your system is updated and extended with RPM Fusion provides a solid foundation for software development, media handling, and hardware compatibility. It prepares your Fedora environment to support a broader ecosystem of tools while maintaining the system's stability and performance.

Chapter 2: Power User Tools and Customization

Fedora’s true strength for developers and technical users lies in its flexibility and support for powerful customizations. This chapter walks through essential enhancements that make the command line more productive, streamline multitasking, and optimize the overall workflow. Whether you're managing projects in multiple shells or customizing system behavior, these tools offer efficiency and control.

2.1 Terminal Multiplexers: tmux and Tilix

For developers and system administrators, efficient multitasking within the terminal is essential. Terminal multiplexers like tmux and advanced terminal emulators like Tilix offer powerful ways to manage multiple sessions, split views, and improve productivity during long or complex workflows.

tmux: A Command-Line Multiplexer

tmux lets you create multiple terminal windows and panes within a single terminal session, maintain persistent environments, and resume sessions after disconnects. It’s particularly valuable when working remotely over SSH or managing multiple builds and logs simultaneously.

Install tmux:

bash
```
sudo dnf install tmux -y
```

To begin a new session named dev :

bash
```
tmux new -s dev
```

This starts a new tmux session and drops you into its shell. Within tmux, you can split the terminal horizontally:

text
```
Ctrl-b "
```

Or vertically:

text
```
Ctrl-b %
```

The prefix Ctrl-b tells tmux to expect a command, followed by a specific character for the action. This lets you manage multiple panes side-by-side—ideal for editing, compiling, and monitoring logs in parallel.

To detach from a session without closing it:

text
```
Ctrl-b d
```

List existing sessions:

bash
tmux ls

And to reattach:

bash
tmux attach -t dev

This feature is invaluable when working on remote servers, as you can resume where you left off even after disconnection. You can also script startup layouts using .tmux.conf to automate pane creation and commands.

Tilix: A GNOME-Tailored Tiling Terminal

While tmux is entirely terminal-based, Tilix provides a modern, graphical tiling terminal emulator for GNOME users. It supports drag-and-drop pane splitting, persistent sessions, quake-style drop-down mode, and profile-specific configurations.

Install Tilix:

bash
sudo dnf install tilix -y

Launch it from the GNOME menu or terminal:

bash
tilix

Right-click inside a terminal pane to split vertically or horizontally. Each pane can run its own shell session, and you can easily close or rearrange them. Tilix remembers your layout and session state, which is helpful when restarting your development environment.

To start Tilix in quake mode (drop-down from the top of the screen), add this to your startup applications:

bash
tilix --quake

It can also be integrated with GNOME terminal profiles for specific shells like Zsh or Fish.

Choosing Between the Two

Use tmux when you need persistence, low-resource multiplexing, or when working remotely via SSH. It's scriptable, fast, and highly configurable. Choose Tilix when working in a local graphical environment where drag-and-drop pane management and visual feedback are more beneficial.

Both tools dramatically reduce the need to open multiple terminal windows, enabling smoother task management, faster navigation, and better system oversight for developers and administrators alike.

2.2 Customizing the Shell with Zsh and Oh-My-Zsh

Switching from Bash to Zsh can significantly enhance your command-line efficiency. Zsh offers intelligent tab completion, command history navigation, globbing improvements, and support for powerful theming and plugins. When combined with the Oh-My-Zsh framework, Zsh becomes a flexible, visually appealing, and productivity-focused shell for developers and power users.

Installing Zsh and Making It the Default Shell

Begin by installing Zsh:

bash

```
sudo dnf install zsh -y
```

Once installed, set it as your default shell:

bash

```
chsh -s /bin/zsh
```

Log out and log back in or open a new terminal window to switch to Zsh. The first time Zsh runs, it will offer to create a configuration file. You can skip the interactive setup for now—we'll handle configuration manually with Oh-My-Zsh.

Installing Oh-My-Zsh

Oh-My-Zsh is a community-driven configuration framework that adds extensibility, themes, and plugin support to Zsh. Install it using the official installer:

bash

```
sh -c "$(curl -fsSL https://raw.githubusercontent.com/ohmyzsh/ohmyzsh/master/tools/install.sh)"
```

This command clones the framework into `~/.oh-my-zsh` and replaces your current `.zshrc` with a starter config. If needed, back up your `.zshrc` beforehand.

The default configuration includes useful aliases and functions for Git, command correction, and directory navigation.

Choosing a Theme

Oh-My-Zsh supports dozens of themes, with `robbyrussell` as the default. For developers, themes like `agnoster`, `powerlevel10k`, or `bureau` offer Git status, command timers, and visual indicators.

To switch themes, open `~/.zshrc` and modify:

bash

```
ZSH_THEME="agnoster"
```

Then reload:

bash

```
source ~/.zshrc
```

For best results with themes like `powerlevel10k`, install a Nerd Font and configure your

terminal emulator to use it.

Enabling Productivity Plugins

Plugins extend Zsh's capabilities. Common ones for developers include:

- `git` : adds shortcuts like `gst` (git status), `gl` (git pull), etc.
- `zsh-autosuggestions` : provides real-time command suggestions from history.
- `zsh-syntax-highlighting` : highlights valid commands, flags, and arguments.

Install additional plugins by cloning them into the custom plugin directory:

bash

```
git clone https://github.com/zsh-users/zsh-autosuggestions ${ZSH_CUSTOM:-~/.oh-my-zsh/custom}/plugins/zsh-autosuggestions

git clone https://github.com/zsh-users/zsh-syntax-highlighting.git ${ZSH_CUSTOM:-~/.oh-my-zsh/custom}/plugins/zsh-syntax-highlighting
```

Enable them in your `.zshrc` :

bash

```
plugins=(git zsh-autosuggestions zsh-syntax-highlighting)
```

Reload the shell:

bash

```
source ~/.zshrc
```

These plugins help reduce typing, prevent syntax errors, and speed up frequent operations.

Creating Custom Aliases and Shortcuts

To add your own aliases, append them to the bottom of `~/.zshrc` . For example:

bash

```
alias gs='git status'
alias gco='git checkout'
alias ll='ls -lah --color=auto'
```

This simplifies frequent commands and saves keystrokes during daily work.

2.3 Productivity with GNOME Extensions and Keyboard Shortcuts

Fedora Workstation ships with GNOME as its default desktop environment, offering a clean and modern interface. However, out of the box, GNOME lacks some features that power users often rely on for rapid navigation and multitasking. With the help of GNOME Shell extensions and custom keyboard shortcuts, you can transform your Fedora desktop into a productivity-focused environment tailored to your workflow.

Enabling Extension Support and Core Tools

To manage and apply GNOME extensions locally, first install the necessary integration tools:

bash
```
sudo dnf install gnome-extensions-app gnome-tweaks -y
```

The `gnome-extensions-app` provides a GUI to enable, disable, and configure extensions. `gnome-tweaks` allows fine-grained control over shell behavior, appearance, startup applications, and window focus settings.

If you're using Firefox or Chromium, add the GNOME browser extension connector:

bash
```
sudo dnf install chrome-gnome-shell -y
```

This enables seamless installation of extensions directly from the GNOME Extensions website.

Recommended Productivity Extensions

Use the `gnome-extensions-app` to browse, enable, and configure your installed extensions. Some high-value options include:

- **Pop Shell**: Introduces dynamic tiling and workspace management, allowing you to split and rearrange windows using only the keyboard.
- **Clipboard Indicator**: Provides persistent clipboard history, useful when copying between code editors, terminals, and browsers.
- **Caffeine**: Prevents the system from sleeping during long builds, downloads, or remote sessions.
- **Just Perfection**: Customizes shell elements like hiding the top bar or animations to reduce distractions.

Install these via GUI, or fetch them using the GNOME extension CLI:

bash
```
gnome-extensions install pop-shell@system76.com
```

Once installed, you can activate or deactivate extensions without rebooting:

bash
```
gnome-extensions enable pop-shell@system76.com
```

Customizing Keyboard Shortcuts

Efficient keyboard navigation reduces reliance on the mouse and accelerates routine actions. Fedora allows custom bindings through the GNOME Settings application.

Navigate to **Settings > Keyboard > View and Customize Shortcuts**, and scroll to **Custom Shortcuts**.

To add a shortcut for launching the terminal:

- Name: Open Terminal
- Command: `gnome-terminal`
- Shortcut: `Ctrl + Alt + T`

For opening your favorite code editor (e.g., VS Code):

- Name: Launch Code
- Command: `code`
- Shortcut: `Ctrl + Alt + E`

These bindings are stored and remembered across sessions. To manage them via command line, use `gsettings` :

bash
```
gsettings set org.gnome.settings-daemon.plugins.media-keys custom-keybindings/custom0/ name 'Open Terminal'
gsettings set org.gnome.settings-daemon.plugins.media-keys custom-keybindings/custom0/ command 'gnome-terminal'
gsettings set org.gnome.settings-daemon.plugins.media-keys custom-keybindings/custom0/ binding '<Control><Alt>t'
```

For window management, GNOME already includes shortcuts like:

- `Super + ←/→` : Snap window left or right
- `Super + Page Up/Down` : Switch workspace
- `Super + D` : Show desktop

Review and refine these under **Settings > Keyboard > Navigation** to suit your usage pattern.

Final Optimization

Once you've applied your preferred extensions and shortcuts, save your GNOME layout using `dconf` for backup or replication on other systems:

bash
```
dconf dump /org/gnome/ > gnome-config-backup.txt
```

This creates a snapshot of your shell configuration, which you can restore later:

bash
```
dconf load /org/gnome/ < gnome-config-backup.txt
```

This setup ensures consistency across systems and helps avoid repeating customization during reinstallation.

By carefully selecting GNOME extensions and defining efficient shortcuts, you can eliminate friction and maximize speed in your daily development routine—bringing the keyboard-driven power of the terminal into the desktop environment.

2.4 File Management with Ranger, fzf, and Bat

Efficient file navigation is a critical skill for developers and power users. While graphical file managers have their place, command-line tools offer unmatched speed, automation potential, and remote usability. Tools like `ranger`, `fzf`, and `bat` significantly enhance the terminal experience by enabling quick file browsing, powerful searching, and context-aware file viewing—all without leaving the keyboard.

Ranger: A TUI File Manager for Power Users

`ranger` is a terminal-based file manager that uses a minimal yet expressive interface for navigating directories and previewing files. Its default layout splits the screen into directory views and a preview pane, making it ideal for scanning large directory trees.

Install ranger:

bash
```
sudo dnf install ranger -y
```

Launch it with:

bash
```
ranger
```

Use arrow keys to navigate, `Enter` to open a directory or file, and `:q` to quit. Press `yy` to copy a file (yank) and `pp` to paste it into a new location. The preview window updates in real time as you move the cursor, showing content or metadata of the selected file.

To open a file in your default editor:

bash
```
l
```

Ranger is fully customizable through its configuration files in `~/.config/ranger/`, where you can define file associations, preview behavior, and keybindings.

You can also launch ranger in a specific directory:

bash
```
ranger ~/projects/
```

This is useful in scripts or for rapid project browsing.

fzf: Fuzzy Finder for Lightning-Fast Search

`fzf` is a general-purpose fuzzy search tool that integrates with many workflows, especially file discovery. It works interactively, filtering input based on approximate text matches in real time.

Install fzf:

bash

```
sudo dnf install fzf -y
```

To search files in the current directory tree:

```
bash
fzf
```

You can pipe output from `find` or `rg` into `fzf` to enhance precision:

```
bash
find . -type f | fzf
```

Once a file is selected, it prints the full path to stdout. This can be combined with command substitution to open files directly:

```
bash
nano "$(fzf)"
```

You can also use `fzf` to search and run shell history:

```
bash
history | fzf
```

And re-execute the selected command:

```
bash
eval "$(history | fzf)"
```

This saves time when repeating lengthy or complex shell commands.

bat: A Smarter `cat` with Syntax Highlighting

`bat` is a drop-in replacement for `cat` that adds syntax highlighting, line numbers, and file metadata. It improves code readability during inspection and can act as a pager in command pipelines.

Install bat:

```
bash
sudo dnf install bat -y
```

On Fedora, the binary is named `batcat` to avoid conflicts with other tools:

```
bash
batcat ~/.bashrc
```

To make it easier to use, create an alias:

```
bash
echo "alias bat='batcat'" >> ~/.bashrc
source ~/.bashrc
```

Now you can run:

```
bash
```

```
bat ~/.zshrc
```

You'll get syntax-highlighted output, ideal for quickly checking config files, scripts, and logs. Use it as a pager for `git diff` :

bash
```
git diff | bat
```

This enhances visibility in large changesets or code reviews.

Combined Use for Maximum Efficiency

These tools work well together. A common pattern is to locate a file using `fzf` , inspect it with `bat` , then open it in an editor:

bash
```
bat "$(fzf)"
```

Or browse directories in `ranger` , launch a script or open a log, and quickly return to the previous task—all from a keyboard-first interface.

By integrating ranger for browsing, fzf for filtering, and bat for inspection, you create a high-speed, terminal-based file management workflow that reduces friction and dramatically improves focus in development tasks.

2.5 Using Systemd for Custom Service Management

Systemd is the default init system and service manager on Fedora. While it handles critical system services, it also empowers users to manage their own custom scripts, daemons, and development workflows as persistent services. Learning how to create, enable, and monitor your own user-defined services with systemd can dramatically simplify automation and background task management.

Creating a Custom User Service

Suppose you want a script to run automatically when you log in—such as syncing a development folder or launching a local server. First, write the script and make it executable:

bash

```
mkdir -p ~/.local/bin
```

```
nano ~/.local/bin/dev-start.sh
```

Example content:

bash
```
#!/bin/bash
# Launch local server and sync folder on login

rsync -a ~/projects/shared/ ~/backups/shared/
python3 -m http.server --directory ~/projects/shared/ 8080
```

Make the script executable:

bash
```
chmod +x ~/.local/bin/dev-start.sh
```

Now define a systemd unit file to manage this as a user-level service:

bash
```
mkdir -p ~/.config/systemd/user
nano ~/.config/systemd/user/dev-start.service
```

Example unit file:

ini
```
[Unit]
Description=Start custom dev sync and server
After=network.target

[Service]
ExecStart=/home/yourusername/.local/bin/dev-start.sh
Restart=on-failure

[Install]
WantedBy=default.target
```

Replace `yourusername` with your actual user name or use `$HOME` where supported. This configuration ensures that the script starts after the network is ready and restarts automatically if it fails.

Enabling and Starting the Service

Reload systemd to register your new unit:

bash
```
systemctl --user daemon-reexec
systemctl --user daemon-reload
```

Enable the service to run at login:

bash
```
systemctl --user enable dev-start.service
```

Start it manually to test:

bash
```
systemctl --user start dev-start.service
```

Check its status:

bash
```
systemctl --user status dev-start.service
```

View logs in real time:

bash
```
journalctl --user -u dev-start.service -f
```

This helps you debug failures, monitor output, and confirm expected behavior.

Managing Background Development Services

You can use systemd in this way to manage web servers, background build watchers, test runners, or custom monitoring scripts. For example, if you frequently run a dev container:

bash
```
ExecStart=/usr/bin/podman start --attach dev-container
```

Or for managing Node.js apps:

bash
```
ExecStart=/usr/bin/node /home/yourusername/dev/app.js
```

Using systemd ensures that these services run reliably in the background and restart as needed without requiring manual terminal sessions.

Stopping or Disabling Services

To stop the service:

bash
```
systemctl --user stop dev-start.service
```

To disable it from starting at login:

bash
```
systemctl --user disable dev-start.service
```

To delete it entirely, remove both the unit file and the script:

bash
```
rm ~/.config/systemd/user/dev-start.service
rm ~/.local/bin/dev-start.sh
```

Then reload the daemon to clean up:

bash
```
systemctl --user daemon-reload
```

Chapter 3: Package Management and Software Sources

Efficient package management is essential for building, maintaining, and troubleshooting Fedora-based systems. Whether you're developing software, running containers, or managing isolated environments, understanding Fedora’s native package tools and ecosystem gives you greater control over software installation, versioning, and configuration.

Fedora uses the **DNF package manager** for handling RPM packages, with a rich plugin system and support for advanced operations such as transaction rollbacks, package version locking, and repository pinning. Beyond DNF, Fedora offers additional methods for distributing and managing software, including **Flatpak**, **modular streams**, and infrastructure automation tools like **Ansible** and **Kickstart**.

This chapter provides practical guidance for managing Fedora’s software stack effectively—from mastering DNF’s advanced features to automating system provisioning. You'll learn how to control repositories, build custom RPMs, and evaluate container-friendly alternatives like Flatpak and Snap in development workflows.

3.1 DNF Advanced Usage and Plugin System

DNF is more than just a package installer—it’s a robust software management engine with support for version control, safe rollbacks, repository tuning, and a modular plugin architecture. Understanding these features can help developers maintain consistency across environments and recover quickly from package conflicts or unexpected updates.

Managing Transaction History and Undoing Changes

DNF tracks all operations in a local history database, enabling you to inspect and reverse package changes. This is especially useful after installing or updating packages that cause dependency issues or break critical tools.

List the transaction log:

bash
```
dnf history
```

Each entry includes a transaction ID, command type, and affected packages. To view full details for a specific transaction:

bash
```
dnf history info 23
```

To undo an unwanted operation, such as a faulty update:

bash
```
sudo dnf history undo 23
```

This restores the system to its state before that transaction, including previous versions of affected packages.

Using Version Locking to Pin Specific Packages

In a development context, it’s sometimes critical to freeze a library or tool version to maintain compatibility with other dependencies. Install the versionlock plugin:

bash
```
sudo dnf install 'dnf-command(versionlock)' -y
```

Lock a package to its current version:

bash
```
sudo dnf versionlock add python3
```

List all version-locked packages:

bash
```
dnf versionlock list
```

To remove a lock:

bash
```
sudo dnf versionlock delete python3
```

This protects against accidental upgrades that might introduce breaking changes during automated system updates.

Optimizing Installs with Parallel Downloads and Fast Mirrors

Fedora’s DNF configuration allows for multiple simultaneous downloads, reducing update time on fast connections. Check your current config:

bash
```
dnf config-manager --dump | grep max_parallel_downloads
```

If not set, enable parallel downloads and fast mirrors:

bash
```
sudo nano /etc/dnf/dnf.conf
```

Append or edit:

ini

```
max_parallel_downloads=6
fastestmirror=True
```

These settings optimize DNF’s performance when installing or upgrading large package sets.

Cleaning Up and Managing Cache

By default, DNF caches RPM files and metadata, which can consume significant space over time. Clean the cache to reclaim disk space:

bash
```
sudo dnf clean all
```

To see how much space is being used:

bash
du -sh /var/cache/dnf

You can automate cleanup by enabling the dnf-automatic plugin or running periodic scripts via systemd timers.

Discovering and Using Additional Plugins

DNF supports modular plugin extensions for tasks like automatic updates, changelog inspection, and repository switching. List all available plugins:

bash
dnf list dnf-plugins*

Install useful ones such as:

bash
sudo dnf install dnf-plugins-core

This bundle includes tools like:

- repoclosure : Checks for missing dependencies in enabled repositories.
- repodiff : Compares package versions across two repo snapshots.
- download : Lets you download RPMs without installing them.

Example: Download an RPM without installing it:

bash
dnf download neovim

This is useful for offline systems, staging, or manual inspection of dependencies.

3.2 Managing Software Repositories and Modular Streams

Fedora's flexible repository system lets you pull packages from multiple curated sources, pin them to defined release channels, and select specific "streams" of language runtimes or frameworks. Mastering these controls helps you maintain predictable build environments and avoid version drift—especially when collaborating across heterogeneous machines or CI pipelines.

Adding, Disabling, and Prioritising Repositories

DNF ships with the config-manager sub-command, which handles most day-to-day repository tasks.
List all enabled repositories:

bash
dnf repolist

Enable the official "updates-testing" repo when you need early bug fixes:

bash
```
sudo dnf config-manager --set-enabled updates-testing
```

If you later discover that a pre-release package introduces regressions, simply disable the repo and downgrade:

bash
```
sudo dnf config-manager --set-disabled updates-testing
sudo dnf downgrade <pkg-name>
```

For third-party or in-house repos, drop a `.repo` file into `/etc/yum.repos.d/`. To force DNF to prefer your internal build farm over the public mirrors, assign a higher numeric priority (1 = highest) inside the repo stanza:

ini
```
priority=1
```

Higher-priority repos override lower ones during dependency resolution, keeping your environment aligned with custom builds.

Modular Streams: Selecting Language and Framework Versions

Fedora's **Modularity** feature lets a single repository carry multiple versions (streams) of the same software, each with its own update cadence. This is invaluable when you need, for example, Node.js 18 for a legacy service while simultaneously running Node.js 20 for new projects.

List available modules and their streams:

bash
```
dnf module list nodejs
```

Output resembles:

css
```
Name   Stream   Profiles        Summary
nodejs 18 [d]   common [d], …    Node.js 18 LTS
nodejs 20       common [d], …    Node.js 20 (current)
```

Here `[d]` marks the default stream (18). To switch the whole system to Node.js 20:

bash
```
sudo dnf module reset  nodejs       # clear existing choice
sudo dnf module enable nodejs:20
sudo dnf install      nodejs
```

`dnf module reset` is vital; without it, DNF may refuse to enable a non-default stream if another is already active.
To roll back:

bash

```
sudo dnf module disable nodejs
sudo dnf install      nodejs      # reinstalls default stream (18)
```

You can mix-and-match modules—e.g., keep the default PostgreSQL 16 stream while opting into Django 4—provided their dependencies do not conflict.

Layering COPR and Extra Repositories

COPR (Cool Other Package Repo) hosts community-maintained builds for bleeding-edge libraries and niche tools. Search and enable a COPR repo in one step:

bash
```
sudo dnf copr enable atim/bottom -y
sudo dnf install bottom
```

Because COPR quality varies, treat it like a staging repo: test locally, then mirror internally if the package becomes critical to production.

Verifying and Auditing Repository Provenance

Before cloning a workstation image or committing a Dockerfile, capture the exact repo state to guarantee reproducibility:

bash
```
dnf repolist --verbose | tee repos-$(date +%F).txt
```

Store this snapshot alongside your project to document where every package originated, simplifying future audits or investigations.

By selectively enabling repositories and pinning modular streams, you gain fine-grained control over software versions without sacrificing Fedora's rapid-update ecosystem. This approach ensures that every developer and CI runner builds against identical, predictable toolchains—eliminating the "works-on-my-machine" dilemma while still letting you adopt newer stacks when you're ready.

3.3 Building and Rebuilding RPM Packages

Being able to build and rebuild RPM packages gives developers more control over software behavior, system integration, and patch management. Whether you want to recompile a package with custom flags, test a patched version of a tool, or bundle your own software for clean installation, RPM packaging is a reliable and repeatable mechanism.

Setting Up the RPM Build Environment

First, install the essential development and RPM tooling:

bash
```
sudo dnf install rpmdevtools rpm-build gcc make -y
```

Create the standard directory structure for building RPMs:

bash
```
rpmdev-setuptree
```

This creates ~/rpmbuild/ with subdirectories like SPECS , SOURCES , and BUILD . All builds will follow this layout, whether you're packaging new software or rebuilding existing ones.

Rebuilding an Existing SRPM

Start by installing the source RPM for the package you want to modify or inspect. You can find SRPMs via DNF or from Koji and Fedora's package sources.

To download and install an SRPM for a package like htop :

bash
```
dnf download --source htop
rpm -ivh htop-*.src.rpm
```

This extracts the .spec file into ~/rpmbuild/SPECS/ and sources (such as .tar.gz) into ~/rpmbuild/SOURCES/ .

Navigate to the SPECS directory and open the spec file for editing:

bash
```
cd ~/rpmbuild/SPECS
nano htop.spec
```

You can now change compiler flags, add patches, or adjust install paths. For example, to build with -O3 optimization, update the %build section:

bash
```
%build
%configure CFLAGS="-O3"
make %{?_smp_mflags}
```

To add a patch, place it in ~/rpmbuild/SOURCES/ , add a Patch0: line in the spec file, and reference it in the %prep section using %patch0 .

Building the RPM

Once your changes are ready, build the package:

bash
```
rpmbuild -ba htop.spec
```

The -ba option builds both binary and source RPMs. Upon success, the .rpm files are stored in ~/rpmbuild/RPMS/ and ~/rpmbuild/SRPMS/ .

To install and test the binary locally:

bash
```
sudo dnf install ~/rpmbuild/RPMS/x86_64/htop-*.rpm
```

If you're packaging for a specific architecture, be sure to match your build environment or use mock or containerized builds.

Building a New RPM from Scratch

For your own software, start by creating a `.spec` file using the helper:

bash
```
rpmdev-newspec myapp
```

Fill out metadata like `Name`, `Version`, `Source0`, and define `%prep`, `%build`, `%install`, and `%files` blocks.

Copy your source tarball into `~/rpmbuild/SOURCES/` and run:

bash
```
rpmbuild -ba myapp.spec
```

This builds a proper RPM you can distribute, version, and sign—making system integration smoother than ad-hoc scripts or `make install`.

Verifying and Troubleshooting

Inspect the built RPM for correctness:

bash
```
rpm -qpl ~/rpmbuild/RPMS/x86_64/myapp-*.rpm
```

This lists included files. To simulate install and catch issues without affecting the system:

bash
```
rpm -Uvh --test ~/rpmbuild/RPMS/x86_64/myapp-*.rpm
```

Review logs under `~/rpmbuild/BUILD/` and `BUILDROOT/` if the build fails, and ensure build dependencies are declared in the spec file with:

css
```
BuildRequires: pkgconfig(libcurl), gcc
```

3.4 Automating Installs with Ansible and Kickstart

For developers managing multiple machines or provisioning fresh Fedora systems, automation ensures consistency, repeatability, and speed. Fedora supports two powerful methods for this: **Ansible** for post-install configuration and **Kickstart** for automated OS installation. Together, they allow full control from bare metal setup to environment customization.

Ansible for Post-Install Configuration

Ansible uses human-readable YAML files to define configuration tasks. It requires no agent on the target system—just SSH and Python—and is ideal for automating software installs, system tweaks, and developer tool setup.

Install Ansible:

bash
```
sudo dnf install ansible -y
```

Create a simple inventory file:

ini

```
[local]
localhost ansible_connection=local
```

Save it as `inventory.ini` . Then write a playbook called `devsetup.yml` :

yaml

```
---
- name: Configure Fedora development environment
  hosts: local
  become: true

  tasks:
    - name: Install developer tools
      dnf:
        name:
          - git
          - vim
          - gcc
          - make
          - podman
        state: present

    - name: Enable RPM Fusion repos
      command: >
        dnf install -y
        https://download1.rpmfusion.org/free/fedora/rpmfusion-free-release-$(rpm -E %fedora).noarch.rpm
        https://download1.rpmfusion.org/nonfree/fedora/rpmfusion-nonfree-release-$(rpm -E %fedora).noarch.rpm
      args:
        creates: /etc/yum.repos.d/rpmfusion-free.repo
```

To run the playbook:

bash

```
ansible-playbook -i inventory.ini devsetup.yml
```

This approach ensures identical environments across machines. You can extend it to configure Zsh, enable GNOME extensions, or pull dotfiles from GitHub.

For more structured setups, group roles, templates, and handlers into an Ansible collection or use `ansible-galaxy init` to scaffold reusable roles.

Kickstart for Automated Fedora Installation

Kickstart is a declarative file format that automates Fedora installation—ideal for spinning up VMs, bare-metal systems, or lab machines without manual input. It's used during Anaconda-based installs and can fully define partitioning, users, software, and post-install scripts.

Start by generating a reference file from an existing installation:

bash

```
sudo dnf install anaconda-install-env-deps -y
sudo cp /root/anaconda-ks.cfg ~/kickstart-fedora.cfg
```

This file reflects how your system was installed and can be edited for reuse. Key sections include:

kickstart
```
lang en_US.UTF-8
keyboard us
timezone UTC --isUtc
rootpw mypassword
user --name=dev --password=secure123 --groups=wheel
auth --enableshadow --passalgo=sha512
bootloader --location=mbr
clearpart --all --initlabel
autopart --type=lvm
%packages
@^development-environment
podman
tmux
vim
%end

%post
echo "Welcome to your preconfigured Fedora system" > /etc/motd
%end
```

You can host the Kickstart file on HTTP or include it on boot media. To initiate a Kickstart install from an ISO:

ini
```
inst.ks=hd:LABEL=BOOT:/ks.cfg
```

Or via PXE:

nginx
```
linux inst.ks=http://server/kickstart-fedora.cfg
```

During unattended installs, Anaconda reads the Kickstart instructions and completes setup with no user interaction. This dramatically reduces deployment time.

Combining Ansible and Kickstart

Use Kickstart for base system provisioning and Ansible for post-install configuration. For example:

- Kickstart defines partitioning, locale, and core packages.
- Ansible configures user dotfiles, installs development tools, and sets GNOME preferences.

This layered automation strategy is effective for setting up reproducible development

environments, onboarding new machines, or deploying CI runners.

3.5 Flatpak vs Snap on Fedora: Integration and Use

Universal packaging formats like **Flatpak** and **Snap** offer application sandboxing, simplified dependencies, and distribution-agnostic compatibility. While both aim to make app deployment more portable and secure, their integration with Fedora differs significantly. Knowing which to use—and when—can streamline your development and application testing workflow.

Flatpak: First-Class Support in Fedora

Flatpak is tightly integrated with Fedora and GNOME, making it the default choice for sandboxed desktop applications. It runs apps in isolated containers while allowing them to access only what they need via permissions managed by portals.

Install the Flatpak runtime if it's not already present:

bash
```
sudo dnf install flatpak -y
```

Enable Flathub, the main Flatpak repository, which hosts thousands of popular applications:

bash
```
sudo flatpak remote-add --if-not-exists flathub https://flathub.org/repo/flathub.flatpakrepo
```

Install a GUI application, such as VLC:

bash
```
flatpak install flathub org.videolan.VLC
```

Flatpak apps are stored independently of system libraries and run in a consistent environment. This eliminates issues caused by mismatched library versions across systems.

Run any Flatpak app with:

bash
```
flatpak run org.videolan.VLC
```

To inspect installed apps and runtimes:

bash
```
flatpak list
```

Flatpak's sandboxing makes it well-suited for untrusted or proprietary GUI applications, reducing the risk of system-level compromise.

For developers, Flatpak is useful when testing apps across different runtime environments or shipping applications that must work across distributions.

Snap: Available but Not Native

Snap, developed by Canonical, is supported on Fedora but not enabled by default. It uses a centralized store (Snapcraft) and mounts each application as a squashfs loop device, which can affect startup time and disk usage.

To use Snap on Fedora:

bash
```
sudo dnf install snapd -y
sudo systemctl enable --now snapd.socket
```

Enable classic support for snaps that require direct access to system paths:

bash
```
sudo ln -s /var/lib/snapd/snap /snap
```

Install a Snap package:

bash
```
sudo snap install code --classic
```

This installs Visual Studio Code from Snapcraft. The `--classic` flag is required for applications that don't fully conform to snap's confinement model.

Snap's permissions and confinement model are stricter by design, but Fedora does not integrate snapd as deeply as Ubuntu. Snap packages cannot interact with native Fedora features like GNOME Software or SELinux policy management without extra configuration.

Snap also tends to download large core runtimes (e.g., `core22`, `gnome-42-2204`) that increase disk usage. Use:

bash
```
snap list
```

to view all installed snaps and:

bash
```
snap remove <name>
```

to uninstall.

Choosing Between Flatpak and Snap

For Fedora users, **Flatpak is the preferred option** due to native integration with the desktop, GNOME Software, and system sandboxing tools. It's well-supported across both Workstation and Silverblue variants, and aligns with Fedora's upstream-first philosophy.

Snap is appropriate if you're targeting cross-distro compatibility with Canonical-backed applications or need access to exclusive Snapcraft content. However, its external lifecycle, slower startup performance, and reliance on systemd confinement may feel less native on Fedora.

In practice:

- Use **Flatpak** for GUI applications, developer tools, and cross-distro packaging.
- Use **Snap** selectively for applications only available via Snapcraft, ensuring you test them in your development environment before deployment.

By understanding the trade-offs and integration depth of each, you can choose the best packaging tool for your workflows without compromising on security, portability, or user experience.

Chapter 4: Setting Up a Developer-Friendly Environment

A well-configured development environment is essential for productivity, reliability, and consistency across projects. Fedora offers a flexible foundation for working with multiple programming languages, containerized development environments, and modern tools like VS Code Dev Containers or CLI package managers. However, configuring this environment for real-world efficiency requires thoughtful setup—especially when working across isolated Python environments, compiling from source, or troubleshooting cross-language builds.

This chapter focuses on building a robust development environment tailored to Fedora. You'll install language runtimes and SDKs, manage virtual environments, set up popular ecosystems like Node.js and Rust, and explore container-based workflows for portability and consistency. You'll also learn to identify and resolve common issues that occur when compiling open-source projects or managing conflicting dependencies.

4.1 Installing Programming Languages and SDKs

Fedora's repositories and modular streams give you direct access to current, secure language runtimes while still allowing you to pin versions for long-term projects. Begin by ensuring the base toolchain is present:

bash
```
sudo dnf groupinstall "Development Tools" "Development Libraries" -y
```

This installs GCC, make, GDB, and essential headers that many language compilers rely on.

Python

Fedora ships with the latest Python 3.x, but isolated per-project environments keep global packages clean. Confirm the version:

bash
```
python3 --version
```

If you require an older or bleeding-edge interpreter, enable the appropriate module stream:

bash
```
sudo dnf module reset python3
sudo dnf module enable python3:3.12
sudo dnf install python3
```

This guarantees system updates respect your chosen minor release.

Java (OpenJDK)

Install the default LTS OpenJDK:

bash
```
sudo dnf install java-17-openjdk-devel -y
```

For builds needing JDK 21 (long-term support after 2024), swap streams:

bash
```
sudo dnf install java-21-openjdk-devel
```

The `alternatives` system keeps multiple JDKs side-by-side; switch with:

bash
```
sudo alternatives --config java
```

Go

The Go compiler is packaged as a modular stream. Enable the version that matches your CI or production servers:

bash
```
sudo dnf module enable go-toolset:1.22
sudo dnf install go-toolset
```

This lays binaries in `/usr/lib/golang`, keeping `/usr/local` free for other tools.

Rust

Rustup remains the simplest way to track stable, beta, or nightly channels without polluting system RPMs:

bash
```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh -s -- -y
source ~/.cargo/env
rustc --version
```

Rustup installs into your home directory, so removal is as easy as deleting `~/.cargo` and `~/.rustup`.

Node.js

Fedora's Node.js module provides maintained LTS streams. To install Node 20 while keeping system updates in sync:

bash
```
sudo dnf module reset nodejs
sudo dnf module enable nodejs:20
sudo dnf install nodejs npm
```

NPM packages install under your home by default, avoiding permission issues.

.NET

Microsoft supports Fedora via RPMs. Add the official repo once, then install SDKs as needed:

bash
```
sudo rpm --import https://packages.microsoft.com/keys/microsoft.asc
sudo tee /etc/yum.repos.d/microsoft-dotnet.repo <<'EOF'
[microsoft-dotnet]
name=Microsoft .NET Packages
baseurl=https://packages.microsoft.com/fedora/40/prod/
enabled=1
```

```
gpgcheck=1
gpgkey=https://packages.microsoft.com/keys/microsoft.asc
EOF

sudo dnf install dotnet-sdk-8.0
```

The installer places `dotnet` in your path and registers it with `dnf` for clean upgrades.

Verification and Path Hygiene

After installing any SDK, verify that binaries resolve correctly:

```
bash
which go rustc node dotnet
```

If a command does not appear in `/usr/bin` or your user-level cargo/bin, check shell startup scripts for misplaced PATH entries. Consistency here prevents unexpected compiler mismatches when switching projects.

By combining Fedora's modular streams with language-specific installers like rustup, you gain a predictable, version-controlled toolchain that integrates cleanly with system updates while remaining flexible for per-project requirements.

4.2 Managing Python Environments with Pipx, venv, and Poetry

Isolating Python tools and libraries prevents dependency conflicts, keeps system packages untouched, and makes projects reproducible. Fedora includes all the ingredients for three complementary approaches—**pipx**, **venv**, and **Poetry**—each solving a different layer of the problem: global CLI utilities, lightweight per-project isolation, and full dependency-locked application management.

Installing the Core Tools

```
bash
sudo dnf install python3-pipx python3-virtualenv python3-poetry -y
python3 -m pipx ensurepath      # adds ~/.local/bin to PATH if missing
source ~/.bashrc                # or reload your Zsh profile
```

`pipx` installs standalone CLI tools in isolated mini-environments, `venv` ships with Python for lightweight project isolation, and Poetry wraps both packaging and environment management in one workflow.

pipx — Global Python CLI Utilities, Zero Conflicts

Use pipx when you need a Python-based command-line program—*httpie*, *black*, *ansible-lint*—available system-wide but sandboxed from other packages.

```
bash
pipx install httpie
http --version
```

Why it works: pipx creates a dedicated virtual environment under `~/.local/pipx/venvs/httpie`, then symlinks the executable into `~/.local/bin`. Updating or uninstalling never touches other software:

bash
```
pipx upgrade httpie
pipx uninstall httpie
```

venv — Lightweight Per-Project Isolation

For small scripts or quick experiments, the stdlib's `venv` keeps dependencies local without extra tooling.

Create and activate a venv in your project folder:

bash
```
python3 -m venv .venv
source .venv/bin/activate
pip install requests rich
python script.py
deactivate    # return to system Python
```

Key points:

- `.venv` lives inside your repo, making it easy to delete and rebuild.
- Activation modifies `PATH` so `pip` and `python` point to the sandbox.
- Ideal for notebooks, throw-away prototypes, or when you don't need Poetry's lock files.

Poetry — Reproducible, Dependency-Locked Projects

Poetry handles dependency resolution, semantic version pinning, packaging, and virtual-env creation in one tool—perfect for applications and libraries you intend to ship or share.

Initialize a new project:

bash
```
mkdir fastapi-demo && cd fastapi-demo
poetry init --name fastapi-demo --dependency fastapi --dependency uvicorn --python "^3.12"
```

Poetry creates a `pyproject.toml` . Install dependencies and auto-create the env:

bash
```
poetry install
poetry shell      # drop into the env
```

Add or update packages:

bash
```
poetry add pytest --group dev
poetry update fastapi
```

Reproduce anywhere with one command:

bash
```
git clone … && cd fastapi-demo && poetry install
```

The generated `poetry.lock` guarantees byte-for-byte identical versions across machines and CI.

Choosing the Right Tool

Need	Best Choice
System-wide Python CLI utilities	**pipx**
Quick, disposable project sandbox	**venv**
Full dependency management & locks	**Poetry**

Combine them freely: keep *black* and *pytest* globally via pipx for every shell, spin up a venv for quick experiments, and rely on Poetry for production applications. This layered strategy delivers maximum flexibility without sacrificing reproducibility or system hygiene.

Modern development rarely sticks to a single language. Fedora's repositories, modular streams, and third-party installers make it straightforward to maintain multiple runtimes—each pinned to the precise version your projects require—without cluttering the base system.

Node.js — LTS Streams via DNF

Fedora packages Node.js in modular streams that track active LTS releases.

bash
```
sudo dnf module reset nodejs
sudo dnf module enable nodejs:20        # switch to Node 20 LTS
sudo dnf install nodejs npm
```

Why: the reset clears any previous stream, ensuring clean dependency resolution. Verify:

bash
```
node --version && npm --version
```

For global CLI tools (ESLint, Vite, etc.) keep your system tidy by directing npm to the user prefix:

bash
```
npm config set prefix ~/.local
echo 'export PATH=$PATH:~/.local/bin' >> ~/.bashrc
source ~/.bashrc
```

Go — Toolset Modules for Reproducible Builds

Fedora's `go-toolset` module bundles the compiler and standard library.

bash
```
sudo dnf module enable go-toolset:1.22
sudo dnf install go-toolset
```

`go env GOPATH` defaults to `~/go`, keeping source code and binaries under your home

directory. To compile a project with vendored dependencies:

bash
```
cd ~/projects/gin-api
go mod vendor
go build -o bin/api .
```

Vendoring locks dependencies and removes network access requirements during CI builds.

Rust — Rustup for Multi-Channel Development

Because Rust evolves quickly, `rustup` offers the most flexibility while remaining self-contained.

bash
```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh -s -- -y
source ~/.cargo/env
rustup default stable          # switch between stable, beta, nightly
```

Update everything with:

bash
```
rustup update
cargo install cargo-edit       # adds `cargo add` / `rm`
```

Rustup installs to `~/.cargo` and `~/.rustup`, so system RPMs stay untouched. Remove channels you don't need with `rustup toolchain uninstall nightly`.

Java — Parallel JDKs with Alternatives

Install multiple OpenJDK versions side by side:

bash
```
sudo dnf install java-17-openjdk-devel java-21-openjdk-devel
```

Switch globally:

bash
```
sudo alternatives --config java      # pick the JVM
sudo alternatives --config javac     # pick the compiler
```

For project-level versioning, use SDKMAN! to manage per-shell JDKs without root:

bash
```
curl -s "https://get.sdkman.io" | bash
sdk install java 21.0.3-tem
sdk use     java 17.0.11-tem
```

This isolates builds that depend on specific language features or byte-code levels.

By leveraging Fedora's modular streams, language-specific installers, and per-user configuration tools, you can maintain clean, version-locked runtimes for Node.js, Go, Rust, and Java—all without risking conflict with system packages or other projects.

4.4 Using Dev Containers with Visual Studio Code

Dev Containers let you package a project's toolchain, dependencies, and environment settings inside a reproducible image. Fedora's native container engine, **Podman**, works seamlessly with Visual Studio Code's *Dev Containers* extension, allowing you to code inside containers as if they were local directories while isolating project dependencies from the host.

Start by installing Podman and enabling its socket, which emulates the Docker API VS Code expects:

bash

```
sudo dnf install podman podman-docker -y
systemctl --user enable --now podman.socket
export DOCKER_HOST=unix:///run/user/$(id -u)/podman/podman.sock
```

`podman-docker` provides a drop-in `/usr/bin/docker` shim that forwards commands to Podman, so VS Code doesn't need modification.

Install VS Code (or Code-OSS) and add the *Dev Containers* extension from the marketplace. Open a project folder and run the command palette: **Dev Containers: Add Dev Container Configuration File**. Choose a base image—e.g., **Fedora** or **Python 3**—and VS Code generates a `.devcontainer` directory with `devcontainer.json` and `Dockerfile`.

Edit the `Dockerfile` to layer project-specific tools:

Dockerfile

```
FROM fedora:40
RUN dnf install -y \
      python3 python3-pip nodejs npm \
      && dnf clean all

USER vscode
WORKDIR /workspace
```

`USER vscode` keeps container processes non-root, matching VS Code's default UID mapping for file permissions.

Add post-create tasks in `devcontainer.json` to set up linters or fetch dependencies automatically:

json

```
{
  "name": "Fedora Dev",
  "build": { "dockerfile": "Dockerfile" },
  "postCreateCommand": "pip install -r requirements.txt && npm install",
  "remoteUser": "vscode",
  "runArgs": ["--security-opt", "label=disable"]   // avoids SELinux relabel delays
}
```

Click **Reopen in Container**. VS Code builds the image with Podman, starts a container, and

connects the editor session over an SSH-like bridge. Inside the terminal you'll see the container prompt, and any code you run uses the container's toolchain.

To commit environment updates back to version control, check in the `.devcontainer` directory. Teammates with Podman or Docker can now clone the repo and open it in an identical workspace with one click—eliminating "works on my machine" drift.

For CI parity, you can trigger the same build file in GitHub Actions:

```
yaml
jobs:
  build:
    runs-on: ubuntu-latest
    container:
     image: ghcr.io/youruser/yourproject:latest
    steps:
      - uses: actions/checkout@v4
      - run: pytest
```

Because the image is the same as your Dev Container, test failures due to environment differences disappear.

Dev Containers with VS Code and Podman give Fedora developers a lightweight yet powerful route to reproducible workspaces, consistent toolchains, and simplified onboarding—while still integrating smoothly with SELinux and cgroup policies on the host.

4.5 Troubleshooting Compilation and Dependency Issues

Even with careful environment setup, builds occasionally fail—stemming from missing headers, incompatible library versions, or SELinux restrictions. A systematic approach helps you diagnose and fix problems quickly while preserving reproducibility.

Diagnose with Verbose Output

Most build systems provide a "verbose" flag. Use it first to expose the exact command that fails:

```
bash
make V=1            # Kernel and many autotools projects
cargo build -vv      # Rust
go build -x         # Go
npm install --verbose # Node.js
```

Reading the raw compiler or linker invocation reveals which header, library, or flag is missing.

Identify Missing Build Dependencies

If GCC bails out with `fatal error: xyz.h: No such file or directory`, query DNF for the header's package:

```
bash
dnf provides */xyz.h
```

Then install:

bash
```
sudo dnf install libxyz-devel
```

*-devel RPMs ship headers and static libraries; runtime *-libs packages alone do not satisfy compile-time needs.

When pkg-config can’t locate a library, confirm its .pc file path:

bash
```
PKG_CONFIG_PATH=/usr/lib64/pkgconfig:/usr/share/pkgconfig pkg-config --cflags libpq
```

Missing? Install:

bash
```
sudo dnf install postgresql-devel
```

Resolve Conflicting Versions

Modular streams let you swap versions cleanly. If a project requires OpenSSL 3 but the system ships 3.2 while you need 3.1, enable the compatibility stream:

bash
```
sudo dnf module enable openssl:3.1
sudo dnf install openssl-devel
```

Keep notes in a BUILD.md so teammates replicate the same module enables.

Work Around SELinux Denials

When a build script writes to unusual paths or spins network daemons, SELinux may block it. Check denials:

bash
```
sudo journalctl -t setroubleshoot -f
```

If the build is legitimate, allow it temporarily:

bash
```
sudo setenforce 0   # build
sudo setenforce 1
```

For permanent fixes, adjust contexts rather than disabling SELinux globally.

Isolate in Toolbx or Podman

If dependency hell persists, spin up an isolated container:

bash
```
toolbox create --release f40 buildbox
toolbox enter buildbox
dnf install gcc make ...
```

This protects your host and keeps the experiment reproducible. Commit successful recipes into a Containerfile , turning ad-hoc fixes into documented steps.

Cache and Pin Successful Builds

Once the build succeeds, freeze versions to prevent future breakage. For Python, run:

bash
```
pip freeze > requirements.lock
```

For Rust:

bash
```
cargo generate-lockfile
```

For Node:

bash
```
npm ci --package-lock-only
```

Commit lock files to version control so CI/CD reproduces exact dependency trees.

Chapter 5: Version Control and DevOps Tools

Modern development hinges on disciplined version control and automated delivery pipelines. Fedora's repositories make advanced Git workflows, local forge hosting, and secret-management tooling straightforward to install and integrate—whether you're working solo or orchestrating multi-service deployments.

5.1 Git Power Features and Configuration

A finely tuned Git setup transforms version control from a passive checkpoint system into an active productivity booster. Fedora ships the latest Git along with optional enhancers such as git-delta (rich diffs) and git-lfs (large-file handling). The goal is to standardize identity, improve readability, and automate common tasks—without locking you into proprietary helpers.

Install Core and Enhancement Tools

bash
```
sudo dnf install git git-delta git-lfs -y
git lfs install                          # one-time LFS hook setup
```

Why: git-delta renders side-by-side diffs with moved-code detection; git-lfs offloads binaries to pointer files, keeping the repository lightweight.

Define Global Identity and Sensible Defaults

bash
```
git config --global user.name  "Your Name"
git config --global user.email dev@example.com
git config --global init.defaultBranch main   # avoid legacy branch names
git config --global pull.rebase true          # fast-forward sync
```

Why: Consistent identity avoids mismatched author lines; rebasing on pull keeps linear history.

Improve Diff and Merge Readability

bash
```
git config --global core.pager "delta --paging=never"
git config --global delta.colorMoved zebra     # highlight relocated blocks
git config --global diff.colorMoved zebra
git config --global merge.conflictstyle diff3  # shows common ancestor
```

Why: Clear visual cues speed up code reviews and conflict resolution.

Speed Up Authentication

For HTTPS remotes:

bash
```
git config --global credential.helper 'cache --timeout=7200'  # two-hour memory
```

For SSH, generate a modern Ed25519 key and enable agent forwarding:

bash

```
ssh-keygen -t ed25519 -C "fedora-dev"
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_ed25519
```

Why: Seamless authentication encourages smaller, more frequent pushes.

Craft Helpful Aliases

bash

```
git config --global alias.st  "status -sb"           # git st
git config --global alias.lg  "log --oneline --graph --decorate"
git config --global alias.co  checkout
git config --global alias.br  branch
```

Why: Fewer keystrokes reduce friction during rapid context switches.

Clean Up History Safely

Interactive rebase for the last five commits:

bash

```
git rebase -i HEAD~5
```

Amend the most recent commit without changing its message:

bash

```
git commit --amend --no-edit
```

Why: Keeps history readable before code reaches shared branches.

Enforce Consistent Line Endings and Whitespace

Create a `.gitattributes` at repo root:

arduino

```
* text=auto eol=lf
*.sh  diff
*.md  diff
```

Enable core whitespace checks:

bash

```
git config --global core.whitespace trailing-space,space-before-tab
git config --global apply.whitespace nowarn   # surface but don’t block
```

Why: Prevents cross-platform merge conflicts and CI failures.

Large Files and Binary Assets

Track design assets or datasets without bloating the repo:

bash

```
git lfs track "*.psd"
```

```
git add .gitattributes
git add path/to/file.psd
git commit -m "Add design asset via LFS"
```

Why: Repositories stay fast-cloning; CI pulls only pointers until needed.

With these configurations, Git becomes an optimized command-line companion—providing clearer diffs, smoother authentication, and guarded history—while staying entirely inside Fedora's open-source toolchain.

5.2 Using GitHub CLI and Forge Integration

The GitHub CLI (`gh`) bridges your terminal and the GitHub platform, allowing you to create issues, pull requests, releases, and workflows without ever opening a browser. For Fedora-based developers, this tight integration streamlines daily GitHub workflows, particularly when used alongside traditional Git and SSH setups.

Installing and Authenticating the GitHub CLI

Install the GitHub CLI package from Fedora's repository:

bash
```
sudo dnf install gh -y
```

Authenticate using your GitHub credentials:

bash
```
gh auth login
```

Choose "GitHub.com," then select SSH or HTTPS. Follow the prompts to authenticate through your default browser or token.

Check status:

bash
```
gh auth status
```

This confirms whether your terminal is connected to GitHub and which user is active.

Managing Repositories from the CLI

To clone a GitHub repository:

bash
```
gh repo clone fedora-infra/bodhi
cd bodhi
```

Create a new repository and push the current directory:

bash
```
gh repo create my-dev-project --public --source=. --push
```

This creates the repository on GitHub and syncs your local commits.

View repo details:

bash
gh repo view --web

This opens the project page in your browser.

Pull Requests and Branch Management

After committing changes to a feature branch:

bash
git push --set-upstream origin feature/new-auth
gh pr create --title "Add new authentication logic" --body "Implements JWT-based login"

This creates a pull request against the default branch. To preview:

bash
gh pr view --we

Check PR status and reviewer checks:

bash
gh pr checks

To fetch and test a pull request locally:

bash
gh pr checkout 135

This checks out PR #135 in a new local branch.

Issues, Comments, and Project Boards

List open issues in the current repo:

bash
gh issue list

Open a new issue with interactive prompts:

bash
gh issue create

You can assign, label, and add to projects directly via CLI. To add a comment:

bash
gh issue comment 82 --body "Confirmed. Reproducible on Fedora 40 with SELinux enforcing."

This is especially useful during triage or when working inside a Dev Container where browser access may be limited.

Forking and Upstream Sync

To fork a repository:

bash

`gh repo fork --clone=true`

This clones your fork locally and sets the origin remote. To sync with upstream later:

bash
`gh repo sync`

This pulls new commits from the upstream repository into your fork's default branch.

Continuous Integration Status

Monitor CI/CD results tied to your branch or pull request:

bash
```
gh run list
gh run watch
```

These commands let you observe GitHub Actions jobs without leaving the terminal.

By integrating GitHub CLI into your Fedora development workflow, you reduce the overhead of switching between terminal and browser. Tasks like opening pull requests, tracking CI runs, managing issues, and cloning repositories become part of your local workflow—making GitHub a seamless extension of your command-line environment.

5.3 CI/CD with GitHub Actions on Fedora

GitHub Actions provides a fully integrated CI/CD platform directly within your repository. It allows you to run automated workflows on every push, pull request, or tag, supporting a wide range of environments including Fedora-based containers and GitHub-hosted runners. For developers on Fedora, this means you can design, test, and validate builds locally before deploying them to production or sharing them with collaborators.

Workflow Structure and Triggers

A GitHub Actions workflow lives under `.github/workflows/` and is triggered by events like `push`, `pull_request`, or `schedule`. A minimal YAML definition might look like:

yaml
```yaml
# .github/workflows/test.yml
name: Fedora CI
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
```

```
      with:
        python-version: 3.12

    - name: Install dependencies
      run: pip install -r requirements.txt

    - name: Run tests
      run: pytest
```

This configuration runs your Python test suite on every push or PR against `main`, using the latest GitHub Ubuntu runner. Even though you use Fedora locally, you can ensure your code runs cleanly in a standardized cloud container.

Running Fedora-Based Workflows

To match your local environment more closely, specify a custom Fedora container:

```
yaml
jobs:
  build:
    runs-on: ubuntu-latest
    container:
      image: fedora:40

    steps:
      - uses: actions/checkout@v4

      - name: Install tools
        run: |
          dnf install -y gcc make python3-pip curl
          pip install -r requirements.txt

      - name: Build and test
        run: make test
```

This ensures the job runs inside a Fedora 40 container using DNF, matching your development environment and reducing inconsistencies between CI and local builds.

Testing with Matrix Builds

You can test across multiple language versions or build targets using a matrix strategy:

```
yaml
jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: [3.10, 3.11, 3.12]

    container:
      image: fedora:40
```

```
    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        run: |
          dnf install -y python${{ matrix.python-version }} python${{ matrix.python-version }}-
pip
          python${{ matrix.python-version }} -m pip install -r requirements.txt
          python${{ matrix.python-version }} -m pytest
```

This ensures your project remains compatible across multiple Python releases.

Validating Locally with act

To debug workflows before pushing them to GitHub, use act:

bash

```
sudo dnf install podman -y
brew install act  # or use the prebuilt binary
act
```

This runs your GitHub workflows using Podman or Docker locally, helping you catch failures before they reach CI.

You can override secrets, simulate events, and test pull request triggers without network round-trips.

Using Secrets and Artifacts

Store deployment keys, tokens, or credentials as encrypted secrets:

- Navigate to your repository → Settings → Secrets and Variables → Actions → New repository secret.

Then access them inside your workflow:

yaml

```
env:
  SECRET_KEY: ${{ secrets.SECRET_KEY }}
```

To persist build outputs between steps or jobs:

yaml

```
- name: Archive results
  uses: actions/upload-artifact@v4
  with:
    name: test-results
    path: reports/
```

This saves logs, binaries, or coverage reports and makes them downloadable after the workflow completes.

By integrating GitHub Actions with Fedora-based containers and CI tools like act , you can maintain a reliable, automated pipeline that mirrors your development environment. This tight feedback loop accelerates testing, reduces human error, and gives you confidence that every commit is production-ready.

5.4 Local Git Server with Gitea

Setting up a local Git server can be invaluable for collaborative development in secure or offline environments. Gitea is a lightweight, open-source Git server that provides a self-hosted web interface, pull request system, issue tracking, and CI integration. It's especially well-suited for Fedora users who want a fast, minimal GitHub alternative with full control.

Installing Gitea on Fedora

To run Gitea as a system service, start by installing the binary or container version. The binary offers better performance for a permanent server setup.

1. **Create a system user and directory:**

bash
```
sudo useradd \
  --system \
  --shell /bin/bash \
  --comment 'Git Version Control' \
  --create-home \
  git
```

2. **Download the Gitea binary:**

bash
```
sudo mkdir -p /usr/local/bin
wget -O gitea https://dl.gitea.io/gitea/latest/gitea-linux-amd64
chmod +x gitea
sudo mv gitea /usr/local/bin/
```

3. **Create required directories and set permissions:**

bash
```
sudo mkdir -p /var/lib/gitea/{custom,data,log}
sudo chown -R git:git /var/lib/gitea/
sudo chmod -R 750 /var/lib/gitea/
```

4. **Create a systemd unit file to manage Gitea as a service:**

bash
```
sudo tee /etc/systemd/system/gitea.service > /dev/null <<EOF
[Unit]
Description=Gitea (Git with a cup of tea)
After=network.target

[Service]
RestartSec=2s
```

```
Type=simple
User=git
Group=git
WorkingDirectory=/var/lib/gitea/
ExecStart=/usr/local/bin/gitea web --config /etc/gitea/app.ini
Restart=always
Environment=USER=git HOME=/var/lib/gitea

[Install]
WantedBy=multi-user.target
EOF
```

5. **Enable and start the service:**

bash
```
sudo mkdir -p /etc/gitea
sudo chown -R git:git /etc/gitea
sudo systemctl daemon-reexec
sudo systemctl enable --now gitea
```

Once the service is running, navigate to http://localhost:3000 in your browser to complete the web-based setup. Configure the database (SQLite for simplicity, or PostgreSQL/MariaDB for robustness), root user, and repo paths.

Setting Up a Local Git Repository

To use Gitea, log into the web interface and create a new repository. Then push a local project:

bash
```
git remote add origin http://localhost:3000/youruser/yourrepo.git
git push -u origin main
```

To clone it on another machine in the network:

bash
```
git clone http://<your-server-ip>:3000/youruser/yourrepo.git
```

Make sure the server is reachable over the LAN, and optionally configure a reverse proxy like Nginx for HTTPS and domain-based access.

Automation and CI Integration

Gitea supports webhooks, enabling it to trigger external CI tools like Drone CI, Jenkins, or GitHub Actions-style workflows hosted locally. Configure this under Settings → Webhooks per repository, providing an endpoint for your CI server.

For example, to use Gitea with Drone CI:

bash
```
docker run \
  -v /var/lib/drone:/data \
  -e DRONE_GITEA_SERVER=http://gitea.local \
```

```
  -e DRONE_GITEA_CLIENT_ID=... \
  -e DRONE_GITEA_CLIENT_SECRET=... \
  -p 80:80 \
  --restart=always \
  --name=drone \
  drone/drone:latest
```

This binds your Gitea repositories to a private CI/CD pipeline—entirely under your control.

Running Gitea on Fedora offers power users a private, reliable Git hosting solution with minimal overhead. Whether for small teams, offline development, or custom DevOps workflows, it provides a streamlined alternative to public Git platforms while keeping you in full control of your source code and infrastructure.

5.5 Secrets Management with SOPS and GPG

Storing API keys, database passwords, or Terraform variables in plain text invites accidental leaks—especially in Git repositories. **Mozilla SOPS** (Secrets OPerationS) encrypts only the sensitive values inside JSON, YAML, ENV, or INI files, keeping the structure human-readable while protecting the data. Pairing SOPS with **GnuPG** gives each team member fine-grained decryption access without introducing a separate secrets service, making it ideal for small-to-medium projects running on Fedora.

Install and Bootstrap GPG

bash

```
sudo dnf install sops gnupg2 -y
gpg --full-generate-key            # choose RSA 4096, set user/email
gpg --list-secret-keys --keyid-format=long
```

Copy the long key ID (e.g., `AB12CD34EF56GH78`) and export the public part for teammates:

bash

```
gpg --armor --export AB12CD34EF56GH78 > public.asc
```

They import with `gpg --import public.asc`, ensuring everyone can decrypt shared secrets after files are committed.

Create an Encrypted YAML File

Prepare a template:

yaml

```
db_host: postgres.internal
db_port: 5432
db_user: admin
db_pass: ENC[]
```

Encrypt with SOPS:

bash

```
sops --encrypt --pgp AB12CD34EF56GH78 secrets.yml > secrets.enc.yml
```

SOPS rewrites only `db_pass` —or any value you leave as `ENC[]` —replacing it with an ASCII-

armored ciphertext while leaving non-secret keys intact for diff clarity.

Opening the file later:

bash
```
sops secrets.enc.yml     # auto-decrypts to $EDITOR
```

SOPS loads your GPG agent, decrypts on-the-fly, and re-encrypts on save.

Automate Decryption in CI/CD

In GitHub Actions or any Podman-based pipeline, store the same public key's **private** half as an encrypted repository secret. At runtime:

bash
```
echo "$GPG_PRIVATE_KEY" | gpg --import --
gpg --pinentry-mode loopback --passphrase "$GPG_PASSPHRASE" --decrypt-files
secrets.enc.yml
```

The workflow then runs `docker login`, `kubectl apply`, or any task requiring the cleartext credentials—without ever committing them.

Rotating Keys and Re-encrypting

When a team member leaves, generate a new GPG key, distribute its public part, then re-encrypt:

bash
```
export SOPS_PGP_FP="NEWKEYID OLDKEYID"
sops -r -i secrets.enc.yml          # -r = rekey in-place
```

SOPS replaces the ciphertext with data encrypted only to the listed fingerprints, instantly revoking access for removed keys while preserving file history.

Using SOPS with GPG on Fedora delivers git-native secret management: transparent in diffs, auditable in commits, and controlled by standard public-key crypto—no additional servers, and no risk of exposing credentials in plain text.

Chapter 6: Working with Containers Using Podman

Red Hat originally created Podman to pair Fedora's security model with the familiar Docker workflow—minus the daemon and root privileges. Today it is a first-class container engine in Fedora, offering drop-in CLI compatibility, systemd integration, and transparent OCI image support. In this chapter you will learn how to build, run, network, and secure containers entirely as an unprivileged user, and how to keep them running under systemd without resorting to ad-hoc shell scripts.

6.1 Understanding Podman and Docker Compatibility

Podman implements the Docker CLI surface while eliminating the always-running, root-privileged daemon. Each `podman` invocation spawns short-lived processes that create user-namespaced containers, then exit—aligning with Fedora's security model (SELinux + cgroups v2) and lowering attack surface. This section shows how to install Podman, enable drop-in Docker compatibility, and verify that existing Docker tooling works unchanged.

Install Podman and the Docker-CLI Shim

bash

```
sudo dnf install podman podman-docker -y   # podman-docker provides /usr/bin/docker
```

`podman-docker` delivers a wrapper that redirects any call to `docker` into Podman, so legacy scripts continue to run without edits.

Confirm Feature Parity

Most Docker commands map 1-to-1:

Docker	Podman Equivalent	Notes
`docker pull alpine`	`podman pull alpine`	OCI images in the same registries
`docker build .`	`podman build .`	Uses Buildah internally
`docker run -it alpine`	`podman run -it alpine`	Defaults to rootless user namespace
`docker ps`	`podman ps`	Same column output; no daemon needed

Try a complete cycle:

bash

```
docker pull quay.io/fedora/fedora:40
docker run --rm quay.io/fedora/fedora:40 cat /etc/fedora-release
```

Because the shim forwards to Podman, the command works even though no Docker engine is installed.

Daemonless Security Advantages

- **Rootless by Default** – Podman maps container UID 0 to an unprivileged host UID ≥ 10 000. If the container is compromised, host root is unaffected.
- **SELinux Integration** – Containers inherit the `container_t` context, preventing unauthorized host file access. No extra flags are required.
- **Socket Exposure** – Without a root-owned `/var/run/docker.sock`, attackers cannot escalate by hijacking the daemon.

Verify rootless isolation:

bash
```
podman run --rm alpine id
# should print uid=0(root) gid=0(root) INSIDE container
grep ^$(id -un): /etc/subuid /etc/subgid
# shows high-range mapping OUTSIDE
```

Migrating Existing Compose Stacks

Docker Compose v2 clients call the Docker API; Podman exposes the same API via a user socket.

bash
```
systemctl --user enable --now podman.socket
export DOCKER_HOST=unix:///run/user/$(id -u)/podman/podman.sock
docker compose up -d        # actually uses Podman under the hood
```

The variable points Compose to Podman, enabling multi-container workflows without installing Docker.

When to Use Docker Instead

- **Non-Linux Hosts** – Docker Desktop bundles a VM for macOS/Windows; Podman requires an additional hypervisor layer there.
- **Plugins That Depend on the Docker Daemon** – Rare utilities query undocumented endpoints. Test with Podman first; if incompatibilities surface, keep Docker in a disposable VM.

By installing the `podman` engine and its Docker-CLI shim, Fedora users retain the familiar Docker UX while gaining rootless execution, SELinux enforcement, and reduced background footprint—an immediate security win with zero workflow disruption.

6.2 Building and Running Containers Without Root

Rootless Podman leverages user namespaces and `subuid` / `subgid` mappings to give every developer a private slice of the host's UID space. You can build images, publish ports, and mount volumes—without ever touching `sudo`. This removes the attack surface of a privileged daemon and aligns perfectly with Fedora's SELinux policies.

Verify User Namespace Configuration

Fedora's installer pre-allocates subordinate IDs for each user. Confirm they exist:

bash
```
grep "^$(id -un):" /etc/subuid /etc/subgid
# Output should resemble: username:100000:65536
```

If absent (rare on upgraded systems), allocate them once:

bash
```
sudo usermod --add-subuids 100000-165535 --add-subgids 100000-165535 "$(id -un)"
```

Log out and back in to refresh the namespace mapping.

Pull and Inspect Images

Without root, pull an image:

bash
```
podman pull quay.io/centos/centos:stream9
podman images
```

Images are stored in `~/.local/share/containers/storage`, isolated from system-wide registries.

Inspect SELinux labels and mount points:

bash

```
podman inspect --format '{{ .GraphDriver.Data.UpperDir }}' centos:stream9
ls -Zd $(podman inspect --format '{{ .GraphDriver.Data.UpperDir }}' centos:stream9)
# should show system_u:object_r:container_file_t:s0:cXYZ,cABC
```

SELinux confines the content to `container_file_t`, blocking accidental host access.

Build an Image from a Dockerfile

Create a simple `Dockerfile`:

dockerfile
```
FROM quay.io/fedora/fedora:40
RUN dnf install -y python3-pip && dnf clean all
COPY app.py /srv/app.py
CMD ["python3", "/srv/app.py"]
```

Build it—no `sudo` needed:

bash
```
podman build -t flask-demo .
```

The build context remains under your UID; Podman invokes Buildah internally, mapping container root to an unprivileged host UID (e.g., 100000).

Run and Expose Ports

Run the container in detached mode:

bash
```
podman run -d --name web -p 5000:5000 flask-demo
```

`-p` works via `slirp4netns`, creating a user-space TCP forwarder that doesn't require privileged `iptables`. Confirm:

bash
```
curl http://127.0.0.1:5000
```

Stop and remove:

bash
```
podman stop web && podman rm web
```

Persistent Volumes and Bind Mounts

Rootless Podman can mount host directories, but the path must have an SELinux label allowing container access:

bash
```
mkdir -p ~/data
chcon -Rt container_file_t ~/data
podman run -d -v ~/data:/var/lib/app:z --name db alpine sleep infinity
```

The `:z` flag relabels only for this mount point, keeping host security intact.

Resource Limits and Cgroups v2

View cgroup assignments:

bash
```
podman stats --no-stream
systemd-cgls | grep $(podman inspect --format '{{.State.Pid}}' web)
```

As an unprivileged user, you cannot set host-wide limits, but you can adjust container CPU shares or memory reservations within your slice:

bash
```
podman run -d --cpus=1 --memory=512m alpine top
```

Cleaning Up

List dangling images and containers:

bash
```
podman images --filter dangling=true
podman container prune
```

Because everything resides in your home directory, removal never affects other users.

By enabling rootless builds, non-privileged networking, and SELinux-enforced storage, Fedora and Podman provide a production-grade container workflow that is as secure as it is convenient—no root required, no daemon to babysit, and no compromises on OCI compatibility.

6.3 Podman Compose and Networking

For developers accustomed to Docker Compose, Podman offers seamless integration via podman-compose , allowing multi-container orchestration without a root daemon. When used with rootless containers, Podman Compose relies on slirp4netns or rootlesskit to simulate networks—bridging containers together and exposing ports to the host—all within user space.

Installing podman-compose

On Fedora, install with:

bash
```
sudo dnf install -y podman-compose
```

This package provides a compatible interface for managing docker-compose.yml files using Podman under the hood.

Defining a Compose File

Let’s deploy a Flask web app with a PostgreSQL backend. Create a docker-compose.yml :

yaml
```
version: "3.9"
services:
  db:
    image: postgres:16
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: appdb
    volumes:
      - pgdata:/var/lib/postgresql/data

  web:
    build: ./web
    ports:
      - "8000:5000"
    environment:
      DATABASE_URL: postgres://user:password@db:5432/appdb
    depends_on:
      - db

volumes:
  pgdata:
```

In the web/ folder, include a simple Flask app and Dockerfile :

Dockerfile
```
FROM python:3.12-slim
RUN pip install flask psycopg2-binary
COPY app.py .
```

```
CMD ["python", "app.py"]
```

`app.py` should read from the `DATABASE_URL` and run a test query.

Launching the Stack

Run the entire stack with:

```
bash
podman-compose up -d
```

This command builds the `web` image, creates two rootless containers (one for the app and one for Postgres), and sets up an isolated bridge network for internal communication.

Check running services:

```
bash
podman ps
podman network ls
podman network inspect podman-compose_default
```

All containers share a network named after your project (e.g., `podman-compose_default`), enabling name-based resolution (`db` for Postgres).

Exposing Services

The `web` container exposes port `8000` on the host, mapped to `5000` in the container. You can access it with:

```
bash
curl http://localhost:8000
```

Podman's rootless mode maps these ports using `slirp4netns`, so no privileged networking setup is required.

Logging and Debugging

Each container logs separately. Tail them with:

```
bash
podman logs <container_id>
```

Or for Compose-managed services:

```
bash
podman-compose logs web
```

To rebuild the app after changes:

```
bash
podman-compose down
podman-compose up --build -d
```

This ensures containers are re-created from the updated image.

Networking Options

You can customize networks manually for finer control:

bash
`podman network create mydevnet`

Attach services by adding `networks:` in your compose file and assigning them explicitly.

For portless inter-container communication in CI or localhost-only setups, disable the `ports` block and use internal names (`db:5432`) directly.

Cleaning Up

To tear down the stack and remove volumes:

bash
`podman-compose down --volumes`

You can also clean up orphaned images:

bash
`podman image prune`

Using `podman-compose` allows Fedora developers to manage multi-container development environments without sacrificing rootless security, Docker Compose compatibility, or performance. Whether for local testing or remote orchestration, it offers a lightweight and efficient DevOps workflow.

6.4 Using Systemd to Manage Podman Containers

For long-running services on Fedora, managing containers with `systemd` provides native startup integration, automatic restarts, user-based isolation, and better log handling through `journald` . Podman simplifies this with built-in systemd unit generation, especially when operating in rootless mode.

Why Use systemd with Podman?

Containers started with `podman run` or `podman-compose up` exist in the current shell session. If the system or user session is restarted, these containers stop unless explicitly managed. By converting container definitions into systemd units, we can ensure they start on boot, restart on failure, and operate independently of the shell or user login.

Example: Managing a Podman Container with systemd

Start with a simple rootless container:

bash
`podman run --name flask-api -d -p 5000:5000 docker.io/library/python:3.12-slim python -m http.server 5000`

Once tested, stop and remove it:

bash

```
podman stop flask-api
podman rm flask-api
```

Now recreate it in a persistent format using `--restart-policy`:

bash
```
podman create --name flask-api --restart=always -p 5000:5000 docker.io/library/python:3.12-slim python -m http.server 5000
```

Generate the systemd unit file:

bash
```
podman generate systemd --name flask-api --files --new
```

This outputs a service file like `container-flask-api.service`. Move it to your user systemd directory:

bash
```
mkdir -p ~/.config/systemd/user
mv container-flask-api.service ~/.config/systemd/user/
```

Enable and start the container as a user service:

bash
```
systemctl --user daemon-reexec
systemctl --user daemon-reload
systemctl --user enable --now container-flask-api.service
```

This ensures that `flask-api` will start automatically every time you log in.

Managing Services

To verify the container is running and track its logs:

bash
```
systemctl --user status container-flask-api.service
journalctl --user -u container-flask-api.service
```

To restart or stop the container:

bash
```
systemctl --user restart container-flask-api.service
systemctl --user stop container-flask-api.service
```

Creating a Pod with Multiple Containers

For more complex setups, create a Pod and add containers to it:

bash
```
podman pod create --name webstack -p 8080:80
podman run -dt --pod webstack --name nginx docker.io/library/nginx:alpine
```

Then generate systemd files for the entire pod:

bash
```
podman generate systemd --name webstack --files --new
```

This creates:

- pod-webstack.service
- container-nginx.service

To start the whole stack automatically:

bash
```
mv *.service ~/.config/systemd/user/
systemctl --user daemon-reload
systemctl --user enable --now pod-webstack.service
```

pod-webstack.service manages the lifecycle of the entire pod and associated containers.

Auto-Start on System Boot (User Services)

To enable auto-start at boot (even before login), enable lingering:

bash
```
loginctl enable-linger $USER
```

This allows user services like containers to start with the system even without an active session.

Best Practices

- Always use --restart=always or --restart=on-failure for production-grade containers.
- Avoid hardcoding paths in the unit file—use podman generate systemd to maintain accuracy.
- Group related services into Pods for single-point systemd management.

Integrating Podman containers with systemd bridges traditional Linux service management with modern container workflows. It gives developers full control over service lifecycles using familiar tools, while maintaining the security and isolation benefits of rootless containers.

6.5 Debugging and Securing Containers

A misconfigured container can fail silently or expose your host to attack. Podman's tooling, Fedora's SELinux policies, and third-party scanners make it straightforward to diagnose issues and harden images—without mounting privileged sockets or running as root.

Live Inspection with Podman Exec and Podman Inspect

To attach an interactive shell to a running container:

bash
podman exec -it myapp /bin/bash

Use `podman inspect` to display JSON metadata:

bash
podman inspect --format '{{ .HostConfig.SecurityOpt }}' myapp

Look for SELinux labels (`label=type:container_t`) and seccomp profiles (`seccomp=unconfined` signals a misconfiguration).

Capturing Startup Failures

If a container exits immediately:

bash
podman logs myapp
podman --log-level=debug run myapp # prints OCI hooks and cgroup setup

Combine with systemd journals for rootless services:

bash
journalctl --user -u container-myapp.service

SELinux Denials and Remedies

SELinux blocks suspicious file or port access inside containers. To view recent AVC denials:

bash
sudo ausearch -m AVC -ts recent | aureport -f -i

If the denial is legitimate (e.g., binding an unconfined port), adjust the label rather than disabling SELinux:

bash

chcon -Rt container_file_t /opt/shared
podman run -v /opt/shared:/data:z myapp

`-Z` or `:z` relabels the mount for the container's MLS range, allowing safe access.

Network Debugging

Check namespace and port mappings:

bash
podman network inspect podman
podman port myapp

For packet-level inspection, enter the container's network namespace:

bash
nsenter --net=$(podman inspect -f '{{.State.Pid}}' myapp) -- tcpdump -n

Rootless users can also run `slirp4netns --disable-host-loopback` if localhost collisions arise.

Image Vulnerability Scanning

Install Trivy to scan for outdated libraries and CVEs:

bash
```
sudo dnf install trivy -y
trivy image --severity HIGH,CRITICAL myapp:latest
```

In CI pipelines, fail builds when critical issues surface:

bash
```
trivy image --exit-code 1 --severity CRITICAL myapp:latest
```

Limiting Capabilities and Resources

Drop unneeded Linux capabilities:

bash
```
podman run --cap-drop=all --cap-add=NET_BIND_SERVICE myapp
```

Cgroup v2 resource limits:

bash
```
podman run --memory=256m --cpus=0.5 myapp
```

Adding `--read-only` and `--tmpfs /tmp` further reduces write attack vectors.

Signing and Verifying Images

Use Sigstore (containers-image) to sign images:

bash
```
podman image sign --sign-by dev@example.com myapp docker://registry.local/myapp:1.0
```

Consumers verify before pulling:

bash
```
podman image trust set registry.local/myapp:1.0 dev@example.com
podman pull registry.local/myapp:1.0
```

By combining runtime inspection, SELinux awareness, vulnerability scanning, and cryptographic signing, you can debug failures quickly and deploy Podman containers with a security posture that rivals—or exceeds—traditional VM workloads, all while remaining rootless on Fedora.

Chapter 7: Developing with Virtual Machines and Toolboxes

Modern development environments benefit from isolation, reproducibility, and flexibility—qualities that virtual machines and containerized sandboxes provide exceptionally well. Fedora Linux offers a developer-friendly ecosystem through tools like GNOME Boxes, Virt-Manager, and Toolbx (Toolbox), enabling you to spin up isolated spaces without disrupting your main system. These environments are especially useful for cross-platform testing, dependency isolation, and experimenting with unstable packages.

This chapter explores how to harness virtual machines (VMs) and Toolbx environments effectively for software development. Whether you need to build a dedicated Fedora Server VM or a temporary container with a minimal developer stack, you'll learn how to set them up, configure them, and integrate them with your preferred development tools. We also cover advanced use cases, including networking across Podman containers, VMs, and IDEs, as well as tips to optimize performance and streamline workflow automation from the command line.

By the end of this chapter, you'll have the skills to use virtualized and containerized environments as modular building blocks of a powerful, flexible, and safe development workflow.

7.1 Creating VMs with GNOME Boxes and Virt-Manager

For developers working with different environments or isolated systems, creating virtual machines (VMs) is a foundational practice. Fedora supports two powerful GUI tools for managing VMs: **GNOME Boxes**, which is ideal for beginners or quick setups, and **Virt-Manager**, a feature-rich interface built on top of libvirt for advanced configurations. Both tools leverage KVM (Kernel-based Virtual Machine) and QEMU under the hood.

GNOME Boxes: Fast and Friendly VM Setup

GNOME Boxes is optimized for simplicity and works well for common development tasks like testing ISO images, creating isolated dev environments, or running alternate Linux distros.

To install:

bash
```
sudo dnf install gnome-boxes
```

Launch it from your application menu and select “+” to create a new VM. Boxes intelligently detects ISO images in your Downloads folder, making it easy to get started. Alternatively, you can manually select an ISO and specify memory, disk size, and CPUs.

GNOME Boxes is suitable when you need:

- A quick disposable test environment
- Lightweight VM management without extensive customization
- Simple drag-and-drop file sharing or clipboard integration

While it lacks the full feature set of libvirt, it’s perfect for fast provisioning or user-friendly interaction with virtualized Fedora, Debian, or Windows guests.

Virt-Manager: Power and Precision for Developers

For full control over your VM setup, **Virt-Manager** gives access to fine-grained virtualization features like bridged networking, PCI passthrough, snapshot management, and advanced storage pools.

To install Virt-Manager and related dependencies:

bash
```
sudo dnf install virt-manager libvirt qemu-kvm
sudo systemctl enable --now libvirtd
```

Once installed, run `virt-manager` from your desktop environment. You'll be prompted to create a new virtual machine. You can either boot from a local ISO or PXE boot from a network.

Why use Virt-Manager over Boxes?

You can specify exact CPU models, configure secure boot, attach cloud-init metadata, or assign multiple virtual NICs and storage volumes. It also supports remote hypervisor connections for managing headless or cloud-based KVM hosts.

Example command to verify that KVM virtualization is supported:

bash
```
egrep -c '(vmx|svm)' /proc/cpuinfo
```

A non-zero output confirms that your CPU supports hardware virtualization.

Creating a Fedora VM with Virt-Manager

1. Download a Fedora Workstation or Server ISO.
2. Launch Virt-Manager → New → Local install media.
3. Select the ISO, allocate RAM (e.g., 4096 MB), CPUs (e.g., 2), and disk space (e.g., 20 GB).
4. Choose “Customize configuration before install” to tweak CPU topology, enable UEFI firmware, or add additional virtual hardware.
5. Start the VM and complete Fedora setup.

Integrating VMs into Development Workflows

After setup, consider:

- Enabling **shared folders** via `9p` or Samba for exchanging code between host and

guest.

- Forwarding ports (e.g., 3000 for a local web server) from the guest to the host using Virt-Manager's network settings.
- Creating **VM snapshots** before major system upgrades or dependency changes.
- Using cloud-init with Virt-Manager to preconfigure SSH keys and scripts during first boot.

Both GNOME Boxes and Virt-Manager are backed by QEMU/KVM, so they can run side-by-side. You might use Boxes for quick testing and Virt-Manager for long-term development environments.

This dual-approach allows Fedora developers to balance ease-of-use with full control depending on the project at hand.

7.2 Using Fedora Toolbx for Isolated Dev Environments

Fedora Toolbox (Toolbx) is a developer-friendly utility that enables you to create isolated, container-based development environments without leaving the comfort of your host system. Unlike virtual machines, Toolbx containers are lightweight and tightly integrated with your desktop, allowing for seamless access to your files, graphical apps, and IDEs—while maintaining complete isolation for your tools and dependencies.

Why Use Toolbx?

Toolbx is ideal for developers who need to:

- Work with multiple versions of runtimes (Python, Node.js, Rust, etc.)
- Test packages in a clean environment
- Avoid cluttering the host system with development dependencies
- Safely experiment with new tools or build chains

It uses Podman behind the scenes but offers a much simpler interface and is tailored to developer workflows.

Installing Toolbx

Toolbx is available by default on Fedora Workstation. To install (or ensure it is installed):

bash
```
sudo dnf install toolbox
```

Verify installation:

bash
toolbox --version

Creating and Entering a Toolbox

To create a toolbox container matching your host system:

bash
toolbox create

You can also specify a different Fedora release for compatibility testing:

bash
toolbox create --release 38

To enter the container:

bash
toolbox enter

This gives you a fully functional Fedora userland that's separate from your host, but with access to your home directory, display server (Wayland/X11), and even USB devices when needed.

Installing Packages Inside the Toolbox

Once inside, use DNF to install packages without affecting the host:

bash
sudo dnf install gcc make python3-devel

This is particularly useful when building C extensions, compiling native libraries, or running sandboxed applications.

Linking with Graphical Applications and Editors

Toolbx integrates smoothly with development tools like:

- **VS Code** via code --remote-container or the Remote Containers extension
- **GNOME Terminal**, allowing you to launch a terminal session directly into the container
- GUI apps, which work due to shared XDG and DBus sessions

For example, if you run gnome-text-editor or firefox inside the toolbox, the applications will launch normally with access to your desktop.

Mounting Project Directories

Your $HOME is mounted into the toolbox by default, making it easy to access code. For more strict isolation, mount a specific directory:

bash

```
toolbox create --container go-dev --volume ~/Projects/go:/src
```

Then:

bash
```
toolbox enter --container go-dev
cd /src
```

This structure prevents toolchain conflicts across projects or languages.

Toolbx for Cross-Distro Testing

You can use images based on other distributions such as RHEL UBI, Ubuntu (with customization), or older Fedora versions to test backward compatibility.

Example using a raw image:

bash
```
toolbox create --image registry.fedoraproject.org/fedora-toolbox:37 --container legacy-dev
```

This approach is useful for developers targeting multiple deployment environments or distributions.

Removing Toolboxes

List all toolboxes:

bash
```
toolbox list
```

Remove an unused one:

bash
```
toolbox rm legacy-dev
```

This makes it easy to keep your dev environments lean and task-specific.

Toolbx offers a clean balance between isolation and integration, making it one of the most efficient tools for Fedora developers who want reproducible builds, experiment-friendly setups, and minimal host pollution.

7.3 Bridging Podman, Toolbox, and IDEs

A polished Fedora workflow often combines Toolbx for day-to-day hacking, Podman for containerized runtime parity, and an IDE such as Visual Studio Code or JetBrains IDEs for a rich editing experience. By wiring these tools together you can debug, test, and ship software from a single laptop while guaranteeing the same dependencies in CI or production.

1 · Mount Your Project Once, Re-use Everywhere

Keep source code on the host and share it with both Toolbx and Podman:

bash
```
export PROJ=~/Projects/agentic-rag
toolbox enter       # inside: cd $PROJ
podman run -it --rm \
  -v "$PROJ":/workspace:Z \
  --workdir /workspace \
  quay.io/fedora/python:3.12 bash
```

Why: SELinux label `:Z` relabels the bind-mount so both the container and host can read/write safely. Now code edits from the host IDE are instantly available inside Toolbx and Podman test containers.

2 · Launch the IDE Inside the Toolbox

The Remote Containers extension (VS Code) detects Toolbx automatically:

bash
```
toolbox enter
code --folder-uri vscode-remote://toolbox/home/$USER/Projects/agentic-rag
```

The IDE runs on the host but opens a server process within Toolbx, providing intelliSense against the toolchain installed in that container—without polluting your host.

3 · Debug a Podman Container from the Same IDE

1. Install the **Containers** extension for VS Code.
2. From the command palette choose **Containers: Attach Shell** → select the container running your app.
3. Add a *launch.json* that attaches to the process (e.g., Python remote attach or Node.js inspect port).

Example for Python attach:

jsonc
```
{
  "name": "Py Attach (Podman)",
  "type": "python",
  "request": "attach",
  "connect": { "host": "localhost", "port": 5678 },
  "pathMappings": [
    { "localRoot": "${workspaceFolder}", "remoteRoot": "/workspace" }
  ]
}
```

Expose the debug port when starting the container:

bash
```
podman run -p 5678:5678 -v "$PROJ":/workspace:Z agentic-rag python -m debugpy --listen 0.0.0.0:5678 app.py
```

4 · Run Integration Tests in Podman from Within Toolbx

Inside Toolbx, call the Podman socket as a non-root user:

bash
```
systemctl --user enable --now podman.socket
export DOCKER_HOST=unix:///run/user/$(id -u)/podman/podman.sock
pytest tests/integration
```

Your tests spin up fresh containers that mirror production images—without leaving the toolbox or requiring root.

5 · Sync Dev → CI

Add the same container build to `.github/workflows/ci.yml` :

yaml

```
jobs:
  test:
    runs-on: ubuntu-latest
    container: quay.io/fedora/python:3.12
    steps:
      - uses: actions/checkout@v4
      - run: pip install -r requirements.txt
      - run: pytest
```

Because you already built and debugged with the identical base image locally, CI surprises disappear.

6 · One-Command Environment Bootstrap

Create a script `devbox.sh` for new teammates:

bash
```
#!/usr/bin/env bash
toolbox create --container fed-dev --release 40
toolbox run --container fed-dev sudo dnf groupinstall "Development Tools" -y
```

Then instruct them to:

bash
```
./devbox.sh
toolbox enter fed-dev
code .
```

Within minutes they're coding in the same containerized environment, running Podman tests, and pushing commits that pass CI.

By binding Toolbx, Podman, and your IDE, you achieve a fluid loop: **edit ➜ build ➜ debug ➜ test ➜ commit** —all while your host stays pristine and your production container images remain the single source of runtime truth.

7.4 Optimizing VM Performance for Development

Virtual machines can match bare-metal responsiveness if you align resources, storage, and hypervisor settings with your workload. Fedora's KVM stack—QEMU, libvirt, Virt-Manager—exposes fine-grained tuning knobs that accelerate compilation, database tests, and UI latency without sacrificing stability.

Allocate vCPUs Strategically

Assign whole cores, not threads, for build-heavy tasks:

bash
```
virsh setvcpus devbox --maximum 8 --config
virsh setvcpus devbox 6 --config --live
```

Why: Matching virtual sockets to physical cores avoids NUMA cross-talk and boosts parallel compiles (e.g., `make -j$(nproc)`).

Enable CPU passthrough so the guest sees host features:

bash
```
<cpu mode='host-passthrough'/>
```

Set this in the XML via `virsh edit` . The guest can now use AVX2/AVX-512 for faster data processing.

Switch to Virtio Devices

Change the disk bus from IDE/SATA to Virtio:

xml
```
<disk type='file' device='disk'>
  …
  <target dev='vda' bus='virtio'/>
</disk>
```

Add the **virtio-scsi** controller for multiple disks:

xml
```
<controller type='scsi' model='virtio-scsi'/>
```

In Virt-Manager, pick **Virtio** or **Virtio-SCSI** under "Advanced options." Performance gain: lower I/O latency and better queue depth—critical when linking large binaries or running unit tests against SQLite / Postgres.

Enable Native I/O and Caching

Set cache mode to `none` and I/O to `native`:

xml
```
<driver name='qemu' type='qcow2' cache='none' io='native'/>
```

Why: Bypasses the host page cache, letting the guest control buffering—results in more predictable build times.

Use HugePages for Memory-Intensive Builds

Reserve HugePages on the host:

bash
```
echo 'vm.hugetlb_shm_group = 1000' | sudo tee -a /etc/sysctl.d/99-hugepages.conf
sudo sysctl -p
sudo grubby --args=default_hugepagesz=1G,hugepagesz=1G,hugepages=4 --update-kernel=ALL
```

Assign them in the XML:

xml
```
<memoryBacking>
  <hugepages/>
</memoryBacking>
```

Reboot host, start the VM, and observe reduced TLB misses during large C++ or Rust builds.

Virtio-GPU and SPICE for Smooth UI

For GUI IDEs inside the guest, attach a Virtio-GPU and enable 3D acceleration:

xml
```
<video>
  <model type='virtio' heads='1' accel3d='yes' vram='65536'/>
</video>
```

Use the **SPICE** display server with the `spice-vdagent` inside the guest to get clipboard sync and dynamic resolution.

Paravirtualized Networking

Change NIC to `virtio` for faster container registry pulls and package installs:

xml
```
<interface type='network'>
  <model type='virtio'/>
</interface>
```

Enable multiqueue to parallelize TCP streams:

xml

<driver name='vhost' queues='4'/>

Snapshot and Trim

Switch to **qcow2** with `discard=unmap` so `fstrim` inside the guest frees host disk space:

```
bash
sudo fstrim -av          # inside guest
```

Take external snapshots before mass package upgrades:

```
bash
virsh snapshot-create-as devbox pre-upgrade --disk-only --atomic
```

This guards against destructive changes and saves rebuild time if upgrades break tooling.

Apply these optimizations incrementally, benchmarking compile times or UI responsiveness after each change. A properly tuned Fedora VM can rival native performance while preserving the flexibility to roll back, clone, or script your entire dev environment.

7.5 CLI Tools for Managing VMs and Toolboxes

Graphical front-ends are convenient, but command-line utilities script repeatable environments, integrate with CI, and scale across headless servers. Fedora ships a mature libvirt toolkit (`virsh` , `virt-install` , `virt-builder`), while Toolbox offers a minimal CLI for containerized dev spaces.

virsh — Everyday VM Control

`virsh` is the swiss-army knife for starting, stopping, and inspecting KVM guests.

```
bash
virsh list --all
# show all VMs
virsh start devbox
# power on
virsh shutdown devbox
# ACPI shutdown
virsh console devbox
# attach serial console
```

Snapshots and resource tweaks:

```
bash
virsh snapshot-create-as devbox snap1
virsh setmem   devbox 6G --config     # change RAM persisted across boots
virsh setvcpus devbox 4  --live
```

Because `virsh` talks to libvirt over a socket, it works identically on remote servers with `virsh -c qemu+ssh://host/system` .

virt-install — Headless VM Provisioning

Automate image creation in one line:

bash
```
virt-install \
  --name dev-fedora \
  --ram 4096 --vcpus 4 \
  --disk path=/var/lib/libvirt/images/dev.qcow2,size=30,bus=virtio \
  --os-variant fedora40 \
  --location
https://download.fedoraproject.org/pub/fedora/linux/releases/40/Everything/x86_64/os/ \
  --graphics none --console pty,target_type=serial \
  --extra-args "console=ttyS0,115200n8 inst.text"
```

Ideal for CI pipelines that need disposable VMs or for generating a cloud-init image you'll upload to OpenStack.

virt-builder & virt-sysprep — Golden Images in Seconds

Pull an up-to-date qcow2 base image and customize it:

bash

```
virt-builder fedora-40 \
  --format qcow2 --size 15G \
  --install "git,ansible" \
  --run-command "useradd -m dev && echo dev:fedora | chpasswd" \
  -o /var/lib/libvirt/images/base-f40.qcow2
```

Reset machine-specific data before cloning:

bash
```
virt-sysprep -a /var/lib/libvirt/images/base-f40.qcow2
```

Now you can copy this disk for rapid VM creation.

toolbox CLI — Lightweight Dev Containers

Common commands:

bash
```
toolbox create --container go-sandbox --release 40
toolbox enter  go-sandbox
toolbox list
toolbox rm     go-sandbox
```

Toolbx recognizes Podman images too:

bash

```
toolbox create --image quay.io/toolbx-images/alpine-toolbox:3.20
```

distrobox (Optional) — Cross-Distro Toolboxes

If you need Debian or Arch inside Fedora:

bash
```
distrobox-create --name deb-dev --image debian:12
distrobox-enter  deb-dev
```

It reuses Podman under the hood but offers broader image support.

Vagrant + Libvirt — Declarative VM Recipes

Install plugin:

bash
```
sudo dnf install vagrant-libvirt
vagrant plugin install vagrant-libvirt
```

`Vagrantfile` snippet:

ruby
```
Vagrant.configure("2") do |config|
  config.vm.box = "fedora/40-cloud-base"
  config.vm.provider :libvirt do |libvirt|
    libvirt.memory = 4096
    libvirt.cpus   = 2
  end
end
```

Then:

bash
```
vagrant up   # builds VM
vagrant ssh  # login
vagrant halt # power off
```

Useful for teams that already share Vagrant recipes on other hypervisors.

Automating with bash + systemd-timers

Example backup of all VMs nightly:

bash
```
#!/usr/bin/env bash
for vm in $(virsh list --name); do
  virsh snapshot-create-as "$vm" auto-$(date +%F)
done
```

Schedule via a user timer:

bash
```
systemd-run --user --on-calendar=03:00 /usr/local/bin/vm-snapshot.sh
```

Mastering these CLI tools turns VM and toolbox management into repeatable code: easy to version, easy to audit, and effortless to reproduce across workstations, servers, or CI runners.

Chapter 8: Networking, Security, and Firewalls

Whether you're configuring development environments, exposing APIs, or deploying containers, network control and security posture are critical on Fedora. This chapter helps you balance functionality and safety using built-in tools like nmcli , firewalld , and SELinux , along with practical tips for remote access and service isolation.

You'll learn to script networking with nmcli , tailor firewall zones for local development, troubleshoot conflicts, and apply hardened but developer-friendly SSH and SELinux policies. These tools are foundational to working securely and confidently in both workstation and server setups.

8.1 Managing Network Interfaces with nmcli

nmcli is NetworkManager's command-line front end, giving you full control over wired, wireless, VPN, and bridge connections without touching GUI tools. Its syntax is stable across Fedora versions, making it ideal for scripts, headless servers, and automated provisioning.

Listing and Inspecting Devices

Start by enumerating hardware and active connections:

bash
```
nmcli device status           # summary of devices and states
nmcli connection show         # all configured profiles
nmcli connection show "$UUID"  # full details of one profile
```

The **TYPE** column distinguishes Ethernet, Wi-Fi, or virtual devices (e.g., tun , bridge , veth).

Creating a Static Wired Profile

For a lab server that needs a fixed address:

bash
```
nmcli connection add type ethernet \
  ifname enp1s0 con-name static-lan \
  ipv4.method manual \
  ipv4.addresses 192.168.10.42/24 \
  ipv4.gateway  192.168.10.1 \
  ipv4.dns      "192.168.10.1 9.9.9.9" \
  autoconnect yes
```

Why: autoconnect guarantees the interface comes up on boot, while separating config into a named profile keeps DHCP fallback intact.

Activate immediately:

bash
```
nmcli connection up static-lan
```

Quick Wi-Fi Connection from Terminal

Useful on minimal installations or when NetworkManager-applet is unavailable:

bash
```
nmcli device wifi list
nmcli device wifi connect MySSID password "supersafePW!"
```

The command creates and stores a WPA 2 profile under `/etc/NetworkManager/system-connections/` .

Switching Proxy or DNS on the Fly

Modify an existing profile without downtime:

bash
```
nmcli connection modify static-lan ipv4.dns "1.1.1.1 1.0.0.1"
nmcli connection reload
```

For corporate proxies:

bash
```
nmcli connection modify static-lan proxy.method manual \
  proxy.http "http://proxy.corp:3128" \
  proxy.https "http://proxy.corp:3128"
```

These settings propagate to GUI apps and CLI utilities that honor environment variables exported by NetworkManager.

Creating a Developer Bridge for VMs and Containers

Bridge the physical NIC with a virtual switch for KVM or Podman network sharing:

bash
```
nmcli connection add type bridge ifname br0 con-name br0
nmcli connection modify br0 ipv4.method manual ipv4.address 10.20.0.1/24
nmcli connection add type bridge-slave ifname enp1s0 master br0
nmcli connection up br0
```

VMs or rootful containers attached to `br0` now receive full LAN access without NAT.

Diagnosing Connectivity

Check IP and routes quickly:

bash
```
nmcli -p device show enp1s0    # pretty-printed key / value list
nmcli connection show --active # which profile is live?
nmcli general connectivity     # returns full, limited, none, or portal
```

Reset a flaky interface:

bash
```
nmcli device disconnect enp1s0
nmcli device connect    enp1s0
```

Scripting Tips

- Prefix scripts with `set -euo pipefail` and use `nmcli -f` to select output columns for predictable parsing.
- Combine with `--ask` to prompt for secrets during playbooks: `nmcli --ask device wifi connect … .`
- Use `nmcli connection reload` rather than restarting NetworkManager to apply bulk edits without dropping other interfaces.

Mastering `nmcli` provides deterministic, version-controlled networking on Fedora—crucial for automated dev environments, CI runners, and remote servers where GUI access is impractical.

8.2 Configuring a Developer-Friendly Firewall with firewalld

`firewalld` sits between your applications and the network stack, applying nftables rules dynamically without service interruption. Fedora ships it enabled, but default settings are often too restrictive for local development (blocking custom ports) or too permissive for production (exposing high-risk services). Tailoring zones and rich rules lets you iterate quickly while staying secure.

Inspect the Current State

bash

```
sudo firewall-cmd --state              # running?
sudo firewall-cmd --get-active-zones   # zone → interfaces map
sudo firewall-cmd --zone=public --list-all
```

Typical desktop installs map all NICs to the **public** zone, allowing only SSH, mDNS, and DHCP.

Create a Personal “dev” Zone

Isolate experimental ports from normal traffic:

bash

```
sudo firewall-cmd --permanent --new-zone=dev
sudo firewall-cmd --permanent --zone=dev --add-interface=enp1s0
sudo firewall-cmd --reload
```

Now the NIC enp1s0 obeys **dev** rules without touching Wi-Fi or VPN adapters that stay in **public**.

Open Common Development Ports

bash

```
sudo firewall-cmd --permanent --zone=dev --add-port=5173/tcp   # Vite
sudo firewall-cmd --permanent --zone=dev --add-port=3000-3001/tcp
sudo firewall-cmd --reload
```

Why `--permanent`: guarantees the rule survives reboot; follow with `--reload` to activate instantly.

Temporarily Expose a Port for Testing

For one-off demos, skip persistence:

bash
```
sudo firewall-cmd --add-port=8080/tcp --timeout=1h
```

After one hour, the rule expires automatically—no manual cleanup.

Use Service Definitions Instead of Ports

Many frameworks register reusable *services*:

bash
```
sudo firewall-cmd --info-service=mysql
sudo firewall-cmd --permanent --zone=dev --add-service=mysql
```

Service files live in `/usr/lib/firewalld/services/`; you can author your own XML to group multiple ports (e.g., HTTP + WebSocket).

Rich Rules for Source-Scoped Access

Lock an internal API to your subnet:

bash
```
sudo firewall-cmd --permanent --zone=dev \
  --add-rich-rule='rule family="ipv4" source address="192.168.10.0/24" port port="9000"
protocol="tcp" accept'
sudo firewall-cmd --reload
```

External hosts now see the port as closed, but local devices can reach it for integration tests.

Allow Container Bridges Safely

Rootless Podman uses `slirp4netns`, but rootful networks appear as `cni-podmanX`. Permit internal traffic only:

bash
```
sudo firewall-cmd --permanent --zone=trusted --add-interface=cni-podman0
```

The **trusted** zone allows all traffic on that interface while the rest of the system remains locked down.

CLI Tips for Rapid Iteration

- **Visualize rules**: `sudo firewall-cmd --direct --get-all-rules` dumps raw nftables for debugging.
- **Batch edits**: wrap multiple `--permanent` changes, then call `--reload` once to avoid packet drops.

- **Dry-run**: use `sudo firewall-cmd --permanent --zone=dev --add-port=4000/tcp --passthrough --dry-run` when scripting installers.

By segmenting interfaces into a **dev** zone, using temporary exceptions, and leveraging rich rules for granular control, you maintain a tight security posture without sacrificing developer velocity. `firewalld` 's dynamic rule engine means no restarts—your editors, compilers, and containers stay running while you iterate on network policy.

8.3 Using SELinux in Targeted Mode for Security and Flexibility

Security-Enhanced Linux (SELinux) provides a powerful, fine-grained access control layer that limits what processes can do—even if they're compromised. Fedora uses SELinux in **targeted** mode by default, applying restrictions only to specific confined services while leaving the rest of the system in standard discretionary access control (DAC) mode. This approach balances usability and security, especially in developer systems where some services need tighter controls without locking down the entire OS.

Confirm SELinux Status and Mode

To verify SELinux is active and enforcing:

bash
```
getenforce
sestatus
```

Expected output should include:

pgsql
```
Current mode: enforcing
Policy from config file: targeted
```

"Enforcing" means SELinux is actively blocking unauthorized actions. "Targeted" means only specific domains (e.g., `httpd_t` , `sshd_t`) are confined.

Inspect Domain Contexts

All files, processes, and ports are labeled with SELinux contexts. Use `ls -Z` to inspect labels:

bash
```
ls -Z /var/www/html
```

You'll see types like `httpd_sys_content_t` indicating the file is accessible to Apache.

To view running processes and their SELinux domains:

bash
```
ps -eZ | grep httpd
```

This helps you debug permission issues tied to SELinux restrictions.

Common Use Case: Allowing Web Server to Bind to Non-Standard Port

By default, Apache (`httpd_t`) can only bind to ports like 80 and 443. To permit another:

bash

```
sudo semanage port -a -t http_port_t -p tcp 8080
```

This command assigns port 8080 to the `http_port_t` type, allowing Apache to bind without policy violations.

Allow Access to Home Directories from Services

Running web apps from your home folder?

bash
```
sudo setsebool -P httpd_enable_homedirs on
```

Also, apply the correct file context:

bash
```
sudo chcon -R -t httpd_sys_content_t ~/myapp
```

This labels files so Apache can serve them safely. If your app needs to write data (logs, uploads), use `httpd_sys_rw_content_t`.

Handling AVC Denials with Audit2Why

When SELinux blocks something, an "AVC denial" appears in `/var/log/audit/audit.log`. To diagnose:

bash
```
sudo ausearch -m avc -ts recent
sudo ausearch -m avc -ts today | audit2why
```

This reveals which policy was violated and offers suggestions like setting booleans or relabeling files.

Relabeling for Persistent Fixes

After moving or restoring files, labels might be incorrect. Use `restorecon` to correct them:

bash
```
sudo restorecon -Rv /var/www/html
```

This ensures files match expected types based on SELinux file contexts.

Temporarily Switching to Permissive Mode for Debugging

You can tell SELinux to log denials without enforcing them:

bash
```
sudo setenforce 0   # switch to permissive
```

This is helpful for debugging policy issues without fully disabling SELinux. Switch back with:

bash
```
sudo setenforce 1
```

For persistent boot-time change (not recommended for production), edit `/etc/selinux/config`.

Creating Custom Policies (Advanced)

If your app runs in a custom location or needs uncommon access patterns:

1. Capture AVC denials in permissive mode.
2. Use audit2allow to generate a policy module.
3. Compile and load it using checkmodule , semodule_package , and semodule .

This level of control is rarely needed for standard dev workflows but offers full extensibility when required.

By understanding SELinux targeted mode and learning how to adjust file contexts, set booleans, and manage port types, you unlock flexible security that adapts to your dev environment without compromising system safety. SELinux becomes a valuable debugging and hardening tool—not a blocker—when used correctly.

8.4 Troubleshooting Port Conflicts and Service Bindings

When a service fails to start or bind to a port on Fedora, it's often due to that port already being in use, blocked by a firewall, or restricted by SELinux policies. Identifying the root cause quickly is essential for maintaining a smooth development experience, especially when working with web servers, databases, or containerized applications.

Identify Which Process is Using the Port

To find out if a specific port is already in use:

bash
```
sudo lsof -i :8080
```

This shows the process name, PID, and user occupying port 8080 . For a more comprehensive view across all ports:

bash
```
sudo ss -tuln | grep LISTEN
```

This displays all active listening sockets, helping you catch services bound to common or custom ports.

Stop or Reassign the Conflicting Service

If a service like firewalld or another instance of a server is using the port, you can stop it:

bash
```
sudo systemctl stop some-service
```

Or, better yet, reconfigure your application to use a non-conflicting port in its config file or systemd unit:

ini

```
ExecStart=/usr/bin/myapp --port=9090
```

Then reload the unit:

bash
```
sudo systemctl daemon-reexec
sudo systemctl restart myapp
```

This avoids interfering with other services.

Check Firewall Rules Blocking the Port

Fedora's firewall may be silently dropping packets. To view open ports and zones:

bash
```
sudo firewall-cmd --list-all
```

To allow a port (e.g., 3000) in the public zone:

bash
```
sudo firewall-cmd --zone=public --add-port=3000/tcp --permanent
sudo firewall-cmd --reload
```

Always test your app after modifying firewall rules to confirm accessibility.

Confirm SELinux Port Policies

SELinux may restrict services from binding to non-standard ports. For example, `httpd_t` can't use port 8888 by default. Use `semanage` to allow it:

bash
```
sudo semanage port -a -t http_port_t -p tcp 8888
```

Check existing policies with:

bash
```
sudo semanage port -l | grep http_port_t
```

Only adjust these policies if SELinux is enforcing and you've verified denial messages in `/var/log/audit/audit.log` .

Examine Systemd Socket Units

If you're using a service with a `.socket` file, like `sshd.socket` , ensure the socket isn't pre-occupying the port. Socket activation can conflict with direct port bindings.

bash
```
systemctl status sshd.socket
```

Disable it if necessary:

bash
```
sudo systemctl disable sshd.socket
```

This is particularly relevant when customizing systemd-based services.

Containers and Network Bindings

When using Podman or Docker, make sure the container isn't already bound to the host port:

bash
```
podman ps
```

To stop a conflicting container:

bash
```
podman stop container_id
```

Use `--publish` or `-p` to bind containers to different ports during creation:

bash
```
podman run -d -p 8081:80 nginx
```

Bind Privileged Ports as Non-Root (Optional)

Ports below 1024 are privileged. If a non-root user needs to bind to them, use capabilities:

bash
```
sudo setcap 'cap_net_bind_service=+ep' /usr/bin/myapp
```

This allows binding to ports like 80 or 443 without needing root privileges.

Effectively diagnosing and resolving port and service binding issues requires a blend of `ss`, `lsof`, `firewall-cmd`, and SELinux tools. Whether you're developing with native apps, containers, or web stacks, proactively managing your system's networking surfaces will prevent downtime, reduce friction, and enhance your confidence working in a complex Fedora development environment.

8.5 SSH Hardening and Secure Remote Access

Secure Shell is the developer's lifeline to servers, CI runners, and remote workstations. Fedora's default OpenSSH configuration is safe for casual use, yet small tweaks greatly reduce brute-force risks and prepare hosts for production. This section walks through hardened key generation, server-side lockdown, agent hygiene, and rate-limiting—without compromising daily convenience.

1 · Generate a Modern Key Pair

Use Ed25519 keys for speed and strong cryptography:

bash
```
ssh-keygen -t ed25519 -a 100 -C "laptop-fedora"
```

Why: the `-a 100` option adds key-stretching rounds, thwarting offline attacks if your private key is stolen.

Add the key to the agent for the current session:

bash

```
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_ed25519
```

For systems with older SSH implementations (rare on Fedora 40+), keep a fallback RSA 4096 key with a distinct comment.

2 · Lock Down `sshd_config`

Open the server file:

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

Recommended hardening directives:

nginx
```
Port 22
Protocol 2
PermitRootLogin no
PasswordAuthentication no
KbdInteractiveAuthentication no
PubkeyAuthentication yes
AuthenticationMethods publickey
ClientAliveInterval 300
ClientAliveCountMax 2
# Optional banner
Banner /etc/issue.net
```

Reload without disconnecting active sessions:

bash
```
sudo systemctl reload sshd
```

Why: disabling password and keyboard-interactive logins eliminates credential stuffing; restricting root forces attackers to breach a non-privileged account first.

3 · Treat the Firewall as an Ally

Limit SSH exposure to trusted subnets:

bash

```
sudo firewall-cmd --permanent \
  --add-rich-rule='rule family=ipv4 source address=203.0.113.0/24 service name=ssh accept'
sudo firewall-cmd --permanent --zone=public --remove-service=ssh
sudo firewall-cmd --reload
```

Why: reducing the source scope cuts log noise and brute-force bandwidth.

For dynamic IPs, expose an alternate port and hide the default:

bash
```
sudo firewall-cmd --permanent --add-port=2222/tcp
```

```
sudo firewall-cmd --reload
```

Then match `Port 2222` in `sshd_config`.

4 · Add Fail2Ban for Rate-Limiting

bash
```
sudo dnf install fail2ban
sudo systemctl enable --now fail2ban
sudo tee /etc/fail2ban/jail.local <<'EOF'
[sshd]
enabled  = true
port     = ssh
maxretry = 4
findtime = 10m
bantime  = 1h
EOF
sudo systemctl restart fail2ban
```

`fail2ban-client status sshd` shows banned IPs; adjust `bantime` as needed.

5 · Enable Two-Factor Authentication (Optional)

Install Google Authenticator PAM module:

bash
```
sudo dnf install google-authenticator -y
```

Run per user:

bash
```
google-authenticator -t -d -f -r 3 -R 30 -w 3
```

Add PAM line to `/etc/pam.d/sshd` **after** key auth succeeds:

swift
```
auth required pam_google_authenticator.so nullok
```

Update `sshd_config`:

r
```
AuthenticationMethods publickey,keyboard-interactive
```

Reload: `sudo systemctl reload sshd`.

Now login requires a key **and** a TOTP code.

6 · Protect the Client Side

Add identities on demand instead of auto-loading all keys:

bash
```
Host *
  AddKeysToAgent yes
```

```
  IdentityAgent $SSH_AUTH_SOCK
```

Use `ssh-add -t 1h keyfile` for one-hour lifetime tokens and revoke with `ssh-add -D` when leaving the workstation.

Disable forwarding unless required:

bash
```
Host *.prod
  ForwardAgent no
```

7 · Monitor and Audit

View recent authentication attempts:

bash
```
sudo journalctl -u sshd -S -2h
```

Report the five most common attacking IPs:

bash
```
sudo journalctl -u sshd | awk '/Failed password/ {print $(NF-3)}' | sort | uniq -c | sort -nr | head
```

Regular audits ensure hardening measures remain effective as threat patterns evolve.

By combining modern keys, strict server policies, network-level filtering, and optional MFA, you can rely on SSH for frictionless yet robust remote access—secure enough for production, streamlined enough for daily development.

Chapter 9: Performance, Power, and Battery Tweaks

Fedora Workstation is a powerful and efficient desktop OS by default, but developers and power users often need more granular control to squeeze out better responsiveness, reduced compile times, longer battery life, or thermal efficiency. This chapter dives into practical tools and techniques for tuning system performance on demand, monitoring hardware behavior in real time, and profiling application resource usage. Whether you're building large projects, working on battery, or debugging bottlenecks, these methods help you stay in control without sacrificing usability.

9.1 Tuning System Performance with tuned and powertop

Fedora provides two essential tools for tuning system performance: tuned and powertop. These utilities cater to different use cases— tuned is ideal for selecting predefined performance or power-saving profiles, while powertop helps identify energy inefficiencies and tweak runtime power settings dynamically. Used together, they can significantly improve responsiveness, thermal behavior, and battery life.

Start by enabling tuned, a daemon that automatically adjusts system settings based on workload characteristics. Fedora includes several profiles targeting common scenarios such as throughput performance, latency, virtual guests, and balanced usage. To begin:

bash
```
sudo dnf install tuned
sudo systemctl enable --now tuned
```

Once installed and running, list available tuning profiles:

bash
```
tuned-adm list
```

To activate a profile optimized for development and compilation performance, such as throughput-performance:

bash
```
sudo tuned-adm profile throughput-performance
```

This profile disables CPU power-saving features and enables aggressive scaling for better I/O throughput and CPU utilization. For laptops or energy-sensitive environments, you can switch to powersave or balanced.

To view the current profile:

bash
```
tuned-adm active
```

While tuned sets broad configuration defaults, powertop provides granular visibility into power consumption across devices, processes, and kernel subsystems. Install it with:

bash

```
sudo dnf install powertop
```

Launch it in interactive mode using:

```
bash
sudo powertop
```

Navigate through tabs to analyze CPU C-states, frequency scaling behavior, wake events, and tunable suggestions. The "Tunables" tab highlights optimizations that can be applied instantly. For example, enabling USB auto-suspend or setting the SATA link power management mode can reduce idle power draw.

To apply suggestions automatically at boot:

```
bash
sudo powertop --auto-tune
```

For persistent settings, integrate `powertop --auto-tune` into a custom systemd service or script during system startup.

When used together, `tuned` sets your base strategy (performance vs. power), and `powertop` allows for real-time fine-tuning. Switching between profiles is fast and does not require reboots, making it practical to optimize dynamically depending on whether you're compiling code, gaming, or preserving battery on the go.

9.2 Optimizing I/O and CPU for Build Tasks

Lengthy compiles are usually bottlenecked by two resources: CPU parallelism and filesystem throughput. Fedora ships everything you need to extract more from both without new hardware.

Start by selecting a performance-centric CPU governor. Modern kernels default to `schedutil`, which balances power and speed well, but “performance” removes ramp-up latency—useful for short, bursty compile steps:

```
bash
sudo cpupower frequency-set -g performance
```

Pair this with a tuned profile that disables deep C-states and boosts I/O elevators:

```
bash
sudo tuned-adm profile throughput-performance
```

Now address filesystem overhead. If your project resides on Btrfs, enable zstd compression to decrease disk writes and speed reads:

```
bash
sudo btrfs filesystem defragment -r -czstd ~/src/linux
```

For new checkouts, mount your work partition with `compress=zstd` in `/etc/fstab` or use per-subvolume property:

```
bash
```

```
sudo btrfs property set ~/src compress zstd
```

During large link stages, linkers perform many random reads; placing build artifacts on tmpfs can cut seconds:

bash
```
mkdir -p /dev/shm/build
cmake -B /dev/shm/build -S .
cmake --build /dev/shm/build -j$(nproc)
```

Tmpfs is RAM-backed, so ensure you have enough memory to avoid swap thrashing. When disk usage is unavoidable, elevate I/O priority for compilers while staying polite to the desktop:

bash
```
ionice -c2 -n0 nice -n -5 make -j$(nproc)
```

`ionice` class 2 sets best-effort scheduling with highest priority; `nice` gives the process CPU precedence without starving interactive apps.

Avoid redundant recompilation with **ccache** (for C/C++) or **sccache** (for Rust):

bash
```
sudo dnf install ccache
export CCACHE_DIR=$HOME/.cache/ccache
export CC="ccache gcc" CXX="ccache g++"
```

First builds populate the cache; subsequent ones skip unchanged objects, slashing build times dramatically.

Finally, ensure the kernel’s dirty writeback interval doesn’t stall your UI under heavy output:

bash
```
sudo sysctl -w vm.dirty_writeback_centisecs=1500
```

Faster flushes smooth desktop responsiveness when dozens of compiler jobs sync simultaneously.

Applied together—performance governor, tuned profile, compressed Btrfs, tmpfs staging, prioritized processes, compiler caches—these techniques consistently chop minutes off large Fedora builds while preserving system usability.

9.3 Managing Logs and Journals with journalctl

Fedora uses `systemd-journald` to collect and manage system logs, storing them in a binary format for fast querying and rich metadata filtering. `journalctl` is the command-line interface for accessing and analyzing this data efficiently, making it indispensable for diagnosing boot issues, monitoring services, and tracking security-relevant events.

To begin with, review the full system log in reverse chronological order:

bash

```
journalctl -xe
```

The `-x` option adds helpful context to messages, while `-e` jumps to the latest entry. This is particularly useful after a service failure or unexpected behavior.

To filter logs by boot session, use:

bash
```
journalctl -b
```

You can specify previous boots using `-b -1` (one boot ago), `-b -2`, and so on. This is invaluable for comparing states across restarts.

For service-specific debugging, focus the output on a systemd unit:

bash
```
journalctl -u nginx.service --since "10 minutes ago"
```

This shows recent logs for `nginx.service`, helping you pinpoint crashes, configuration issues, or port conflicts. Use `--follow` to tail the logs in real-time:

bash
```
journalctl -u nginx.service -f
```

To isolate logs by priority, such as warnings or errors, use:

bash
```
journalctl -p warning
```

You can increase precision with range filtering (`-p err..alert`) to catch serious issues only.

If you're interested in logs related to a specific user session:

bash
```
journalctl _UID=1000
```

Alternatively, filter by binary or executable:

bash
```
journalctl _COMM=sshd
```

This isolates logs from commands or services started with a given executable name.

Logs can also be viewed based on time:

bash
```
journalctl --since "2024-12-01 08:00:00" --until "2024-12-01 10:00:00"
```

This is especially helpful for correlating events across multiple services or understanding the sequence of system changes during a deployment window.

For export or archival, convert logs to plain text:

bash
```
journalctl -u sshd.service --since today > sshd-today.log
```

This makes it easy to share logs with support teams or for attaching to bug reports.

By default, logs are stored in `/var/log/journal/`. To check disk usage:

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

If you want to limit journal size to conserve disk space:

bash
```
sudo journalctl --vacuum-size=200M
```

Or to remove old entries based on time:

bash
```
sudo journalctl --vacuum-time=2weeks
```

In high-throughput systems or development environments with frequent logging, these limits prevent journal bloat.

`journalctl` also supports persistent logs if `/var/log/journal/` exists. To ensure logs survive reboots:

bash
```
sudo mkdir -p /var/log/journal
sudo systemd-tmpfiles --create --prefix /var/log/journal
```

Managing logs with `journalctl` empowers developers to diagnose and trace issues with high granularity—across boots, users, services, and timestamps—while keeping system logs maintainable and efficient.

9.4 Resource Monitoring with btop and dstat

Monitoring resource usage is essential for keeping Fedora development systems performant and responsive. Tools like `btop` and `dstat` provide intuitive and detailed insights into CPU, memory, disk I/O, and network traffic—allowing developers to diagnose bottlenecks, monitor build workloads, and maintain system health in real time.

`btop` is a modern, interactive system monitor with a responsive terminal interface and mouse support. To install it:

bash
```
sudo dnf install btop
```

Once installed, launch it with:

bash
```
btop
```

`btop` displays real-time graphs for CPU cores, memory usage, swap, disk read/write speeds,

network throughput, and active processes. Its layout allows you to instantly identify overused resources, runaway processes, or underperforming disks. You can sort processes by CPU, memory, or I/O, and filter or kill them directly from the interface.

Use the settings menu (press `M`) to adjust update frequency or switch themes for better visibility in low-contrast terminals.

For users who need exportable, script-friendly data or lightweight monitoring, `dstat` is a powerful CLI utility that merges functionalities of tools like `vmstat`, `iostat`, and `netstat`. Install it via:

bash
```
sudo dnf install dstat
```

A simple usage example:

bash
```
dstat -cdngyt
```

This command tracks CPU usage, disk I/O, network, paging, system interrupts, and process context switches—updating continuously. The output is printed as a structured table, ideal for comparing metrics side by side during heavy builds or deployments.

To monitor a specific disk device (e.g., `nvme0n1`), use:

bash
```
dstat -D nvme0n1
```

To capture output to a CSV file for later analysis:

bash
```
dstat --output perf_log.csv 10
```

This records system performance every 10 seconds, which is useful for benchmarking scripts, evaluating application behavior over time, or visualizing data in tools like LibreOffice Calc.

If you're debugging high I/O during builds or data-intensive tasks, `dstat` helps you correlate CPU saturation with disk or network throughput anomalies.

Together, `btop` and `dstat` serve different but complementary purposes: `btop` for interactive, user-friendly diagnostics, and `dstat` for continuous, customizable system telemetry. Fedora developers can rely on these tools to stay proactive about system performance, catch issues early, and optimize workflows across both graphical and headless environments.

9.5 Benchmarking and Profiling Applications

To build fast and efficient software, it’s essential to benchmark and profile applications directly in your development environment. Fedora provides a range of native tools and packages that let you measure execution time, memory usage, CPU hotspots, and I/O overhead precisely—whether you’re optimizing a CLI tool or a multi-threaded service.

Benchmarking Basics with hyperfine

For CLI applications, hyperfine is a simple yet powerful benchmarking tool that runs commands

multiple times and computes statistical results.
Install it via:
bash
sudo dnf install hyperfine

Benchmark two versions of a CLI script:
bash
hyperfine './mytool --mode fast' './mytool --mode safe'

It displays mean execution time, standard deviation, and a helpful comparison, allowing you to make performance decisions confidently.

Profiling Python, Rust, or C/C++ with perf

For deep system-level profiling, perf provides detailed insights into function calls, CPU cycles, and cache misses.
To install:
bash
sudo dnf install perf

Run a profiling session:
bash

perf record ./mybinary
perf report

The first command runs your binary while collecting kernel-level performance data; the second shows a navigable TUI with call graphs and hotspots. This is especially valuable for identifying bottlenecks in native code, loops, or system calls.
If your binary is stripped or built without debug symbols, compile it with flags like -g -O2 (GCC or Clang) to enable accurate source-level tracing.

Visual Profiling with sysprof and gnome-builder

sysprof is a system-wide visual profiler integrated with GNOME. It can capture CPU, I/O, scheduling events, and even GTK rendering performance. When paired with gnome-builder, it offers real-time visual flame graphs and a more intuitive view of CPU usage across threads.
bash
sudo dnf install sysprof
sudo sysprof

Capture a session for a custom binary or a system process, then explore aggregated metrics.

Memory Profiling with valgrind and massif

To inspect heap usage and detect memory leaks:

bash
sudo dnf install valgrind
valgrind --tool=massif ./myapp
ms_print massif.out.<PID>

This reveals memory allocation over time, helping optimize excessive heap growth or pinpoint unexpected memory retention in long-running apps.
Use valgrind --leak-check=full for direct memory leak reporting with line-level granularity.

Flame Graphs with py-spy and cargo-flamegraph

For Python developers, py-spy captures live stack traces without modifying the code:

bash
sudo dnf install py-spy
py-spy top --pid <PID>

For Rust, cargo flamegraph integrates tightly with perf:

bash
cargo install flamegraph
cargo flamegraph

This renders interactive SVG flame graphs to visualize CPU-intensive sections, making performance optimization far more actionable.

Case Example: Profiling a Flask App

You can profile a running Flask web server using py-spy like this:

bash
ps aux | grep flask # find PID
sudo py-spy record -p <PID> -o profile.svg

Open profile.svg in your browser to explore stacked call traces and identify slow endpoints or inefficient handlers.

By integrating tools like hyperfine, perf, valgrind, and flame graph generators into your dev workflow, you gain a measurable edge in delivering high-performance software. Fedora's native support for these utilities ensures that performance tuning is not an afterthought—it becomes a core development discipline.

Chapter 10: Backup, Recovery, and Maintenance

As your Fedora development system evolves, so does the risk of data loss, system corruption, or misconfiguration. Whether you're coding locally, running containerized services, or experimenting with low-level tweaks, a solid backup and maintenance strategy is non-negotiable. This chapter guides you through practical tools and workflows to safeguard your system, automate protection, and confidently recover from failure.

You'll learn how to create secure and efficient backups using tools like Borg, Restic, and rsync; how to manage system snapshots using Btrfs; how to automate recurring backup jobs with systemd timers; and how to perform rollbacks and keep your Fedora environment clean and responsive. Each section focuses on actionable, developer-friendly practices that minimize overhead and maximize reliability—ensuring that your work and configuration remain safe no matter what.

10.1 Backing Up with Borg, Restic, and rsync

A dependable backup plan combines three qualities: encryption, deduplication, and automation. Fedora's repositories include Borg Backup, Restic, and rsync—each excels in different situations, and together they cover local disks, off-site cloud buckets, and ad-hoc mirrors.

Borg Backup — encrypted deduplicated archives on disks or SSH servers

Install and initialize a new repository (local path or user@host:path):

bash
```
sudo dnf install borgbackup
borg init --encryption=repokey-blake2 /mnt/backup/borg
```

Create a daily archive of key directories:

bash
```
borg create --stats /mnt/backup/borg::home-'{now:%Y-%m-%d}' ~/Code ~/Documents
```

Prune with a sensible retention policy—seven daily, four weekly, six monthly:

bash
```
borg prune --keep-daily=7 --keep-weekly=4 --keep-monthly=6 /mnt/backup/borg
```

Restore a single file after accidental deletion:

bash
```
borg extract /mnt/backup/borg::home-2025-07-11 home/Code/main.py
```

Borg is ideal for USB drives or remote SSH servers where full deduplication and compression cut storage costs.

Restic — painless snapshots to S3, Backblaze, or any rclone-compatible target

Configure once in your shell profile:

bash

```
sudo dnf install restic
export RESTIC_REPOSITORY="s3:https://s3.us-east-1.amazonaws.com/dev-backups"
export RESTIC_PASSWORD="ChangeMe!"
export AWS_ACCESS_KEY_ID="AKIA…"
export AWS_SECRET_ACCESS_KEY="abcd…"
```

Initialize the bucket and push first snapshot:

bash
```
restic init
restic backup ~/Projects --tag fedora_dev
```

List and prune old snapshots automatically:

bash
```
restic snapshots --tag fedora_dev
restic forget --tag fedora_dev --keep-daily 7 --keep-monthly 3 --prune
```

Restic's built-in encryption, resumable uploads, and small binary footprint make it ideal for off-site or hybrid cloud backups.

rsync — quick mirrors and bootable clones

For a one-shot mirror to an external SSD:

bash
```
sudo dnf install rsync
rsync -aHAXv --delete ~/Projects /run/media/$USER/DEV_SSD/
```

Options explained:
`-a` archive mode, `-H` preserve hard links, `-A` ACLs, `-X` extended attributes, `--delete` prune removed files.
Add `--info=progress2` when copying huge directories to view throughput.

rsync shines when you need an exact file-level mirror or a bootable disk clone without the overhead of deduplication databases.

Choosing the right tool

- Use **Borg** for encrypted, deduplicated archives on local drives or SSH shares.
- Use **Restic** for straightforward, cloud-friendly snapshots and easy retention management.
- Use **rsync** for fast, no-frills folder mirrors—great for raw media or VM images.

Automate any of these with `cron` or `systemd` timers (see 10.3) to ensure backups happen even when you forget. Store recovery commands in a README inside the backup destination so anyone can restore under pressure.

10.2 Managing Snapshots with Btrfs on Fedora

Fedora Workstation uses Btrfs as the default filesystem, which offers built-in snapshot capabilities ideal for fast backups, rollbacks, and system recovery. Snapshots capture the state of a subvolume at a specific point in time without copying data, enabling near-instant, low-overhead protection of your system and data.

Understanding Btrfs Subvolumes and Snapshots

Btrfs organizes the filesystem into subvolumes, each of which can be snapshotted independently. Common subvolumes in Fedora include `/` (root), `/home`, and sometimes `/var`.

To list existing subvolumes:

bash
```
sudo btrfs subvolume list /
```

You might see paths like `@` (for `/`) or `@home` for `/home`, depending on how Fedora was installed.

Creating Manual Snapshots

Snapshots can be taken instantly using `btrfs subvolume snapshot`. For example, to snapshot your home directory:

bash
```
sudo btrfs subvolume snapshot /home /home/.snapshots/home-$(date +%F)
```

This command creates a read-write snapshot. For recovery purposes, read-only snapshots are preferable:

bash
```
sudo btrfs subvolume snapshot -r / /mnt/btrfs_snapshots/root-$(date +%F)
```

This is useful before running system updates or making risky changes.

Restoring from a Snapshot

To revert your system to a previous state, mount the snapshot and manually copy files back, or replace the current subvolume entirely. Example:

bash
```
sudo mount -o subvol=root-2025-07-11 /dev/nvme0n1p3 /mnt
sudo rsync -aAXv /mnt/ /  # Restore selectively or completely
```

This assumes `/mnt` is a mount point and the snapshot subvolume exists there.

To fully rollback (at boot level), advanced users can configure GRUB to boot from a snapshot by pointing it to the read-only subvolume.

Automating Snapshots with snapper or Timeshift

snapper is a powerful snapshot manager:

bash

```
sudo dnf install snapper
sudo snapper -c root create --description "Before system upgrade"
```

It integrates well with DNF to automatically snapshot before and after upgrades if configured.

For GUI users or those coming from Ubuntu, **Timeshift** offers a friendly alternative:

bash
```
sudo dnf install timeshift
sudo timeshift --create --comments "Weekly backup" --tags D
```

Timeshift uses rsync or Btrfs, depending on your filesystem, and focuses on system snapshots (excluding `/home` unless configured).

Best Practices

- Always store snapshots outside the subvolume being snapshotted to avoid circular references.
- Clean up old snapshots to conserve disk space using `btrfs subvolume delete` .
- Mount a dedicated `.snapshots` directory to make them easily accessible or browsable.
- Combine Btrfs snapshots with `systemd` timers for regular, automated protection (see 10.3).

Btrfs snapshots offer a modern, performant safety net for Fedora systems. When used regularly, they provide near-instant rollbacks for package upgrades, configuration mistakes, or even accidental deletions—without the need for a full backup.

10.3 Automating Backups with systemd Timers

Creating reliable and automated backups is essential for maintaining system integrity, especially in development environments where frequent changes are the norm. Fedora's systemd infrastructure allows users to automate backup tasks without relying on external cron services, offering tighter integration with the boot and user session lifecycle.

Why Use systemd Timers for Backups?

Unlike `cron` , systemd timers are more flexible, support dependencies, and integrate well with modern Linux services. You can create custom backup routines using `rsync` , `borg` , `restic` , or Btrfs snapshot commands and schedule them using systemd timer units.

Step-by-Step: Automating a Daily rsync Backup

Let's automate a simple backup using `rsync` to copy `/home` to an external disk mounted at `/mnt/backup` .

Step 1: Create the Backup Script

bash
```
sudo nano /usr/local/bin/daily-backup.sh
```

bash
```
#!/bin/bash
rsync -aAXv /home/ /mnt/backup/home-backup/
```

Make the script executable:

bash
```
sudo chmod +x /usr/local/bin/daily-backup.sh
```

Step 2: Create a systemd Service Unit

bash
```
sudo nano /etc/systemd/system/daily-backup.service
```

ini
```
[Unit]
Description=Daily Home Directory Backup
Wants=network-online.target
After=network-online.target

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

This defines the backup behavior— `Type=oneshot` means it runs and exits.

Step 3: Create a systemd Timer Unit

bash
```
sudo nano /etc/systemd/system/daily-backup.timer
```

ini
```
[Unit]
Description=Run Daily Backup Script

[Timer]
OnCalendar=*-*-* 03:00:00
Persistent=true

[Install]
WantedBy=timers.target
```

This schedules the backup to run daily at 3 AM. The `Persistent=true` option ensures the task runs even if the system was off at the exact scheduled time.

Step 4: Enable and Start the Timer

bash
```
sudo systemctl daemon-reexec
sudo systemctl enable --now daily-backup.timer
```

To verify:

bash
```
systemctl list-timers --all | grep daily-backup
```

This will show the next scheduled time and the last run result.

Automating with Btrfs Snapshots or Borg/Restic

You can modify the same timer structure to automate snapshotting:

bash
```
ExecStart=/usr/bin/btrfs subvolume snapshot -r / /mnt/snapshots/root-$(date +%%F)
```

Or for restic:

bash
```
ExecStart=/usr/bin/restic backup /home --repo /mnt/restic-repo
```

Just ensure that required environment variables or passphrases are available via systemd `EnvironmentFile` or pre-configured keys.

Monitoring and Logs

To check execution logs:

bash
```
journalctl -u daily-backup.service
```

This allows debugging any failures in execution or permission issues.

Best Practices

- Ensure external destinations are mounted before execution; use `RequiresMountsFor=/mnt/backup` in the service unit if needed.
- Avoid running backups during high-load periods or when system suspend may interrupt it.
- Consider using `RandomizedDelaySec` for timers if you're backing up many systems to avoid simultaneous I/O spikes.

Automating backups with systemd timers provides a native, reliable way to ensure data safety without relying on third-party scheduling tools. Combined with tools like rsync, Btrfs, or restic, this approach offers a lightweight yet powerful foundation for regular, unattended protection of critical files and system states.

10.4 Performing System Rollbacks and Recovery

When a system update or configuration change breaks functionality, Fedora users can leverage rollback techniques to restore stability. Thanks to Fedora’s native Btrfs filesystem and the robust snapshot tools available, it’s possible to roll back either specific directories or the entire system state with minimal effort. For critical systems and development environments, these techniques

can drastically reduce downtime and restore confidence.

Leveraging Btrfs Snapshots for Rollback

If your Fedora installation uses Btrfs (the default on Fedora Workstation), you can manually or automatically create read-only snapshots before performing upgrades or risky configurations. Rolling back is as simple as deleting the current subvolume and renaming the snapshot.

For example, to create a snapshot before upgrading:

bash
```
sudo btrfs subvolume snapshot -r / /mnt/btrfs-root/@root-before-upgrade
```

If a rollback is necessary:

bash
```
sudo mv /mnt/btrfs-root/@ /mnt/btrfs-root/@broken
sudo btrfs subvolume snapshot /mnt/btrfs-root/@root-before-upgrade /mnt/btrfs-root/@
```

Then regenerate your initramfs or GRUB if needed and reboot. This works best when `/` is on Btrfs and separate from `/home`, which should ideally be snapshot individually as well.

Rollbacks with Snapper (Optional Integration)

Snapper offers snapshot management with automatic pre/post snapshots for system updates. First, create a configuration for your root:

bash
```
sudo snapper -c root create-config /
```

Before performing `dnf upgrade`, Snapper automatically saves a snapshot pair. If the upgrade causes issues:

bash
```
sudo snapper list
sudo snapper rollback <snapshot_id>
```

This restores the filesystem to its previous state. Snapper is especially useful for systems that receive frequent updates and need version control over system-level changes.

Recovering from Broken Systems with Live USB

In severe cases where the system won't boot, recovery is still possible from a Fedora Live USB environment. After booting into the Live session, mount your root Btrfs partition:

bash

```
sudo mount -o subvolid=5 /dev/mapper/fedora-root /mnt
```

From there, navigate to the snapshot:

bash
```
sudo btrfs subvolume list /mnt
sudo btrfs subvolume delete /mnt/@
sudo btrfs subvolume snapshot /mnt/@root-before-upgrade /mnt/@
```

Chrooting into the restored root allows you to reinstall bootloaders or regenerate initramfs if needed:

bash
```
sudo mount --bind /dev /mnt/dev
sudo mount --bind /proc /mnt/proc
sudo mount --bind /sys /mnt/sys
sudo chroot /mnt
```

Once inside, use tools like `grub2-install` or `dracut -f` to repair the boot process.

Using Bootloader Snapshots (grub-btrfs Integration)

With `grub-btrfs`, Btrfs snapshots can be exposed in the GRUB boot menu. This allows you to boot directly into a previous system state without using recovery tools or Live USBs.

Install and enable grub-btrfs:

bash
```
sudo dnf install grub-btrfs
sudo systemctl enable --now grub-btrfs.path
```

Now, every time you create a Btrfs snapshot, it appears as a selectable option in GRUB under "Snapshots".

Application-Level Rollbacks

Tools like Flatpak support their own versioning and rollback functionality:

bash
```
flatpak update --commit=<previous_commit_hash> <app_id>
```

This is useful for reverting specific applications without restoring full system snapshots.

10.5 Keeping Fedora Clean and Lean Over Time

A streamlined Fedora system enhances developer productivity, boot speed, package management, and overall reliability. Over time, however, unused packages, orphaned dependencies, bloated logs, and outdated kernels can clutter the system. Preventive maintenance ensures Fedora remains fast, efficient, and free from unnecessary overhead.

Prune Unused Packages and Dependencies

Packages accumulate through testing, temporary installations, or indirect dependencies. Identifying and removing orphaned libraries helps reclaim disk space and avoid conflicts:

bash
```
sudo dnf autoremove
```

This command removes packages installed as dependencies that are no longer needed. It’s safe to run regularly, especially after uninstalling development toolchains or switching desktop environments.

To review manually installed packages:

bash
```
dnf repoquery --userinstalled
```

This helps audit the system and prune obsolete tools.

Clean Package Cache and Metadata

Fedora stores downloaded `.rpm` files in the local cache. Over time, these can occupy gigabytes of space. Use:

bash
```
sudo dnf clean packages
sudo dnf clean all
```

This clears both the cached packages and metadata. Cleaning is especially useful after a system upgrade or failed transaction.

Limit Installed Kernel Versions

Fedora installs new kernels with updates but keeps older versions for fallback. However, if left unchecked, this can waste space in `/boot` . Limit retained kernels with `dnf` :

bash
```
sudo dnf install dnf-plugin-remove-retired
sudo dnf remove --oldinstallonly --setopt installonly_limit=2 kernel
```

This command removes all but the two most recent kernels. For long-term maintenance, add the setting in `/etc/dnf/dnf.conf` :

ini
```
installonly_limit=2
```

Purge Old Logs and Journal Data

System logs can grow over time. `journald` provides built-in log rotation, but you can manually enforce size and retention:

bash
```
sudo journalctl --vacuum-time=7d
```

This retains only the last 7 days of logs. You can also limit log size:

bash
```
sudo journalctl --vacuum-size=100M
```

These help avoid disk exhaustion on smaller SSDs or containers.

Trim Btrfs Filesystem

If your Fedora installation uses Btrfs, trimming helps reclaim unused blocks and optimize disk performance, especially on SSDs:

bash
```
sudo fstrim -av
```

Schedule regular trimming with a systemd timer:

bash
```
sudo systemctl enable fstrim.timer --now
```

This keeps your filesystem compact and fast without manual intervention.

Uninstall Unused Flatpaks and Containers

Development environments often accumulate Flatpaks and Podman containers. Review unused resources:

bash
```
flatpak uninstall --unused
podman image prune
```

This removes stale Flatpak runtimes and dangling container layers that no longer serve a purpose.

Monitor Disk Usage and Growth Patterns

Use `ncdu` to visualize disk usage in a terminal UI:

bash
```
sudo dnf install ncdu
sudo ncdu /
```

It helps quickly identify oversized directories and rogue logs or build artifacts that should be purged.