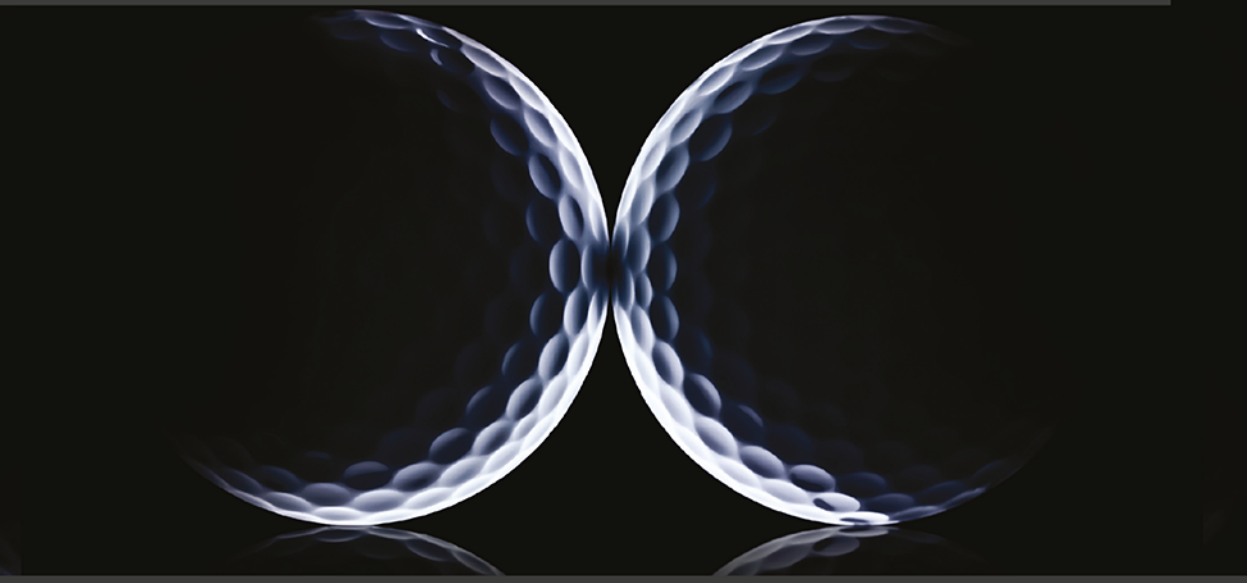
DevOps
w praktyce
Wydanie II

Wdrażanie narzędzi Terraform, Azure DevOps, Kubernetes i Jenkins



Mikael Krief

Helion

Tytuł oryginału: Learning DevOps: A comprehensive guide to accelerating DevOps culture adoption with Terraform, Azure DevOps, Kubernetes, and Jenkins, 2nd Edition

Tłumaczenie: Łukasz Wójcicki

ISBN: 978-83-8322-198-4

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/devpr2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

CZĘŚĆ 2. Potok CI/CD

CZĘŚĆ 4. Testowanie aplikacji

Kultura DevOps i praktyki kodowania infrastruktury

Rozdział 1

DevOps, termin, który coraz częściej słyszymy w przedsiębiorstwach z frazami typu „stosujemy DevOps" czy „używamy narzędzi DevOps", to skrót pochodzący od słów „Development" i „Operations".

DevOps to kultura, która różni się od tradycyjnych kultur korporacyjnych i wymaga zmiany sposobu myślenia, procesów i narzędzi. Często wiąże się to z praktykami **ciągłej integracji** (ang. *continuous integration* — CI) i **ciągłego dostarczania** (ang. *continuous delivery* — CD), które są związane z inżynierią oprogramowania, ale także z **infrastrukturą jako kod** (ang. *Infrastructure as Code* — IaC), polegającą na *kodyfikacji* struktury i konfiguracji infrastruktury.

W tym rozdziale zobaczymy, czym jest kultura DevOps, jakie są zasady DevOps i jakie korzyści przynoszą firmie. Następnie wyjaśnimy praktyki CI/CD, a na koniec omówimy szczegółowo IaC z jej wzorcami i praktykami.

W tym rozdziale omówimy następujące tematy:

- pierwsze kroki z DevOps,
- wdrażanie CI/CD i ciągłe wdrażanie,
- zrozumienie praktyk IaC.

Obejrzyj następujący film na kanale Code in Action: *https://bit.ly/3JJAMAb*.

Pierwsze kroki z DevOps

Termin *DevOps* został wprowadzony w latach 2007 – 2009 przez Patricka Debois, Gene'a Kima i Johna Willisa i reprezentuje połączenie słów *Development* (Dev) i *Operations* (Ops). Dało to początek ruchowi, który opowiada się za łączeniem razem programistów i operacji. Zapewnia to użytkownikom dodaną wartość biznesową dużo szybciej, co czyni ją bardziej konkurencyjną na rynku.

Kultura DevOps to zestaw praktyk zmniejszających bariery między *programistami*, którzy chcą szybciej wprowadzać innowacje i dostarczać, a *zespołami operacyjnymi*, które chcą zagwarantować stabilność systemów produkcyjnych i jakość wprowadzanych zmian systemowych.

Kultura DevOps to także rozszerzenie zwinnych (ang. *agile*) procesów (scrum, XP itd.), co pozwala skrócić czas dostawy, angażując programistów i zespoły biznesowe. Procesy te są często trudne do przeprowadzenia z powodu niewłączania operacji do tych samych zespołów.

Komunikacja i to połączenie między Dev i Ops umożliwiają lepsze śledzenie kompleksowych wdrożeń produkcyjnych i częstsze wdrożenia o wyższej jakości, co pozwala zaoszczędzić pieniądze dla firmy.

Aby ułatwić tę współpracę i poprawić komunikację między programistami a zespołami operacyjnymi, w procesach należy wprowadzić kilka kluczowych elementów, jak pokazano poniżej:

- Częstsze wdrożenia aplikacji z integracją i ciągłym dostarczaniem (tzw. **CI/CD**).
- Wdrażanie i automatyzacja testów jednostkowych i integracyjnych z procesem skoncentrowanym na **projektowaniu opartym na zachowaniu** (ang. *behawior-driven design* — **BDD**) lub **projektowaniu opartym na testach** (ang. *test-driven design* — **TDD**).
- Wdrożenie sposobu zbierania informacji zwrotnych od użytkowników.
- Monitorowanie aplikacji i infrastruktury.

Ruch DevOps opiera się na trzech założeniach:

- **Kultura współpracy.** To jest istota DevOps — fakt, że zespoły nie są już rozdzielone (jeden zespół programistów, jeden zespół Ops, jeden zespół testerów itd.). Ludzie ci łączą się, tworząc multidyscyplinarne zespoły, które mają ten sam cel: jak najszybsze dostarczenie wartości dodanej do produktu.
- **Procesy.** Aby oczekiwać szybkiego wdrożenia, zespoły te muszą śledzić procesy rozwoju oparte na metodologiach agile z iteracyjnymi fazami, które pozwalają na lepszą funkcjonalność, wyższą jakość i szybszą informację zwrotną. Te procesy powinny być zintegrowane nie tylko z przepływem pracy programistycznej z ciągłą integracją, ale także z przepływem pracy wdrażania z ciągłym dostarczaniem i wdrażaniem. Proces DevOps podzielony jest na kilka faz:
  A. Planowanie i ustalanie priorytetów funkcjonalności.
  B. Rozwój.
  C. Ciągła integracja i dostarczanie.

D. Ciągłe wdrażanie.

E. Ciągłe monitorowanie.

Fazy te są przeprowadzane cyklicznie i iteracyjnie przez cały czas trwania projektu.

- **Narzędzia.** Wybór narzędzi i produktów używanych przez zespoły jest bardzo ważny w DevOps. W rzeczywistości, kiedy zespoły zostały podzielone na Dev i Ops, każdy zespół używał swoich specyficznych narzędzi — narzędzi do wdrażania dla programistów i narzędzi infrastruktury dla Ops — co jeszcze bardziej zwiększyło luki komunikacyjne.

W zespołach, które łączą programowanie i operacje, oraz w tej kulturze jedności używane narzędzia muszą być użyteczne i możliwe do wykorzystania przez wszystkich członków.

Deweloperzy muszą zapoznać się z narzędziami monitorującymi używanymi przez zespoły Ops w celu jak najwcześniejszego wykrywania problemów z wydajnością i z narzędziami bezpieczeństwa dostarczanymi przez Ops w celu ochrony dostępu do różnych zasobów.

Ops z kolei musi zautomatyzować proces tworzenia i aktualizacji infrastruktury oraz zintegrować kod z menedżerem kodu. Wszystkie te czynności tworzą praktyki IaC. Można je jednak wykonać tylko we współpracy z programistami, którzy znają infrastrukturę potrzebną dla aplikacji. Zadania zespołów operacyjnych należy również zintegrować z procesami i narzędziami wydawania aplikacji.

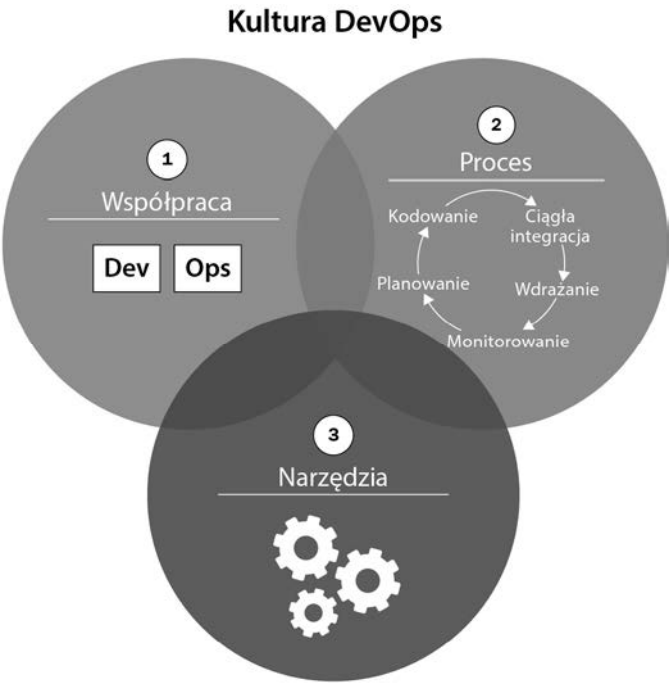
Poniższy diagram ilustruje trzy założenia kultury DevOps — współpracę między Dev i Ops, procesy i wykorzystanie narzędzi.

Możemy nawiązać do kultury DevOps za pomocą definicji Donovana Browna (*http://donovanbrown.com/post/what-is-devops*):

> *„DevOps to połączenie ludzi, procesów i produktów, które umożliwia ciągłe dostarczanie wartości naszym użytkownikom końcowym”.*

Korzyści z ustanowienia kultury DevOps w przedsiębiorstwie są następujące:

- Lepsza współpraca i komunikacja w zespołach, co ma wpływ na ludzi i na społeczne więzi w firmie.
- Krótsze czasy realizacji produkcji, co skutkuje lepszą wydajnością i satysfakcją użytkownika końcowego.
- Zmniejszone koszty infrastruktury dzięki IaC.



Rysunek 1.1. Kultura DevOps

- Znaczna oszczędność czasu dzięki cyklom iteracyjnym zmniejszającym liczbę błędów aplikacji i narzędziom automatyzacji, które ograniczają liczbę zadań wykonywanych ręcznie, dzięki czemu zespoły skupiają się bardziej na opracowywaniu nowych funkcji o wartości dodanej dla biznesu.

Uwaga

Aby uzyskać więcej informacji na temat kultury DevOps i jej wpływu na transformację przedsiębiorstw, przeczytaj książki *The Phoenix Project: A Novel about IT, DevOps and Helping Your Business Win* Gene'a Kima i Kevina Behra oraz *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations* Gene'a Kima, Jeza Humble'a, Patricka Debois i Johna Willisa.

W tej sekcji poznaliśmy podstawowe pojęcia kultury DevOps. Przyjrzyjmy się teraz pierwszej praktyce kultury DevOps: implementacji CI/CD i ciągłemu wdrażaniu.

Wdrażanie CI/CD i ciągłe wdrażanie

Wcześniej dowiedzieliśmy się, że jedną z kluczowych praktyk DevOps jest proces ciągłej integracji i ciągłego dostarczania, znany również jako **CI/CD**. W rzeczywistości za akronimami CI/CD kryją się trzy praktyki:

- **ciągła integracja** (**CI**),
- **ciągłe dostarczanie** (**CD**),
- **ciągłe wdrażanie**.

Czemu odpowiada każda z tych praktyk? Jakie są ich wymagania wstępne i które z nich są najlepsze? Gdzie mają zastosowanie?

Przyjrzyjmy się szczegółowo każdej z tych praktyk, zaczynając od ciągłej integracji.

Ciągła integracja (CI)

W poniższej definicji podanej przez Martina Fowlera wymienia się trzy kluczowe rzeczy: *członkowie zespołu integrują się i to jak najszybciej*:

> *„Ciągła integracja to praktyka tworzenia oprogramowania,*
> *w której członkowie zespołu często integrują swoją pracę...*
> *Każda integracja jest weryfikowana przez zautomatyzowaną kompilację*
> *(w tym test), aby jak najszybciej wykryć błędy integracji".*

Oznacza to, że CI to automatyczny proces, który pozwala na sprawdzenie kompletności kodu aplikacji za każdym razem, gdy członek zespołu dokona zmiany. Tę weryfikację należy przeprowadzić jak najszybciej.

Kulturę DevOps w CI widzimy bardzo wyraźnie, w duchu współpracy i komunikacji, ponieważ realizacja CI wpływa na wszystkich członków pod względem metodologii pracy, a tym samym współpracy; ponadto CI wymaga implementacji procesów (wprowadzanie zmian, zatwierdzanie, pobieranie i przegląd kodu itd.) wykorzystujących automatyzację za pomocą narzędzi dostosowanych do całego zespołu (Git, Jenkins, Azure DevOps itd.). Wreszcie CI musi działać szybko, aby jak najszybciej zebrać informacje zwrotne na temat integracji kodu, a tym samym być w stanie szybciej dostarczać nowe funkcje użytkownikom.

Wdrażanie CI

Dlatego aby skonfigurować CI, konieczne jest posiadanie **menedżera kodu źródłowego** (ang. *Source Code Manager* — **SCM**), który scentralizuje kod wszystkich członków. Ten menedżer może być dowolnego typu: Git, SVN lub *Team Foundation Version Control* (**TFVC**). Ważne jest również posiadanie automatycznego menedżera kompilacji (serwera CI), który obsługuje ciągłą integrację, takiego jak Jenkins, GitLab CI, TeamCity, Azure Pipelines, GitHub Actions, Travis CI i Circle CI.

> **Uwaga**
> W tej książce jako SCM użyjemy Gita i przyjrzymy się nieco dokładniej jego konkretnym zastosowaniom.

Każdy członek zespołu będzie pracował nad kodem aplikacji codziennie, iteracyjnie i przyrostowo (np. w metodach agile i scrum). Każde zadanie lub funkcja musi być oddzielona od innych rozwiązań za pomocą stosowania gałęzi (ang. *branch*).

Regularnie, nawet kilka razy dziennie, członkowie archiwizują lub zatwierdzają (ang. *commit*) swój kod, najlepiej za pomocą małych paczek (ang. *trunks*), które można łatwo naprawić w przypadku błędu. Zostanie on zintegrowany z resztą kodu aplikacji, z resztą zatwierdzeń innych członków.

Integracja wszystkich zatwierdzeń jest punktem wyjścia procesu CI.

Ten proces, który jest wykonywany przez serwer CI, musi być zautomatyzowany i uruchamiany przy każdym zatwierdzeniu. Serwer pobierze kod, a następnie wykona następujące czynności:

- Zbuduje pakiet aplikacji — kompilacja, transformacja plików itd.
- Wykona testy jednostkowe (z testem pokrycia — ang. *code coverage*).

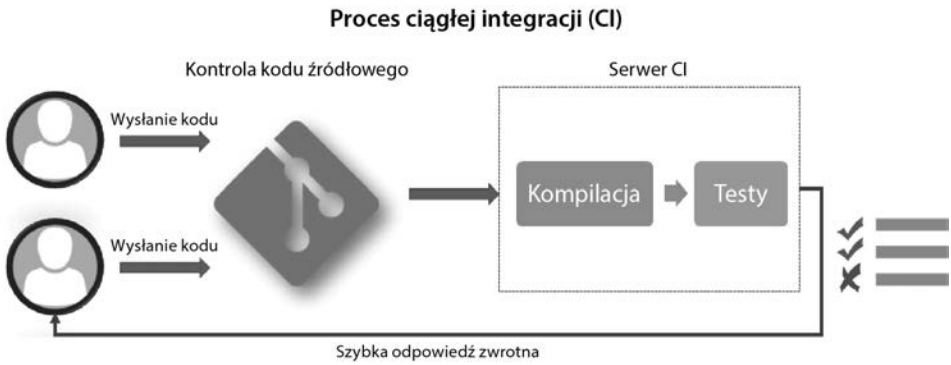
> **Uwaga**
> Możliwe jest również wzbogacenie tego procesu o statyczną analizę kodu i podatności. Przyjrzymy się temu w rozdziale 12., „Statyczna analiza kodu za pomocą SonarQube", który jest poświęcony testom.

Proces CI musi zostać zoptymalizowany najwcześniej, jak to możliwe, aby mógł działać natychmiast, a programiści mogli szybko zebrać informacje zwrotne na temat integracji ich kodu. Na przykład kod, który został zarchiwizowany i nie kompiluje się lub dla którego test kończy się niepowodzeniem, może wpływać na cały zespół i blokować go.

Czasami złe praktyki mogą spowodować niepowodzenie testów podczas CI. Przed dezaktywacją wykonania tego testu musisz wziąć pod uwagę to, że *nie jest ważne, aby koniecznie zadbać o szybkie dostarczenie kodu*.

Wręcz przeciwnie, ta praktyka może mieć poważne konsekwencje, gdy błędy wykryte przez testy zostaną ujawnione w produkcji. Czas zaoszczędzony podczas CI zostanie stracony na naprawianie błędów za pomocą poprawek i ich szybkie ponowne wdrażanie, co może powodować stres. Jest to przeciwieństwo kultury DevOps, ponieważ użytkownicy końcowi otrzymują niską jakość aplikacji i nie ma prawdziwych informacji zwrotnych; zamiast opracowywać nowe funkcje, spędzamy czas na poprawianiu błędów.

Dzięki zoptymalizowanemu i kompletnemu procesowi CI programista może szybko naprawić swoje problemy i ulepszyć kod lub przedyskutować go z resztą zespołu i zatwierdzić swój kod do nowej integracji. Spójrzmy na poniższy diagram:



Rysunek 1.2. Proces ciągłej integracji

Ten diagram przedstawia cykliczne etapy ciągłej integracji. Obejmuje to kod przesyłany do SCM przez członków zespołu oraz kompilację i test wykonywany przez serwer CI. Celem tego procesu jest zapewnienie członkom szybkiej informacji zwrotnej.

Teraz, gdy dowiedzieliśmy się, czym jest ciągła integracja, spójrzmy na ciągłe dostarczanie.

Ciągłe dostarczanie (CD)

Po zakończeniu ciągłej integracji następnym krokiem jest automatyczne wdrożenie aplikacji w co najmniej jednym środowisku nieprodukcyjnym, określanym jako ***staging***. Ten proces nazywa się **ciągłym dostarczaniem** (ang. *continuous delivery* — **CD**).

CD często rozpoczyna pracę z pakietem aplikacji przygotowywanym przez CI, który zostanie zainstalowany na podstawie listy zautomatyzowanych zadań. Mogą to być zadania dowolnego typu: rozpakowanie, zatrzymanie i ponowne uruchomienie usługi, skopiowanie plików, zamiana konfiguracji itd. Podczas procesu CD mogą być również wykonane testy funkcjonalne i akceptacyjne.

W przeciwieństwie do CI, CD ma na celu przetestowanie całej aplikacji ze wszystkimi jej zależnościami. Jest to szczególnie widoczne w aplikacjach mikroserwisowych składających się z kilku usług i interfejsów API; CI przetestuje tylko rozwijaną mikrousługę, podczas gdy po wdrożeniu w środowisku przejściowym będzie można przetestować i zweryfikować całą aplikację, a także interfejsy API i mikrousługi, z których się składa.

W praktyce obecnie bardzo często łączy się CI z CD w środowisku integracyjnym; oznacza to, że CI wdraża się w tym samym czasie w środowisku. Jest to konieczne, aby programiści mogli nie tylko wykonywać testy jednostkowe, ale także weryfikować aplikację jako całość (UI i funkcjonalną) przy każdym zatwierdzeniu, wraz z integracją opracowań innych członków zespołu.

Ważne jest, aby pakiet generowany podczas CI, który zostanie wdrożony również podczas CD, był tym samym, który zostanie zainstalowany we wszystkich środowiskach, i tak powinno być do czasu produkcji. Mogą jednak występować zmiany pliku konfiguracyjnego, które różnią się w zależności od środowiska, ale kod aplikacji (pliki binarne, pliki DLL, obrazy Dockera i JAR) musi pozostać niezmieniony.

Niezmienny charakter kodu jest jedyną gwarancją, że aplikacja weryfikowana w środowisku będzie tej samej jakości co wersja, która została wdrożona w poprzednim środowisku, a także ta sama, która zostanie wdrożona w następnym środowisku. Jeśli zmiany (ulepszenia lub poprawki błędów) mają zostać wprowadzone w kodzie po weryfikacji w jednym z tych środowisk, po ich wykonaniu modyfikacje będą musiały ponownie przejść cykl CI i CD.

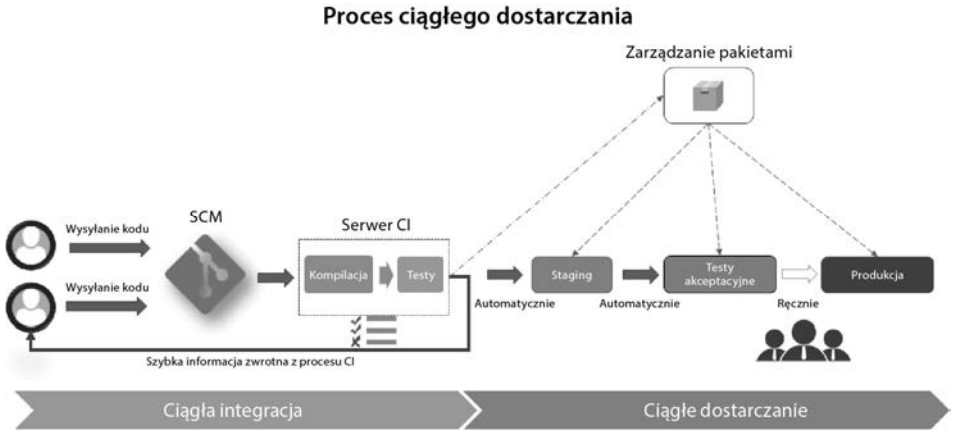
Narzędzia skonfigurowane dla CI/CD są często używane z innymi rozwiązaniami, takimi jak:

- **Menedżer pakietów** — stanowi przestrzeń przechowywania pakietów wygenerowanych przez CI i odzyskanych przez CD. Menedżery te muszą obsługiwać źródła danych, wersjonowanie i różne typy pakietów. Na rynku jest ich kilka, np.: Nexus, ProGet, Artifactory i Azure Artifacts.
- **Menedżer konfiguracji** — umożliwia zarządzanie zmianami konfiguracji podczas procesu CD; większość narzędzi CD zawiera mechanizm konfiguracyjny z systemem zmiennych.

Podczas procesu CD wdrażanie aplikacji w każdym środowisku przejściowym jest wyzwalane w następujący sposób:

- Może być wyzwolone automatycznie po pomyślnym wykonaniu w poprzednim środowisku. Na przykład możemy sobie wyobrazić przypadek, w którym wdrożenie w środowisku przedprodukcyjnym jest uruchamiane automatycznie po pomyślnym przeprowadzeniu testów integracyjnych w środowisku dedykowanym.
- Może być uruchomione ręcznie w przypadku wrażliwych środowisk, takich jak środowisko produkcyjne, po ręcznym zatwierdzeniu przez osobę odpowiedzialną za sprawdzenie poprawności działania aplikacji w środowisku.

W procesie CD ważne jest to, że wdrożenie do środowiska produkcyjnego — czyli dla użytkownika końcowego — jest uruchamiane ręcznie przez zatwierdzonych użytkowników.



Rysunek 1.3. Proces ciągłego dostarczania

Powyższy diagram wyraźnie pokazuje, że proces CD jest kontynuacją procesu CI. Reprezentuje łańcuch kroków CD, które są automatyczne dla środowisk przejściowych i ręczne dla wdrożeń produkcyjnych. Pokazuje również, że pakiet jest generowany przez CI i przechowywany w menedżerze pakietów oraz że jest to ten sam pakiet, który jest wdrażany w różnych środowiskach.

Teraz, gdy przyjrzeliśmy się CD, przyjrzyjmy się praktykom ciągłego wdrażania.

Ciągłe wdrażanie

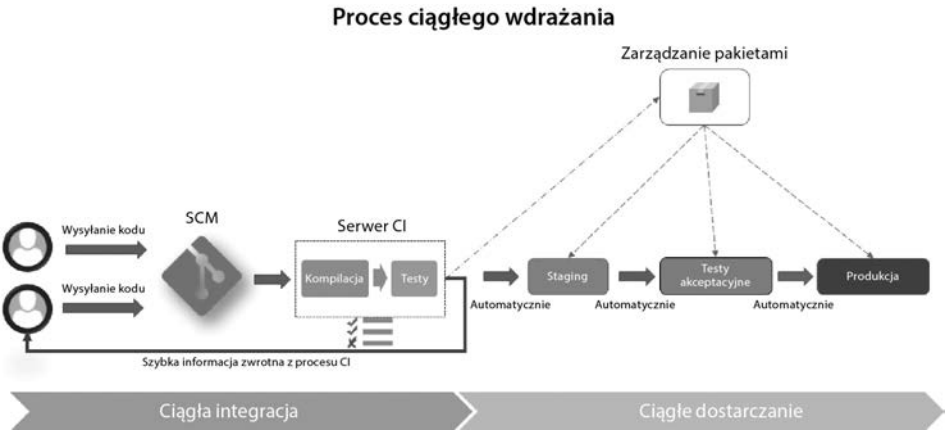
Ciągłe wdrażanie jest rozszerzeniem procesu CD. Automatyzuje cały potok CI/CD od momentu, gdy programista zatwierdzi swój kod, do wdrożenia w środowisku produkcyjnym, przez wszystkie etapy weryfikacji.

Taka praktyka jest rzadko wdrażana w przedsiębiorstwach, ponieważ wymaga wykonania różnych testów (jednostkowych, funkcjonalnych, integracyjnych, wydajnościowych itd.) dla aplikacji. Pomyślne wykonanie tych testów jest wystarczające do sprawdzenia poprawności działania aplikacji pod kątem wszystkich tych zależności. Pozwala również na automatyczne wdrażanie w środowisku produkcyjnym bez konieczności wykonywania jakichkolwiek czynności zatwierdzania.

Ciągły proces wdrażania musi również uwzględniać wszystkie etapy przywracania aplikacji w razie wystąpienia problemu produkcyjnego.

Ciągłe wdrażanie można zastosować przy użyciu i implementacji technik przełączników (ang. *feature toggle*), co obejmuje hermetyzację funkcji aplikacji i aktywowanie danej funkcjonalności na żądanie, bezpośrednio w środowisku produkcyjnym, bez konieczności ponownego wdrażania kodu aplikacji.

Inną techniką jest wykorzystanie niebiesko-zielonej infrastruktury produkcyjnej, która składa się z dwóch środowisk produkcyjnych: jednego niebieskiego i jednego zielonego. Najpierw wdrażamy się do niebieskiego środowiska, a następnie do zielonego; zapewni to, że nie będą wymagane żadne przestoje.



Rysunek 1.4. Proces ciągłego wdrażania

Uwaga

Przełącznikom i niebiesko-zielonemu wykorzystaniu wdrożeń bardziej szczegółowo przyjrzymy się w rozdziale 15., „Skrócenie czasu przestoju wdrażania".

Powyższy diagram jest prawie taki sam jak w przypadku CD, ale z tą różnicą, że przedstawia automatyczne, kompleksowe wdrożenie.

Procesy CI/CD są zatem istotną częścią kultury DevOps, przy czym CI pozwala zespołom integrować i testować spójność kodu oraz regularnie uzyskiwać szybkie informacje zwrotne. CD automatycznie wdraża się w jednym lub kilku środowiskach staging, dzięki czemu oferuje możliwość przetestowania całej aplikacji, dopóki nie zostanie ona wdrożona w środowisku produkcyjnym.

Wreszcie ciągłe wdrażanie automatyzuje możliwość wdrażania aplikacji od zatwierdzenia do środowiska produkcyjnego.

Uwaga

W rozdziale 7., „Ciągła integracja i ciągłe wdrażanie", dowiemy się, jak wdrożyć wszystkie te procesy we współpracy z narzędziami Jenkins, Azure DevOps i GitLab CI.

W tej sekcji omówiliśmy praktyki, które są niezbędne dla kultury DevOps, czyli ciągłą integrację, ciągłe dostarczanie i ciągłe wdrażanie.

W następnej sekcji przyjrzymy się innej praktyce DevOps, znanej jako IaC.

Zrozumienie praktyk IaC

IaC to praktyka polegająca na pisaniu kodu zasobów składających się na infrastrukturę.

Praktyka ta zaczęła przynosić efekty wraz z rozwojem kultury DevOps i modernizacją infrastruktury chmury. Rzeczywiście, zespoły operacyjne, które ręcznie wdrażają infrastrukturę, poświęcają czas na wprowadzenie zmian w infrastrukturze ze względu na niespójną obsługę i ryzyko błędów. Ponadto, wraz z modernizacją chmury i jej skalowalnością, sposób budowania infrastruktury wymaga przeglądu wyposażenia (ang. *provisioning*) i praktyki zmian poprzez dostosowanie bardziej zautomatyzowanej metody.

IaC to proces pisania kodu etapów udostępniania i konfiguracji komponentów infrastruktury, który pomaga zautomatyzować jej wdrażanie w powtarzalny i spójny sposób.

Zanim przyjrzymy się korzystaniu z IaC, zobaczymy, jakie są korzyści płynące z tej praktyki.

Korzyści IaC

Korzyści płynące z IaC są następujące:

- Standaryzacja konfiguracji infrastruktury zmniejsza ryzyko błędów.
- Kod opisujący infrastrukturę jest wersjonowany i kontrolowany w menedżerze kodu źródłowego.
- Kod jest zintegrowany z potokami CI/CD.
- Wdrożenia, które wprowadzają zmiany w infrastrukturze, są szybsze i bardziej wydajne.
- Lepsze zarządzanie, kontrola i redukcja kosztów infrastruktury.

IaC przynosi również korzyści zespołowi DevOps, umożliwiając Ops większą wydajność w zakresie zadań związanych z ulepszaniem infrastruktury zamiast poświęcania czasu na ręczną konfigurację. Daje również Dev możliwość ulepszania swojej infrastruktury i wprowadzania zmian bez konieczności proszenia o więcej zasobów operacyjnych.

IaC umożliwia też tworzenie samoobsługowych, efemerycznych środowisk, które zapewnią programistom i testerom większą elastyczność w testowaniu nowych funkcji w izolacji i niezależnie od innych środowisk.

Języki i narzędzia IaC

Języki i narzędzia używane do napisania konfiguracji infrastruktury mogą być różnych typów: **skryptowe**, **deklaratywne** i **programowe**. Zbadamy je w kolejnych sekcjach.

Typy skryptowe

Zaliczają się do nich języki skryptowe, takie jak Bash, PowerShell lub inne, które korzystają z różnych klientów (SDK) dostarczanych przez dostawcę chmury. Na przykład możesz uzyskać dostęp do zasobów platformy Azure za pomocą interfejsu wiersza polecenia platformy Azure lub programu Azure PowerShell.

Oto polecenie, które tworzy grupę zasobów na platformie Azure:

- Korzystając z interfejsu Azure CLI (dokumentacja jest dostępna pod adresem *https://bit.ly/2V1OfxJ*):

  ```
  az group create --location westeurope --resource-group MyAppResourcegroup
  ```

- Korzystając z Azure PowerShell (dokumentacja jest dostępna pod adresem *https://bit.ly/2VcASeh*):

  ```
  New-AzResourceGroup -Name MyAppResourcegroup -Location westeurope
  ```

Problem z tymi językami i narzędziami polega na tym, że wymagają one dużej ilości linijek kodu. Dzieje się tak, ponieważ musimy zarządzać różnymi stanami manipulowanych zasobów i konieczne jest napisanie wszystkich kroków tworzenia lub aktualizowania pożądanej infrastruktury.

Jednak te języki i narzędzia mogą być bardzo przydatne w wypadku zadań, które automatyzują powtarzalne czynności wykonywane na liście zasobów (wybór elementu i zapytania) lub które wymagają złożonego przetwarzania z pewną logiką do wykonania na zasobach infrastruktury, np. skrypt automatyzujący maszyny wirtualne, które mają usuwany określony tag.

Typy deklaratywne

Zaliczają się do nich języki, w których wystarczy zapisać pożądany stan systemu lub infrastruktury w postaci konfiguracji i właściwości. Tak jest np. w przypadku narzędzi Terraform i Vagrant firmy HashiCorp, Ansible, szablonu Azure ARM, Azure Bicep (*https://learn.microsoft.com/en-us/azure/azure-resource-manager/bicep/overview?tabs=bicep*), PowerShell DSC, Puppet i Chef. Wszystko, co użytkownik musi zrobić, to opisać ostateczny stan pożądanej infrastruktury; narzędzie zadba o jego zastosowanie.

Na przykład poniższy kod Terraform umożliwia zdefiniowanie żądanej konfiguracji grupy zasobów platformy Azure:

```
resource "azurerm_resource_group" "myrg" {
    name = "MyAppResourceGroup"
    location = "West Europe"

    tags = {
        environment = "Bookdemo"
    }
}
```

W tym przykładzie, jeśli chcesz dodać lub zmodyfikować tag, po prostu zmodyfikuj właściwość `tags` w poprzednim kodzie, a Terraform sam wykona aktualizację.

Oto kolejny przykład, który pozwala zainstalować i uruchomić usługę nginx na serwerze za pomocą Ansible:

```
---
- hosts: all
 tasks:
 - name: install and check nginx latest version
 apt: name=nginx state=latest
 - name: start nginx
 service:+
 name: nginx
 state: started
```

Aby się upewnić, że usługa nie jest zainstalowana, po prostu zmień poprzedni kod dla elementu `service` — ustaw wartość `stopped` dla właściwości `state`:

```
---
- hosts: all
 tasks:
 - name: stop nginx
 service:
 name: nginx
 state: stopped
 - name: check nginx is not installed
 apt: name=nginx state=absent
```

W tym przykładzie wystarczyło zmienić właściwość `state`, aby wskazać pożądany stan usługi.

> **Uwaga**
>
> Aby uzyskać szczegółowe informacje na temat korzystania z Terraform i Ansible, zobacz rozdział 2., „Udostępnianie infrastruktury chmury za pomocą Terraform", i rozdział 3., „Używanie Ansible do konfigurowania infrastruktury IaaS".

Typy programowe

Od kilku lat obserwuje się, że dwa rodzaje kodu IaC, czyli języki skryptowe lub deklaratywne, używane są przez zespół **operacyjny**. Zwykle nie dotyczy to programistów.

W celu zacieśnienia więzi między programistami a operacjami widzimy pojawianie się narzędzi IaC, które są bardziej oparte na językach znanych programistom, takich jak TypeScript, Java, Python i C#.

Wśród narzędzi IaC, które pozwalają nam udostępniać infrastrukturę za pomocą języka programowania, mamy **Pulumi** (*https://www.pulumi.com/*) i **Terraform CDK** (*https://github.com/hashicorp/terraform-cdk*).

Poniżej znajduje się przykład kodu TypeScriptu napisanego za pomocą Terraform CDK:

```
import { Construct } from 'constructs';
import { App, TerraformStack, TerraformOutput } from 'cdktf';
import {
  ResourceGroup,
} from './.gen/providers/azurerm';
class AzureRgCDK extends TerraformStack {
  constructor(scope: Construct, name: string) {
    super(scope, name);
    new AzurermProvider(this, 'azureFeature', {
      features: [{}],
    });
    const rg = new ResourceGroup(this, 'cdktf-rg', {
```

```
      name: 'MyAppResourceGroup',
      location: 'West Europe',
    });
  }
}
const app = new App();
new AzureRgCDK(app, 'azure-rg-demo');
app.synth();
```

W tym przykładzie, napisanym w języku TypeScript, używamy bibliotek dwuwarstwowych: pakietu `npm` i Terraform CDK o nazwie `cdktf`. Pakiet `npm` używany do udostępniania zasobów platformy Azure nosi nazwę `'gen/providers/azurerm'`.

Następnie deklarujemy nową klasę, która inicjuje dostawcę platformy Azure, i definiujemy tworzenie grupy zasobów za pomocą metody `new ResourceGroup`.

Na koniec, aby utworzyć grupę zasobów, tworzymy wystąpienie tej klasy i wywołujemy metodę `app.synth` zestawu CDK.

Uwaga

Aby uzyskać więcej informacji na temat Terraform CDK, proponuję przeczytać następujące wpisy na blogu i obejrzeć następujący film:

https://www.hashicorp.com/blog/cdk-for-terraform-enabling-python-and-typescript-support,

https://www.hashicorp.com/blog/announcing-cdk-for-terraform-0-1,

https://www.youtube.com/watch?v=5hSdb0nadRQ.

Topologia IaC

W infrastrukturze chmurowej IaC dzieli się na kilka typologii:

- wdrażanie i udostępnianie infrastruktury,
- konfiguracja serwera i tworzenie szablonów,
- konteneryzacja,
- konfiguracja i wdrożenie w Kubernetesie.

Przyjrzymy się szerzej każdej z nich.

Wdrażanie i udostępnianie infrastruktury

Udostępnianie to czynność tworzenia instancji zasobów tworzących infrastrukturę. Mogą to być zasoby typu **platforma jako usługa** (ang. *Platform-as-a-Service* — PaaS) i bezserwerowe, takie jak aplikacja internetowa, funkcja platformy Azure, Event Hub, ale także cała zarządzana część sieci, taka jak sieć wirtualna, podsieci, tabele routingu

lub Azure Firewall. W przypadku zasobów maszyn wirtualnych proces udostępniania tworzy lub aktualizuje tylko zasób w chmurze maszyny wirtualnej, ale nie jego zawartość.

Dostępnych jest wiele narzędzi do udostępniania, takich jak Terraform, szablon ARM, Azure CLI, Azure PowerShell, a także Google Cloud Deployment Manager. Oczywiście jest ich znacznie więcej, ale trudno je wszystkie wymienić. W tej książce przyjrzymy się szczegółowo wykorzystaniu Terraform.

Konfiguracja serwera

Ten krok dotyczy konfigurowania maszyn wirtualnych, takich jak utwardzenie (ang. *hardening*), montowanie dysków, konfiguracja sieci (zapora ogniowa, proxy itd.) oraz instalacja middleware.

Istnieją różne narzędzia konfiguracyjne, takie jak Ansible, PowerShell DSC, Chef, Puppet i SaltStack. Oczywiście jest ich znacznie więcej, ale w tej książce szczegółowo przyjrzymy się wykorzystaniu Ansible w celu konfiguracji maszyny wirtualnej.

Aby zoptymalizować czas udostępniania i konfiguracji serwera, można tworzyć modele serwerów, zwanych obrazami, i ich używać. Zawierają one całą konfigurację (hardening, middleware itd.) serwerów. Podczas przygotowywania serwera wskażemy odpowiadający mu szablon. Tak więc za kilka minut będziemy mieli serwer, który został skonfigurowany i jest gotowy do użycia.

Istnieje również wiele narzędzi IaC do tworzenia szablonów serwerów, np. **Aminator** (używany przez Netfliksa) i **HashiCorp Packer**.

Oto przykład kodu Packera służącego do tworzenia obrazu Ubuntu z aktualizacjami pakietów:

```
{
"builders": [{
    "type": "azure-arm",
    "os_type": "Linux",
    "image_publisher": "Canonical",
    "image_offer": "UbuntuServer",
    "image_sku": "16.04-LTS",
    "managed_image_resource_group_name": "demoBook",
    "managed_image_name": "SampleUbuntuImage",
    "location": "West Europe",
    "vm_size": "Standard_DS2_v2"
}],
"provisioners": [{
    "execute_command": "chmod +x {{ .Path }}; {{ .Vars }} sudo -E sh
    ↪'{{ .Path }}'",
```

```
    "inline": [
    "apt-get update",
    "apt-get upgrade -y",
    "/usr/sbin/waagent -force -deprovision+user && export
    HISTSIZE=0 && sync"
 ],
 "inline_shebang": "/bin/sh -x",
 "type": "shell"
 }]
}
```

Ten skrypt tworzy obraz szablonu dla maszyny wirtualnej `Standard_DS2_V2` na podstawie systemu operacyjnego Ubuntu (sekcja `builders`). Dodatkowo podczas tworzenia obrazu Packer aktualizuje wszystkie pakiety za pomocą polecenia `apt-get update`.

Następnie Packer wyrejestruje obraz, aby usunąć wszystkie informacje o użytkowniku (sekcja `provisioners`).

Uwaga

Packer zostanie szczegółowo omówiony w rozdziale 4., „Optymalizacja wdrażania infrastruktury za pomocą Packera".

Niezmienna infrastruktura z kontenerami

Konteneryzacja polega na wdrażaniu aplikacji w kontenerach zamiast na maszynach wirtualnych.

Obecnie najpowszechniejszym oprogramowaniem służącym do konteneryzacji jest **Docker**. Obraz Dockera jest konfigurowany za pomocą kodu w pliku *Dockerfile*. Plik ten zawiera deklarację obrazu bazowego, który reprezentuje używany system operacyjny, dodatkowe oprogramowanie middleware, pliki i binaria niezbędne dla aplikacji oraz konfigurację sieciową portów. W przeciwieństwie do maszyn wirtualnych, kontenery są uważane za niezmienne; konfiguracja kontenera nie może być modyfikowana podczas jego wykonywania.

Oto prosty przykład pliku *Dockerfile*:

```
FROM ubuntu
RUN apt-get update
RUN apt-get install -y nginx
ENTRYPOINT ["/usr/sbin/nginx","-g","daemon off;"]
EXPOSE 80
```

W tym obrazie Dockera używamy podstawowego obrazu Ubuntu, instalujemy `nginx` i udostępniamy port `80`.

Uwaga

Część dotycząca platformy Docker zostanie szczegółowo omówiona w rozdziale 9., „Konteneryzacja aplikacji za pomocą Dockera".

Konfiguracja i wdrożenie w Kubernetesie

Kubernetes jest orkiestratorem kontenerów — to technologia, która w większości uosabia IaC (moim zdaniem) ze względu na sposób wdrażania kontenerów, architekturę sieci (równoważenie obciążenia, porty itd.) i zarządzanie woluminami. Chroni także poufne informacje, które są zapisane w plikach specyfikacji YAML.

Oto prosty przykład pliku specyfikacji YAML:

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: nginx-demo
    labels:
        app: nginx
spec:
    replicas: 2
    selector:
        matchLabels:
            app: nginx
template:
metadata:
    labels:
        app: nginx
spec:
    containers:
    - name: nginx
        image: nginx:1.7.9
ports:
- containerPort: 80
```

W poprzednim pliku specyfikacji możemy zobaczyć nazwę obrazu do wdrożenia (`ngnix`), port do otwarcia (`80`) i liczbę replik (`2`).

Uwaga

Część dotycząca Kubernetesa zostanie szczegółowo omówiona w rozdziale 10., „Efektywne zarządzanie kontenerami za pomocą Kubernetesa".

IaC, podobnie jak tworzenie oprogramowania, wymaga od nas wdrożenia praktyk i procesów, które umożliwiają ewolucję i utrzymanie kodu infrastruktury.

Wśród tych praktyk są te związane z tworzeniem oprogramowania:

- Stosuj dobre zasady nazewnictwa.
- Nie przeciążaj kodu niepotrzebnymi komentarzami.
- Używaj małych funkcji.
- Zaimplementuj obsługę błędów.

Uwaga

Aby dowiedzieć się więcej o dobrych praktykach tworzenia oprogramowania, przeczytaj doskonałą książkę związaną z tym tematem, *Clean Code* Roberta Martina.

Są jeszcze inne konkretne praktyki, które moim zdaniem zasługują na więcej uwagi:

- **Kod musi być w pełni zautomatyzowany.** Podczas wykonywania IaC należy zakodować i zautomatyzować wszystkie etapy udostępniania i nie należy pozostawiać ręcznych działań poza kodem, które zakłócają automatyzację infrastruktury — co może generować błędy. A jeśli to konieczne, nie wahaj się użyć kilku narzędzi, takich jak Terraform i Bash, ze skryptami Azure CLI.
- **Kod musi się znajdować w menedżerze kodu źródłowego.** Kod infrastruktury musi również znajdować się w SCM, aby można go było wersjonować, śledzić, scalać i przywracać, a tym samym zapewniać lepszą widoczność kodu między programistami i zespołem operacyjnym.
- **Kod infrastruktury musi być trzymany z kodem aplikacji.** W niektórych przypadkach może to być trudne, ale jeśli to możliwe, znacznie lepiej jest umieścić kod infrastruktury w tym samym repozytorium co kod aplikacji. Ma to na celu zapewnienie lepszej organizacji pracy między programistami i operatorami, którzy będą dzielić ten sam obszar roboczy.
- **Oddzielenie ról i katalogów.** Dobrze jest oddzielić kod od infrastruktury — zgodnie z rolą kodu. Umożliwia to utworzenie jednego katalogu do udostępniania i konfigurowania maszyn wirtualnych i innego, który będzie zawierał kod do testowania integracji całej infrastruktury.
- **Integracja z procesem CI/CD.** Jednym z celów IaC jest możliwość automatyzacji wdrażania infrastruktury. Tak więc od początku jego implementacji konieczne jest utworzenie procesu CI/CD, który zintegruje kod, przetestuje go i wdroży w różnych środowiskach. Niektóre narzędzia, takie jak Terratest, umożliwiają tworzenie testów dla kodu infrastruktury. Jedną z najlepszych praktyk jest integracja procesu CI/CD infrastruktury w tym samym potoku co aplikacja.
- **Kod musi być idempotentny.** Wykonanie kodu wdrażania infrastruktury musi być idempotentne — czyli kod powinien być ciągle automatycznie wykonywalny. Oznacza to, że skrypty muszą uwzględniać stan infrastruktury

podczas jej uruchamiania i nie generować błędu, jeśli zasób do utworzenia już istnieje lub zasób do usunięcia został już usunięty. Zobaczymy, że języki deklaratywne, takie jak Terraform, natywnie przyjmują ten aspekt idempotencji. Kod infrastruktury raz w pełni zautomatyzowany musi umożliwiać budowę i niszczenie infrastruktury aplikacji.

- **Kod jako dokumentacja.** Kod infrastruktury musi być jasny i musi służyć jako dokumentacja. Dokumentacja infrastruktury zajmuje dużo czasu i w wielu przypadkach nie jest aktualizowana w miarę rozwoju infrastruktury.
- **Kod musi być modułowy.** W infrastrukturze komponenty często mają ten sam kod — jedyną różnicą jest wartość ich właściwości. Również te komponenty są kilkakrotnie wykorzystywane w aplikacjach firmy. Dlatego ważne jest, aby zoptymalizować czasy pisania kodu przez podzielenie go na moduły (lub role w przypadku Ansible), które będą wywoływane jako funkcje. Kolejną zaletą korzystania z modułów jest możliwość standaryzacji nazewnictwa zasobów i zgodność niektórych właściwości.
- **Posiadanie środowiska programistycznego.** Problem z IaC polega na tym, że trudno jest testować kod infrastruktury będącej w fazie rozwoju w środowiskach używanych do integracji, a także testować aplikację, ponieważ zmiana infrastruktury może wpływać na kod. Dlatego ważne jest, aby mieć środowisko programistyczne nawet dla IaC, które może w każdej chwili ulec uszkodzeniu lub nawet zostać zniszczone.

W przypadku testów infrastruktury lokalnej niektóre narzędzia symulują środowisko lokalne, np. Vagrant (od HashiCorp), więc należy ich używać do testowania skryptów kodu tak często, jak to możliwe.

Oczywiście pełna lista dobrych praktyk jest dłuższa; również wszystkie metody i procesy praktyk inżynierii oprogramowania mają zastosowanie.

Dlatego IaC, podobnie jak procesy CI/CD, jest kluczową praktyką **kultury DevOps**, która umożliwia wdrażanie i konfigurowanie infrastruktury poprzez pisanie kodu. Jednak IaC może być skuteczna tylko przy użyciu odpowiednich narzędzi i wdrażaniu dobrych praktyk.

W tej sekcji zawarliśmy omówienie niektórych najlepszych praktyk DevOps. Teraz przedstawimy krótki przegląd ewolucji kultury DevOps.

Ewolucja kultury DevOps

Z czasem i doświadczeniem zdobytym przy stosowaniu kultury DevOps możemy zaobserwować ewolucję praktyk, a także zespołów pragnących dołączyć do tego ruchu.

Tak jest np. w przypadku praktyki **GitOps**, która coraz częściej zaczyna się pojawiać w firmach.

Przepływ pracy GitOps, który jest powszechnie stosowany w Kubernetesie, polega na użyciu Gita jako jedynego źródła prawdy; oznacza to, że repozytorium Gita zawiera kod stanu infrastruktury, a także kod aplikacji do wdrożenia.

Kontroler będzie nadzorował pobieranie źródła Gita podczas zatwierdzania kodu, wykonywania testów i ponownego wdrażania aplikacji.

Uwaga

Aby uzyskać więcej informacji o kulturze, praktykach i przepływach pracy GitOps, przeczytaj oficjalny przewodnik po GitOps: *https://www.weave.works/technologies/gitops/*.

Podsumowanie

W tym rozdziale zobaczyliśmy, że kultura DevOps to złożenie współpracy, procesów i narzędzi. Następnie szczegółowo opisaliśmy poszczególne etapy procesu CI/CD i wyjaśniliśmy różnicę między ciągłą integracją, ciągłym dostarczaniem i ciągłym wdrażaniem. W ostatniej części wyjaśniliśmy, jak korzystać z IaC z jej najlepszymi praktykami, a także omówiliśmy ewolucję kultury DevOps.

Poznaliśmy podstawy kultury DevOps i jej praktyk, co nadaje ton pozostałym rozdziałom tej książki, w których omówimy, jak zastosować tę kulturę za pomocą narzędzi i praktyk.

W następnym rozdziale zaczniemy od omówienia implementacji procesu *infrastruktury jako kodu* i sposobu dostarczania infrastruktury za pomocą Terraform.

Pytania

1. Od jakich słów pochodzi skrót DevOps?
2. Czy DevOps to termin, który reprezentuje nazwę narzędzia, kulturę, społeczeństwo czy tytuł książki?
3. Jakie są trzy założenia kultury DevOps?
4. Jaki jest cel ciągłej integracji?
5. Jaka jest różnica między ciągłym dostarczaniem a ciągłym wdrażaniem?
6. Co oznacza IaC?

Dalsza lektura

Jeśli chcesz dowiedzieć się więcej o kulturze DevOps, oto kilka zasobów:

- Centrum zasobów DevOps (zasoby firmy Microsoft) — *https://docs.microsoft.com/en-us/azure/devops/learn/*.
- Raport na temat DevOps z 2020 r. (zasoby Puppeta) — *https://puppet.com/resources/report/2020-state-of-devops-report*.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

T

U

V

W

Y

Z

Ż

Potrzebujesz efektywności? Praktykuj DevOps i wygrywaj na rynku!

DevOps jest doskonałym rozwiązaniem dla każdej organizacji, która musi zwiększyć przepływ pracy technicznej przy zachowaniu odpowiedniej jakości i niezawodności. Pozwala też na uzyskanie trwałości projektów i wzorową współpracę programistów z zespołem operacyjnym. Wiele organizacji decyduje się na wdrożenie praktyk DevOps. Pomyślne przeprowadzenie tego procesu wymaga przygotowań, w ich ramach zaś kluczowe znaczenie ma wybór odpowiednich do potrzeb wzorców i narzędzi.

To drugie, zaktualizowane i uzupełnione wydanie książki poświęconej wdrażaniu najlepszych praktyk DevOps przy użyciu nowoczesnych narzędzi. Przedstawiono w niej informacje o kulturze DevOps, opisano różne narzędzia i techniki stosowane do jej wdrażania, takie jak IaC, potoki Git i CI/CD, a także automatyzację testów i analizę kodu. Sporo miejsca poświęcono konteneryzacji aplikacji za pomocą Dockera i platformy Kubernetes. Znajdziemy tutaj również kwestię skracania przestojów podczas wdrażania oprogramowania i omówienie możliwości stosowania praktyk DevOps w projektach open source. Warto zwrócić uwagę na ostatni rozdział, w którym pokazano zasady wdrażania niektórych praktyk DevOps w całym cyklu życia projektów.

Najciekawsze zagadnienia:

- infrastruktura jako kod (IaC)
- udostępnianie i konfigurowanie infrastruktury chmurowej
- tworzenie lokalnego środowiska programistycznego i konteneryzowanie aplikacji
- zastosowanie DevSecOps do testowania zgodności i zabezpieczania infrastruktury
- potoki DevOps CI/CD i zielononiebieskie praktyki wdrażania
- praktyki DevOps dla projektów open source

Mikael Krief jest inżynierem DevOps, autorem książek technicznych i blogerem. Angażuje się w wiele różnych projektów, często występuje na prestiżowych konferencjach. Specjalizuje się w stosowaniu Terraform. Od lat rokrocznie otrzymuje tytuł Microsoft MVP, a od 2020 roku jest wybierany na ambasadora HashiCorp.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-8322-198-4

Cena: 109,00 zł

Packt