DevOps Concepts Training

Days 1-5: Foundation & Core Concepts

Building the DevOps Mindset & Technical Foundation

Training Period: 23rd July - 19th September, 2025

Focus: DevOps Culture, CI/CD, Cloud Computing, Infrastructure as Code & SRE

: Rashi Rana
e Trainer)

Rashi Rana

DevOps Expert & Corporate Trainer

What Drives Me

Passionate DevOps Evangelist & Cloud Architect

Transforming Traditional Development Teams

Making Complex DevOps Concepts Simple & Practical

Training Philosophy

Culture First, Tools Second

Hands-on Learning Approach

Real-world Problem Solving

Continuous Improvement Mindset

"Ready to transform your development approach?"

: Rashi Rana
e Trainer)

Days 1-5: Learning Objectives

"Building Your DevOps Foundation - From Culture to Code"

Day 1-2: DevOps Culture

- Understanding DevOps Philosophy
- Traditional vs DevOps Approach
- Building Collaborative Mindset
- SDLC & DevOps Integration

Day 3: CI/CD Mastery

- Continuous Integration Concepts
- Pipeline Design & Implementation
- Real-world CI/CD Examples
- Hands-on Pipeline Activities

☁ Day 4: Cloud & AWS

- Cloud Computing Fundamentals
- AWS Services Deep Dive
- Manual vs Automated Deployment
- Cloud-native DevOps Practices

Day 5: Infrastructure as Code

- IaC Principles & Benefits
- Terraform Fundamentals
- Providers, Resources & State
- Hands-on GitHub Integration
- SRE Best Practices

By the End of Days 1-5, You Will:

Understand DevOps culture and mindset

Design and implement CI/CD pipelines

Navigate cloud computing concepts

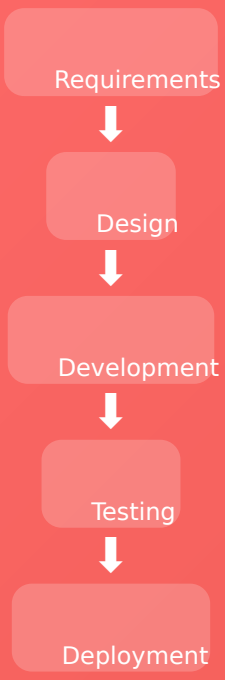
Write Infrastructure as Code with Terraform

Apply SRE principles in practice

Set up complete development environment

⚡ The Great Transformation: Traditional vs DevOps

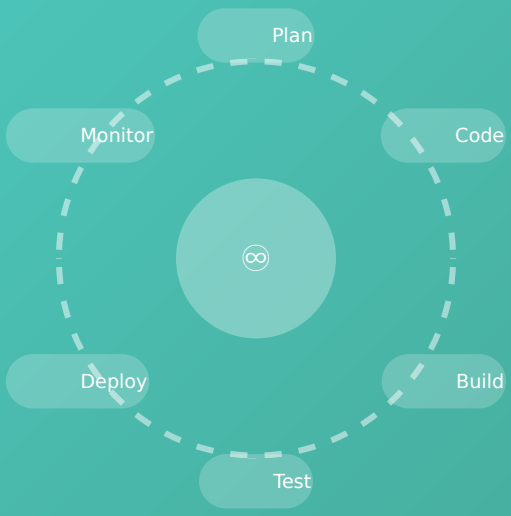
Traditional Waterfall



Pain Points:

- Months between releases
- Silos between teams
- Late bug discovery
- Manual deployments
- Blame culture

DevOps Revolution



Benefits:

- Continuous delivery
- Cross-functional teams
- Early feedback loops
- Automated everything
- Shared responsibility

The DevOps Impact

200% Faster
Deployment Frequency

50% Fewer
Production Failures

⚡

24x Faster
Recovery Time

What is DevOps?

DevOps = Development + Operations

A culture, movement, and set of practices that bridge the gap between software development and IT operations

Real-World Analogy: Building a House

Traditional Approach: Architect designs → Builder builds → Electrician wires → Plumber installs → Each works in isolation

DevOps Approach: All specialists collaborate from day one, share tools, communicate constantly, and deliver faster with fewer mistakes!

Core DevOps Values

- **C**ollaboration over silos
- **A**utomation over manual processes
- **L**earning from failures
- **M**onitoring everything
- **S**haring responsibility

: Rashi Rana
e Trainer)

Why DevOps Matters

Before DevOps		With DevOps
Months to deploy	→	Minutes to deploy
High failure rates		Faster recovery
Blame culture		Shared success

Key Benefits

Speed

- Faster deployments
- Quick feedback loops
- Rapid innovation

Quality

- Automated testing
- Fewer bugs
- Better monitoring

DevOps Culture & Collaboration

It's Not Just About Tools - It's About People!

Traditional Silos vs DevOps Collaboration

Traditional: DEV | → | QA | → | OPS
(Walls between teams)

DevOps: DEV ↔ QA ↔ OPS
(Continuous collaboration)

Cultural Shifts

- **From "It's not my job" → To "How can we solve this together?"**
- **From blame games → To shared responsibility**
- **From manual processes → To automation first**
- **From perfection → To fail fast and learn**

Think About It

Can you think of a time when poor communication between teams caused problems? How would DevOps culture help?

Continuous Integration/Continuous Deployment (CI/CD)

CI/CD: The Heartbeat of DevOps

Assembly Line Analogy

Think of CI/CD like a modern car factory assembly line:

- **CI (Continuous Integration):** Every part is tested before being added to the car
- **CD (Continuous Deployment):** Completed cars automatically roll off the production line

Continuous Integration (CI)

- Developers merge code changes frequently (multiple times per day)
- Automated tests run on every code change
- Quick feedback on code quality
- Catch bugs early when they're cheaper to fix

Continuous Deployment (CD)

- Automated deployment to production
- No manual intervention required
- Faster time to market
- Consistent deployment process

CI/CD Pipeline Flow

The DevOps Pipeline

Code → Commit → Build → Test → Package → Deploy → Monitor

Each step happens automatically!

Developer		CI System		CD System
Writes code	→	Builds & tests	→	Deploys to production
Commits changes		Reports results		Monitors performance

Interactive Question

Imagine you're ordering food through an app. How is this similar to a CI/CD pipeline? What are the different "stages" from order to delivery?

Real-World CI/CD: Netflix Example

How Netflix Deploys 1000+ Times Per Day

Developer

Pushes code for new feature

Auto Build

Code compiled & packaged

Auto Test

1000+ tests run in minutes

Auto Deploy

Live to 200M+ users!

⚡ **Result:** New features reach users in **minutes**, not months!

Hands-On Activity: "Pizza Delivery Pipeline"

Step 1: Manual Activities

Scenario: You're opening a pizza restaurant

Think & List:

- What manual steps are needed?
- From order to delivery?
- What can go wrong?
- Where are the delays?

Step 2: Create Pipeline

Challenge: Automate the process!

⚡ **Design:**

- Automated order system
- Kitchen workflow
- Quality checks
- Delivery tracking

Step 3: Compare with SDLC

Pizza Process

Order → Prepare → Cook → Quality Check → Deliver

Software SDLC

Plan → Code → Build → Test → Deploy

CI/CD Pipeline

Commit → Build → Test → Deploy → Monitor

Discussion Questions:

- How does automation reduce errors in both pizza delivery and software deployment?

- What happens when a "quality check" fails in each process?
- How do feedback loops improve both pizza quality and software quality?

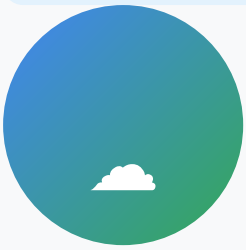
☁ Welcome to the Cloud: Your Digital Playground

> **"Someone else's computer, but way cooler!"**

What is Cloud Computing?

Instead of buying and maintaining your own servers, you **rent computing power** from companies like Amazon, Microsoft, or Google. It's like using Uber instead of buying a car - you get what you need, when you need it, without the hassle of ownership!

> "Pay only for what you use, scale instantly, access from anywhere!"



The Cloud

Traditional IT

- Buy expensive servers upfront
- Set up your own data center
- ⚡ Pay for electricity and cooling
- Hire IT staff for maintenance
- Guess future capacity needs
- Takes weeks to add new servers

☁ Cloud Computing

- Pay only for what you use

- Access servers worldwide instantly
- ⚡ No electricity or cooling costs
- Automated maintenance and updates
- Scale up or down automatically
- ⚡ New servers ready in minutes

The Big Three Cloud Providers

Amazon AWS

The pioneer and market leader. Powers Netflix, Airbnb, and millions of websites.

Microsoft Azure

Great for businesses already using Microsoft products. Growing rapidly.

Google Cloud

Strong in AI/ML and data analytics. Powers Google's own services.

Amazon Web Services (AWS): The Cloud Giant

"The company that started the cloud revolution and still leads it"

AWS by the Numbers

200+

Services Available

Millions

Active Customers

$80B+

Annual Revenue

32%

Market Share

Compute Services

- **EC2** - Virtual servers
- **Lambda** - Serverless functions
- **ECS** - Container hosting
- **Batch** - Large-scale processing

Storage Services

- **S3** - Object storage

: Rashi Rana
e Trainer)

- **EBS** - Block storage
- **EFS** - File storage
- **Glacier** - Archive storage

Database Services

- **RDS** - Relational databases
- **DynamoDB** - NoSQL database
- **Redshift** - Data warehouse
- **ElastiCache** - In-memory cache

Why AWS Matters for DevOps

- **Infrastructure as Code** - Everything can be automated
- **Global reach** - Deploy anywhere in minutes
- **Pay-as-you-go** - Perfect for testing and development
- **Massive ecosystem** - Tools for every need
- **Enterprise ready** - Security and compliance built-in
- **DevOps tools** - CodePipeline, CodeBuild, CodeDeploy

😲 Manual AWS Resource Creation: The Painful Journey

> **"Let's create a simple web server... manually"**

The 47-Step Manual Process (Yes, Really!)

Networking Setup (Steps 1-15)

1. Log into AWS Console
2. Navigate to VPC service
3. Click "Create VPC"
4. Choose VPC name and CIDR block
5. Configure DNS settings
6. Create Internet Gateway
7. Attach Gateway to VPC
8. Create public subnet
9. Configure subnet settings
10. Create route table
11. Add routes to route table
12. Associate subnet with route table
13. Create security group
14. Configure inbound rules
15. Configure outbound rules

Server Setup (Steps 16-30)

1. Navigate to EC2 service
2. Click "Launch Instance"
3. Choose Amazon Machine Image
4. Select instance type
5. Configure instance details
6. Add storage
7. Add tags
8. Configure security group
9. Review and launch
10. Create/select key pair
11. Wait for instance to start
12. Connect to instance

13. Update system packages
14. Install web server
15. Configure web server

Additional Configuration (Steps 31-47)

And we're not done yet! Still need to: Configure load balancer, set up auto-scaling, configure monitoring, set up backups, configure SSL certificates, set up logging, configure alerts, set up database, configure caching, test everything, document the setup, create disaster recovery plan, set up CI/CD integration, configure cost monitoring, set up access controls, and more...

😱 What Could Go Wrong? (Everything!)

Time Issues

- Takes 2-4 hours for one server
- Need to repeat for dev/test/prod
- Total: 6-12 hours of clicking

Human Errors

- Wrong security group settings
- Typos in configuration
- Forgot to enable monitoring

Consistency Issues

- Each environment slightly different
- Hard to remember all settings
- No version control for changes

Activity: "The Great AWS Challenge"

Challenge: Deploy a 3-Tier Web Application

Your startup needs: **Web Server + Database + Load Balancer**
Deploy to: **Development, Testing, and Production environments**

Your Mission (10 minutes):

Calculate the time, effort, and potential problems for both approaches. Work in teams of 3-4 people.

Manual Approach

Calculate:

- **Time per environment:** _____ hours
- **Total time (3 environments):** _____ hours
- **Number of manual steps:** _____ steps
- **Potential error points:** _____ places

Consider:

- What if you need to make changes?
- What if someone leaves the team?
- How do you ensure consistency?
- What about disaster recovery?

Terraform Approach

Calculate:

- **Time to write code:** _____ hours
- **Time per deployment:** _____ minutes
- **Total time (3 environments):** _____ minutes
- **Manual steps required:** _____ steps

Benefits:

- Changes applied to all environments
- Code is documentation
- Version controlled and reviewable
- Disaster recovery is automatic

Team Discussion

Cost Impact:
Which approach costs more in the long run?
Team Scaling:
What happens when your team grows from 2 to 20 people?
Business Speed:
Which approach helps you launch features faster?

Meet Terraform: The Infrastructure Wizard

> "Write once, deploy anywh

What is Terraform?

Terraform is like a **universal translator** for cloud infrastructure. You write simple configuration files, and Terraform talks to AWS, Azure, Google Cloud, and 1000+ other services to build exactly what you described.

> "It's like having a super-smart assistant who can set up entire data centers just by reading your shopping list!"

Terraform

☁

Multi-Cloud

Works with AWS, Azure, Google Cloud, and 1000+ providers

Declarative

Describe what you want, not how to build it

State Management

Tracks what exists and what needs to change

Your First Terraform Code

```
resource "aws_instance" "my_server" {
  ami = "ami-12345678"
  instance_type = "t2.micro"
  tags = {
    Name = "My Web Server"
  }
}
```

Translation: "Hey AWS, please create a small virtual server with this specific image and name it 'My Web Server'"

Infrastructure as Code (IaC) - The Foundation

"Now you understand Terraform - let's dive deeper into IaC principles"

The Traditional Infrastructure Problem (You've Seen This!)

Manual Setup Process:

- Log into cloud console
- Click through 20+ configuration screens
- Remember all the settings
- Hope you don't make mistakes
- Document everything manually

What Goes Wrong:

- Human errors in configuration
- Inconsistent environments
- Takes hours to set up one server
- No way to track changes
- Disaster recovery is nightmare

The Infrastructure as Code Solution (Terraform is IaC!)

Write Code Instead:

- Define infrastructure in text files
- Version control like application code
- Review changes before applying
- Automate deployment
- Replicate environments instantly

Amazing Benefits:

- Consistent, repeatable results
- Deploy 100 servers in minutes
- Track every change
- Easy rollback if issues
- Disaster recovery in minutes

Core IaC Principles (What Makes Terraform Powerful)

Declarative

Describe what you want, not how to build it

Idempotent

Run it 100 times, get the same result

Immutable

Replace, don't modify existing resources

Interactive Activity: "Terraform vs Manual - The Ultimate Showdown"

Scenario: You're the New DevOps Engineer

Your startup needs to deploy identical development, testing, and production environments. Each environment needs: **3 web servers, 2 database servers, 1 load balancer, and proper networking.**

Your Mission (10 minutes):

Now that you know about Terraform, compare the manual AWS approach vs Terraform approach. Work in groups of 3-4 people and be ready to present your findings!

Manual AWS Console Approach

Calculate & List:

- **Steps per environment:** _____ steps
- **Time per environment:** _____ hours
- **Total time (3 environments):** _____ hours
- **Potential error points:** _____ places
- **Documentation needed:** _____ pages

Consider: What happens when you need to add a 4th environment? What if AWS changes their interface?

Terraform Infrastructure as Code

Calculate & Design:

- **Time to write Terraform code:** _____ hours

- **Time per deployment:** _____ minutes
- **Total time (3 environments):** _____ minutes
- **Manual steps required:** _____ steps
- **Lines of code needed:** ~_____ lines

Think: How easy is environment #4, #5, #6? What about disaster recovery?

Terraform Code Snippet (What You'll Learn)

```
resource "aws_instance" "web_servers" {
  count = 3
  ami = "ami-12345678"
  instance_type = "t2.micro"
  tags = {
    Name = "WebServer-${count.index + 1}"
  }
}
```

This creates 3 web servers automatically! Compare this to 47+ manual steps per server.

Team Discussion & Presentation

ROI Analysis:
Calculate cost savings over 1 year
Team Impact:
How does this affect team productivity?
Business Speed:
Impact on time-to-market for features
Future Scaling:
What about 10 environments? 100?

Terraform Lifecycle: The Magic

Essential Terraform Commands

`terraform plan`
Shows what will change

`terraform apply`
Creates/updates resources

`terraform destroy`
Removes all resources

: Rashi Rana
e Trainer)

Terraform State: The Memory of Your Infrastructure

> **"Terraform's brain that remembers everything"**

What is Terraform State?

- **Memory system** that tracks what Terraform has created
- **JSON file** (terraform.tfstate) storing resource details
- **Mapping** between your code and real-world resources
- **Performance optimizer** - knows what exists without checking cloud

Why State Matters

- **Knows what to update** vs what to create
- **Prevents duplicates** - won't create the same resource twice
- **Enables collaboration** - team members see same state
- **Tracks dependencies** - knows deletion order

How State Works in Practice

1. You Write Code

Define desired infrastructure

2. Terraform Checks

Compares code vs state file

⚡

3. Makes Changes

Creates/updates/deletes resources

4. Updates State

Records what was done

⚠ **State File Warnings**

- **Never edit manually** - let Terraform manage it
- **Don't delete** - you'll lose track of resources
- **Keep it secure** - contains sensitive information
- **Backup regularly** - it's your infrastructure memory

State Best Practices

- **Remote state** - store in cloud (S3, Azure Storage)
- **State locking** - prevent concurrent modifications
- **Version control** - track state file changes
- **Team access** - shared state for collaboration

Terraform Providers: The Connection Layer

> **"How Terraform talks to different services"**

What are Terraform Providers?

Providers are **plugins** that enable Terraform to interact with different services. Think of them as **translators** - they understand how to communicate with AWS, Azure, GitHub, Docker, and 1000+ other services.

> "One Terraform language, thousands of services!"

Providers

☁️

Cloud Providers

- AWS
- Azure
- Google Cloud
- DigitalOcean

DevOps Tools

- GitHub

- Docker
- Kubernetes
- Datadog

Databases

- PostgreSQL
- MySQL
- MongoDB
- Redis

How to Configure Providers

Basic Provider Block:

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}

provider "aws" {
  region = "us-west-2"
}
```

Multiple Providers:

```
provider "aws" {
  region = "us-west-2"
}

provider "github" {
  token = var.github_token
}

provider "docker" {
  host = "unix:///var/run/docker.sock"
}
```

Pro Tip: Always specify provider versions to ensure consistent deployments!

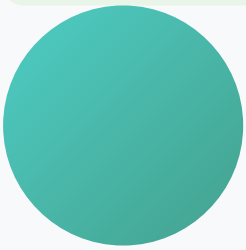
Terraform Resources: The Building Blocks

> "The actual things Terraform creates manages"

What are Terraform Resources?

Resources represent **infrastructure objects** that Terraform manages. Each resource corresponds to a real-world component like a server, database, network, or even a GitHub repository. Resources are the **"nouns"** of your infrastructure.

> "If providers are the 'how', resources are the 'what'!"



Resources

Resource Syntax Breakdown

```
resource "resource_type" "resource_name" {
  argument1 = "value1"
  argument2 = "value2"
  tags = {
    Name = "My Resource"
  }
}
```

resource_type
What kind of thing

resource_name
Your local identifier

arguments
Configuration details

☁ AWS Resources

```hcl
resource "aws_instance" "web" {
  ami = "ami-12345"
  instance_type = "t2.micro"
}

resource "aws_s3_bucket" "data" {
  bucket = "my-unique-bucket"
}
```

Creates EC2 instance and S3 bucket

GitHub Resources

```hcl
resource "github_repository" "app" {
  name = "my-app"
  description = "My awesome app"
  private = false
}
```

Creates a GitHub repository

Terraform State: Deep Dive & Best Practices

> "Understanding Terraform's memory system"

What's in the State File?

- **Resource mappings** - Links code to real resources
- **Resource metadata** - IDs, attributes, dependencies
- **Provider information** - Which provider manages what
- **Terraform version** - Compatibility tracking
- **Workspace info** - Environment separation

⚠ State File Warnings

- **Contains secrets** - Passwords, keys, sensitive data
- **Never edit manually** - Can corrupt your infrastructure
- **Don't commit to Git** - Security and conflict issues
- **Always backup** - Losing state = losing control
- **Team coordination** - Concurrent changes cause problems

Essential State Commands

Inspection

```
terraform state list
terraform state show
terraform show
```

Management

```
terraform state mv
terraform state rm
terraform import
```

Refresh

```
terraform refresh
terraform plan -refresh-only
terraform apply -refresh-only
```

☁ Remote State: Production Best Practice

Why Remote State?

- Team collaboration
- State locking (prevents conflicts)
- Automatic backups
- Encryption at rest
- Access control

S3 Backend Example:

```
terraform {
  backend "s3" {
    bucket = "my-terraform-state"
    key = "prod/terraform.tfstate"
    region = "us-west-2"
    encrypt = true
  }
}
```

⚡ Hands-On Execution & Advanced GitHub Features

> **"Time to run it and see the magic happen!"**

Execution Commands

Step 5: Set Your Token (terraform.tfvars)

```
github_token = "ghp_your_token_here"
repository_name = "my-awesome-project"
```

⚡ Step 6: Execute Terraform

```
# Initialize Terraform
terraform init

# Plan the changes
terraform plan

# Apply the changes
terraform apply

# Check what was created
terraform show
```

What You'll See:

- **terraform init** - Downloads GitHub provider
- **terraform plan** - Shows what will be created
- **terraform apply** - Creates the repository
- **Success!** - Repository appears in your GitHub

Verification:

Go to github.com/yourusername and see your new repository with README, .gitignore, license, and topics - all created automatically!

Advanced GitHub Resources

Branch Protection

```
resource "github_branch_protection" "main" {
  repository_id = github_repository.demo_repo.name
  pattern = "main"
  enforce_admins = true

  required_status_checks {
    strict = true
  }
}
```

Repository Files

```
resource "github_repository_file" "readme" {
  repository = github_repository.demo_repo.name
  branch = "main"
  file = "README.md"
  content = file("${path.module}/README.md")
}
```

Your Turn!

Challenge:

- Create a repository for your favorite programming language
- Add appropriate .gitignore template
- Include relevant topics
- Add branch protection rules
- Create a custom README file

Cleanup

Important: Don't forget to clean up your resources!

```
# Destroy the repository
terraform destroy

# Confirm with 'yes'
# Repository will be deleted!
```

⚠ **Warning:** terraform destroy will permanently delete your repository!

Site Reliability Engineering (SRE)

SRE: Bridging Development and Operations

Google's approach to DevOps

Hospital Emergency Room Analogy

Traditional Operations: Like having only doctors who treat patients after they get sick

SRE Approach: Like having doctors who also focus on preventing illness, monitoring patient health, and improving hospital systems

What is SRE?

- Software engineers who specialize in reliability
- Apply software engineering practices to operations
- Focus on automation and monitoring
- Balance between new features and system stability

Key SRE Metrics

SLI

Service Level Indicator

What we measure

SLO

Service Level Objective

Our target

SLA

Service Level Agreement

Promise to customers

⚖ Key SRE Principles

1. Error Budgets

Credit Card Analogy

Imagine you have a "downtime credit card" with a monthly limit. You can "spend" some downtime on new features, but if you max out your credit card, you must focus only on stability until next month!

2. Automation Over Toil

- **Toil:** Repetitive, manual, interrupt-driven work
- **Goal:** Automate toil to focus on engineering work
- **Rule of thumb:** If you do it more than twice, automate it!

3. Monitoring and Observability

Real-world Question

If Netflix promised 99.9% uptime (SLA), but their actual system can handle 99.95% (SLO), what would you do with that extra 0.05% reliability buffer?

What's Next?

Your DevOps Journey Continues!

Days 12-45: Advanced Topics & Real-World Projects

Upcoming Learning Modules

- **Containerization:** Docker and Kubernetes
- ☁ **Cloud Platforms:** AWS, Azure, GCP
- **Monitoring:** Prometheus, Grafana, ELK Stack
- **Security:** DevSecOps practices
- **Advanced IaC:** Terraform modules, Ansible playbooks
- **GitOps:** Advanced CI/CD patterns

Personal Action Plan

Before you leave today:

- Choose one concept to explore deeper this week
- Set up a practice environment at home
- Follow DevOps practitioners on social media
- Join DevOps communities and forums

Remember: DevOps is a journey, not a destination!
Keep learning, keep practicing, keep growing!

: Rashi Rana
e Trainer)

Questions & Discussion

Let's Talk DevOps!

Your questions make everyone smarter

Common Questions We Might Explore

- "How do I convince my team to adopt DevOps practices?"
- "What if something goes wrong with automated deployments?"
- "How much automation is too much automation?"
- "What's the difference between DevOps and traditional IT?"
- "How do I measure DevOps success?"

Discussion Format

- Ask any questions about concepts we covered
- Share your thoughts on real-world applications
- Discuss challenges you anticipate
- Connect concepts to your previous experience

Thank you for an amazing first 10 days!

Ready for your assessment challenge?

: Rashi Rana
e Trainer)

Installing Git: Your Version Control Foundation

> **"Let's get your development environment ready!"**

Windows Installation

Step-by-Step:

1. Go to **git-scm.com**
2. Click "Download for Windows"
3. Run the downloaded .exe file
4. Choose "Git Bash Here" option
5. Select "Use Git from Git Bash only"
6. Choose "Checkout Windows-style"
7. Select "Use Windows' default console"
8. Click "Install" and wait

Verify Installation:

```
git --version
```

Should show: git version 2.x.x

macOS Installation

Method 1 - Homebrew (Recommended):

```
# Install Homebrew first
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
install/HEAD/install.sh)"

# Then install Git
brew install git
```

Method 2 - Direct Download:

1. Go to **git-scm.com**
2. Click "Download for Mac"
3. Open the .dmg file
4. Run the installer

Verify Installation:

```
git --version
```

Should show: git version 2.x.x

⚙ Initial Git Configuration (Required!)

Run these commands in your terminal/Git Bash:

```
# Set your name (replace with your actual name)
git config --global user.name "Your Full Name"

# Set your email (replace with your actual email)
git config --global user.email "your.email@example.com"

# Verify configuration
git config --list
```

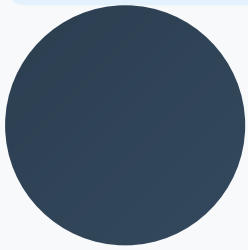
Git Bash: Your Command Line Companion

> "Unix-like terminal for Windows us

What is Git Bash?

Git Bash provides a **Unix-like command line interface** on Windows. It comes automatically with Git installation and gives you access to powerful command-line tools that work the same way on Windows, macOS, and Linux.

> "One terminal experience across all operating systems!"



Git Bash

Windows Users

Good News!

Git Bash is **automatically installed** when you install Git! No additional steps needed.

How to Access:

- **Method 1:** Right-click in any folder → "Git Bash Here"
- **Method 2:** Start Menu → Search "Git Bash"
- **Method 3:** Windows Terminal → New Tab → Git Bash

Pro Tip:

Pin Git Bash to your taskbar for quick access!

macOS Users

Even Better News!

macOS already has a **Unix-based terminal**! You can use the built-in Terminal app.

How to Access:

- **Method 1:** Cmd + Space → Type "Terminal"
- **Method 2:** Applications → Utilities → Terminal
- **Method 3:** Launchpad → Other → Terminal

Pro Tip:

Add Terminal to your Dock for quick access!

Essential Commands to Test Your Setup

Navigation

```
pwd # Show current directory
ls # List files
cd ~ # Go to home directory
```

Git Commands

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

File Operations

```
touch test.txt
echo "Hello" > test.txt
cat test.txt
```

Installing Terraform: Your Infrastructure Automation Tool

> **"Time to install your infrastructure wizard!"**

Windows Installation

Method 1 - Chocolatey (Recommended):

```
# Install Chocolatey first (if not installed)
# Run PowerShell as Administrator
Set-ExecutionPolicy Bypass -Scope Process -Force
iex ((New-Object System.Net.WebClient).DownloadString('https://
chocolatey.org/install.ps1'))

# Then install Terraform
choco install terraform
```

Method 2 - Manual Download:

1. Go to **terraform.io/downloads**
2. Download Windows AMD64 zip file
3. Extract terraform.exe to C:\terraform\
4. Add C:\terraform\ to your PATH
5. Restart your terminal

Verify Installation:

```
terraform --version
```

Should show: Terraform v1.x.x

macOS Installation

Method 1 - Homebrew (Recommended):

```
# Install Terraform
brew tap hashicorp/tap
brew install hashicorp/tap/terraform
```

Method 2 - Manual Download:

1. Go to **terraform.io/downloads**
2. Download macOS AMD64 zip file
3. Extract and move to /usr/local/bin/
4. Make it executable: chmod +x terraform

Verify Installation:

```
terraform --version
```

Should show: Terraform v1.x.x

Essential Terraform Commands to Test

Basic Commands

```
terraform --version
terraform --help
terraform init
```

Planning & Apply

```
terraform plan
terraform apply
terraform destroy
```

State & Info

```
terraform show
terraform state list
terraform validate
```

Quick Installation Test

Create a simple test to verify everything is working:

```
# Create a test directory
mkdir terraform-test
cd terraform-test

# Create a simple main.tf file
echo 'terraform {
  required_version = ">= 1.0"
}' > main.tf

# Initialize and validate
terraform init
terraform validate
```

If you see "Success! The configuration is valid." - you're ready to go!

Complete Setup Verification & Troubleshooting

> **"Let's make sure everything works perfectly!"**

Complete Verification Checklist

Git Verification

```
git --version
git config --global --list
git status
```

Should show version, your name/email, and "not a git repository" message

Terraform Verification

```
terraform --version
terraform --help
terraform fmt --version
```

Should show Terraform version and help information

Common Issues & Quick Fixes

Git Issues

"git: command not found"

Solution: Restart terminal/Git Bash after installation

"Please tell me who you are"

Solution: Run git config commands with your name/email

Terraform Issues

"terraform: command not found"

Solution: Check PATH variable or reinstall

Permission denied (macOS)

Solution: Run chmod +x /usr/local/bin/terraform

Final Success Test

Run this complete test to confirm everything is working:

```
# Test Git
git --version && echo "    Git is working!"

# Test Terraform
terraform --version && echo "    Terraform is working!"

# Create a test workspace
mkdir devops-test && cd devops-test
git init
echo "# DevOps Test" > README.md
git add README.md
git commit -m "Initial commit"
echo "    Complete setup successful!"
```

You're Ready! Next Steps:

Learn Git Basics

Start with basic Git commands and workflows

Practice Terraform

Create your first infrastructure code

Join the Community

Connect with other DevOps learners