DevOps

UNIDADE 03

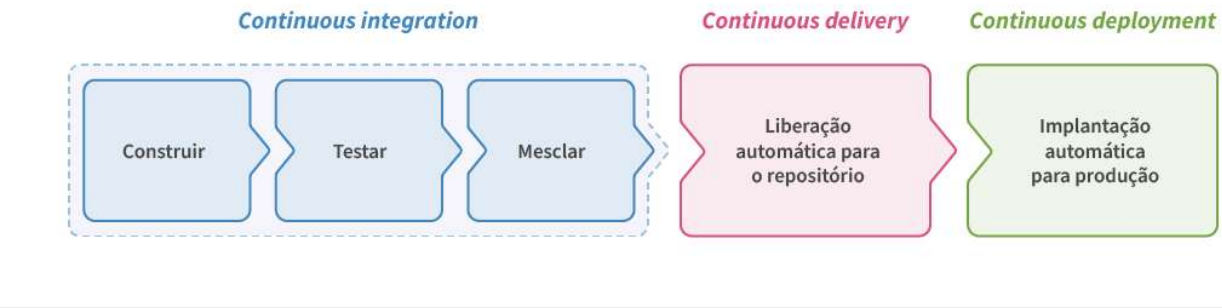
Trabalhando do jeito que fazem no serviço: CI/CD

PIPELINES DE CI/CD

Lembrando o que comentamos anteriormente: CI/CD é um conjunto de práticas e ferramentas que permite aos desenvolvedores entregar código rápido, de um jeito seguro e eficiente. Ele é composto de duas partes:

1. **CI** (***continuous integration***): aqui, a ideia é integrar pequenas partes do código em um repositório compartilhado várias vezes ao dia, garantindo que cada mudança seja testada e validada automaticamente.
2. **CD** (***continuous delivery***): aqui, montamos uma infraestrutura de um jeito que o código é sempre mantido em um estado que pode ser implantado em produção a qualquer momento, mas a implantação real ainda é feita **manualmente**.
    a. Também existe outro termo que também se chama CD: o ***continuous deployment***. O código é automaticamente implantado em produção assim que passa por todas as etapas de teste e verificação, **sem necessidade de intervenção manual**. É como se fosse um passo além do *continuous delivery*.

#ParaTodosVerem



Fonte: Red Hat, 2020 (Adaptado).

DICA

O CD no termo CI/CD significa continuous delivery, e não continuous deployment. Contudo, muita gente se confunde com os dois termos, incluindo a pessoa que pode te entrevistar para um novo emprego. Portanto, se alguém se perguntar sobre CI/CD em uma entrevista, é legal você falar da diferença entre os dois tipos de CD.

FAZENDO CI/CD COM GITHUB ACTIONS

Antes de continuarmos é importante entendermos as principais diferenças entre CI e os dois tipos de CD. Portanto, veja a tabela comparativa abaixo antes de continuarmos.

Característica	*Continuous integration* (CI)	*Continuous delivery* (CD)	*Continuous deployment* (CD)
Analogia	É como verificar e preparar os ingredientes para uma refeição. Você garante que tudo está fresco e pronto para ser usado.	É como preparar a refeição e deixá-la pronta para servir. Você só precisa da aprovação final de outra pessoa para colocá-la na mesa.	Assim que a refeição está pronta, ela é imediatamente colocada na mesa sem nenhuma intervenção adicional.
Objetivo	Automatizar a construção e testes do código.	Manter o código sempre pronto para ser implantado.	Automatizar o *deploy* do código em produção.
Foco principal	Verificação contínua da integridade do código.	Preparação contínua do código para produção.	*Deploy* contínuo do código em produção.
Gatilho	Cada *commit* ou *pull request*	Cada *commit* ou *pull request* após passar pela fase de CI.	Cada *commit* ou *pull request* após passar pela fase de CI.
Automação	Construção e testes.	Construção, testes e preparação de artefatos.	Construção, testes, preparação de artefat e deploy.
Intervenção manual	Nenhuma.	*Deploy* manual em produção.	Nenhuma (*deploy* automático em produção)
Benefícios	- *Feedback* rápido sobre a qualidade do código.	- Menor tempo de preparação para *deploy*.	- Menor tempo de entrega de funcionalidades.
	- Identificação precoce de problemas.	- Maior confiança na prontidão do código.	- Entrega rápida e contínua de novas funcionalidades.
Risco	Baixo.	Médio.	Alto.
Complexidade	Moderada.	Moderada a alta.	Alta.
Exemplo de Atividades	- Compilar código.	- Compilar código.	- Compilar código.
	- Executar testes.	- Executar testes.	- Executar testes.
	- *Feedback* rápido sobre erros.	- Preparar artefatos de lançamento.	- Preparar artefatos de lançamento.

		- Revisão manual antes da implantação.	- *Deploy* automático no ambiente de produção.
Ferramentas Usadas	GitHub Actions, Jenkins, Travis CI, CircleCI		
Exemplo de Configuração	push ou pull_request desencadeia o *build* e os testes.	push ou pull_request desencadeia o *build*, testes e preparação de artefatos.	push ou pull_request desencadeia o build, testes, preparação de artefatos e *deploy*.

CI

No mercado de trabalho, CI significa automatizar a integração do código que os desenvolvedores estão criando. Basicamente, cada vez que alguém faz uma alteração no código e a envia (ou faz o *commit*) para o repositório compartilhado, um sistema de CI entra em ação. Esse sistema pode ser algo como **Jenkins, Travis CI, CircleCI,** ou **GitHub Actions**.

Esses sistemas normalmente pegam o código novo, integram com o código existente, e executam uma **série de testes automáticos**. Se os testes passarem, ótimo! Sabemos que essa nova parte do código que introduzimos não quebrou nada do que já estava funcionando. Se falharem, os desenvolvedores são notificados imediatamente para que possam corrigir os problemas.

DICA
Com o tempo, você escreverá códigos que quebrem menos e saberá resolver os erros com maior velocidade. É por isso que dizemos que colocar a mão na massa é fundamental em nossa área.

Ter esses testes automáticos ajuda a detectar erros rapidamente, antes que eles se tornem grandes problemas. Além disso, garantem que todos os desenvolvedores estejam sempre trabalhando com a versão mais atualizada e funcional do código, o que ajuda muito quando temos equipes grandes em que várias pessoas podem estar modificando diferentes partes do sistema ao mesmo tempo.

Agora, dá para fazer CI do jeito certo e do jeito errado. Diria que alguns erros comuns que vemos no uso de CI são:

1. **Falta de testes abrangentes**: muitas vezes, as empresas implementam CI, mas os testes automáticos não cobrem todas as funcionalidades importantes. Isso faz com que problemas passem despercebidos, mesmo tendo criado um CI. Por isso que é bem importante investir tempo para escrever testes que realmente validem todas as partes críticas do sistema.
2. **Integração ocasional**: às vezes, os desenvolvedores evitam enviar suas mudanças frequentemente. No final, acabam criando *pull requests* (PRs) gigantescas, o que não é nada legal para validarmos tudo o que essas pessoas fazem. A ideia do CI é justamente integrar frequentemente e, de preferência, várias vezes ao dia. Quando as mudanças são integradas esporadicamente, também se aumenta o risco de conflitos complicados e *bugs* difíceis de resolver.
3. **Falhas nos *scripts* de automação**: os scripts que automatizam os testes e integrações podem ter falhas. Por isso, é legal revisarmos estes *scripts* periodicamente e garantir que estão funcionando corretamente, para que o processo de CI seja confiável.
4. **Subestimar a infraestrutura**: testes automatizados requerem uma infraestrutura própria. Se os servidores de CI forem lentos ou instáveis, o processo de integração contínua pode se tornar um gargalo. Por isso, também vale a pena monitorarmos essa infraestrutura para manter a produtividade.
5. **Ignorar a qualidade do código**: implementar CI não significa que podemos criar qualquer código sem nos preocuparmos com a qualidade. Boas práticas de desenvolvimento, revisões de código e refatorações contínuas ainda são necessárias. CI é uma ferramenta que auxilia, mas a responsabilidade pela qualidade do código é de todos na equipe.

Nesse ponto, você já sabe qual é a diferença entre CI e CD, mas não colocamos a mão na massa até agora. Logo, é o momento de criarmos o nosso primeiro *pipeline* de CI. Vamos juntos nessa?

CD (*delivery*, e não o *deployment*)

Neste momento, temos CI. Agora, e quanto ao CD? Afinal de contas, queremos ver como seria uma *pipeline* CI/CD. Lembre-se: **a ideia central do CD é garantir que o código que passa pelos testes de CI esteja sempre em um estado que pode ser implantado em produção a qualquer momento**. Isto quer dizer que após cada mudança no código, ele não só é testado automaticamente, mas também preparado para ser lançado com segurança.

Essa fase de CD significa, na prática, termos coisas como **testes de integração** e **testes de sistema**. Se os testes unitários do CI verificam pequenos pedaços do código, os testes integrados garantem que todo o sistema esteja funcionando como deveria. Também podemos ter definições da **infraestrutura como código** (*infrastructure-as-code*, ou IaC). A ideia é usar ferramentas como o Ansible ou o Terraform para definirmos toda a infraestrutura necessária para rodar um sistema em código.

O que acha de vermos juntos como podemos melhorar aquele *workflow* de CI para que seja, de fato, CI/CD?

CD (agora sim, o *deployment*)

Beleza, já fizemos o CI/CD. Contudo, lembre-se que também temos um segundo tipo de CD: o *continuous deployment*. No CD, cada mudança no código que passa pelos testes automáticos é imediatamente implantada em produção, sem intervenção humana. Isso significa que qualquer *commit* bem-sucedido pode resultar em uma nova versão do *software* sendo disponibilizada para os usuários finais quase que instantaneamente.

Isso, na prática, implica em coisas como:

1. ***Deploy* automático para produção:** reforçando o que comentei acima – assim que o código passa por todos os testes e verificações, ele é automaticamente implantado no ambiente de produção. Isso requer um sistema robusto de automação de *deploy* que pode ser implementado com ferramentas como o próprio GitHub Actions ou, ainda o Jenkins, CircleCI, GitLab CI, entre outras soluções de mercado.
2. **Monitoramento contínuo**: automatizar o *deploy* aumenta os riscos de subirmos alguma coisa que não deveríamos ou, ainda, introduzir algum tipo de erro ou falha no nosso *software* sem querer. Por isso, ferramentas de monitoramento como Prometheus, Grafana e o que o mercado chama de **ELK Stack** (Elasticsearch + Logstash + Kibana) ajudam a detectar problemas rapidamente.

3. ***Feature flags***: para evitar o risco de lançarmos uma nova funcionalidade em produção sem querer, temos o conceito de *feature flags*. São configurações de sistema que permitem ativar ou desativar funcionalidades específicas sem precisar implantar novo código, facilitando testes e *rollbacks*.
4. ***Rollback* automático**: aconteceu algum problema em produção e precisamos resolver o mais rápido possível? É aí que os mecanismos de *rollback* automático são essenciais. Se uma mudança causa falhas, o sistema pode automaticamente reverter para a versão anterior que era estável.

É claro que implementar esse tipo de CD é legal, mas é importante tomar cuidado com a forma pela qual fazemos uma implementação. Aqui, é importante termos um cuidado adicional ao prezarmos pela qualidade do nosso código, bem como tomar cuidado com:

1. **Preguiça em criar testes**: *continuous deployment* significa usarmos testes automáticos, certo? Logo, para diminuir os riscos de problemas em produção, é importante termos bons testes, e em uma boa quantidade. Se os testes não forem abrangentes o suficiente, *bugs* podem chegar aos usuários finais.
2. **Monitoramento e alerta inadequados**: uma das chaves do sucesso do CD é identificar rapidamente os problemas. Para identificar rapidamente os problemas, precisamos, obrigatoriamente, de um monitoramento proativo e alertas em tempo real. Sem isso, problemas em produção podem passar despercebidos até que os usuários reclamem.
3. **Falta de cultura de responsabilidade**: é importante ter uma cultura de confiança e colaboração entre as equipes. De fato, CD requer uma cultura de alta responsabilidade e colaboração entre todas as equipes envolvidas. Sem isso, os problemas podem ser jogados por baixo do tapete e não seriam resolvidos rapidamente.

E agora, o que acha de implementarmos esse segundo tipo de CD (o *continuous deployment*) no nosso repositório?

CONCLUSÃO

E chegamos ao final dos estudos desta terceira semana. Com isso, quero reforçar a importância dessas práticas para a sua carreira. CI/CD é essencial no desenvolvimento de *software* hoje em dia porque ajuda a entregar código mais rápido e com menos erros. Aqui, usamos o GitHub Actions. Ele permite a você automatizar o processo de integração e entrega contínua, garantindo que cada mudança no código seja testada e pronta para ser lançada. Isso não só melhora a qualidade do *software*, mas também aumenta (ao menos na teoria) a eficiência do time de desenvolvimento.

Além disso, saber CI/CD e usar ferramentas como o GitHub Actions é um grande diferencial no mercado de trabalho – especialmente nos anos iniciais da sua carreira. As empresas buscam desenvolvedores que entendam a importância da automação e da entrega contínua, e geralmente perguntam sobre esses temas em entrevistas de emprego. Dominar essas habilidades mostra que você está preparado para enfrentar os desafios modernos do desenvolvimento de *software* e, também, um novo emprego na área.

REFERÊNCIAS

FREEMAN, E. **DevOps para leigos**. Rio de Janeiro: Editora Alta Books, 2021.

KIM, G.; BEHR, K.; SPAFFORD, G. **O projeto Fênix**. Rio de Janeiro: Editora Alta Books, 2020.

KIM, G.; HUMBLE, J.; DEBOIS, P.; WILLIS, J. **Manual de DevOps**. Rio de Janeiro: Editora Alta Books, 2018.

O QUE é entrega contínua (CD)? Red hat, 26 nov. 2020. Disponível em: https://www.redhat.com/pt-br/topics/devops/what-is-continuous-delivery. Acesso em: 13 ago. 2024.