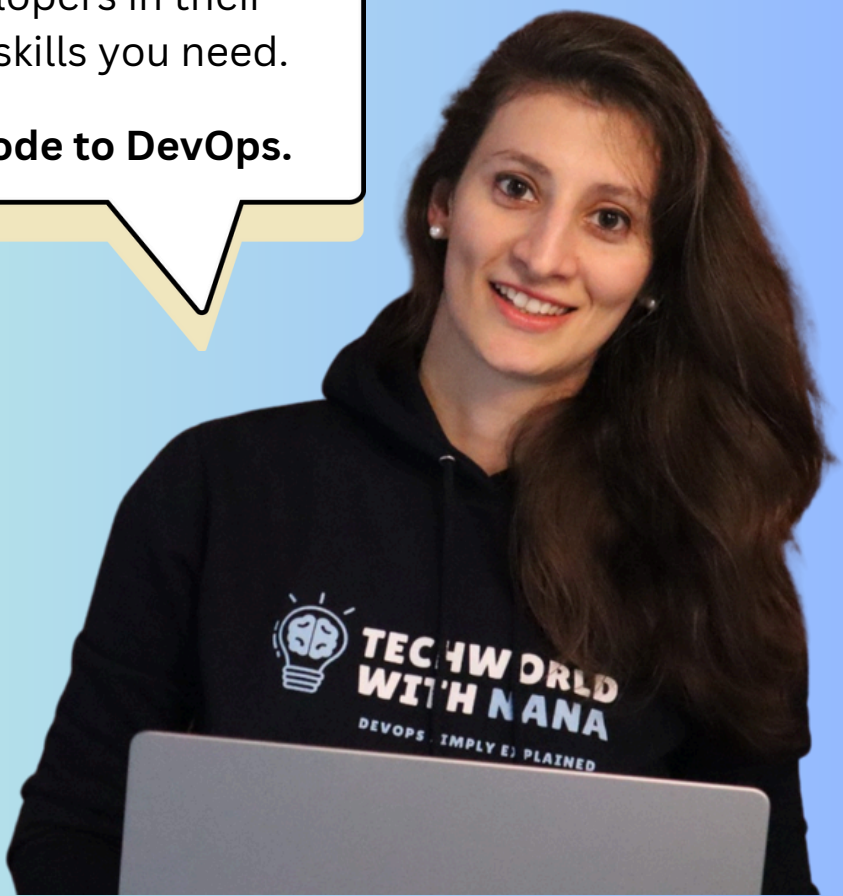
DEVOPS ROADMAP

for Software Engineers

Hi, I'm Nana. Creator of #1 DevOps training

After helping 70,000+ developers in their careers, I know exactly what skills you need.

This is your blueprint from code to DevOps.



TECHWORLD WITH NANA

Why Learning DevOps

is game changer for Developers

If you downloaded this guide, **you likely fall into 1 of these categories**, as many of our DevOps Bootcamp students:

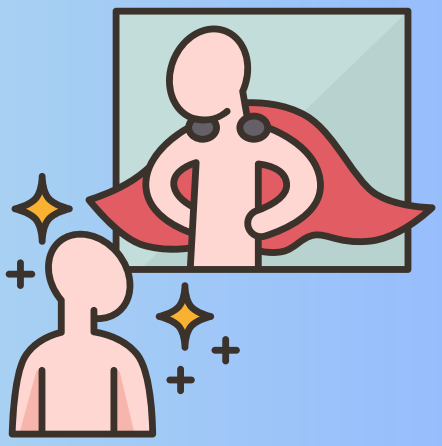
1) Upskill as a Software Engineer

You understand, knowing DevOps has many benefits:

Higher Salary: Many companies pay premium for a developer, who also knows DevOps

Better Engineer: You want to be a full fledged engineer, who understands the complete engineering process - development as well as deployment cycle

Job Security: You don't want to fall behind, and with AI, having DevOps expertise will make you indispensable



3) Manage DevOps tasks thrown to you

Often companies don't hire or don't find skilled DevOps Engineers.

No matter if you want or not, more and more DevOps tasks land on your desk and you're trying to keep up!

2) Transition to DevOps Engineer Role

There are many reasons for that:

Preference: To work on holistic end-to-end processes, automating everything and doing diverse tasks

Tangible Impact: You love seeing immediate results by optimizing and improving team workflows

Cross-Team Collaboration: You like working across teams, communicating and collaborating to solve challenges system-wide

Your Existing Superpowers

Great news! Your coding skills are your foundation -
here's how they translate to DevOps

DevOps Pre-Requisites - Part 1

1.1 - Programming Language

You know programming!

DevOps engineers work closely with developers and system administrators to automate tasks. They often need to **write scripts and small applications** to automate them. So your software development background is a huge plus.

DevOps is all about automation.

Automation logic is written as code. So your development skills of writing clean, maintainable code, thinking logically, writing automated tests will help you write infrastructure automation scripts, release pipeline scripts etc.

Software Development Lifecycle

Your work as a DevOps engineer is a holistic one.

So it's also crucial to:

- Understand what the whole lifecycle covers from idea to code, all the way to releasing it to the end user
- Understand how developers collaborate (Agile, Jira workflows)
- How applications are configured and built (Build Tools)

1.2 - Version Control - Git

Just like application code, **DevOps automation code files are also managed and hosted with a version control tool,** like Git.

So you have a massive advantage, knowing these:

- Git workflows, Branching strategies
- Git commands and Merge conflict resolution
- Collaborative development, Code review processes

OS & Linux Fundamentals

Most software engineers jump straight into the DevOps technologies, but here's the thing - these tools are built on Linux concepts.

Master the foundation that every container, cloud instance, and server runs on!

DevOps Pre-Requisites - Part 2

Why Linux?

From Software Engineer POV

As a developer, your new REST API service deployed to Kubernetes **isn't receiving requests** from other services.

What do you do?

You SSH into your Linux test container, use simple *curl* commands to test connectivity between services, and check basic network connectivity with *ping*

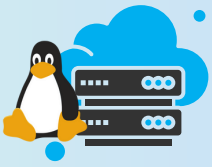
You discover that while your application is running, it's not actually listening on the expected port - a quick *netstat -tulpn* shows no process on port 8080.

This basic Linux networking knowledge helps you find that you misconfigured the container port, fix it, and get your service running properly.

The **Importance of Linux in DevOps** environment

Without Linux skills, you'd be stuck in a modern DevOps environment - unable to verify network connectivity, check container health, or troubleshoot basic infrastructure issues.

Every container **runs on Linux**



Every cloud instance **runs Linux**

And without understanding the **operating system that runs your entire infrastructure**, you can't effectively deploy or debug your applications.

Operating Systems

A big part of DevOps is about preparing and maintaining the infrastructure (servers) on which the application is deployed.

So you need to **know the basics of how to administer a server and install different tools on it.**

Linux OS

Since most servers use Linux OS, you need to be comfortable using **Linux,** especially its Command Line Interface.

Basic concepts of Operating Systems you need to understant:

- Shell Commands
- Linux File System & Permissions
- SSH Key Management
- Virtualization
- Packages

You also need to know the **basics of Networking & Security** in order to configure the infrastructure:

- Configure Firewalls to secure access
- Understand IP addresses, ports, DNS
- Load Balancers
- Proxies
- HTTP/HTTPS

However, to **draw a line** here between DevOps and IT Operations: You don't need to be the SysAdmin. So **no advanced knowledge** of server administration is needed here. There are own professions like SysAdmins, Networking or Security Professionals for more advanced use cases.

Before diving in,
WHO are we and WHAT do we do?

#1 Educator for
DevOps & Cloud Engineering

TECHWORLD WITH NANA

At TechWorld with Nana, we make DevOps easy for developers everywhere!

With our hands-on tutorials and DevOps bootcamp certification, we've brought together a massive community of 1.5+ million engineers worldwide.

Our secret? We take complex concepts - turn them into practical, real-world knowledge you can ACTUALLY use.

Our straight-to-the-point teaching style—made us one of the most trusted names in DevOps & Cloud education.

We're training everyone—from individuals around the world to teams in startups and Fortune 500 companies.

We've helped over 70,000+ developers master the tools that drive today's top tech companies.

How? No fluff—just real, hands-on learning that connects writing code to running production systems at scale

Enjoy the DevOps Roadmap for software engineers!

Cloud Provider

Where modern applications live

-

flexible, scalable, pay-as-you-go infrastructure

Why Cloud?

From Software Engineer POV

As a developer, you've built a Node.js API that needs to **handle varying traffic** - 100 users at night, 10,000 during day peaks.

You deploy it to AWS EC2 instances behind an Application Load Balancer. At 3 AM, your app runs on just two t3.small instances ($15/month each).

Cloud Infrastructure

When morning traffic hits, AWS Auto Scaling automatically detects increased CPU usage and **launches four more instances within minutes**.

At 6 PM, as traffic drops, instances automatically shut down. You **pay only for actual server usage**.

Then Black Friday hits - your monitoring shows traffic surging to 50,000 users!

While your on-premise competitors' servers crash under load, your Auto Scaling Group automatically scales to 20 instances across three availability zones.

The Application Load Balancer distributes traffic evenly, keeping response times stable.

Cloud

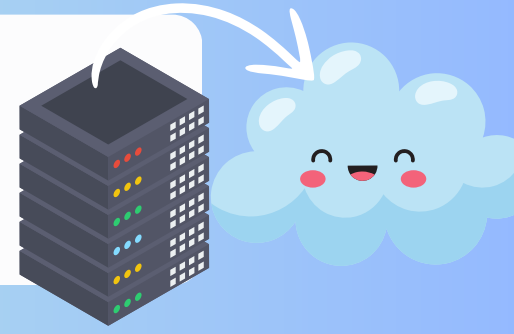
After the sales end, the system scales back down automatically. Extra cost: just $100 for that day's additional servers.

On-Premise Comparison

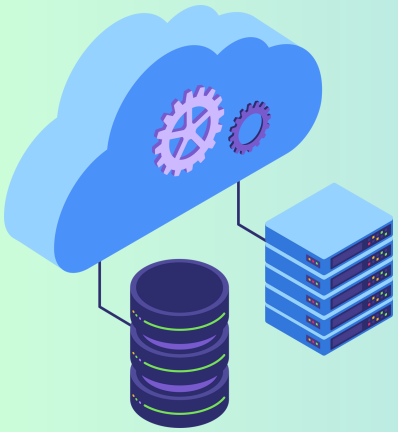
On-premise, you'd need to permanently maintain a server farm ($50,000+ in hardware) year-round just to handle these few peak days, plus redundant power, cooling, and maintenance staff.

That's the reason why companies are moving to cloud.
And why you need to learn specific Cloud services that are important for application deployment.

DevOps Engineering includes cloud skills!



What is Cloud?



Cloud is essentially a network of data centers - huge facilities filled with computers - that allow you to **rent computing resources** instead of buying and maintaining your own hardware.

It's like having a powerful computer that you can access from anywhere with an internet connection.

These are **Infrastructure as a Service (IaaS)** platforms, which offer a range of additional services, like backup, security, load balancing etc.

Why Companies Use Cloud

Cost Efficiency

- Pay-as-you-go pricing: Only pay for resources you actually use
- No upfront infrastructure costs
- Reduce operational costs (fewer IT staff needed)

Scalability & Flexibility

- Scale up/down instantly based on demand
- Handle traffic spikes automatically
- Deploy globally in minutes

Reliability & Security

- Automatic failover if one location has issues
- Built-in redundancy and backup systems
- Enterprise-level security features

Focus

By using cloud services, you can focus on your application and improving user experience, instead of worrying about managing servers and infrastructure.

Learn 1 Cloud Provider

These cloud services are **platform-specific**.

So you need to learn the services of that specific cloud platform and learn how to manage the whole infrastructure on it.

AWS is the most powerful and most widely used IaaS platform.

Other popular ones: Microsoft Azure, Google Cloud

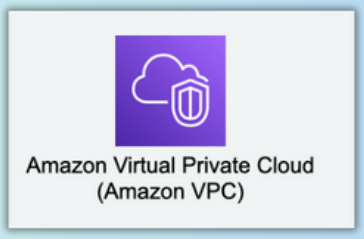
Once you learn one IaaS platform, it's easy to learn others

AWS has 100s of services, but you only need to learn handful of cloud services you/your company actually uses. E.g. when the K8s cluster runs on AWS, you need to learn the EKS service as well.

Core Cloud Services

Managing
Users & Permissions

Control Access



The Primary
Networking Service

Your private Network

The Main
Compute Services

Virtual Servers

Containerization

Solving “*it works on my machine*”

-

Package once, run anywhere

Why Containers?

From Software Engineer POV

As a developer, your team reports that your Python API **works on their machines but crashes in staging.**

With Containers

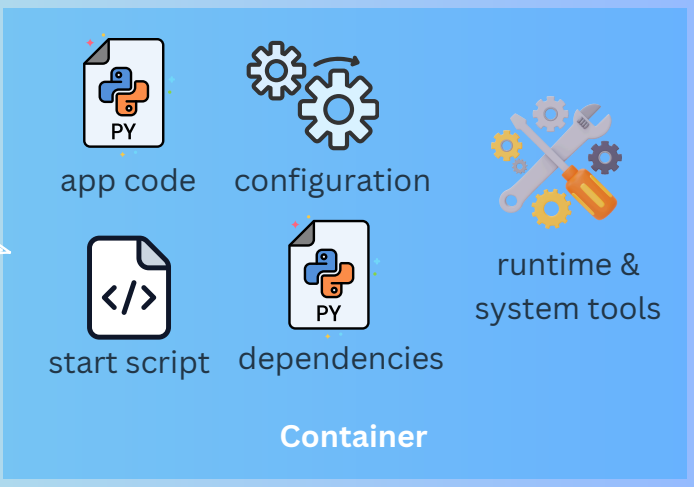
You quickly spin up a Docker container with the exact same Python 3.8.5 version, system libraries, and dependencies defined in your Dockerfile.

Running the API inside this container on your macOS laptop perfectly reproduces the error happening in the Linux staging environment.

You discover a dependency conflict, update the requirements.txt, rebuild the container, and push it to the repository.

Now your application runs identically everywhere:

- your MacBook
- colleague's Windows PC
- Linux staging
- and production



Because **the container packages the ENTIRE runtime environment, not just the code:**

Before Containers

You'd spend days documenting every system library version, Python package, and OS-level dependency in a wiki.

Then your operations team would painstakingly try to replicate this environment on each server, often discovering hidden dependencies months later when code that worked in testing would crash in production due to a slightly different library version or missing system package.

What are Containers?

Containers have become the new **standard of software packaging**, so you will run your application as a container.

This means you need to generally understand:

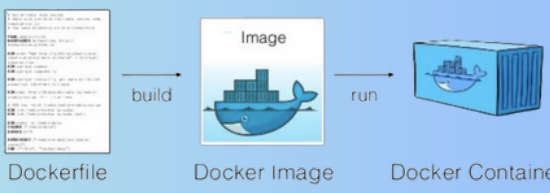
- concepts of virtualization
- concepts of containerization
- how to manage containerized applications on a server.

A container is a standard unit of software that packages up code and all its dependencies, so the application **runs quickly and reliably on any computing environment.**

Containers and virtual machines have similar resource isolation and allocation benefits, but function differently. **VMs virtualize the whole OS. Containers virtualize only the application level of the OS.** Therefore, containers are more lightweight and faster.

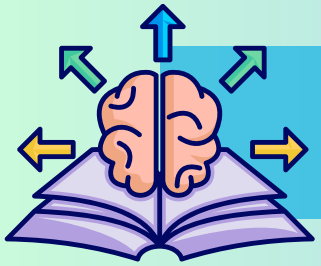
Learn Docker

Docker is by far the **most popular container technology**!



What you **should know:**

- Build Docker images
- Starting, stopping containers
- Inspect active containers
- Docker Networking
- Persist data with Docker Volumes
- Write Dockerfiles to dockerize apps
- Run multiple containers using Docker-Compose
- Work with Docker Repository



Free **Learning Resource**

- Docker in 1 Hour
- Docker Docs

Container Orchestration

Managing thousands of containers,

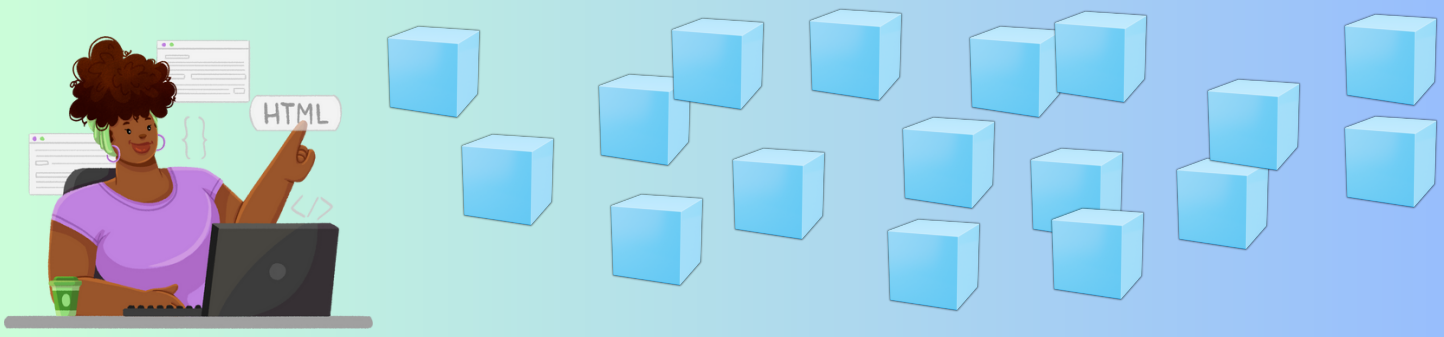
-

across hundreds of servers

Why Kubernetes?

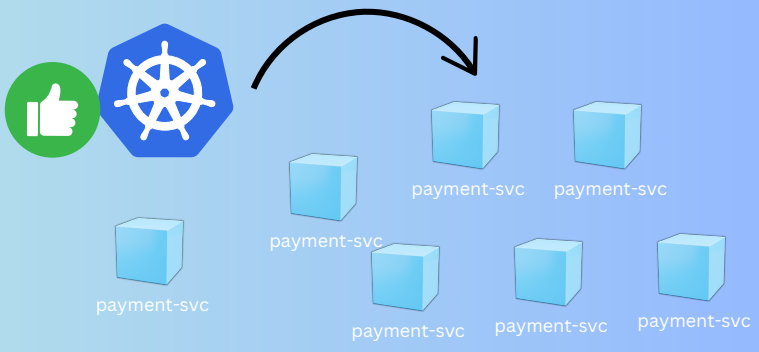
From Software Engineer POV

As a developer, your e-commerce app consists of 30 microservices in Docker containers.



During Black Friday, your **payment service struggles with high load**.

Kubernetes detects unhealthy payment containers, automatically restarts them, and **ensures at least 5 copies always run**, redirecting customer traffic only to healthy ones.



When servers get too busy, Kubernetes automatically creates more payment service containers, scaling from 6 to 15 copies.

If a server starts running out of resources, Kubernetes moves all containers from it to other servers automatically.

Before Kubernetes

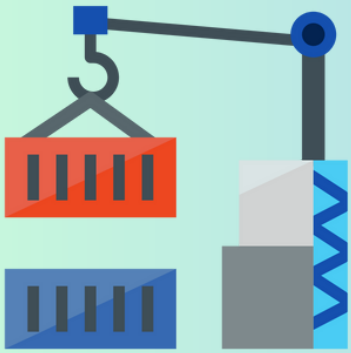
Your ops team would wake up at 3 AM to manually restart crashed applications, add more copies by hand, update traffic routing rules, and migrate everything from failing servers

- Taking hours to do what Kubernetes now handles automatically in minutes.

Container Orchestration - Kubernetes

Since containers are popular and easy to use, many companies are running **hundreds or thousands of containers on multiple servers**. This means these containers need to be managed somehow.

For this purpose there are container orchestration tools.



Container orchestration tools like Kubernetes, **automate** the deployment, scaling and management of containerized applications.

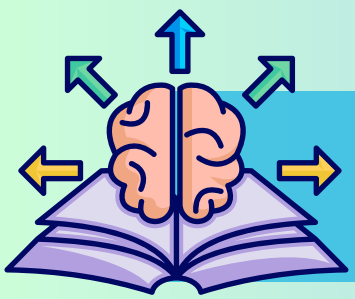
Kubernetes (also known as K8s) is the most popular container orchestration tool

So you need to learn:

- How Kubernetes works
- How to administer and manage the K8s cluster
- How to deploy applications on K8s

Specific K8s knowledge needed:

- Learn core components like, Deployment, Service, ConfigMap, Secret, StatefulSet, Ingress
- Kubernetes CLI (Kubectl)
- Persisting data with K8s Volumes
- Namespaces



Free **Learning Resource**

- Kubernetes in 1 Hour
- Kubernetes Docs

CI/CD Pipelines

From commit to production:

-

automating the software delivery lifecycle

Why CI/CD?

From Software Engineer POV

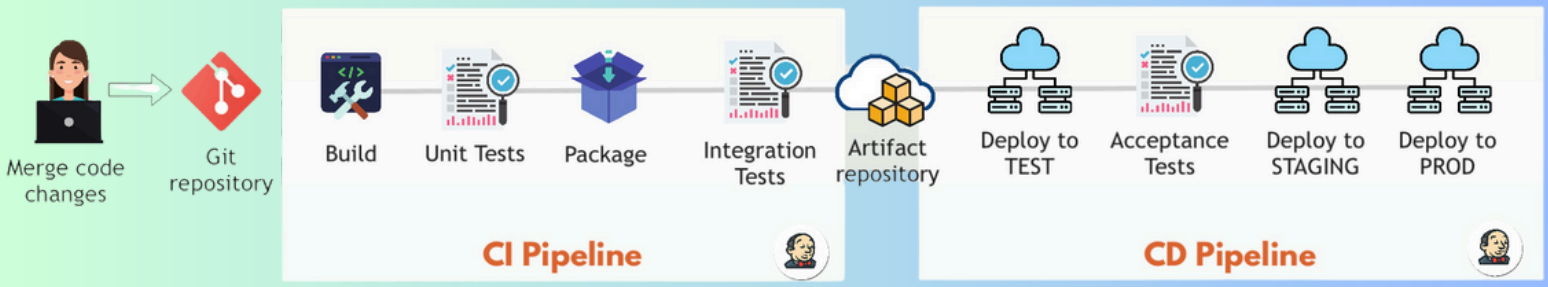
As a developer, you push a bug fix to your feature branch.

Your CI/CD pipeline automatically starts:

1) Builds your Java application
2) Runs 300 unit tests
3) Packages it into a Docker container
4) Deploys to a test environment
5) Runs integration tests, security scans, and performance checks

20 minutes later, you get a Slack notification that **your code failed a security test** - an outdated dependency has a critical vulnerability.

You update the dependency, push again, and after all tests pass automatically, create a pull request. When merged to main, the pipeline deploys to staging, then production with zero downtime:



Before CI/CD

You would:

- Manually run tests on your machine (missing half of them)
- Wait for QA team to test (2-3 days)
- Then schedule a deployment window with ops team - a process taking days and often missing critical issues until they hit production.

CI/CD Pipelines

CI/CD is the heart of DevOps.

In DevOps, all code changes, like new features or bug fixes, need to be **integrated** in the existing application and **deployed** for the end user **continuously** and in an **automated** way.

Hence the term:
Continuous **I**ntegration and
Continuous **D**eployment (CI/CD)

When the feature or bugfix is done, a pipeline running on a CI server (e.g. Jenkins) should be triggered automatically, which:

1. runs the tests
2. packages the application
3. builds a container Image
4. pushes the container Image to an image repository
5. deploys the new version to a server

There are many CI/CD platforms out there:

The most used ones at companies are **Jenkins, GitLab CI, GitHub Actions**

Skills you need to learn here:

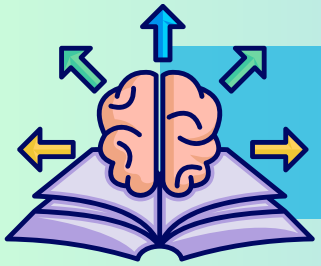
- Setting up the CI/CD server
- Integrate code repository to trigger pipeline automatically
- Build & Package Manager Tools to execute the tests and package the application
- Configuring artifact repositories (like Nexus) and integrate with pipeline
- Configuring deployment to different environments (cloud, K8s cluster)

CI - Continuous Integration

Automate the integration of code changes into a shared repository and test

CD - Continuous Delivery

Automate the delivery of applications to staging or production environments after successful testing



Free **Learning Resource**

- GitLab CI in 1 Hour
- CI/CD Tutorials

Read these stories of software developers, who **dared to DevOps**

From Embedded Software Developer to DevOps:...

Andrijana recently started her DevOps engineer position and shares valuable insights and tips for...

How to bring DevOps knowledge in-house and...

Previously relying on external agencies for critical digital services, Thomas realized that deepening his...

Mastering DevOps: A Software Developer's Pat...

Kevin, a dedicated software developer, saw the increasing importance of DevOps in his work...

Read their stories and get inspired

or copy link:

https://www.techworld-with-nana.com/success-stories

Infrastructure as Code (Iac)

Treating servers like software:

-

version-controlled, reproducible infrastructure

Why IaC?

From Software Engineer POV

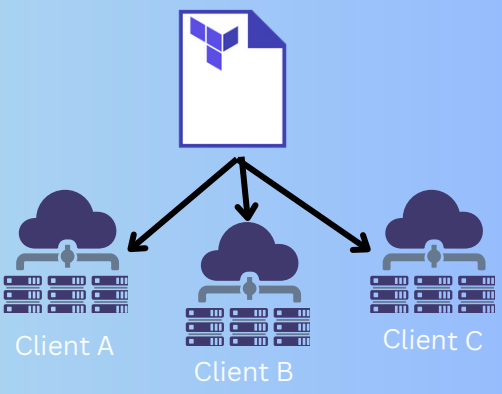
As a developer, you need to deploy the same Node.js application for three different clients, **each requiring their own isolated environment**.

You create a Terraform script defining the standard infrastructure:

- EC2 instances with auto-scaling
- Application Load Balancer
- RDS database, and correct VPC networking.

For each client, you only change a few variables - instance sizes, scaling thresholds, and SSL certificates.

One `terraform apply` command deploys a complete environment in 15 minutes, with every security rule and network configuration exactly matching your approved template:



When you need to update all environments with a new security patch, you modify one line in your base module, run Terraform, and it consistently applies the change across all client environments.

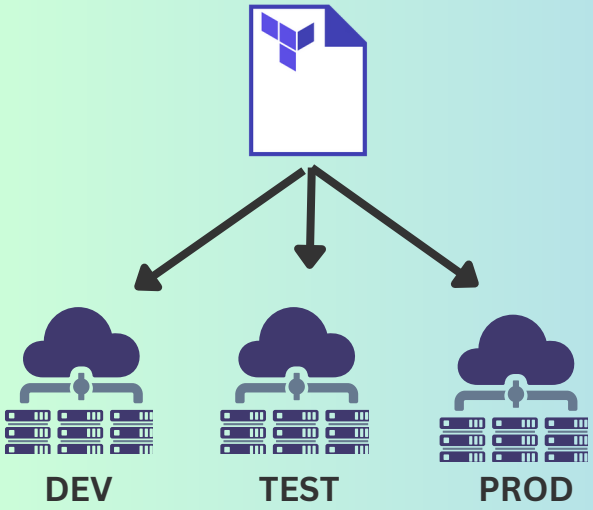
Before IaC

Each client environment would be a **unique snowflake**, **manually configured and documented** in wiki pages, making maintenance and security updates error-prone and time-consuming.

Infrastructure as Code

Manually creating and maintaining infrastructure is time consuming and error prone. Especially when you need to **replicate the infrastructure**, e.g. for a Development, Testing and Production environment.

In DevOps, we want to automate as much as possible and that's where Infrastructure as Code comes into the picture.



With IaC we **use code to create and configure infrastructure** and there are 2 types of IaC tools you need to know:

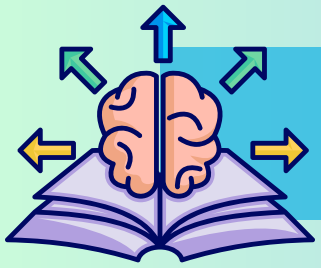
1. Infrastructure provisioning
2. Configuration management

Terraform is the most popular **infrastructure provisioning** tool

Ansible is the most popular **configuration management** tool

Benefits of having everything as code :

- **Encourage collaboration** in a team
- **Document** changes to infrastructure
- **Transparency** of the infrastructure state
- **Accessibility** to that information in a centralized place versus being scattered on people's local machines in the form of some scripts.



Free **Learning Resource**

- Terraform explained
- IaC Tutorials

Monitoring & Logging

From metrics to insights:

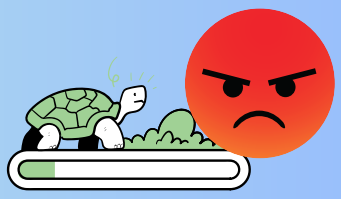
-

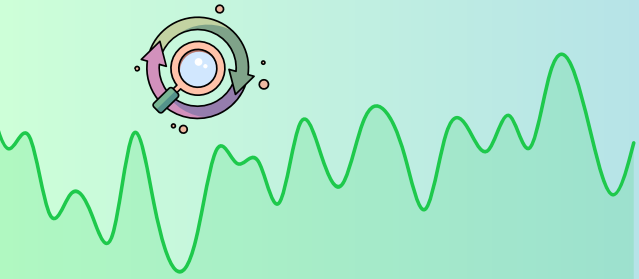
making sense of your application's behavior

Why Monitoring?

From Software Engineer POV

As a developer, users report your e-commerce checkout is **"sometimes slow."**



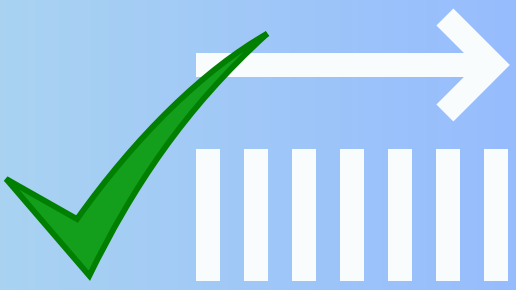


You check your Grafana dashboard, which **shows Prometheus metrics for the past 24 hours.**

You notice that every 2 hours, response times jump from 200ms to 2 seconds, exactly when your inventory sync job runs.

The graphs show that during these spikes, your database CPU hits 90% and connection pool runs out of available connections.

A quick look at resource usage metrics reveals the sync job is running on the same database as user checkouts.



You move the sync job to a read replica, and **Grafana immediately shows checkout response times staying stable at 200ms.**

Before Monitoring

You'd **spend days trying to catch the issue in action**, manually watching server metrics, and asking users: *"when exactly did you see it slow down?"* - leading to frustrated customers and lost sales.

Monitoring & Logging

Once software is in production, it is important to monitor it to **track the performance, discover problems** in your infrastructure and the application.

DevOps encompasses to setup proper monitoring system

- Setup software monitoring
- Setup infrastructure monitoring, e.g. for your Kubernetes cluster and underlying servers
- Visualize the data

You will need to:

- Collect metrics
- Write queries
- Set up alerts to notify team
- Create custom dashboards

Prometheus:
A popular monitoring and alerting tool

Grafana:
Analytics and interactive visualization tool

What to monitor

Infrastructure Monitoring

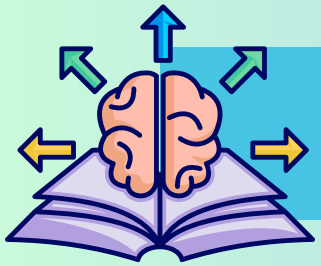
- CPU, Memory, Disk usage
- Network traffic & latency
- Container metrics
- Cloud resource utilization

Application Monitoring

- Response times
- Error rates
- Request throughput
- Database performance

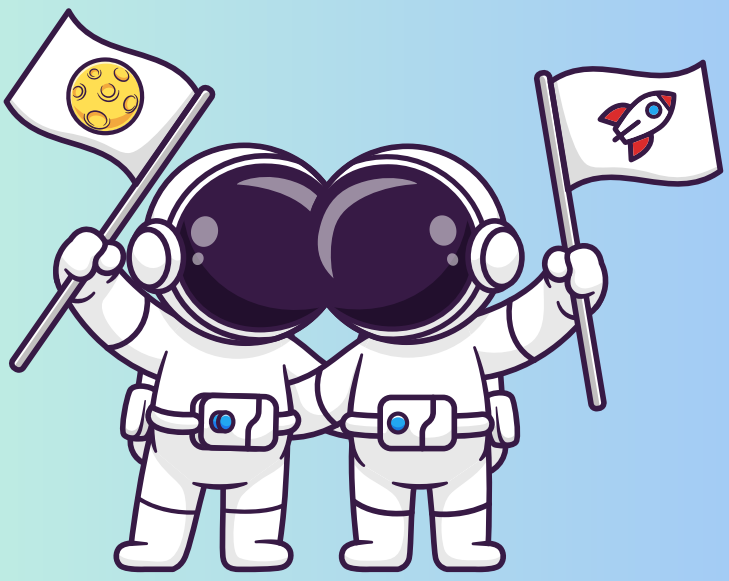
You should also understand how systems can collect and aggregate data with the goal of using it to troubleshoot, gain business insights etc.

ELK Stack:
A popular log management stack



Free **Learning Resource**

- Prometheus explained



Was this guide helpful for you?

How can we make it better?

Let me know

nana@techworld-with-nana.com

All the best on your

DevOps and Cloud journey! :)

Join 1.2M+ engineers

Join 200k+ engineers