A TEXT BOOK OF

DevOps IN MACHINE LEARNING

(Elective - IV)

FOR
SEMESTER – VII

FINAL YEAR (B.E.) DEGREE COURSE IN ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

Strictly According to New Credit System Syllabus of
Savitribai Phule Pune University
(w.e.f. 2023-2024)

NITIN N. SAKHARE
M. E. (Comp. Networks)
Assistant Professor,
Dept. of Comp. Engg.,
Vishwakarma Institute of Information Technology,
Kondhwa (Bk.), Pune.

RUSHIKESH S. TANKSALE
Network Engineer,
Piyush IT Services,
Pune.

Price ₹ 260.00

NIRALI PRAKASHAN
ADVANCEMENT OF KNOWLEDGE

N6460

DevOps IN MACHINE LEARNING (BE, AI & ML) **ISBN : 978-81-19116-27-0**

First Edition : September 2023
© : Authors

The text of this publication, or any part thereof, should not be reproduced or transmitted in any form or stored in any computer storage system or device for distribution including photocopy, recording, taping or information retrieval system or reproduced on any disc, tape, perforated media or other information storage device etc., without the written permission of Authors with whom the rights are reserved. Breach of this condition is liable for legal action.

Every effort has been made to avoid errors or omissions in this publication. In spite of this, errors may have crept in. Any mistake, error or discrepancy so noted and shall be brought to our notice shall be taken care of in the next edition. It is notified that neither the publisher nor the authors or seller shall be responsible for any damage or loss of action to any one, of any kind, in any manner, therefrom. The reader must cross check all the facts and contents with original Government notification or publications.

Published By :
NIRALI PRAKASHAN
Abhyudaya Pragati, 1312 Shivaji Nagar
Off J.M. Road, PUNE 411005
Tel : (020) 25512336/37/39
Email : niralipune@pragationline.com

Polyplate

Printed By :
YOGIRAJ PRINTERS AND BINDERS
Survey No. 10/1A, Ghule Industrial Estate
Nanded Gaon Road
Nanded, Pune 411041

DISTRIBUTION CENTRES

PUNE

Nirali Prakashan
(For orders outside Pune)
S. No. 28/27, Dhayari Narhe Road, Near Asian College
Pune 411041, Maharashtra
Tel : (020) 24690204; Mobile : 9657703143
Email : bookorder@pragationline.com

Nirali Prakashan
(For orders within Pune)
119, Budhwar Peth, Jogeshwari Mandir Lane
Pune 411002, Maharashtra
Mobile : 9657703145, 9890997937
Email : niralilocal@pragationline.com

MUMBAI

Nirali Prakashan
Rasdhara Co-op. Hsg. Society Ltd., 'D' Wing Ground Floor, 385 S.V.P. Road
Girgaum, Mumbai 400004, Maharashtra
Mobile : 7045821020, Tel : (022) 2385 6339 / 2386 9976
Email : niralimumbai@pragationline.com

DISTRIBUTION BRANCHES

DELHI

Nirali Prakashan
Room No. 2 Ground Floor
4575/15 Omkar Tower, Agarwal Road
Darya Ganj, New Delhi 110002
Mobile : 9555778814/9818561840
Email : delhi@niralibooks.com

BENGALURU

Nirali Prakashan
Maitri Ground Floor, Jaya
Apartments, No. 99, 6th Cross, 6th
Main, Malleswaram, Bengaluru
560003 Karnataka; Mob : 9686821074
Email : bengaluru@niralibooks.com

NAGPUR

Nirali Prakashan
Above Maratha Mandir, Shop No. 3,
First Floor, Rani Jhanshi Square,
Sitabuldi Nagpur 440012 (MAH)
Tel : (0712) 254 7129
Email : nagpur@niralibooks.com

KOLHAPUR

Nirali Prakashan
438/2, Bhosale Plaza, Ground Floor,
Khasbag, Opp. Balgopal Talim,
Kolhapur 416 012 Maharashtra
Mob : 9850046155
Email : kolhapur@niralibooks.com

JALGAON

Nirali Prakashan
34, V. V. Golani Market, Navi Peth,
Jalgaon 425001, Maharashtra
Tel : (0257) 222 0395
Mob : 94234 91860
Email : jalgaon@niralibooks.com

SOLAPUR

Nirali Prakashan
R-158/2, Avanti Nagar, Near Golden
Gate, Pune Naka Chowk
Solapur 413001, Maharashtra
Mobile 9890918687
Email : solapur@niralibooks.com

marketing@pragationline.com | www.pragationline.com

Also find us on www.facebook.com/niralibooks

PREFACE

It gives us great pleasure to present the book **'DevOps in Machine Learning' (Elective - IV)** for the students of **Final Year (BE) Degree Course in Artificial Intelligence and Machine Learning** of Savitribai Phule Pune University. This book is strictly as per the **New Credit System Syllabus 2020 Pattern** effective from the Academic Year 2023-24.

As per the New Revised Examination Scheme, which has been implemented from the previous academic year, the Mid-semester examination allots 30 marks for the first two units and the End-Semester Examination allots 70 marks for the entire syllabus.

The theory subject will have 3 Credits.

In the book, all the basic concepts are explained in a simplified manner. Subject matter is presented in a conceptual manner rather than mathematical, as required by the new examination system. It is our objective to keep the presentation systematic, and clear through explanatory notes and figures.

Main feature of this book is, Complete Coverage of the New Credit System Syllabus with Exercises and Model Question Papers of Mid-Sem. and End-Sem. Exams.

We are sure that this book will cater to all needs of the students for this subject.

We also take this opportunity to express our sincere thanks to Shri. Dineshbhai Furia, Shri. Jignesh Furia, Mrs. Nirali Verma and entire team of Nirali Prakashan, namely Mrs. Deepali Lachake (Co-ordinator), who really have taken keen interest and untiring efforts in publishing this text.

We wish to make a special mention of the valuable contribution made by Late Shri M. P. Munde during the last two decades in reaching out to students, parents and teachers which eventually made **Nirali Prakashan, a brand of trust and quality in technical books**. Without his perseverance and zeal, successive editions would not have been possible.

The advice and suggestions of our esteemed readers to improve the text are most welcome and will be highly appreciated.

Pune **Authors**

SYLLABUS

Unit I: Introduction to DevOps and the Culture (6 hrs)

What is DevOps? Role of DevOps Engineer, Developer responsibility, Introduction to Continuous Integration and Continuous Delivery Policies, DevOps Culture: Dilution of barriers in IT departments, Process automation, Agile Practices, Reason for adopting DevOps, What and Who Are Involved in DevOps? Changing the Coordination, Introduction to DevOps pipeline phases, Defining the Development Pipeline, Centralizing the Building Server, Monitoring Best Practices and Best Practices for Operations.

Unit II: Continuous Integration and Continuous Delivery (6 hrs)

Implementing Continuous Integration-Version control, automated build, Continuous Integration Practices using Continuous Integration Software (Jenkins as an example tool), Jenkins Architecture, Integrating Source code management, build, testing tools etc., with Jenkins – plugins, Artefacts management, setting up the Continuous Integration pipeline, Continuous delivery to a staging environment or the pre-production environment, Self-healing systems.

Unit III: Continuous Deployment and Continuous Monitoring (6 hrs)

Implementing a testing Strategy: Types of Tests, Integration testing, managing defect backlogs, what is Continuous Deployment? Changes moving through the deployment pipeline, Trade-offs in the deployment pipeline, Basic Deployment pipeline, Deployment pipeline practices & Commit stage, Automated Acceptance Test Gate, Subsequent test stages, preparing to release, Implementing a deployment pipeline, Factors involved in monitoring systems, why monitoring is important, white-box and black-box monitoring, building a monitoring system, monitoring infrastructure and applications, collecting data, logging, creating the dashboard, behaviour-driven monitoring, what is site reliability engineering? SRE and DevOps, roles, and responsibilities of SRE, common tools used by SREs.

Unit IV: DevOps Tooling and Case Studies (6 hrs)

Continuous Development/ Version Control: Git, Serverless orchestration: Kubernetes, Container Technology: Docker, Continuous Integration: Jenkins, Continuous delivery: Jenkins, Continuous Deployment: Ansible, Continuous Testing: Selenium, Monitoring: Prometheus, Bug tracking tool: Jira, elk stack. Case study: Spotify: Using Docker, Bank of New Zealand, EtSy Virtualization and Containerization: Virtualization, Virtualization vs Containerization, Containerization using Dockers, Docker Images, Micro-services and Containerization, orchestration, Difference between orchestration and automation.

Unit V: Introduction to MLOps (6hrs)

What is MLOps & MLOps Motivation, Solutions and Future Trends, MLOps Components, Different Roles involved in MLOps (ML Engineering + Operations), Machine Learning Life Cycle, MLOps Vs DevOps, Tools to create ML pipelines.

Unit VI: ML Model Deployment (6hrs)

MLOps Maturity Model Levels, MLOps - Stages Of CI / CD, Creating and deploying ML/AI models, ML Pipelines, automation of ML through Pipelines, Tools to create ML pipelines, Monitoring and Logging, Data Quality and Integrity, Model Retraining and Model replacement, Model Versioning, MLOps: Infrastructure, MLOps: Testing, Monitoring and Maintenance.

CONTENTS

Unit I: Introduction to DevOps and the Culture		**1.1-1.24**
1.1	Introduction to DevOps	1.1
1.2	Role of DevOps Engineer	1.1
1.2.1	DevOps Engineer Job Description	1.2
1.3	Introduction to Continuous Integration and Continuous Delivery Policies	1.2
1.3.1	What is Continuous Integration and Why is it Helpful?	1.2
1.3.2	What is Continuous Delivery and Why is it Helpful?	1.3
1.3.3	What is Continuous Deployment and Why is it Helpful?	1.3
1.3.4	Key Concepts and Practices for Continuous Processes	1.4
1.3.5	Types of Testing	1.5
1.4	DevOps Culture	1.7
1.5	Barriers to DevOps	1.8
1.6	DevOps Automation	1.9
1.7	What and Who are Involved in DevOps?	1.11
1.8	Changing the Coordination	1.13
1.9	Introduction to DevOps Pipeline Phases	1.13
1.10	Continuous Everything	1.15
1.11	Centralizing the Building Server	1.16
1.12	Monitoring Best Practices	1.18
1.12.1	DevOps Monitoring vs. Observability	1.19
1.12.2	The Importance of DevOps Monitoring	1.19
1.12.3	Key Capabilities of DevOps Monitoring	1.21
1.12.4	DevOps Monitoring Tools	1.21
1.13	Best Practices for Operations	1.22
• **Exercise**		1.24
Unit II: Continuous Integration and Continuous Delivery		**2.1-2.18**
2.1	Implementing Continuous Integration	2.1
2.1.1	Version Control	2.2
2.1.2	Automated Build	2.3
2.1.3	Continuous Integration Practices using Continuous Integration Software (Jenkins as an Example Tool)	2.4
2.1.4	Jenkins Architecture	2.5
2.2	Integrating Source Code Management, Build, Testing Tools Etc., with JENKINS	2.7
2.2.1	Plugins	2.8
2.2.2	Artefacts Management	2.9
2.2.3	Setting Up the Continuous Integration Pipeline	2.10
2.2.4	Continuous Delivery to a Staging Environment or the Pre-Production Environment	2.15
2.2.5	Self-Healing Systems	2.16
• **Exercise**		2.18

Unit III: Continuous Deployment and Continuous Monitoring **3.1-3.36**

3.1 Implementing a Testing Strategy 3.1
3.2 Types of Tests 3.2
3.2.1 Integration Testing 3.2
3.2.2 How Traditional Testing Worked 3.2
3.2.3 Features of Testing in a DevOps Environment 3.2
3.2.4 DevOps Adopts Shift Left Testing Approach 3.3
3.3 DevOps Testing Best Practices 3.3
3.4 Purpose of Integration Testing 3.6
3.4.1 Types of Integration Testing 3.6
3.4.2 Integration Testing in an Enterprise DevOps Environment 3.7
3.5 Managing Defect Backlogs 3.7
3.5.1 What is a Defect? 3.7
3.5.2 Types of Defects 3.7
3.5.3 Defect Logging and Tracking 3.8
3.5.4 Defect Details 3.8
3.5.5 Defect Status 3.9
3.5.6 Defect Triage and Assignment 3.10
3.5.7 Defect Resolution Process 3.11
3.5.8 Defect Management Artifacts 3.11
3.6 What is Continuous Deployment? 3.14
3.6.1 Continuous Deployment Tools 3.14
3.7 Changes Moving through the Deployment Pipeline 3.15
3.8 Basic Deployment Pipeline 3.16
3.9 Deployment Pipeline Practices and Commit Stage, Only Build Your Binaries Once 3.17
3.9.1 Deploy the Same Way to Every Environment 3.18
3.9.2 Smoke-Test Your Deployments 3.18
3.9.3 Deploy into a Copy of Production 3.18
3.10 Automated Acceptance Test Gate 3.19
3.11 Subsequent Test Stages 3.19
3.12 Preparing to Release 3.21
3.13 Implementing a Deployment Pipeline 3.23
3.14 What is a Monitoring System? 3.26
3.14.1 Factors Involved in Monitoring Systems 3.26
3.14.2 Why Monitoring is Important? 3.27
3.15 White-Box and Black-Box Monitoring 3.28
3.15.1 What is White Box Monitoring? 3.28
3.15.2 What is Black Box Monitoring? 3.28
3.15.3 Difference between Black Box and White Box Monitoring 3.28

3.16 Building a Monitoring System 3.29
3.16.1 Common Pitfalls in Monitoring 3.29
3.16.2 Ways to Improve Monitoring? 3.30
3.17 Monitoring Infrastructure and Applications 3.30
3.17.1 Infrastructure Monitoring 3.30
3.17.2 Application Monitoring 3.31
3.18 Collecting Data 3.31
3.19 Logging 3.31
3.20 Creating Dashboard 3.32
3.21 Behavior Driven Monitoring 3.33
3.22 What is Site Reliability Engineering? 3.35
3.23 SRE and DevOps 3.35
3.24 Roles and Responsibilities of SRE 3.35
3.24.1 Daily Roles and Responsibilities of an SRE 3.36
3.25 Common Tools Used by SREs 3.36
- **Exercise** 3.36

Unit IV: DevOps Tooling and Case Studies **4.1-4.36**

4.1 Continuous Development/Version Control 4.1
4.1.1 Benefits of Version Control Systems 4.2
4.2 Git 4.2
4.2.1 Version Control with Git 4.3
4.2.2 Git is a Quality Open Source Project 4.4
4.2.3 Criticism of Git 4.4
4.3 Serverless Orchestration 4.4
4.3.1 Serverless DevOps Evolves 4.4
4.3.2 Get Familiar with Serverless Cloud Services 4.4
4.3.3 Make a Plan 4.5
4.4 Kubernetes 4.5
4.5 Container Technology 4.6
4.5.1 Docker 4.6
4.6 Continuous Integration: Jenkins 4.7
4.6.1 What is Continuous Integration? 4.7
4.6.2 How does Jenkins Work? 4.7
4.6.3 Jenkin History 4.7
4.6.4 Why Use Continuous Integration with Jenkins? 4.7
4.6.5 Real-World Case Study of Continuous Integration 4.8
4.7 Continuous Delivery: Jenkins 4.9
4.7.1 What is a Jenkins Pipeline? 4.9
4.8 Continuous Deployment: Ansible 4.10

4.9 Continuous Testing: Selenium 4.11
4.9.1 How is Continuous Testing Different? 4.11
4.9.2 Continuous DevOps Process 4.12
4.10 Monitoring: Prometheus 4.12
4.10.1 What Do We Monitor? 4.12
4.11 Bug Tracking Tool 4.13
4.11.1 JIRA 4.13
4.11.2 ELK Stack 4.15
4.12 Case Study 4.15
4.12.1 Spotify: Using Docker 4.15
4.12.2 Bank of New Zealand 4.16
4.12.3 EtSy 4.17
4.13 Virtualization and Containerization 4.20
4.13.1 Virtualization 4.20
4.13.2 Virtualization vs Containerization 4.21
4.13.3 Containerization using Dockers 4.23
4.13.4 Docker Images 4.31
4.13.5 Micro-Services and Containerization 4.32
4.13.6 Orchestration 4.32
4.13.7 Difference between Orchestration and Automation 4.33
- **Exercise** 4.36

Unit V: Introduction to MLOps **5.1-5.14**

5.1 What is MLOps and MLOps Motivation 5.1
5.2 Solutions and Future Trends 5.5
5.3 MLOPS Components 5.7
5.4 Different Roles involved in MLOps (ML Engineering + Operations) 5.8
5.5 Machine Learning Life Cycle 5.9
5.6 Difference between MLOps and DevOps 5.11
5.7 Tools to Create ML Pipelines 5.13
- **Exercise** 5.13

Unit VI: ML Model Development **6.1-6.44**

6.1 MLOps Maturity Model Levels 6.1
6.2 MLOps - Stages of CI / CD 6.4
6.3 Creating and Deploying ML/AI Models 6.5
6.4 ML Pipelines 6.6
6.5 Automation of ML through Pipelines 6.7
6.6 Tools to Create ML Pipelines 6.8
6.7 Monitoring and Logging 6.10
6.8 Data Quality and Integrity 6.11

6.9 Model Retraining and Model Replacement 6.12
6.10 Model Versioning 6.12
6.11 MLOps - Infrastructure 6.13
6.12 MLOps - Testing 6.20
6.13 Monitoring and Maintenance 6.31
- **Exercise** 6.43

- **Model Question Papers**
  - ➢ **Mid-Sem. Exam. (30 Marks)** **P.1 – P.1**
  - ➢ **End-Sem. Exam. (70 Marks)** **P.2 – P.2**

✠ ✠ ✠

UNIT - I

INTRODUCTION TO DevOps AND THE CULTURE

1.1 INTRODUCTION TO DevOps

What is DevOps?

- DevOps helps to improve communication between teams, increase collaboration and ensure that applications are always meeting the needs of their users.
- With a focus on automation, monitoring and feedback, DevOps aims for efficient and continuous delivery of products. It strives to create a culture where developers can be agile and responsive to user needs while ensuring that systems are stable and secure.
- DevOps project managers believe that developers and operations must work together effectively to break down silos. By doing so, you can create better applications and services that are more reliable and meet customer needs.

  The Key Benefits of DevOps Include:

  ➢ Reduced time needed to launch new features,

  ➢ Improved customer satisfaction,

  ➢ Reduced cost of ownership.

Four Pillars of DevOps:

(i) Culture,

(ii) Automation,

(iii) Metrics and

(iv) Sharing

- In DevOps, development and operations teams collaborate and communicate with one another. The goal is to break down the silos that have traditionally existed between these two teams and work together to deliver software faster and more efficiently. They do so by incorporating the DevOps pillars.
- Collaboration, communication and integration between the operations and development teams are the keys to a thriving DevOps culture. The success of a DevOps team depends on each of these pillars:

(i) Culture: DevOps culture revolves around culture as its first pillar. Culture is the basis on which everything else is built. Development and operations teams can collaborate, communicate and integrate with a robust DevOps culture. It also encourages a focus on continuous improvement and customer satisfaction.

(ii) Automation: The second pillar of DevOps culture is automation. Automation helps to speed up the software delivery process by automating repetitive tasks and eliminating manual errors. Automation also allows teams to focus on more strategic tasks, such as improving the quality of the codebase or adding new features.

(iii) Metrics: The third pillar of DevOps culture is metrics. Metrics help measure the team's speed, quality and reliability success. Additionally, they provide feedback to improve the process. Several metrics are commonly used, including lead time, mean time to repair and mean time to recover.

(iv) Sharing: The fourth pillar of DevOps culture is sharing. Sharing knowledge and best practices across the organization helps to improve the quality of the software delivered and speeds up the process. It also helps to build a community of practice that can share lessons learned and support others.

1.2 ROLE OF DevOps ENGINEER

- A DevOps engineer is responsible for the smooth operation of a company's IT infrastructure. They work with developers to deploy and manage code changes and with operations staff to ensure that systems are up and running smoothly. To be successful in this role, a DevOps engineer must have a deep understanding of both development and operations processes, as well as a strong technical background.
- As the world of business becomes increasingly reliant on technology, the role of a DevOps engineer is becoming more and more important. Companies are looking for individuals who can help them to

streamline their operations and make the most of their IT infrastructure. If you have a strong technical background and are interested in working with both development and operations staff, then a career as a DevOps engineer could be the perfect fit for you.

1.2.1 DevOps Engineer Job Description

Roles and Responsibilities

- A DevOps engineer's roles and responsibilities are a combination of technical and management roles. It is essential to have excellent communication and coordination skills to successfully integrate various functions in a coordinated manner and deliver the responsibilities to the customer's satisfaction.
- The DevOps engineer's responsibilities are multi-prong - they need to be agile enough to wear a technical hat and manage operations simultaneously.

Some of the Core Responsibilities of DevOps Engineer Include:

- Understanding customer requirements and project KPIs.
- Implementing various development, testing, automation tools and IT infrastructure.
- Planning the team structure, activities and involvement in project management activities.
- Managing stakeholders and external interfaces.
- Setting up tools and required infrastructure.
- Defining and setting development, test, release, update and support processes for DevOps operation.
- Have the technical skill to review, verify and validate the software code developed in the project.
- Troubleshooting techniques and fixing the code bugs.
- Monitoring the processes during the entire lifecycle for its adherence and updating or creating new processes for improvement and minimizing the wastage.
- Encouraging and building automated processes wherever possible.
- Identifying and deploying cybersecurity measures by continuously performing vulnerability assessment and risk management.
- Incidence management and root cause analysis.
- Coordination and communication within the team and with customers.
- Selecting and deploying appropriate CI/CD tools.
- Strive for continuous improvement and build continuous integration, continuous development and constant deployment pipeline (CI/CD Pipeline).
- Mentoring and guiding the team members.
- Monitoring and measuring customer experience and KPIs.
- Managing periodic reporting on the progress to the management and the customer.

1.3 INTRODUCTION TO CONTINUOUS INTEGRATION AND CONTINUOUS DELIVERY POLICIES

Introduction

- Developing and releasing software can be a complicated process, especially as applications, teams and deployment infrastructure grow in complexity. Often, challenges become more pronounced as projects grow. To develop, test and release software in a quick and consistent way, developers and organizations have created three related but distinct strategies to manage and automate these processes.
- Continuous integration focuses on integrating work from individual developers into a main repository multiple times a day to catch integration bugs early and accelerate collaborative development. Continuous delivery is concerned with reducing friction in the deployment or release process, automating the steps required to deploy a build so that code can be released safely at any time. Continuous deployment takes this one step further by automatically deploying each time a code change is made.
- Here, we will discuss each of these strategies, how they relate to one another and how incorporating them into your application life cycle can transform your software development and release practices. To get a better idea of the differences between various open-source CI/CD projects, check out our CI/CD tool comparison.

1.3.1 What is Continuous Integration and Why is it Helpful?

- **Continuous integration** is a practice that encourages developers to integrate their code into a main branch of a shared repository early and often. Instead of building out features in isolation and integrating them at the end of a development cycle, code is integrated with the shared repository by each developer multiple times throughout the day.

- The idea is to minimize the cost of integration by making it an early consideration. Developers can discover conflicts at the boundaries between new and existing code early, while conflicts are still relatively easy to reconcile. Once the conflict is resolved, work can continue with confidence that the new code honors the requirements of the existing codebase.
- Integrating code frequently does not, by itself, offer any guarantees about the quality of the new code or functionality. In many organizations, integration is costly because manual processes are used to ensure that the code meets standards, does not introduce bugs and does not break existing functionality. Frequent integration can create friction when an approach to automation does not align with quality assurance measures in place.
- To address this friction within the integration process, in practice, continuous integration relies on robust test suites and an automated system to run those tests. When a developer merges code into the main repository, automated processes kick off a build of the new code. Afterwards, test suites are run against the new build to check whether any integration problems were introduced. If either the build or the test phase fails, the team is alerted so that they can work to fix the build.
- The end goal of continuous integration is to make integration a simple, repeatable process that is part of the everyday development workflow in order to reduce integration costs and respond to defects early. Working to make sure the system is robust, automated and fast while cultivating a team culture that encourages frequent iteration and responsiveness to build issues is fundamental to CI success.

1.3.2 What is Continuous Delivery and Why is it Helpful?

- **Continuous delivery** is an extension of continuous integration. It focuses on automating the software delivery process so that teams can easily and confidently deploy their code to production at any time. By ensuring that the codebase is always in a deployable state, releasing software becomes an unremarkable event, without any complicated rituals. Teams can be confident that they can release whenever they need to without complex coordination or late-stage testing. As with continuous integration, continuous delivery is a practice that requires a mixture of technical and organizational improvements to be effective.
- On the technology side, continuous delivery leans heavily on deployment pipelines to automate the testing and deployment processes. A deployment pipeline is an automated system that runs increasingly rigorous test suites against a build as a series of sequential stages. This picks up where continuous integration leaves off, so a reliable continuous integration setup is a prerequisite to implementing continuous delivery.
- At each stage, the build either fails the tests, which alerts the team, or passes the tests, which results in automatic promotion to the next stage. As the build moves through the pipeline, later stages deploy the build to environments that mirror the production environment as closely as possible. This way the build, the deployment process and the environment can be tested in tandem. The pipeline ends with a build that can be deployed to production at any time in a single step.
- The organizational aspects of continuous delivery encourage prioritization of "deployability" as a principle concern. This has an impact on the way that features are built and hooked into the rest of the codebase. Thought must be put into the design of the code so that features can be safely deployed to production at any time, even when incomplete. A number of techniques have emerged to assist in this area.
- Continuous delivery is compelling because it automates the steps between checking code into the repository and deciding on whether to release well-tested, functional builds to your production infrastructure. The steps that help assert the quality and correctness of the code are automated, but the final decision about what to release is left in the hands of the organization for maximum flexibility.

1.3.3 What is Continuous Deployment and Why is it Helpful?

- **Continuous deployment** is an extension of continuous delivery that automatically deploys each build that passes the full test cycle. Instead of waiting for a human gatekeeper to decide what and when to deploy to production, a continuous deployment system deploys everything that has successfully traversed the deployment pipeline. Keep in mind that when new

code is automatically deployed, new features can still be activated conditionally at a later time or for a subset of users. Deploying automatically pushes features and fixes to customers quickly, encourages smaller changes with limited scope and helps avoid confusion over what is currently deployed to production.

- This fully automated deploy cycle can be a source of anxiety for organizations worried about relinquishing control to their automation system of what gets released. The trade-off offered by automated deployments is sometimes considered too dangerous for the payoff they provide.
- Other groups leverage this approach as a means of ensuring that best practices are always followed. Without a final manual verification before deploying a piece of code, developers must take responsibility for ensuring that their code is well-designed and that the test suites are up-to-date. This consolidates any decision-making around what and when to commit to the main repository and what and when to release to production into a single decision point for the development team.
- Continuous deployment also allows organizations to benefit from consistent early feedback. Features can immediately be made available to users and defects or unhelpful implementations can be caught early before the team devotes extensive effort in an unproductive direction. Getting fast feedback that a feature is not helpful lets the team shift focus rather than sinking more energy into an area with minimal impact.

1.3.4 Key Concepts and Practices for Continuous Processes

- While continuous integration, delivery and deployment vary in the scope of their involvement, there are some concepts and practices that are fundamental to the success of each.

1. Small, Iterative Changes

- One of the most important practices when adopting continuous integration is to encourage small changes. Developers should practice breaking up larger work into small pieces and committing those early. Special techniques like branching by abstraction and feature flags help to protect the functionality of the main branch from in-progress code changes.
- Small changes minimize the possibility and impact of integration problems. By committing to the shared branch at the earliest possible stage and then continually throughout development, the cost of integration is diminished and unrelated work is synchronized regularly.

2. Trunk-Based Development

- With trunk-based development, work is done in the main branch ("trunk") of the repository or merged back into the shared repository at frequent intervals. Short-lived feature branches are permissible as long as they represent small changes and are merged back as soon as possible.
- The idea behind trunk-based development is to avoid large commits that violate of concept of small, iterative changes discussed above. Code is available to peers early so that conflicts can be resolved when their scope is small.
- Releases are performed from the main branch or from a release branch created from the trunk specifically for that purpose. No development occurs on the release branches in order to maintain focus on the main branch as the single source of truth.

3. Keep the Building and Testing Phases Fast

- Each of the processes relies on automated building and testing to validate correctness. Because the build and test steps must be performed frequently, it is essential that these processes be streamlined to minimize the time spent on these steps.
- Increases in build time should be treated as a major problem because the impact is compounded by the fact that each commit kicks off a build. Because continuous processes force developers to engage with these activities daily, reducing friction in these areas is very worthwhile.
- When possible, running different sections of the test suite in parallel can help move the build through the pipeline faster. Care should also be taken to make sure the proportion of each type of test makes sense. Unit tests are typically very fast and have minimal maintenance overhead. In contrast, acceptance testing is often complex and prone to breakage. To account for this, it is often a good idea to rely heavily on unit tests, conduct a fair number of integration tests and minimize the number of more complex tests.

4. Consistency Throughout the Deployment Pipeline

- Because a continuous delivery or deployment implementations is supposed to be testing release worthiness, it is essential to maintain consistency

during each step of the process the build itself, the deployment environments and the deployment process itself:

- ➢ **Code Should Be Built Once at the Beginning of the Pipeline:** The resulting software should be stored and accessible to later processes without rebuilding. By using the exact same artifact in each phase, you can be certain that you are not introducing inconsistencies as a result of different build tools.
- ➢ **Deployment Environments Should be Consistent:** A configuration management system can control the various environments and environmental changes can be put through the deployment pipeline itself to ensure correctness and consistency. Clean deployment environments should be provisioned each test cycle to prevent legacy conditions from compromising the integrity of the tests. The staging environments should match the production environment as closely as possible to reduce unknown factors present when the build is promoted.
- ➢ **Consistent Processes Should be Used to Deploy the Build in Each Environment:** Each deployment should be automated and each deployment should use the same centralized tools and procedures. Ad-hoc deployments should be eliminated in favor of deploying only with the pipeline tools.

5. Decouple Deployment and Release

- Separating the deployment of code from its release to users is an extremely powerful part of continuous delivery and deployment. Code can be deployed to production without initially activating it or making it accessible to users. Then, the organization decides when to release new functionality or features independent from deployment.
- This gives organizations a great deal of flexibility by separating business decisions from technical processes. If the code is already on the servers, then deployment is no longer a delicate part of the release process, which minimizes the number of participants and the amount of work involved at the time of release.
- There are a number of techniques that help teams deploy the code responsible for a feature without releasing it. Feature flags set up conditional logic to check whether to run code based on the value of an environmental variable. Branching by abstraction allows developers to rewrite processes incrementally by creating an abstraction layer between process input and output. Careful planning to incorporate these techniques gives you the ability to decouple these two processes.

1.3.5 Types of Testing

- Continuous integration, delivery and deployment all rely heavily on automated tests to determine the efficacy and correctness of each code change. Different types of tests are needed throughout these processes to assert confidence in a given solution.
- While the categories below in no way represent an exhaustive list and although there is disagreement on the exact definition of each type, these broad categories of tests represent a variety of ways to evaluate code in different contexts.

1. Smoke Testing

- Smoke tests are a special kind of initial checks designed to ensure core functionality as well as some fundamental implementation and environmental assumptions. Smoke tests are generally run at the very start of each testing cycle as a sanity check before running a more complete test suite.
- The idea behind this type of test is to help to catch big red flags in an implementation and to bring attention to problems that might indicate that further testing is either not possible or not worthwhile. Smoke tests are not very extensive, but should be extremely quick. If a change fails a smoke test, it is an early signal that core assertions were broken and that you should not devote any more time to testing until the problem is resolved.
- Context-specific smoke tests can be employed at the start of any new phase testing to assert that the basic assumptions and requirements are met. For instance, smoke tests can be used both prior to integration testing or deploying to staging servers, but the conditions to be tested will vary in each case.

2. Unit Testing

- Unit tests are responsible for testing individual elements of code in an isolated and highly targeted way. The functionality of individual functions and classes are tested on their own. Any external dependencies are replaced with stub or mock implementations to focus the test completely on the code in question.

- Unit tests are essential to test the correctness of individual code components for internal consistency and correctness before they are placed in more complex contexts. The limited extent of the tests and the removal of dependencies makes it easier to hunt down the cause of any defects. It also is the best time to test a variety of inputs and code branches that might be difficult to reproduce later on. Often, after any smoke tests, unit tests are the first tests that are run when any changes are made.
- Unit tests are typically run by individual developers on their own workstation prior to submitting changes. However, continuous integration servers almost always run these tests again as a safeguard before beginning integration tests.

3. Integration Testing

- After unit testing, integration testing is performed by grouping together components and testing them as an assembly. While unit tests validate the functionality of code in isolation, integration tests ensure that components cooperate when interfacing with one another. This type of testing has the opportunity to catch an entirely different class of bugs that are exposed through interaction between components.
- Typically, integration tests are performed automatically when code is checked into a shared repository. A continuous integration server checks out the code, performs any necessary build steps (usually performing a quick smoke test to make sure the build was successful) and then runs unit and integration tests. Modules are hooked together in different combinations and tested.
- Integration tests are important for shared work because they protect the health of the project. Changes must prove that they do not break existing functionality and that they interact with other code as expected. A secondary aim of integration testing is to verify that the changes can be deployed into a clean environment. This is frequently the first testing cycle that is not performed on the developers' own machines, so unknown software and environmental dependencies can also be discovered during this process. This is usually also the first time that new code is tested against real external libraries, services and data.

4. System Testing

- Once integration tests are performed, another level of testing called system testing can begin. In many ways, system testing acts as an extension to integration testing. The focus of system tests are to make sure that groups of components function correctly as a cohesive whole.
- Instead of focusing on the interfaces between components, system tests typically evaluate the outward functionality of a full piece of software. This set of tests ignores the constituent parts in order to gauge the composed software as a unified entity. Because of this distinction, system tests usually focus on user- or externally-accessible interfaces.

5. Acceptance Testing

- Acceptance tests are one of the last types of tests that are performed on software prior to delivery. Acceptance testing is used to determine whether a piece of software satisfies all of the requirements from the business or user's perspective. These tests are sometimes built against the original specification and often test interfaces for some expected functionality and for usability.
- Acceptance testing is often a more involved phase that might extend past the release of the software. Automated acceptance testing can be used to make sure the technological requirements of the design were met, but manual verification usually also plays a role.
- Frequently, acceptance testing begins by deploying the build to a staging environment that mirrors the production system. From here, the automated test suites can run and internal users can access the system to check whether it functions the way they need it to. After releasing the software or offering beta access to users, further acceptance testing is performed by evaluating how the software functions in real-world use and by collecting additional feedback.

Additional Terminology

- While we have discussed some of the broader ideas above, there are many related concepts that you may come across as you learn about continuous integration, delivery and deployment. Let us define a few other terms you are likely to see:
  - ➢ **Blue-Green Deployment:** Blue-green deployment is a strategy for testing code in a production-like environment and for deploying code with minimal

downtime. Two sets of production-capable environments are maintained and code is deployed to the inactive set where testing can take place. When ready to release, production traffic is routed to the servers with the new code, instantly making the changes available.

- **Branching by Abstraction:** Branching by abstraction is a method of performing major refactoring operations in an active project without long-lived development branches in the source code repository, which continuous integration practices discourage. An abstraction layer is built and deployed within an existing implementation so that the new implementation can be built out behind the abstraction in parallel.
- **Build (Noun):** A build is a specific version of software created from source code. Depending on the language, this might be compiled code or a consistent bundle of interpreted code.
- **Canary Releases:** Canary releases are a strategy for releasing changes to a limited subset of users. The idea is to make sure everything works correctly with production workloads while minimizing the impact if there are problems.
- **Dark Launch:** Dark launching is the practice of deploying code to production that receives production traffic but does not impact the user experience. New changes are deployed alongside existing implementations and the same traffic is often routed to both places for testing. The old implementation is still hooked up to the user's interface, but behind the scenes, the new code can be evaluated for correctness using real user requests in the production environment.
- **Deployment Pipeline:** A deployment pipeline is a set of components that moves software through increasingly rigorous testing and deployment scenarios to evaluate its readiness for release. The pipeline typically ends by automatically deploying to production or providing the option to do so manually.
- **Feature Flags or Feature Toggles:** Feature flags are a technique of deploying new features behind conditional logic that determines whether or not to run based on the value of an environment variable. New code can be deployed to production without being activated by setting this flag appropriately. Feature flags often contain logic that allows for subsets of users to gain access to the new feature, creating a mechanism to gradually roll out the new code.
- **Promoting:** In the context of continuous processes, promoting means moving a software build through to the next stage of testing.
- **Soak Test:** Soak testing involves testing software under significant production or production-like load for an extended period of time.

1.4 DevOps CULTURE

- At its essence, a DevOps culture involves closer collaboration and a shared responsibility between development and operations for the products they create and maintain. This helps companies align their people, processes and tools toward a more unified customer focus.
- It involves cultivating multidisciplinary teams who take accountability for the entire lifecycle of a product. DevOps teams work autonomously and embrace a software engineering culture, workflow and toolset that elevates operational requirements to the same level of importance as architecture, design and development. The understanding that developers who build it, also run it, brings developers closer to the user, with a greater understanding of user requirements and needs. With operations teams more involved in the development process, they can add maintenance requirements and customer needs for a better product.
- At the heart of DevOps culture is increased transparency, communication and collaboration between teams that traditionally worked in siloes. But there are important cultural shifts that need to happen to bring these teams closer together. DevOps is an organizational culture shift that emphasizes continuous learning and continuous improvement, especially through team autonomy, fast feedback, high empathy and trust and cross-team collaboration.



Related material

Embrace a customer-first mindset



Related Material

Advantages of DevOps

- **DevOps Entails Shared Responsibilities:** Development and operations staff should both be responsible for the success or failure of a product. Developers are expected to do more than just build and hand off to operations they are expected to share the responsibility of overseeing a product through the entire course of its lifetime, adopting a "you build it, you run it" mentality. They test and operate software and collaborate more with QA and IT Ops. When they understand the challenges faced by operations, they are more likely to simplify deployment and maintenance. Likewise, when operations understand the system's business goals, they can work with developers to help define the operational needs of a system and adopt automation tools.
- **Autonomous Teams** are another important aspect of DevOps. For the development and operations teams to collaborate effectively, they need to make decisions and implement changes without a cumbersome and lengthy approval process. This involves handing over trust to teams and establishing an environment where there is no fear of failure. These teams should have the right processes and tools in place to make decisions faster and easier, for each level of risk to the customer.

  For example, a typical development workflow might require engagement from several contributors on different teams to deploy code changes. The developer makes a code change and pushes it to a source control repository, a build engineer builds and deploys the code to a test environment, a product owner updates the status of the work in an issue tracking tool, etc. An autonomous team will take advantage of tools that automate these processes, so that pushing new code triggers the building and deployment of a new feature into a test environment and the issue tracking tool is updated automatically.

  For example, a team is handicapped by requirements like having to open a ticket with a separate operations team to make a trivial infrastructure change, such as a new DNS entry. A task that should take seconds to complete ends up taking days or weeks to satisfy. An autonomous team has the ability to implement such changes themselves, whether by having an individual on the team who has the correct skills and experience, or by having access to self-service tooling.
- A DevOps team culture values fast **feedback** that can help with continuous improvement of a unified development and operations team. In an environment where the development and operations teams are in isolated silos, feedback about the performance and stability of application software in production is often slow to make it back to the development team, if it makes it at all. DevOps ensures that developers get the fast feedback they need to rapidly iterate and improve on application code by requiring collaboration between operations folks in designing and implementing application monitoring and reporting strategies. For example, any sufficiently capable continuous integration tool will enable automated build and testing of new code pushes and provide the developer with immediate feedback on the quality of their code.
- **Automation** is essential to DevOps culture, since it allows great collaboration and frees up resources. Automating and integrating the processes between software development and IT teams helps them to build, test and release software faster and more reliably.

1.5 BARRIERS TO DevOps

A recent survey of 2,000 IT executives conducted by cloud sandbox software company Quali highlighted these top 9 barriers to DevOps implementation:

- **Culture:** Many understand the importance of culture when it comes to DevOps. Few IT executives understand HOW to cultivate a successful DevOps culture.
- **Test Automation:** Continuous testing a vital part of the DevOps model but often gets forgotten in the organized chaos. Implement AlertOps and set triggers to automatically kick-off tests.
- **Legacy Systems:** Introducing new tools or systems into existing ones can be tough. Implement AlertOps, with an open API, for easy integration into all your systems allowing for enhanced visibility and incident response automation.