DEVOPS:

DevOps is a set of practices, cultural philosophies, and tools that aim to improve an organization's ability to deliver applications and services at high velocity. It integrates and automates the work of software development (Dev) and IT operations (Ops) teams to enhance collaboration and productivity. Here's a detailed overview of DevOps:

Key Principles of DevOps

- **Collaboration and Communication:** Break down silos between development and operations teams. Foster a culture of shared responsibility and teamwork.
- **Automation:** Automate repetitive and manual processes throughout the software development lifecycle.
- **Continuous Integration (CI) and Continuous Delivery (CD)** are key practices in automation.
- **Continuous Improvement:** Encourage frequent, small, incremental changes. Implement feedback loops to continuously learn and improve processes.
- **Infrastructure as Code (IaC):** Manage and provision infrastructure through code and automation tools. Ensure environments are consistent and reproducible.
- **Monitoring and Logging:** Implement robust monitoring and logging to gain visibility into the system's performance and issues. Use these insights to proactively address problems and improve system reliability.

Key Practices in DevOps

- **Continuous Integration (CI):** Developers frequently merge their code changes into a shared repository. Automated builds and tests run to detect issues early.

- **Continuous Delivery (CD):** Extends CI by automatically deploying code changes to a staging or production environment after successful tests. Ensures software can be reliably released at any time.
- **Microservices:** Architectural style that structures an application as a collection of small, loosely coupled services. Each service can be developed, deployed, and scaled independently.
- **Version Control:** Use of version control systems (e.g., Git) to manage code changes. Facilitates collaboration, code review, and rollback capabilities.
- **Configuration Management:** Tools like Ansible, Chef, Puppet, and Terraform automate the setup and maintenance of computing resources. Ensures that environments are consistent and that configurations are version-controlled.
- **Containerization:** Use of containers (e.g., Docker) to package and isolate applications and their dependencies. Ensures consistency across development, testing, and production environments.
- **Orchestration:** Tools like Kubernetes manage and scale containerized applications. Automates deployment, scaling, and operation of application containers.

DevOps Tools

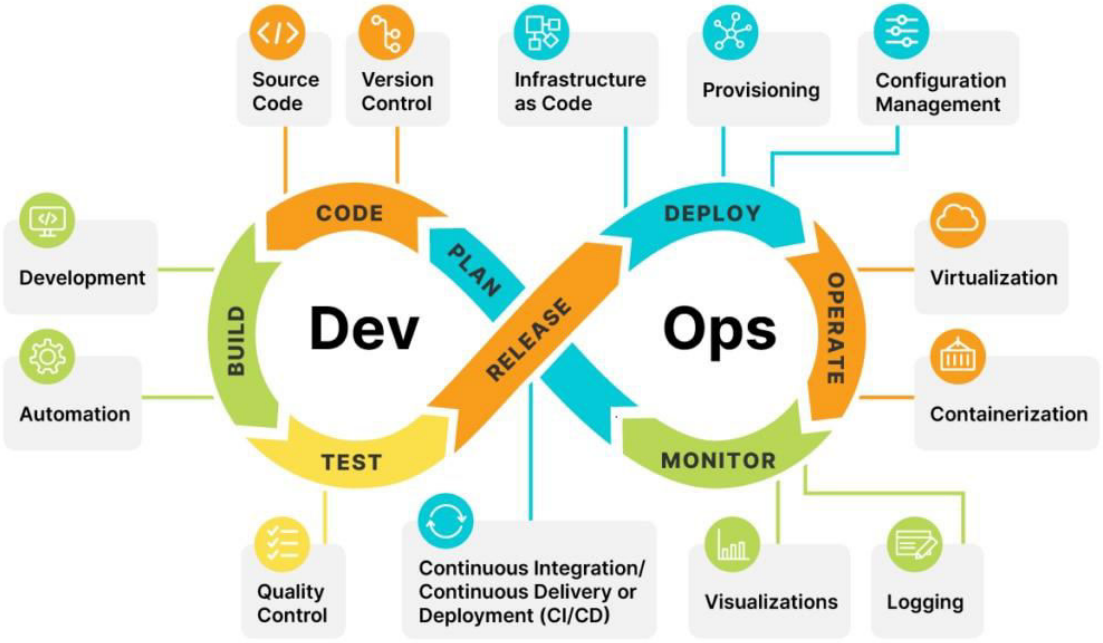
- **Version Control:** Git, GitHub, GitLab, Bitbucket
- **CI/CD:** Jenkins, GitLab CI, CircleCI, Travis CI, Bamboo
- **Configuration Management and IaC:** Ansible, Puppet, Chef, Terraform
- **Containerization:** Docker, Podman
- **Container Orchestration:** Kubernetes, Docker Swarm, OpenShift
- **Monitoring and Logging:** Prometheus, Grafana, ELK Stack (Elasticsearch, Logstash, Kibana), Splunk

Benefits of DevOps

- **Faster Delivery:** Accelerates the release of features and updates through automation and streamlined processes.
- **Improved Collaboration:** Enhances communication and collaboration between development and operations teams, leading to more efficient workflows.
- **Increased Reliability:** Ensures stable and reliable systems through continuous testing, monitoring, and automation.
- **Scalability:** Facilitates the scaling of applications and infrastructure efficiently.
- **Better Quality:** Improves software quality through automated testing and continuous integration practices.
- **Cost Efficiency:** Reduces costs associated with manual processes, downtime, and operational inefficiencies.

Conclusion

DevOps is a transformative approach that leverages collaboration, automation, and continuous improvement to enhance software development and operations. By adopting DevOps practices and tools, organizations can deliver software more rapidly, reliably, and efficiently, ultimately driving better business outcomes. DevOps was originally envisioned as a way to structure engineering organizations to efficiently integrate user feedback.



PCBT-RDOM

Lesson 1: What is DevOps

Three pillars of DEVOPS engineering:

- Pull Request Automation: GitHub, Bit-Bucket, GitLab
- Deployment Automation: Deployment and Rollback
- Application Performance Management: Metrics, Logging, Monitoring, Alerting

Lesson 2: Test Driven Development

A coding methodology where tests are written before code is written.

- Unit tests: Ensure individual components work on their own. Does the heater work? Does the tank hold water?
- Integration tests: Ensure a few components work together. Does the heater heat the water in the tank?
- System (end-to-end) tests: Ensure everything works together. Does the coffee maker brew a cup of coffee?
- Acceptance tests: After being launched (sent to consumers), are they satisfied with the result? Are they confused with the button layout or breaking their coffee maker within the warranty period?

Forces the team to prioritize tasks. Test Driven Development is an alternative way of writing well tested software. It uses tests as a way of building a specification for how things should work before a single line of code is written.

Lesson 3: Continuous Integration

Continuous Integration (CI) refers to developers continuously pushing small changes to a central Git repository numerous times per day. These changes are verified by automated software that runs comprehensive tests and ensures that no major issues are ever seen by customers.

Ensures that a newly added feature does not break existing functionality.

Benefits: Code collaboration, Reduces Customer churn, prevent errors and improves speed.

Lesson 4: What is code coverage

Code Coverage quantitatively measures how comprehensive a code base's tests are. Increasing code coverage often increases stability and reduces bugs. Basically, it is number of non-syntax line tested divided by the total number of non-syntax line. Another similar concept is branch coverage.

Lesson 5: Linting best practices

Linters are programs that look at a program's source code and find problems automatically. They are a common feature of pull request automation because they ensure that "obvious" bugs do not make it to production. Examples: pylint, flake8 for python.

Nit small comments for future review can also be used.

Lesson 6: Ephemeral Environments:

Ephemeral environments are temporary deployments that contain a self-contained version of your application, generally for every feature branch. They are often spun up by a Slack bot, or automatically on every commit using hosted DevOps platforms like webapp.io or Heroku.

The most common reason to adopt an ephemeral environment workflow is that it accelerates the software development lifecycle. Developers can review the results of changes visually, instead of needing to exclusively give feedback on the code change itself.

Lesson 7: The difference between VMs (Powerful) and Containers (Faster):

Containers virtualise over the operating system whereas VMs virtualise over the hardware itself using a hypervisor. The big change for moving programs into containers or VMs is that each will have its own version of shared resources like files and network ports. The container running Chrome might create a file at ~/.chrome/cache while the container running Notepad would not see that file at all.

Chapter 8: Rolling Deployments

Rolling deployments are a strategy to deploy a new version of an application without causing downtime. They work by creating a single instance of the new version of an application, then shutting off one instance of the old version until all instances have been upgraded.

Chapter 9: Blue/Green Deployments

Blue/green deployments are so-called because they maintain two distinct clusters, one named "blue" and one named "green" out of convention. If the current version of the application is deployed to "blue", we would deploy the new version to "green" and use it as a sort of staging environment to ensure that the new version of the app works correctly. After we're confident that the new version of the software works correctly, we would move over production load from "blue" to "green", and then repeat the cycle in the opposite direction.

Chapter 10: Aut-Scaling

Autoscaling automates horizontal scaling to ensure that the number of workers is proportional to the load on the system. Autoscaling is usually discussed on the timeline of 1 hour chunks of work. If you took the concept of autoscaling and took it to its limit, you'd get serverless: Define resources that are quickly started, and use them on the timeline of ~100ms.

Chapter 11: Service Discovery

Service discovery is how different services "learn" about each-other in order to connect. A key problem in deployment is getting services to be able to find each-other. A database might be at 10.1.1.1:6543 while the webserver is at 10.1.1.2:8080. Things get more complicated as you add more copies of your webserver, or add entirely new services.

The idea for a zero-downtime deployment is simple:

- Start the version of the backend and frontend.
- Wait until the new version is up, then divert the traffic to them.
- Shut off the old version of the backend and frontend.

However, it's difficult to update the IP addresses in our DNS provider for various reasons (DNS is can take days to update, so it's difficult to change things quickly). The solution is to add a webserver that acts as the "gateway" to the frontend and backend. We'll be able to change where it points without causing any downtime or waiting. Webservers like these are called reverse proxies.

Chapter 12: Log Aggregation

It's a way of collecting and tagging application logs from many different services into a single dashboard that can easily be searched. Eg ELK

Chapter 13: Vital Production Metrics

Metric aggregation tools measure and store numerical data to understand what is happening in production. If Log Aggregation is the first tool to set up for production monitoring, metrics monitoring should be the second. They are both indispensable for finding production faults and debugging performance and stability problems. Log Aggregation primarily deals with text - logs are textual, of course. In contrast, metric aggregation deals with numbers: How long did something take? How much memory is being used?

The sorts of metrics to collect

There is obviously a lot of subjectivity about which metrics are important based on what your product does and what your users are, but here are a few ideas of metrics and what they would be used for:

Request fulfilment times: These are very useful for understanding when systems are getting overloaded or if a newly pushed change has negatively impacted performance. Fulfillment times are often parsed out of logs (using a regular expression, for example) or taken out of a field in a database. For a website or REST API, a common request fulfillment time would be a "time to response" for each API and webpage, that way slow webpages can be discovered and investigated in production.

Request counts: A related metric that is very indicative of problems in production is the total number of requests. If there is a huge spike in the number of requests per second, it's very likely that at least a few production systems will have trouble scaling. Watching request counts can also be used to identify and mitigate Denial of Service attacks, a common type of attack used against publicly accessible resources.

Server resources: The last metric that we'll talk about is server resources - finite quantities that are used in production. Here are a few examples:

- Database size and maximum database size
- Web server memory (currently used as well as maximum)
- Network throughput and capacity (900mb/s out of 1gb/s)
- TLS certificate expiry time and life left

By measuring server resources and automatically alerting as they reach critical levels, you can avoid outages like Google Voice faced in 2021.

Quartile analysis

Production faults rarely look like "no user can access anything", often they are a gradual ramp - certain APIs taking longer and longer to load until everything falls apart, for example.

Quartile analysis is an easy way to pare down production time statistics into something actionable. A website might measure how long it takes visitors to fully load their landing page to notice when there is a production issue. With quartile analysis, they would split the request times into many different buckets:

- How long did the slowest 1% of requests take?

- How long did the slowest 5% of requests take?
- How long did the slowest 25% of requests take?

For one example, stackoverflow.com was automatically notified of an outage because their landing page was taking a long time to respond to requests.

Common production metrics tools:

- Prometheus / Grafana - OSS, often seen in cloud native (Kubernetes/Docker) settings
- Datadog - newer, primarily SaaS, often described as expensive
- New Relic - older, perhaps more reliable and mature
- AWS CloudWatch Metrics - common choice for AWS users
- Google Cloud Monitoring - less mature version of CloudWatch Metrics
- Azure Monitor Metrics - perhaps the best interface and experience of the three top cloud providers

Containers are lightweight, portable, and self-sufficient units that package an application and its dependencies, allowing it to run consistently across different environments. Containers are isolated from each other and the host system, providing a clean and consistent runtime environment.

Container orchestration refers to the automated management of containerized applications at scale. It involves the scheduling, deployment, scaling, networking, and management of containers across a cluster of machines.