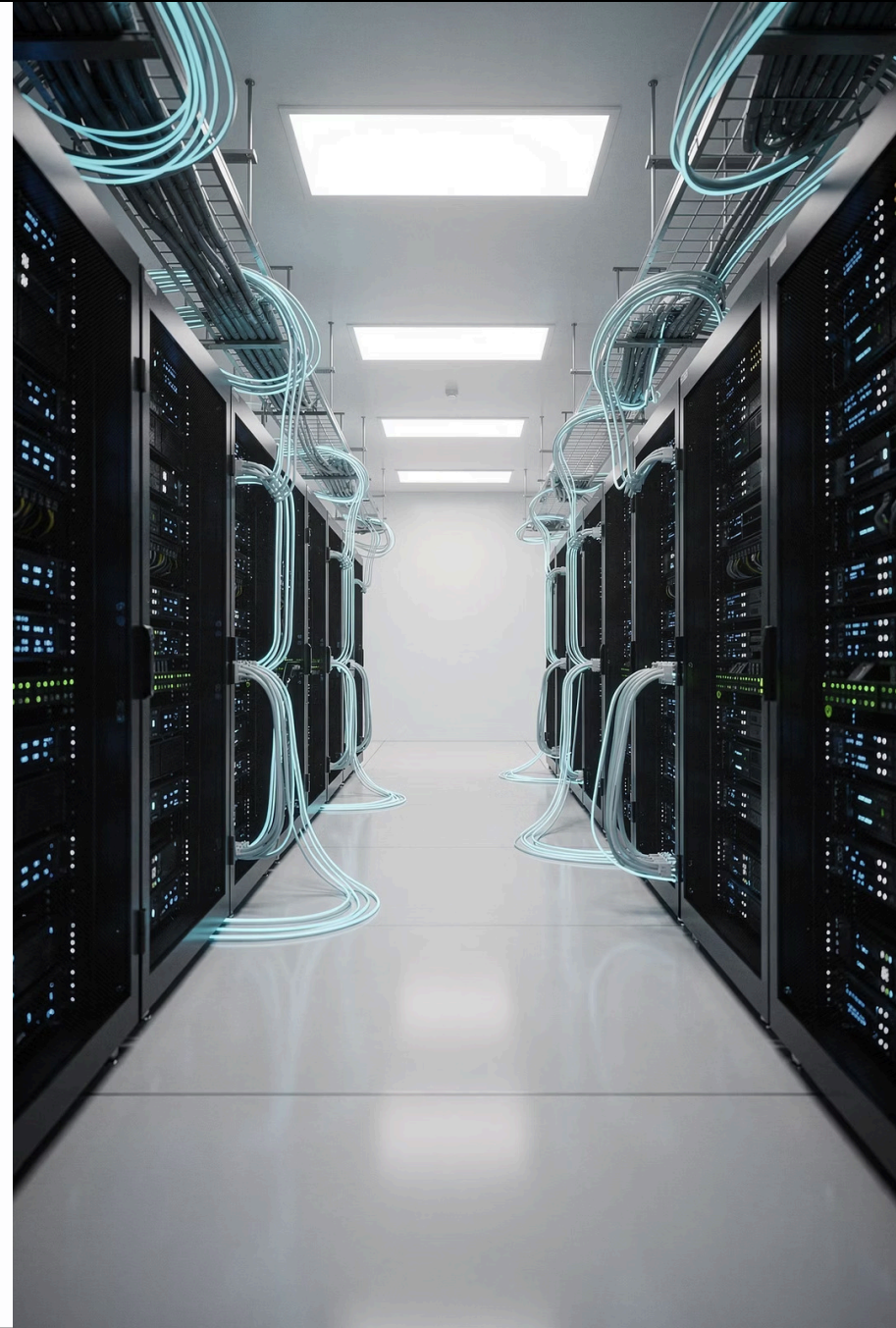
Automation & Version Control for DevOps



PART 1

Ansible Automation Platform

Why Automation Matters

Imagine manually configuring 50 web servers with the same settings. It would take days and be error-prone. Ansible lets you do this in minutes with a single command.

Ansible is an open-source automation tool that helps you configure systems, deploy software, and orchestrate complex IT tasks. It's agentless, meaning you don't need to install software on target machines.

Think of Ansible as your personal assistant that can configure hundreds of servers exactly the same way, every time.

Save Time

Automate repetitive tasks

Reduce Errors

Consistent configurations

Scale Easily

Manage thousands of servers

YAML: The Language of Ansible

Quick YAML Refresher

YAML (YAML Ain't Markup Language) is a human-readable data format. Think of it as a way to write configuration files that both humans and computers can easily understand.

YAML uses indentation (spaces, not tabs!) to show structure. It's like an outline you'd write on paper, but with specific rules that computers can parse.

- Key-value pairs: `name: webserver`
- Lists start with dashes: `- item1`
- Indentation matters: Use 2 spaces per level
- No tabs allowed: Always use spaces

Example YAML Structure

```
# A simple server configuration
server:
  name: web-prod-01
  type: nginx
  port: 80
  enabled: true
  packages:
    - nginx
    - ssl-cert
    - firewall
```

Pro Tip: YAML is whitespace-sensitive. Always use a YAML linter to validate your files before running them. A single incorrect space can break your entire playbook!

Setting Up Your Ansible Environment

Install Ansible

On Ubuntu/Debian: `sudo apt install ansible`

On RHEL/CentOS: `sudo yum install ansible`

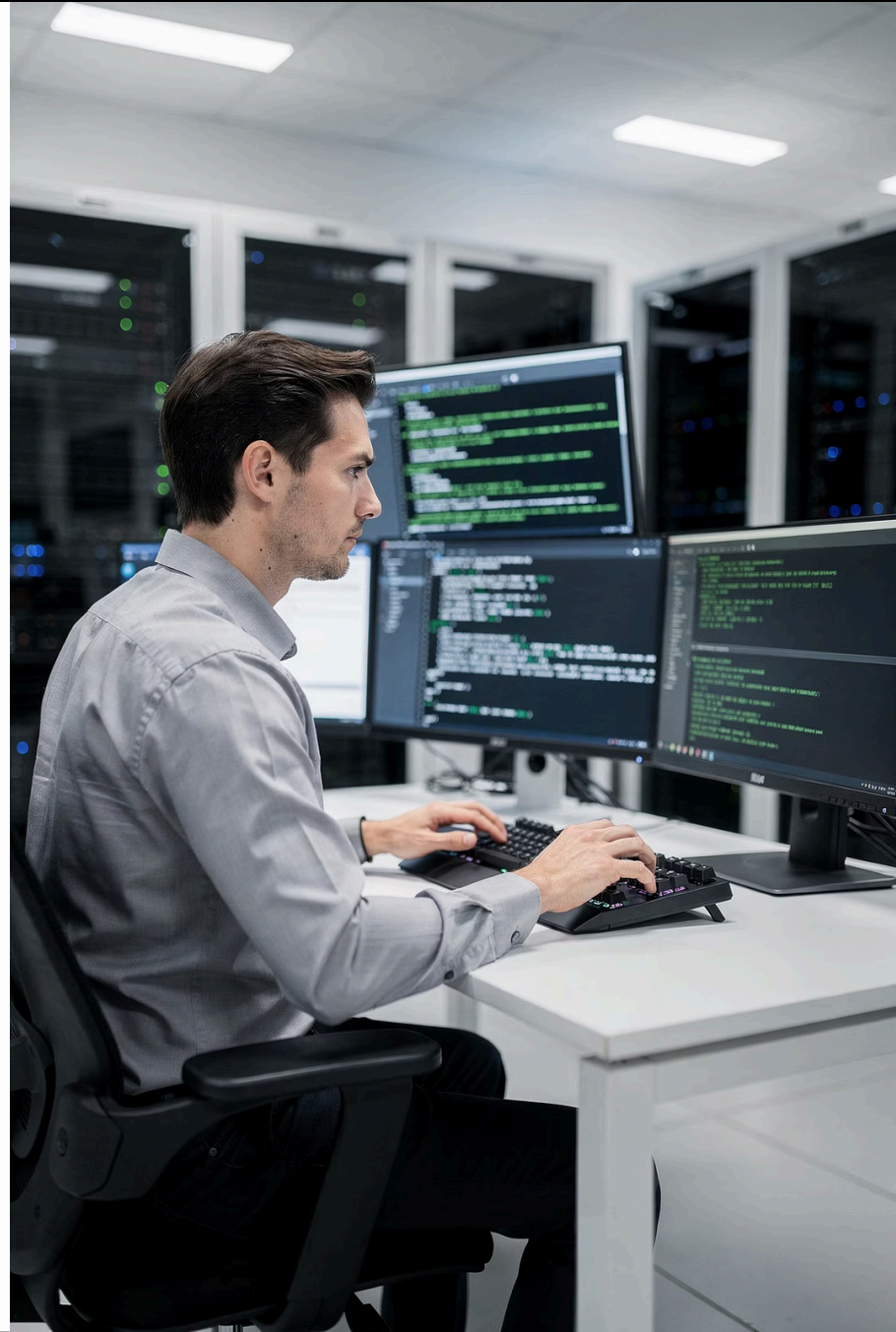
Configure Nodes

Set up SSH keys for passwordless authentication between control node and managed nodes

Test Connection

Run `ansible all -m ping` to verify connectivity to your managed hosts

The Ansible control node is where you run commands from. Managed nodes are the servers you want to configure. You'll need Python installed on all nodes and SSH access configured properly. Think of it like setting up a remote control for your TV—once configured, you can control everything from one place.



Understanding Ansible Architecture

The Three Core Components

Ansible's architecture is beautifully simple. You have an inventory file that lists your servers, modules that perform specific tasks, and playbooks that orchestrate everything together.

No databases, no daemons, no complex setup. Just files that describe what you want, and Ansible makes it happen.

1

Inventory

A list of servers you want to manage. Can be a simple text file or dynamically generated from cloud providers.

Example: `web1.example.com`, `db-server-prod`

2

Modules

Pre-built scripts that perform specific tasks like installing packages, copying files, or restarting services.

Example: `apt`, `copy`, `service`

3

Playbooks

YAML files that define the tasks you want to run, in what order, and on which servers.

Example: Install nginx, copy config, restart service

Inventory Files: Your Server Directory

Organizing Your Infrastructure

An inventory file is like a phone book for your servers. It tells Ansible which machines to talk to and how they're organized.

You can group servers by function (webservers, databases), by environment (production, staging), or any way that makes sense for your infrastructure.

Real-World Example

Imagine you're managing an e-commerce site. You have web servers handling customer traffic, database servers storing orders, and cache servers for performance. Your inventory organizes all of these.

Sample Inventory File

```
[webservers]
web1.example.com
web2.example.com
web3.example.com

[databases]
db1.example.com
db2.example.com

[cache]
redis1.example.com

[production:children]
webservers
databases
cache
```

Groups can contain other groups using the `:children` suffix. This lets you run tasks against all production servers at once.

Ansible Modules: Your Automation Toolkit

Modules are the building blocks of Ansible. Each module does one specific thing really well. Think of them as specialized tools in a toolbox—you wouldn't use a hammer to tighten a screw, and you wouldn't use the `copy` module to install packages.

apt / yum

Install, update, or remove software packages on Debian or RedHat-based systems. Example: `apt: name=nginx state=present`

copy

Copy files from control node to managed nodes. Perfect for deploying configuration files. Example: `copy: src=/local/config dest=/etc/app/`

service

Start, stop, restart, or enable services. Example: `service: name=nginx state=restarted`

user

Manage user accounts and groups. Create users, set passwords, manage SSH keys. Example: `user: name=deploy group=admin`

command / shell

Run arbitrary commands on remote systems. Use sparingly—prefer specific modules when available for better idempotency.

template

Deploy Jinja2 template files with variables substituted. Great for dynamic configurations that change per environment.

Playbooks: Orchestrating Your Automation

A playbook is like a recipe or a musical score. It tells Ansible exactly what to do, in what order, and on which servers. Playbooks are written in YAML and can contain one or more "plays," each targeting specific hosts with specific tasks.

The beauty of playbooks is that they're idempotent—you can run them multiple times and get the same result. If a package is already installed, Ansible won't try to install it again. This makes playbooks safe to run repeatedly.

Playbook Structure

- **Hosts:** Which servers to target
- **Variables:** Values you can reuse
- **Tasks:** Actions to perform in order
- **Handlers:** Tasks triggered by changes

Each task uses a module and can include conditions like "only run this on Ubuntu servers" or "skip this if the file already exists."

Your First Playbook: Installing Nginx

```
---
- name: Configure web servers
  hosts: webservers
  become: yes

  tasks:
    - name: Install nginx
      apt:
        name: nginx
        state: present
        update_cache: yes

    - name: Start nginx service
      service:
        name: nginx
        state: started
        enabled: yes

    - name: Copy custom homepage
      copy:
        src: files/index.html
        dest: /var/www/html/index.html
        mode: '0644'
      notify: Reload nginx

  handlers:
    - name: Reload nginx
      service:
        name: nginx
        state: reloaded
```

This playbook installs nginx, ensures it's running and enabled at boot, then copies a custom homepage. The handler only runs if the file actually changes, avoiding unnecessary service reloads.

Jinja2 Templates: Dynamic Configuration Files

Why Templates?

Imagine you have 20 web servers, but each needs a slightly different configuration—different server names, IP addresses, or memory limits. Writing 20 separate config files would be tedious and error-prone.

Jinja2 templates solve this by letting you create one template file with variables that get filled in automatically for each server. It's like a mail merge for configuration files.

Common Use Cases

- Nginx/Apache virtual host configs
- Database connection strings
- Application settings per environment
- Monitoring agent configurations

Template Example: nginx.conf.j2

```
server {
    listen 80;
    server_name {{ inventory_hostname }};

    location / {
        proxy_pass http://{{ backend_ip }}:{{ backend_port }};
        proxy_set_header Host $host;
    }

    {% if enable_ssl %}
    listen 443 ssl;
    ssl_certificate {{ ssl_cert_path }};
    {% endif %}
}
```

Using in Playbook

```
- name: Deploy nginx config
  template:
  src: nginx.conf.j2
  dest: /etc/nginx/sites-available/myapp
```

Ad-Hoc Commands: Quick One-Off Tasks

Sometimes you don't need a full playbook—you just need to check disk space, restart a service, or copy a file to all servers quickly. That's where ad-hoc commands shine. They're perfect for one-time tasks or quick checks.

Check Disk Usage

```
ansible webservers -m shell -a "df -h"
```

See disk space on all web servers instantly

Restart Service

```
ansible databases -m service -a "name=mysql state=restarted" --become
```

Restart MySQL on all database servers

Copy File

```
ansible all -m copy -a "src=/tmp/file.txt dest=/tmp/file.txt"
```

Distribute a file to all managed hosts

Install Package

```
ansible webservers -m apt -a "name=htop state=present" --become
```

Install htop monitoring tool on web servers

Ad-hoc commands use the same modules as playbooks. The syntax is: `ansible [hosts] -m [module] -a "[arguments]"`. Add `--become` for sudo privileges.

Using Modules Effectively in Playbooks

Modules are more powerful when combined in playbooks. You can chain tasks together, use variables to make them flexible, and add logic like loops and conditionals. Here's how to level up your module usage.

Use Variables

Define variables at the top of your playbook or in separate files. This makes your playbooks reusable across environments.

```
vars:
  app_port: 8080

tasks:
  - name: Configure port
    lineinfile:
      path: /etc/app/config
      line: "PORT={{ app_port }}"
```

Add Conditionals

Run tasks only when certain conditions are met, like operating system type or file existence.

```
- name: Install package (Ubuntu)
  apt:
    name: nginx
    state: present
  when: ansible_os_family == "Debian"
```

Loop Through Items

Install multiple packages or create multiple users with a single task using loops.

```
- name: Install packages
  apt:
  name: "{{ item }}"
  state: present
  loop:
  - nginx
  - git
  - htop
```

PART 2

Git & Version Control Fundamentals

Why Version Control Matters

Imagine you're working on a critical configuration file. You make changes, something breaks, and you can't remember exactly what you changed. Or worse—three people edit the same file simultaneously and overwrite each other's work.

Git solves these problems by tracking every change to your files, allowing you to roll back mistakes, see who changed what and when, and merge multiple people's work intelligently.

For DevOps engineers, Git isn't optional—it's essential. Infrastructure as Code means your server configurations are code, and code needs version control.

Time Machine

See entire history of changes

Collaboration

Work together without conflicts

Backup

Never lose work again

Experimentation

Try new ideas safely

Git SCM Overview

Git is a distributed version control system (VCS). Unlike older systems where there's one central server, every developer has a complete copy of the entire project history on their machine. This makes Git fast, reliable, and perfect for both solo work and team collaboration.

Created by Linus Torvalds in 2005 for Linux kernel development, Git has become the de facto standard for version control. It's free, open-source, and works on Windows, Mac, and Linux.

Distributed Architecture

Every developer has a full copy of the repository. You can work offline and commit changes locally, then sync with others when you're ready.

Snapshots, Not Differences

Git stores complete snapshots of your project at each commit, not just the changes. This makes branching and merging lightning-fast.

Data Integrity

Everything in Git is checksummed before being stored. You can't lose information or get file corruption without Git detecting it.

Installing & Configuring Git

Install Git

Linux: `sudo apt install git`

Mac: `brew install git`

Windows: Download from git-scm.com

Configure Identity

```
git config --global user.name "Your Name"
git config --global user.email
"you@example.com"
```

This info appears in every commit you make

Verify Setup

```
git --version
git config --list
```

Confirm Git is installed and configured correctly

Recommended Settings

```
git config --global core.editor "vim"
git config --global init.defaultBranch main
git config --global pull.rebase false
```

The `--global` flag applies settings to all repositories on your machine. Omit it to configure settings for just the current repository.

Git Architecture & Key Concepts

Understanding Git's architecture helps you use it effectively. Git has three main areas where your files can exist, and understanding the flow between them is crucial.

1	2	3
Working Directory	**Staging Area (Index)**	**Repository (.git directory)**
The actual files on your computer where you're making changes. This is where you edit code, create files, and do your work. Changes here aren't tracked yet.	A holding area for changes you want to include in your next commit. Think of it as a shopping cart—you add items (changes) before checking out (committing).	The permanent record of your project's history. Once you commit changes from the staging area, they're saved here forever with a unique identifier.

The workflow is: modify files in working directory → add changes to staging area → commit to repository. This three-stage process gives you fine-grained control over what gets saved in each commit.

Essential Git Terminology

Repository (Repo)

A directory containing your project files and the entire Git history in a hidden `.git` folder. Think of it as your project's complete time machine.

Commit

A snapshot of your project at a specific point in time. Each commit has a unique ID, author info, timestamp, and message describing the changes.

Branch

An independent line of development. Like a parallel universe where you can experiment without affecting the main codebase. Branches are cheap and easy to create.

Merge

Combining changes from different branches. Git automatically figures out how to combine the work, and alerts you if there are conflicts to resolve manually.

Remote

A version of your repository hosted elsewhere (like GitHub). You push your changes to remotes and pull other people's changes from them.

Clone

Creating a local copy of a remote repository. You get all the files, all the history, and all the branches—everything needed to work independently.

Cloning Repositories

Cloning is how you get a copy of an existing Git repository onto your local machine. When you clone, you download the entire project history, not just the latest files. This gives you the full power of version control locally.

Command Line Cloning

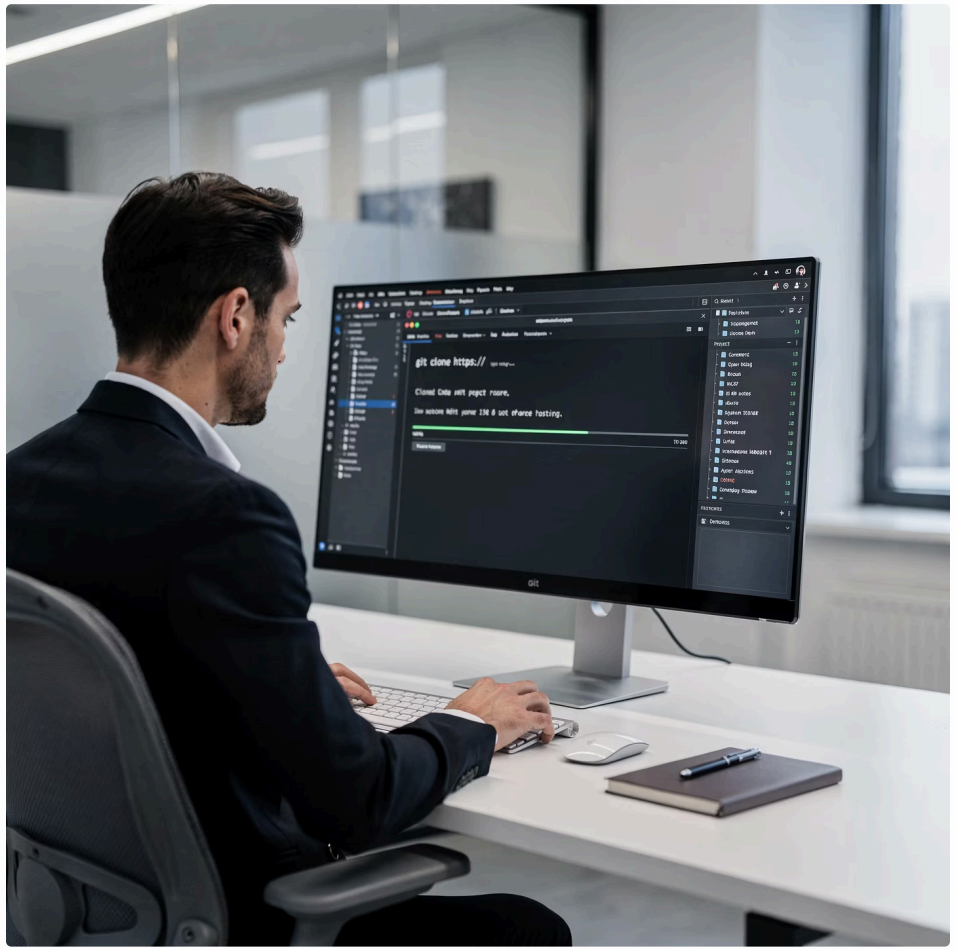
```
# Clone via HTTPS
git clone https://github.com/username/repo.git

# Clone via SSH (recommended)
git clone git@github.com:username/repo.git

# Clone into specific directory
git clone https://github.com/username/repo.git my-folder

# Clone only latest commit (shallow clone)
git clone --depth 1 https://github.com/username/repo.git
```

SSH cloning requires setting up SSH keys with GitHub but is more secure and convenient—no passwords needed.

IDE Integration

Most modern IDEs (VS Code, IntelliJ, PyCharm) have built-in Git support. You can clone repositories through the GUI:

- **VS Code:** Command Palette → Git: Clone
- **IntelliJ:** File → New → Project from Version Control
- **PyCharm:** VCS → Get from Version Control



After cloning, Git automatically sets up a remote called "origin" pointing to the source repository. This makes pushing and pulling changes easy.

Your First Commits

Committing is the heart of Git. A commit is like taking a photo of your project at a specific moment. You can always return to that exact state later. Good commits are small, focused, and have clear messages explaining what changed and why.

01

Check Status

```
git status
```

See which files have changed and which are staged for commit

02

Stage Changes

```
git add filename.txt
git add .  # stage all changes
git add *.py  # stage all Python files
```

Move changes from working directory to staging area

03

Commit Changes

```
git commit -m "Add user authentication feature"
```

Save staged changes permanently to repository with a descriptive message

Real-World Example

```
# You fixed a bug in config.py
$ git status
modified: config.py

$ git add config.py
$ git commit -m "Fix database connection timeout issue"
[main 3d5a2f1] Fix database connection timeout issue
 1 file changed, 3 insertions(+), 1 deletion(-)
```

Writing Great Commit Messages

Your commit messages are documentation for future you and your teammates. A good commit message explains what changed and why, making it easy to understand the project's evolution.

✅ Good Examples

```
"Add error handling for API timeout"
"Update nginx config to support HTTP/2"
"Fix typo in user registration form"
"Refactor database connection pooling"
```

Clear, concise, starts with a verb

❌ Bad Examples

```
"fix stuff"
"updates"
"asdfasdf"
"Changed some files"
```

Vague, unhelpful, will confuse you later

Best Practices

- Use present tense: "Add feature" not "Added feature"
- Keep first line under 50 characters
- Capitalize first letter
- No period at the end
- Add detailed explanation in body if needed

Multi-line Message

```
git commit -m "Fix memory leak in cache module

The cache was not properly releasing memory
when items expired. Added explicit cleanup
in the expiration handler.

Fixes #234"
```

Pushing Changes to Remote

Once you've made commits locally, you need to push them to a remote repository (like GitHub) so others can see your work and you have a backup in the cloud. Pushing uploads your commits while preserving the entire history.

1

Link Remote

```
git remote add origin git@github.com:user/repo.git
```

Usually done automatically when you clone

2

Push Commits

```
git push origin main
```

Upload your commits to the remote repository

3

Set Upstream

```
git push -u origin main
```

After this, you can just use `git push`

Common Push Scenarios

```
# Push current branch to remote
git push

# Force push (careful! rewrites history)
git push --force

# Push all branches
git push --all

# Push tags
git push --tags
```

Warning: Never force push to shared branches! It can destroy other people's work. Only force push to your own feature branches if absolutely necessary.

Branches: Parallel Development

Branches are Git's killer feature. They let you work on new features, bug fixes, or experiments without touching the stable main codebase. When you're done, you merge your branch back. It's like having multiple save files in a video game.

Create Branch

```
git branch feature-login
git checkout feature-login

# Or combine both:
git checkout -b feature-login
```

List Branches

```
git branch          # local
git branch -a       # all
git branch -d old   # delete
```

Switch Branches

```
git checkout main
git checkout feature-login

# Modern syntax:
git switch main
git switch feature-login
```

Common branching strategy: keep `main` stable, create a branch for each feature or bug fix, merge back when ready. Some teams use additional branches like `develop` for integration and `release` for staging.

Merging Branches

Merging combines the changes from one branch into another. When you finish a feature, you merge it back into the main branch. Git is smart about merging—it can automatically combine changes, even if multiple people edited the same file in different places.

1 Switch to Target

```
git checkout main
```

Switch to the branch you want to merge INTO

2 Merge Source

```
git merge feature-login
```

Merge the feature branch into current branch

3 Push Result

```
git push origin main
```

Upload the merged code to remote

Fast-Forward Merge

If the target branch hasn't changed since you created your feature branch, Git just moves the pointer forward. Clean and simple.

```
$ git merge feature-login
Updating 3d5a2f1..8b2c4d3
Fast-forward
 login.py | 45 ++++++++++++++++++++++++++++
 1 file changed, 45 insertions(+)
```

Three-Way Merge

If both branches have new commits, Git creates a merge commit combining both histories. You'll see both branches in the log.

```
$ git merge feature-search
Merge made by the 'recursive' strategy.
 search.py | 67 +++++++++++++++++++++++++++
 1 file changed, 67 insertions(+)
```

Handling Merge Conflicts

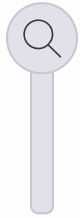
A merge conflict happens when Git can't automatically combine changes—usually when two people edited the same lines of code differently. Don't panic! Conflicts are normal and easy to resolve. Git marks the conflicting sections and lets you choose what to keep.

Conflict Example

```
$ git merge feature-config
Auto-merging config.py
CONFLICT (content): Merge conflict in config.py
Automatic merge failed; fix conflicts and commit.

# config.py looks like:
<<<<<<< HEAD
database_host = "prod-db.example.com"
=======
database_host = "staging-db.example.com"
>>>>>>> feature-config
```

Everything between `<<<<<<<` and `=======` is YOUR version. Everything between `=======` and `>>>>>>>` is THEIR version.

Resolving Conflicts



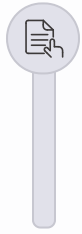
Find Conflicts

```
git status
```

Lists conflicted files

Edit Files

Open files, remove conflict markers, choose correct code



Mark Resolved

```
git add config.py
git commit
```

Many IDEs and editors have built-in merge conflict resolution tools with visual interfaces. VS Code, for example, shows buttons to "Accept Current Change," "Accept Incoming Change," or "Accept Both."

Git Hooks: Automation Triggers

Git hooks are scripts that run automatically when certain Git events occur. They're incredibly powerful for enforcing code quality, running tests, or automating deployment. Hooks live in the `.git/hooks` directory of your repository.

pre-commit

Runs before a commit is created. Perfect for linting code, checking formatting, or running quick tests. If the hook exits with an error, the commit is aborted.

commit-msg

Validates commit messages. You can enforce message format, minimum length, or require ticket numbers. Helps maintain clean git history.

post-receive

Runs on the server after receiving a push. Commonly used to trigger deployments, send notifications, or update documentation sites.

Example pre-commit Hook

```
#!/bin/bash
# .git/hooks/pre-commit

# Run Python linter
pylint **.py
if [ $? -ne 0 ]; then
 echo "Linting failed. Commit aborted."
 exit 1
fi

# Run tests
pytest tests/
if [ $? -ne 0 ]; then
 echo "Tests failed. Commit aborted."
 exit 1
fi
```

Setting Up Git Hooks

Create Hook File

```
touch .git/hooks/pre-commit
chmod +x .git/hooks/pre-commit
```

Hook files must be executable shell scripts

Write Hook Logic

Add your script—bash, Python, Ruby, whatever. Exit with 0 for success, non-zero to abort the Git action.

Test Hook

Try the Git action (commit, push, etc.) to verify your hook runs correctly. Debug with `echo` statements.

Common Hook Use Cases

- Run linters and formatters before commits
- Verify commit messages follow conventions
- Run test suite before pushing
- Prevent commits to protected branches
- Automatically deploy code after push
- Send notifications to chat systems
- Update issue trackers

Sharing Hooks with Team

Hooks in `.git/hooks` aren't tracked by Git. To share hooks, keep them in a versioned directory:

```
# Create hooks directory
mkdir git-hooks

# Create a hook
echo '#!/bin/bash\npylint **.py' > git-hooks/pre-commit

# Configure Git to use it
git config core.hooksPath git-hooks
```

Now everyone on the team uses the same hooks!

Git & GitHub Best Practices

1

Commit Often, Push Regularly

Make small, focused commits as you work. Push to remote at least daily. This creates a backup and makes it easier to collaborate. Small commits are easier to review and debug.

2

Use Branches for Everything

Never commit directly to main. Create a branch for each feature, bug fix, or experiment. This keeps main stable and makes collaboration smoother. Delete branches after merging.

3

Pull Before You Push

Always run `git pull` before pushing your changes. This fetches the latest code and helps you resolve conflicts locally before they become a problem for others.

4

Write Meaningful Messages

Future you will thank present you for clear commit messages. Explain what changed and why. Reference issue numbers when applicable. Messages are documentation.

5

Review Before Committing

Use `git diff` to see exactly what changed before staging. Use `git status` frequently. Don't blindly commit everything—be intentional about what you're saving.

6

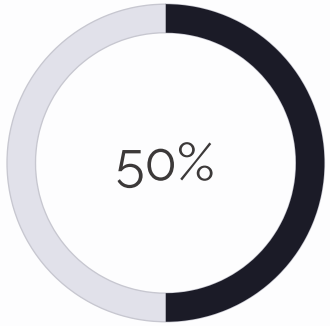
Use .gitignore Files

Never commit sensitive data, build artifacts, or IDE settings. Create a `.gitignore` file to exclude these automatically. GitHub provides templates for common languages.

You're Ready to Automate!

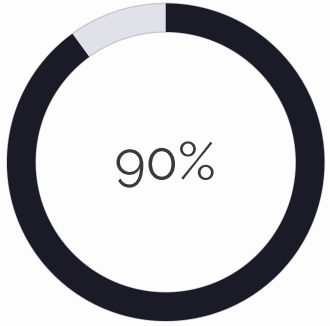
What You've Accomplished

You now have the foundational skills to automate infrastructure with Ansible and manage code with Git. These are real-world tools used by DevOps engineers at companies of all sizes.



Time Saved

With automation over manual config



Error Reduction

Using version control and IaC

Next Steps

- ### Practice Daily

  Use Git for everything, even personal projects. Automate one manual task per week with Ansible.

- ### Explore Advanced Features

  Learn about Ansible roles, Git rebase, and CI/CD pipelines. Join DevOps communities online.

- ### Build Real Projects

  Create an Ansible playbook that sets up your entire dev environment. Contribute to open-source projects on GitHub.

Remember: Every expert was once a beginner. Keep experimenting, make mistakes in safe environments, and ask questions. The DevOps community is welcoming and eager to help. You've got this!