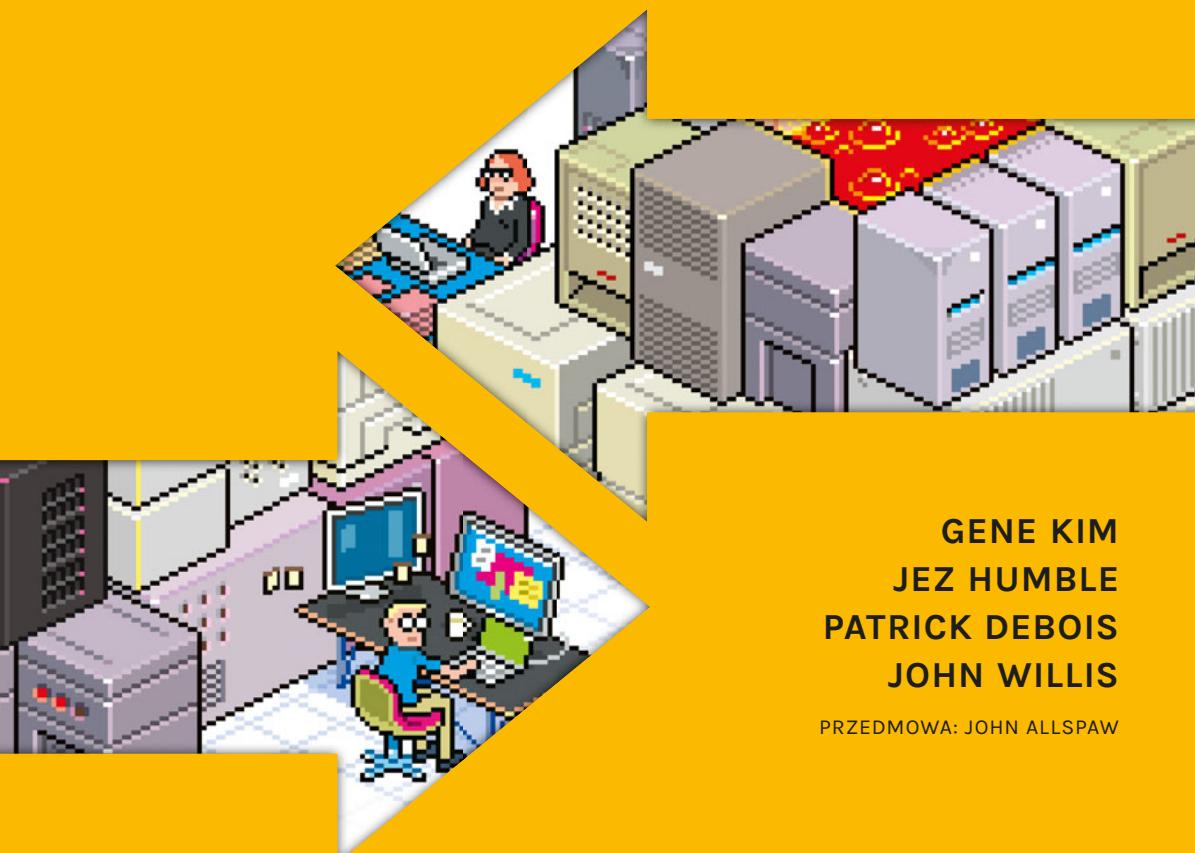


DevOps

ŚWIATOWEJ KLASY
ZWINNOŚĆ, NIEZAWODNOŚĆ
I BEZPIECZEŃSTWO
W TWOJEJ ORGANIZACJI



GENE KIM
JEZ HUMBLE
PATRICK DEBOIS
JOHN WILLIS

PRZEDMOWA: JOHN ALLSPAW

onepress

Helion

Tytuł oryginału: The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations

Tłumaczenie: Radosław Meryk

ISBN: 978-83-283-3454-0

Copyright © 2016 by Gene Kim, Jez Humble, Patrick Debois, and John Willis
All rights reserved

Polish edition copyright © 2017 by Helion SA
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione.
Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicielami.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Autor studium przypadku zespołu 18F na stronie 355 przekazał swoje dzieło do domeny publicznej zrzekając się praw autorskich oraz praw zależnych do dzieła na terenie całego świata, w zakresie dopuszczonym przez prawo.
Można kopiować, modyfikować oraz rozpowszechniać i wykonywać studium przypadku zespołu 18F, także do celów komercyjnych, bez występowania o zgodę.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

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

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

SPIS TREŚCI

Przedmowa. Aha!	7
Słowo wstępne	15
Wyobraź sobie świat, w którym Dev i Ops tworzą DevOps.	
Wprowadzenie do podręcznika DevOps	17
CZĘŚĆ I. TRZY DROGI	33
1. Agile, ciągłe dostarczanie i trzy drogi	39
2. Pierwsza droga: Zasady przepływu.....	47
3. Druga droga: Zasady sprzężenia zwrotnego	59
4. Trzecia droga: Zasady ciągłego uczenia się i eksperymentowania.....	69
CZĘŚĆ II. OD CZEGO ZACZĄĆ?	79
5. Wybór strumieni wartości, od których należy zacząć	83
6. Zrozumienie pracy w strumieniu wartości, zapewnienie jej widoczności i rozszerzenie zrozumienia na całą organizację.....	93
7. Projektowanie organizacji i jej architektury z uwzględnieniem praw Conwaya	107
8. Jak uzyskać świetne efekty poprzez zintegrowanie zadań działu Ops z codzienną pracą działu Dev?	125

CZĘŚĆ III. PIERWSZA DROGA	
<i>TECHNICZNE PRAKTYKI PRZEPŁYWU</i>	137
9. Podstawy potoku wdrożeń	141
10. Szybkie i niezawodne testowanie automatyczne	153
11. Wdrożenie i stosowanie praktyk ciągłej integracji.....	173
12. Automatyzacja i zapewnienie wydań niskiego ryzyka.....	183
13. Architektura dla wydań niskiego ryzyka	209
CZĘŚĆ IV. DRUGA DROGA	
<i>TECHNICZNE PRAKTYKI SPRZĘŻEŃ ZWROTNYCH</i>	221
14. Tworzenie telemetrii umożliwiające dostrzeganie i rozwiązywanie problemów.....	225
15. Analizowanie telemetrii w celu lepszego przewidywania problemów i realizowania zadań	245
16. Sprzężenia zwrotne poprawiają bezpieczeństwo wdrażania kodu przez zespoły Dev i Ops.....	259
17. Integracja technik wytwarzania oprogramowania sterowanego hipotezami i testowania A/B w codziennej pracy	273
18. Tworzenie procesów przeglądu i koordynacji w celu poprawy jakości bieżącej pracy	281
CZĘŚĆ V. TRZECIA DROGA	
<i>TECHNICZNE PRAKTYKI CIĄGŁEGO UCZENIA SIĘ I EKSPERYMENTOWANIA</i>	299
19. Stworzenie warunków do uczenia się podczas codziennej pracy	303
20. Konwersja lokalnych odkryć w globalne usprawnienia	317
21. Zarezerwuj czas na stworzenie organizacyjnego systemu uczenia się i doskonalenia.....	329
CZĘŚĆ VI. ZARZĄDZANIE ZMIANAMI	
<i>I ZAPEWNIEŃIE ZGODNOŚCI Z PRZEPISAMI</i>	339
22. Bezpieczeństwo informacji jako codzienne zadanie każdego z nas	343
23. Ochrona potoku wdrożeń	363
Wezwanie do działania. Podsumowanie podręcznika DevOps	377

MATERIAŁY DODATKOWE	381
Dodatki.....	383
Zasoby dodatkowe.....	397
Przypisy końcowe	401
Skorowidz.....	437
Podziękowania.....	445
Biogramy autorów.....	449

PRZEDMOWA

Aha!

Podróż, której celem było ukończenie książki *DevOps. Światowej klasy zwinnosć, niezawodnośc i bezpieczeństwo w Twojej organizacji*, była długa. Zaczęło się w lutym 2011 r. od przeprowadzanych raz w tygodniu na Skypie rozmów pomiędzy współautorami. Mieliśmy wizję stworzenia przewodnika, który będzie służył jako lektura uzupełniająca do wtedy niedokończonej jeszcze książki *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win**

Ponad pięć lat później, po wykonaniu pracy, którą oceniamy na ponad 2000 godzin, książka *DevOps. Światowej klasy zwinnosć, niezawodnośc i bezpieczeństwo w Twojej organizacji* ostatecznie stała się faktem. Jej pisanie było niezwykle długotrwałym procesem, chociaż przyniosł on wiele satysfakcji, pozwolił się dużo nauczyć, a ostateczny zakres publikacji jest znacznie szerszy, niż pierwotnie przewidywaliśmy. Podczas trwania projektu wszystkich współautorów łączyło przekonanie, że idea DevOps jest naprawdę ważna. Tę myśl sformowała osobista chwila „aha”, która nastąpiła dużo wcześniej w karierze zawodowej każdego z nas. Podejrzewam, że wielu naszych Czytelników także jej doświadczyło.

* Wydanie polskie: *Projekt Feniks. Powieść o IT, modelu DevOps i o tym, jak pomóc firmie w odniesieniu sukcesu*, Helion 2016.

Gene Kim

Miałem przywilej studiowania wysokowydajnych organizacji technicznych od 1999 roku. Jednym z pierwszych moich odkryć było to, że kluczem do sukcesu jest przekraczanie granic pomiędzy różnymi grupami funkcjonalnymi: operacji IT (Ops), bezpieczeństwa informacji (Infosec) i rozwoju oprogramowania (Dev). Wciąż jednak pamiętam chwilę, w której zaobserwowałem po raz pierwszy skalę upadku wynikającego z działania tych funkcji w kierunku przeciwnych celów.

To był rok 2006. Miałem okazję spędzić tydzień z grupą osób zarządzających firmą dostarczającą usługi Ops, działającą na zlecenie dużej firmy świadczącej usługi rezerwacji lotniczych. Osoby te opisywały ujemne konsekwencje dostarczania rozbudowanych, corocznych wydań oprogramowania. Każde wydanie powodowało ogromny chaos i zakłócenia zarówno dla wydawcy, jak i klientów. Zdarzały się kary wynikające z naruszeń umów **SLA** (ang. *service level agreement* — dosł. „umowa zapewnienia poziomu usług”) z powodu wpływających na klienta przestojów. Były zwolnienia najbardziej utalentowanych i doświadczonych pracowników ze względu na krótkoterminowe spadki zysków. Było dużo niezaplanowanej pracy i „gaszenia pożarów”, do tego stopnia, że pozostały personel nie był w stanie sprostać ciągle rosnącej liczbie zaległych zleceń usług od klientów. Kontrakt był kontynuowany tylko dzięki heroizmowi menedżerów średniego szczebla i wszyscy czuli, że będzie poddany ponownemu przetargowi w okresie trzech lat.

Powstałe poczucie beznadziejności i bezsilności zainspirowało mnie do rozpoczęcia moralnej krucjaty. Zadania rozwoju oprogramowania zawsze były postrzegane jako strategiczne, natomiast operacje IT jako taktyczne, często delegowane lub w całości powierzane podmiotom zewnętrznym. A po pięciu latach okazywało się, że są one w jeszcze gorszym stanie niż w momencie ich przekazania.

Przez długie lata wielu z nas przeczuwało, że musi istnieć lepszy sposób. Pamiętam rozmowy prowadzone na konferencji Velocity Conference 2009, gdy opisywano niesamowite wyniki uzyskiwane dzięki architekturze, rozwiązaniom technicznym i normom kulturowym znany dziś jako DevOps. Byłem nimi podekscytowany, ponieważ wyraźnie wskazywały na lepszy sposób, którego wszyscy szukaliśmy. Pomoc w rozpowszechnianiu tego poglądu była jedną z moich osobistych motywacji do bycia współautorem książki *The Phoenix Project*. Łatwo sobie wyobrazić, jak bardzo satysfakcjonujące było doświadczenie reakcji szerszej społeczności na tę książkę — wypowiedzi, jak bardzo pomogła ludziom doświadczyć ich własnych momentów „aha”.

Jeż Humble

Swój moment „aha” w związku z DevOps przeżyłem w 2000 roku, pracując w *startupie*. Była to moja pierwsza posada po ukończeniu studiów. Przez jakiś czas byłem jednym z dwóch pracowników technicznych. Robiłem wszystko: sieci, programowanie, wsparcie, administrację systemami. Wdrażaliśmy oprogramowanie do produkcji przez FTP — bezpośrednio z naszych stacji roboczych.

Potem, w 2004 r., dostałem pracę w ThoughtWorks — firmie doradczej, gdzie w początkowym okresie byłem zaangażowany w projekt, w którym pracowało około 70 osób. Byłem w zespole ośmiu inżynierów, których zadaniem było wdrażanie naszego oprogramowania w środowisku przypominającym produkcyjne. Na początku to było bardzo stresujące. Ale w ciągu kilku miesięcy przeszliśmy od wdrożeń ręcznych, które zajmowały dwa tygodnie, do instalacji automatycznych, trwających godzinę, gdzie błyskawicznie mogliśmy przesuwać proces do przodu i wstecz, stosując wzorzec instalacji niebieski-zielony. Co ważne, robiliśmy to w normalnych godzinach pracy.

Ten projekt stał się inspiracją dla wielu pomysłów — zarówno dla książki *Continuous Delivery* (Addison-Wesley, 2000), jak i dla tej pozycji. Bardzo wiele inspiracji dla mnie oraz innych osób pracujących w tej dziedzinie wywodzi się z przekonania, że niezależnie od ograniczeń zawsze można robić coś lepiej, oraz z dążenia do pomagania ludziom w ich podróży.

Patrick Debois

Dla mnie to było kilka chwil. W 2007 roku pracowałem z kilkoma zespołami Agile w projekcie migracji centrum danych. Byłem zazdrosny, że mieli taką wysoką wydajność — potrafili zrobić tak wiele w tak krótkim czasie.

Podczas wykonywania mojego kolejnego zadania zacząłem eksperymentować z techniką Kanban i przekonałem się, w jak dynamiczny sposób zmienił się zespół. Później, na konferencji Agile Toronto w 2008 roku, zaprezentowałem artykuł IEEE na ten temat, ale miałem wrażenie, że nie wywarł wielkiego wpływu na społeczność Agile. Utworzyliśmy grupę administracyjną pracującą zgodnie z systemem Agile, ale przeoczyliśmy czynnik ludzki.

Po obejrzeniu na Velocity Conference 2009 prezentacji Johna Allspawa i Paula Hammonda zatytułowanej „10 Deploys per Day”* byłem przekonany, że inni myśleli w podobny sposób. Postanowiłem zatem zorganizować pierwszą konferencję DevOpsDays. W ten sposób przypadkowo stworzyłem termin DevOps.

* 10 wdrożeń dziennie — przyp. tłum.

Energia tego wydarzenia była unikatowa i zaraźliwa. Kiedy ludzie zaczęli mi dziękować za zmianę na lepsze, która nastąpiła w ich życiu, zrozumiałem, jaki wpływ wywarł nowy nurt. Od tamtej pory nie przestałem promować DevOps.

John Willis

W 2008 roku, wkrótce po tym, jak sprzedałem firmę konsultingową zajmującą się wielkoskalowymi praktykami operacji IT w zakresie zarządzania konfiguracją i monitorowaniem (Tivoli), po raz pierwszy spotkałem Luke'a Kaniesa (założyciela Puppet Labs). Luke wygłaszał prezentację na temat firmy Puppet na organizowanej przez O'Reilly konferencji open source poświęconej zarządzaniu konfiguracją (ang. *configuration management — CM*).

Początkowo po prostu biernie siedziałem z tyłu, myśląc sobie: „Co ten dwudziestolatek mógłby powiedzieć mi o zarządzaniu konfiguracją?”. W końcu dosłownie całe moje życie pracowałem w największych przedsiębiorstwach na świecie, pomagając im tworzyć architektury CM i inne rozwiązania zarządzania operacjami. Jednak nie minęło pięć minut jego sesji, gdy przeniosłem się do pierwszego rzędu i zdałem sobie sprawę, że wszystko, co robiłem w ciągu ostatnich 20 lat, było niewłaściwe. Luke opisywał coś, co ja teraz nazywam CM drugiej generacji.

Po tej sesji miałem okazję usiąść z nim i napić się kawy. Byłem bezgranicznie przekonany do koncepcji, którą dziś nazywamy „infrastruktura jako kod”. Jednak podczas spotkania na kawie Luke zaczął dokładniej wyjaśniać swoje pomysły. Zaczął mi opowiadać, że uważa, iż działy operacji powinny zacząć zachowywać się tak jak programiści. Powinni utrzymywać konfiguracje w systemach zarządzania kodem źródłowym oraz zaadaptować wzorce dostaw CI/CD w swoich przepływach pracy. Będąc wówczas starym lisem w dziedzinie operacji IT, odpowiedziałem mu coś w stylu: „Ten pomysł utonie wśród ludzi z Ops, tak jak Led Zeppelin” (wyraźnie nie miałem racji).

Następnie jakiś rok później, w 2009 roku, na innej konferencji O'Reilly, Velocity, spotkałem Andrew Clayego Shafera wygłaszającego prezentację na temat infrastruktury Agile. W swojej prezentacji Andrew pokazał kultowy obraz ściany pomiędzy programistami a pracownikami działów operacyjnych, z metaforecznym przedstawieniem pracy przerzucanej przez mur. Nazwał to „murem nieporozumień” (ang. *the wall of confusion*). Pomyślały, które wyraził w tej prezentacji, kodyfikowały wiedzę, którą Luke próbował przekazać mi rok wcześniej. W mojej głowie „zaświeciła się żarówka”. W tym samym roku byłem jedynym Amerykaninem zaproszonym na pierwszą konferencję DevOpsDays w Gandawie. Kiedy to wydarzenie się zakończyło, coś, co dziś nazywamy DevOps, było obecne w mojej krwi.

Najwyraźniej wszyscy współautorzy tej książki przeżyli podobne objawienie — nawet jeśli doszli do tych samych wniosków z bardzo różnych kierunków. Obecnie jednak istnieje przytłaczający ciężar dowodów, że problemy opisane powyżej występowały niemal wszędzie i że rozwiązania dotyczące DevOps mają niemal uniwersalne zastosowanie.

Celem tej książki jest opisanie sposobu replikowania przekształceń związanych z DevOps, których doświadczaliśmy albo które zaobserwowaliśmy, a także rozwianie wielu mitów na temat powodów, dla których w pewnych sytuacjach DevOps się nie sprawdzi. Poniżej opisano kilka najczęstszych mitów, które można usłyszeć na temat DevOps.

Mit — *DevOps jest tylko dla startupów.* Chociaż pionierami praktyk DevOps były duże internetowe „jednorożce” — czyli takie firmy, jak Google, Amazon, Netflix i Etsy — to każdej z tych organizacji w pewnym momencie groziło bankructwo z powodu problemów typowych dla firm tradycyjnych (tzw. „koni”): bardzo niebezpiecznych wydań kodu, które stwarzały ryzyko katastrofalnych awarii, braku zdolności do wydawania cech funkcjonalnych wystarczająco szybko, by pokonać konkurencję, kłopotów ze zgodnością, braku zdolności do skalowania, wysokiego poziomu nieufności pomiędzy działami rozwoju i operacji itd.

Jednak każdej z tych organizacji udało się przekształcić swoją architekturę, praktyki techniczne i kulturę w taki sposób, że firmy te uzyskały niesamowite wyniki. Zasługę tę przypisujemy DevOps. Jak zażartował dr Branden Williams, dyrektor ds. bezpieczeństwa informacji: „Zaprzestaśmy dysput na temat jednorożców i koni DevOps. Mówmy o koniach czystej krwi i koniach innych ras zmierzających w kierunku fabryki kleju”.

Mit — *DevOps zastępuje Agile.* Zasady i praktyki DevOps są zgodne z Agile. Wiele osób zauważało, że DevOps jest logiczną kontynuacją podróży Agile, która rozpoczęła się w 2001 roku.

Wiele praktyk DevOps wyłania się w wyniku kontynuacji zarządzania pracą wykraczającą poza cel „kodu potencjalnie nadającego się do wydania” na końcu każdej iteracji. Cel ten należy rozszerzyć o postulat, by kod zawsze był w stanie gotowym do instalacji, aby programiści codziennie wgrywali swój kod do repozytorium, a cechy funkcjonalne były demonstrowane w środowiskach przypominających środowiska produkcyjne.

Mit — *DevOps jest niezgodny z ITIL.* Wiele osób uważa DevOps za zaprzeczenie ITIL (ang. *Information Technology Infrastructure Library*) lub ITSM (ang. *IT Service Management* — dosł. „zarządzanie usługami IT”). Praktyki ITIL wywarły ogromny wpływ na wiele pokoleń praktyków operacji IT, w tym na jednego ze współautorów tej książki. Jest to ciągle zmieniająca się biblioteka praktyk, mająca na celu kodyfikację procesów i technik stanowiących podstawę światowej klasy operacji IT, obejmujących strategię usług, ich projektowanie i pomoc techniczną.

Praktyki DevOps mogą być zgodne z procesem ITIL. Jednak w celu wspomagania osiągnięcia krótszego czasu realizacji i większych częstotliwości wdrażania związa-

nych z DevOps wiele obszarów procesów ITIL w pełni zautomatyzowano, rozwiązuje mnóstwo problemów dotyczących procesów zarządzania konfiguracją i wydaniami (np. utrzymywanie aktualności bazy danych i bibliotek oprogramowania). Ponieważ DevOps wymaga szybkiego wykrywania zdarzeń serwisowych i przeprowadzania działań naprawczych po ich wystąpieniu, to dziedziny ITIL dotyczące projektowania usług, zarządzania incydentami i problemami pozostają równie istotne, jak bez stosowania DevOps.

Mit — *DevOps jest niezgodny z zasadami bezpieczeństwa informacji i zgodności z przepisami.* Brak tradycyjnych praktyk kontrolnych (np. podział obowiązków, procesy zatwierdzania zmian, ręczne kontrole zabezpieczeń pod koniec projektu) może przerażać specjalistów w dziedzinie bezpieczeństwa informacji i zgodności z przepisami.

Jednak to nie oznacza, że organizacje stosujące DevOps nie mają skutecznych mechanizmów kontroli. Zamiast realizacji zadań związanych z bezpieczeństwem i zgodnością z przepisami tylko pod koniec projektu elementy kontroli są zintegrowane z każdym z etapów codziennej pracy w cyklu tworzenia oprogramowania. Dzięki temu można uzyskać lepsze wyniki jakości, bezpieczeństwa i zgodności.

Mit — *mechanizmy DevOps eliminują funkcje operacji IT.* Wiele osób błędnie uważa, że techniki DevOps całkowicie eliminują funkcje operacji IT. Jednak taka sytuacja nie zdarza się często. Chociaż charakter operacji IT może być inny, to pozostają one równie ważne. Działy operacji IT współpracują z działami rozwoju znacznie wcześniej w cyklu życia oprogramowania. Programiści współpracują z pracownikami działu operacji IT na dłużej po przekazaniu kodu do produkcji.

Zamiast wykonywania przez pracowników działów operacji IT ręcznej pracy w odpowiedzi na zlecenia usług DevOps promuje korzystanie z API oraz platform samoobsługowych. W ten sposób powstają środowiska, mechanizmy testowania i wdrażania kodu, monitorowania i przeglądania telemetrii produkcji itd. Dzięki temu praca personelu operacji IT przypomina działania programistów (a także inżynierów validacji i ekspertów zabezpieczeń). Są oni zaangażowani w rozwój produktu, który jest platformą wykorzystywaną do bezpiecznego, szybkiego i niezawodnego testowania, wdrażania i uruchamiania usług IT w produkcji.

Mit — *DevOps to po prostu „infrastruktura jako kod” lub automatyzacja.* Podczas gdy wiele wzorców DevOps zaprezentowanych w tej książce wymaga automatyzacji, to stosowanie ich wymaga również przestrzegania norm kulturowych i architektury. Dzięki temu można osiągać wspólne cele za pośrednictwem strumienia wartości IT. To znacznie więcej niż tylko automatyzacja. Jak napisał Christopher Little, praktyk DevOps i jeden z pierwszych autorów publikacji na ten temat: „DevOps nie jest automatyzacją — podobnie jak teleskopy nie są astronomią”.

Mit — *DevOps dotyczy tylko oprogramowania open source.* Chociaż wiele pomyślnych zastosowań DevOps miało miejsce w organizacjach korzystających z oprogramowania stosu LAMP (Linux, Apache, MySQL, PHP), to uzyskanie pozytywnych wyni-

ków z DevOps jest niezależne od stosowanych technologii. Osiągano sukcesy przy zastosowaniu aplikacji napisanych przy użyciu Microsoft.NET, COBOL-a oraz na platformach mainframe, jak również z SAP, a nawet w przypadku systemów wbudowanych (np. firmware HP LaserJet).

ROZPRZESTRZENIANIE MOMENTU „AHA!”

Każdego ze współautorów tej książki zainspirowała niesamowite innowacje wprowadzone przez społeczność DevOps oraz uzyskane dzięki temu efekty: stworzenie bezpiecznych systemów pracy i umożliwienie niewielkim zespołom szybkiego i niezależnego rozwijania i walidacji kodu, który można bezpiecznie wdrożyć u klientów. Biorąc pod uwagę nasze przekonanie, że DevOps jest przejawem tworzenia dynamicznych, uczących się organizacji, które nieustannie wzmacniają normy kulturowe wysokiego poziomu zaufania, wydaje się nieuniknione, że organizacje te będą w dalszym ciągu wprowadzać innowacje i wygrywać na rynku.

Mamy szczerą nadzieję, że książka *DevOps. Światowej klasy zwinnosć, niezawodność i bezpieczeństwo w Twojej organizacji* będzie służyć jako cenny zasób dla wielu osób. Może stać się pomocna jako:

- przewodnik planowania i przeprowadzania przekształceń DevOps;
- zbiór studiów przypadku do analizy i nauki;
- kronika historii DevOps;
- mechanizm wspierający koalicję obejmującą właścicieli produktu, architektów, działa rozwoju, walidacji, operacji IT i ekspertów bezpieczeństwa informacji;
- środek do uzyskania najwyższego poziomu wsparcia menedżerskiego dla inicjatyw DevOps;
- moralny nakaz zmiany sposobu zarządzania organizacjami technicznymi w celu osiągnięcia lepszej skuteczności i wydajności;
- droga do stworzenia szcześliwszego i bardziej ludzkiego środowiska pracy, tak aby pracownicy mogli stać się osobami uczącymi się przez całe życie — to nie tylko pomaga wszystkim ludziom osiągnąć najwyższe osobiste cele, ale również ułatwia osiągnięcie sukcesu ich organizacjom.

SŁOWO WSTĘPNE

W przeszłości w wielu dziedzinach inżynierii doświadczano swego rodzaju ewolucji polegającej na nieustannym podnoszeniu poziomu zrozumienia własnej pracy. Chociaż istnieją wydziały uniwersyteckie i profesjonalne organizacje wspierające poszczególne dziedziny inżynierii (budownictwo, mechanikę, elektryczność, technikę jądrową itp.), to nie ulega wątpliwości, że nowoczesne społeczeństwo potrzebuje korzyści ze współdziałania wszystkich form inżynierii w sposób wielodyscyplinarny.

Spróbujmy pomyśleć o projektowaniu pojazdu wysokiej wydajności. Gdzie kończy się praca inżyniera mechanika, a gdzie zaczyna się praca inżyniera elektryka? Gdzie (jak i kiedy) ktoś z wiedzą z dziedziny aerodynamiki (kto z pewnością ma dobrze ugruntowane opinie na temat kształtu, wielkości i rozmieszczenia okien) powinien podjąć współpracę z ekspertem z dziedziny ergonomii pasażerów? A co z chemicznym wpływem przez cały okres użytkowania pojazdu mieszkańców paliwowej i oleju na materiały, z których są zbudowane silnik i skrzynia biegów? Można zadać także inne pytania dotyczące projektowania samochodu, ale efekt będzie taki sam: sukces we współczesnych przedsięwzięciach technicznych absolutnie wymaga wielu perspektyw i współdziałania wielu ekspertów.

Aby nastąpił postęp w jakiejś dziedzinie i aby ta dziedzina dojrzała, musi dotrzeć do punktu, w którym można dokładnie zastanowić się nad jej początkami, poszukać zbioru różnych perspektyw wynikających z tych refleksji i umieścić tę syntezę w kontekście, który pomoże społeczności nakreślić jej przyszłość.

Niniejsza książka reprezentuje taką syntezę i powinna być postrzegana jako przełomowy zbiór perspektyw (śmiem twierdzić, wciąż rozwijających się i szybko zmieniających się) dziedzin inżynierii oprogramowania i operacji IT.

Niezależnie od tego, w jakiej branży pracujesz lub jaki produkt czy jaką usługę dostarcza Twoja organizacja, ten sposób myślenia jest najważniejszy i niezbędny dla przetrwania każdego lidera biznesowego i technicznego.

— **John Allspaw, dyrektor techniczny (CTO) Etsy,**
Brooklyn w stanie Nowy Jork, sierpień 2016

WYOBRAŹ SOBIE ŚWIAT, W KTÓRYM DEV I OPS TWORZĄ DEVOPS

Wprowadzenie do podręcznika DevOps

Wyobraź sobie świat, w którym właściciele produktu, deweloperzy, inżynierowie validacji (QA), operacji IT i bezpieczeństwa informacji współpracują ze sobą nie tylko po to, aby sobie nawzajem pomóc, ale także po to, aby zapewnić sukces całej organizacji. Dzięki ukierunkowaniu na wspólny cel zapewniają szybki przepływ zaplanowanej pracy do produkcji (np. wykonanie dziesiątek, setek lub nawet tysięcy instalacji kodu dziennie), a jednocześnie zachowują światowej klasy stabilność, niezawodność, dostępność i bezpieczeństwo.

W tym świecie interdyscyplinarne zespoły rygorystycznie testują swoje hipotezy na temat tego, które funkcje najbardziej spodobażą się użytkownikom oraz które najbardziej przyczynią się do osiągnięcia celów organizacji. Dbają nie tylko o implementację cech funkcjonalnych dla użytkownika, ale również aktywnie działają na rzecz tego, aby ich praca płynnie i często przechodziła przez cały strumień wartości, bez powodowania chaosu i zakłóceń dla operacji IT lub dowolnych innych klientów, zarówno wewnętrznych, jak i zewnętrznych.

Jednocześnie inżynierowie validacji, operacji IT i bezpieczeństwa informacji zawsze pracują nad sposobami zminimalizowania tarów wewnętrz zespołu. Dzięki temu tworzą systemy pracy pozwalające deweloperom zwiększyć wydajność i uzyskać lepsze wyniki. Udostępnienie zespołom realizującym dostawy kodu wiedzy inżynierów validacji, operacji IT i bezpieczeństwa informacji, a także dostarczenie zautomatyzowanych

narzędzi i samoobsługowych platform pozwala im skorzystać z tych wartości w ich codziennej pracy w sposób niezależny od innych zespołów.

To umożliwia organizacjom stworzenie bezpiecznego systemu pracy, w którym niewielkie zespoły potrafią szybko i samodzielnie rozwijać, testować i wdrażać kod i usługi u klientów szybko, bezpiecznie, pewnie i niezawodnie. Dzięki temu organizacje mogą zmaksymalizować wydajność pracy deweloperów, stworzyć warunki do uczenia się w organizacji, uzyskać wysoki poziom zadowolenia i wygrać na rynku.

Są to wyniki stosowania technik DevOps. Większość z nas nie żyje w takim świecie. Bardzo często system, z którym pracujemy, jest uszkodzony, co powoduje bardzo słabe wyniki — znacznie poniżej potencjału. W naszym świecie zadania rozwoju i operacji IT wzajemnie sobie przeszkadzają. Zadania testowania i weryfikacji zabezpieczeń są wykonywane wyłącznie pod koniec projektu — za późno na to, by rozwiązać wszystkie znalezione problemy. Do tego prawie każda kluczowa aktywność wymaga zbyt wiele „ręcznych” działań i zbyt wiele przekazywania pracy, co sprawia, że prawie zawsze musimy na kogoś czekać. Taka sytuacja nie tylko przyczynia się do bardzo długich okresów realizacji czegokolwiek, ale również jakość naszej pracy, zwłaszcza wdrażanie produkcyjne, wywołuje problemy, co wywiera negatywny wpływ na klientów i kondycję naszego biznesu.

W rezultacie jesteśmy dalecy od osiągnięcia oczekiwanych celów, a cała organizacja jest niezadowolona z wydajności IT, co skutkuje redukcjami budżetu i frustracją, powodującą niezadowolenie wśród pracowników, którzy czują się bezsilni wobec zadania wprowadzenia zmian w procesie i poprawy jego wyników*. Czy istnieje jakieś rozwiązanie? Trzeba zmienić sposób naszej pracy. DevOps pokazuje nam najlepszą drogę.

Aby lepiej zrozumieć potencjał rewolucji DevOps, przyjrzyjmy się rewolucji produkcji przemysłowej z lat 80. Przyjęcie zasad i praktyk metodyki Lean pozwoliło przedsiębiorstwom produkcyjnym znacznie poprawić wydajność, skrócić czasy realizacji, poprawić jakość produktów i zadowolenie klientów. Wszystko to umożliwiło przedsiębiorstwom osiągnięcie sukcesu na rynku.

Przed rewolucją średni czas realizacji zlecenia produkcyjnego wynosił sześć tygodni i tylko niespełna 70% zamówień dostarczano w terminie. Do roku 2005 dzięki powszechnemu wdrożeniu praktyk Lean przeciętny czas realizacji produktu spadł poniżej trzech tygodni i ponad 95% zamówień było dostarczanych terminowo. Przedsiębiorstwa, które nie wdrożyły praktyk Lean, utraciły część rynku, a wiele z nich całkowicie z niego zniknęło.

Na podobnej zasadzie podniosła się poprzeczka dla dostarczania produktów i usług technicznych — to, co było wystarczająco dobre w poprzednich dekadach, nie jest wystarczająco dobre teraz. W każdej z czterech ostatnich dekad czas i koszty wymagane do tworzenia i wdrażania strategicznych funkcji biznesowych zmniejszyły się o rząd

* To tylko niewielka próbka problemów występujących w typowych organizacjach IT.

wielkości. W latach 70. i 80. większość nowych funkcji wymagała od jednego roku do pięciu lat na opracowanie i wdrożenie i często kosztowała dziesiątki milionów dolarów.

W dekadzie 2000 – 2010 ze względu na postęp technologiczny i wdrożenie zasad i praktyk Agile czas potrzebny na opracowanie nowych funkcji skrócił się do tygodni lub miesięcy, ale wdrożenie rozwiązań do produkcji nadal wymagało tygodni lub miesięcy, co często miało katastrofalne skutki.

Po roku 2010 dzięki wprowadzeniu DevOps i niekończącej się komodyfikacji sprzętu, oprogramowania, a obecnie chmury funkcje (a nawet całe startupy) można utworzyć w kilka tygodni, a następnie szybko wdrożyć do produkcji w ciągu zaledwie kilku godzin lub minut — dla tych organizacji wdrażanie wreszcie stało się rutyną i zadaniem o niskim ryzyku. Organizacje te mogą wykonywać eksperymenty w celu testowania pomysłów biznesowych, odkrywać, jakie pomysły przynoszą największą wartość dla klientów i organizacji jako całości, a następnie rozwijać te pomysły w cechy funkcjonalne, które można szybko i bezpiecznie wdrożyć do produkcji. Trend w kierunku coraz szybszego wdrażania oprogramowania zilustrowano w tabeli 1.

Tabela 1. Nieustannie postępujący trend w kierunku wdrażania oprogramowania szybszego, tańszego i obarczonego mniejszym ryzykiem

	lata 70. i 80.	lata 90.	2000 – obecnie
Era	Komputery mainframe	klient-serwer	Komodyfikacja i chmura
Reprezentatywna technologia ery	COBOL, DB2 na MVS itp.	C++, Oracle Solaris itp.	Java, MySQL, Red Hat, Ruby on Rails, PHP itp.
Czas cyklu	1 – 5 lat	3 – 12 miesięcy	2 – 12 tygodni
Koszty	1 mln – 100 mln dolarów	100 tys. – 10 mln dolarów	10 tys. – 1 mln dolarów
Ryzyko	Cała firma	Linia produktu lub dział	Funkcja produktu
Koszty awarii	Upadłość, sprzedaż firmy, masowe zwolnienia	Utrata przychodów, zwolnienia inżynierów	Nieistotne

(Źródło: Adrian Cockcroft, „Velocity and Volume (or Speed Wins)” — prezentacja na konferencji FlowCon, San Francisco, Kalifornia, listopad 2013 r.)

Dzisiejsze organizacje stosujące zasady i praktyki DevOps często wdrażają zmiany setki lub nawet tysiące razy dziennie. W dobie, w której uzyskanie przewagi konkurencyjnej wymaga krótkiego czasu wprowadzenia produktu na rynek i nieustannego eksperymentowania, organizacje, które nie są w stanie replikować tych wyników, są skazane na porażkę na rynku z bardziej zwinnymi konkurentami. Mogą nawet całkowicie zbankrutować — podobnie jak firmy produkcyjne, które nie przyjęły zasady metodyki Lean.

Obecnie niezależnie od branży, w której konkurujemy, sposób pozyskiwania klientów i dostarczania im wartości zależy od strumienia wartości technologii. Można to ująć jeszcze zwięzlej. Jak powiedział Jeffrey Immelt, dyrektor zarządzający w firmie

General Electric: „Każda branża i firma, która nie dostarczy oprogramowania do rdzenia swojej działalności, zniknie z rynku”. Można też zacytować Jeffreya Snovera, posiadacza tytułu Technical Fellow w firmie Microsoft: „W poprzednich epokach gospodarczych przedsiębiorstwa tworzyły wartość przez przenoszenie atomów. Teraz tworzą ją, przenosząc bity”.

Trudno przecenić ogrom tego problemu — dotyczy on każdej organizacji, niezależnie od branży, w której działały, rozmiarów organizacji oraz tego, czy jest to organizacja zorientowana na osiąganie zysków, czy non profit. Teraz bardziej niż kiedykolwiek sposób działania i zarządzania technologią pozwala prognozować, czy organizacja osiągnie przewagę konkurencyjną na rynku lub nawet czy przetrwa. W wielu przypadkach trzeba przyjąć zasady i praktyki, które znaczco różnią się od tych, które gwarantowały sukces w ostatnich dziesięcioleciach (dodatek 1.).

Teraz, kiedy opisaliśmy powagę problemów, jakie rozwiązuje DevOps, poświęćmy trochę miejsca na bardziej szczegółową analizę symptomatologii problemu — dlaczego występuje i dlaczego bez dramatycznej interwencji z biegiem czasu się nasila.

PROBLEM — COŚ W TWOJEJ ORGANIZACJI WYMAGA POPRAWY (INACZEJ NIE CZYTAŁBYŚ TEJ KSIĄŻKI)

Większość organizacji nie jest w stanie wdrożyć zmian w produkcji w ciągu kilku minut lub godzin. Zwykle wymaga to kilku tygodni lub miesięcy. Nie są również w stanie wdrażać setek lub tysięcy zmian w produkcji dziennie. Taka liczba zmian zazwyczaj wymaga miesiąca lub nawet kwartału. Wdrożenia do produkcji nie są także rutynowe. Najczęściej wymagają przestojów, „gaszenia pożarów” i heroizmu.

W czasach, kiedy uzyskanie przewagi konkurencyjnej wymaga krótkiego czasu wprowadzenia produktu na rynek, wysokiego poziomu obsługi i nieustanego eksperymentowania, organizacje o opisanych powyżej cechach tracą na konkurencyjności. Jest to w dużej mierze spowodowane niezdolnością do rozwiązywania w nich podstawowych, przewlekłych konfliktów.

PODSTAWOWE, PRZEWLEKŁE KONFLIKTY

W prawie każdej organizacji IT istnieje wewnętrzny konflikt pomiędzy działem rozwoju a działem operacji IT. Konflikt ten powoduje spiralę negatywnych zjawisk. Skutkuje coraz dłuższym czasem wprowadzania na rynek nowych produktów i funkcji, gorszą wydajnością, częstszymi przestojami, a co najgorsze, coraz większymi niedoborami technologicznymi (tzw. długiem technicznym — ang. *technical debt*).

Termin „dług techniczny” został wymyślony przez Warda Cunninghama. Dług techniczny analogicznie do finansowego pokazuje, że podejmowane przez nas decyzje prowadzą do problemów, które z biegiem czasu stają się coraz bardziej trudne do rozwiązania.

Jednocześnie przyczyniają się do stałego zmniejszania w przyszłości dostępnych możliwości. Nawet jeśli dług został zaciągnięty rozważnie, to i tak trzeba płacić odsetki.

Jednym z czynników, które przyczyniają się do wzrostu dłużu technicznego, są występujące często ze sobą w sprzeczności cele działów rozwoju i operacji IT. Organizacje IT są odpowiedzialne za wiele spraw. Wśród nich są dwa cele, które muszą być osiągnięte jednocześnie:

- reagowanie na szybko zmieniający się krajobraz konkurencji;
- zapewnienie klientom stabilnych, niezawodnych i bezpiecznych usług.

Dział rozwoju często przejmuje odpowiedzialność za reagowanie na zmiany na rynku i stara się, aby nowe funkcje i zmiany były jak najszybciej wdrażane do produkcji. Dział operacji IT przejmuje odpowiedzialność za dostarczanie klientom usług IT, które są stabilne, niezawodne i bezpieczne. Ze względu na to wprowadzanie do produkcji zmian, co mogłoby wymagać przestojów w świadczeniu usług, staje się utrudnione lub wręcz niemożliwe. Skonfigurowane w ten sposób działy rozwoju i operacji IT mają przeciwne cele i motywacje do działania.

Dr Eliyahu M. Goldratt, jeden z założycieli ruchu zarządzania produkcją, nazwał ten rodzaj konfiguracji „podstawowym, przewlekłym konfliktem” — sytuacją, gdy parametry organizacyjne i motywacje do działania w obrębie różnych silosów w organizacji przeszkadzają w osiągnięciu globalnych celów organizacyjnych*.

Wspomniany konflikt tworzy spiralę degradacji tak potężną, że osiągnięcie pożądanego wyników biznesowych, zarówno wewnętrz, jak i na zewnątrz organizacji IT, staje się niemożliwe. Te przewlekłe konflikty często prowadzą do słabej jakości oprogramowania i usług oraz niskiego poziomu zadowolenia klientów, a także wprowadzają codzienną potrzebę obejść „gaszenia pożarów” i heroicznego poświęcenia — zdarza się to często w działach zarządzania produktem, rozwoju i bezpieczeństwa informacji (dodatek 2.).

SPIRALA DEGRADACJI W TRZECH AKTACH

Spirala degradacji w IT składa się z trzech aktów, które prawdopodobnie są znane większości informatyków.

Pierwszy akt rozpoczyna się w dziale operacji IT, którego celem jest utrzymanie działania aplikacji i infrastruktury w taki sposób, aby organizacja mogła dostarczać wartość dla klientów. Powodem wielu problemów w naszej codziennej pracy są aplikacje i infrastruktura, które są skomplikowane, słabo udokumentowane i niezwykle

* W firmach produkcyjnych występuje podobny przewlekły konflikt: potrzeba jednoczesnego zapewnienia klientom dostaw na czas i kontrolowanie kosztów. Sposób rozwiązania tego podstawowego, przewlekłego konfliktu opisano w dodatku 2.

kruche. To jest dług techniczny oraz codzienne obejścia, z którymi stale żyjemy, zawsze obiecując, że wyeliminujemy chaos, gdy znajdziemy trochę więcej czasu. Ale ten czas nigdy nie nadchodzi.

Na domiar złego najbardziej kruche artefakty zwykle obsługują najważniejsze systemy, generujące dochód, lub najbardziej kluczowe projekty. Innymi słowy, systemy najbardziej podatne na awarie są jednocześnie najważniejsze i znajdują się w epicentrum najpilniejszych zmian. Kiedy te zmiany zakończą się niepowodzeniem, narażają na szwank najważniejsze obietnice organizacyjne, takie jak dostępność dla klientów, cele finansowe, bezpieczeństwo danych klienta, dokładne raporty finansowe itd.

Akt drugi rozpoczyna się w chwili, gdy ktoś musi zrekompensować ostatnią złamąć obietnicę — może to być menedżer produktu, który obiecuje lepszą, bardziej rozbudowaną, mającą olśnić klientów wersję, albo menedżer biznesowy, który określa jeszcze wyższy docelowy przychód. Następnie, niezależnie od tego, co może, a czego nie może technologia albo jakie czynniki doprowadziły do niepowodzenia w osiąganiu wcześniejszego zobowiązania, kreowane jest zobowiązanie spełnienia nowej obietnicy.

W rezultacie działa rozwoju otrzymuje zadanie realizacji innego pilnego projektu, który nieuchronnie wymaga rozwiązania nowych problemów technicznych i wyznaczenia nowych dróg na skróty w celu dotrzymania obiecanej daty wydania. To tylko zwiększa dług techniczny, zaciągnięty, co oczywiste, przez obietnicę, że wszelkie powstałe problemy rozwiążemy, jeśli znajdziemy trochę więcej czasu.

To ustawia scenę dla trzeciego i ostatniego aktu, kiedy wszystko staje się trochę bardziej — krok po kroku każdy staje się coraz bardziej zajęty, praca zajmuje trochę więcej czasu, komunikacja staje się trochę wolniejsza, a kolejka pracy do wykonania jest trochę dłuższa. Praca staje się ścisłe powiązana, mniejsze działania powodują większe niepowodzenia. Stajemy się bardziej bojaźliwi i mamy mniej tolerancji dla wprowadzania zmian. Praca wymaga więcej komunikacji, koordynacji i uzgodnień; zespoły muszą czekać dłużej na wykonanie pracy, od której zależą, a jakość coraz bardziej się obniża. Tryby poruszają się coraz wolniej i potrzeba coraz więcej wysiłku, aby nadal się obracały (dodatek 3.).

Choć trudno zauważyć ją natychmiast, to wystarczy zrobić krok wstecz, aby wyraźnie zobaczyć spirali degradacji. Zauważamy, że wdrożenia kodu produkcyjnego zajmują coraz więcej czasu — na ich ukończenie potrzeba już nie minut, tylko godzin, nie dni, tylko tygodni. I co gorsza, wyniki wdrażania przyczyniają się do jeszcze większych problemów. Powodują coraz większą liczbę awarii mających wpływ na klienta. To wymaga jeszcze więcej „heroizmu” i „gaszenia pożarów” w dziale operacji IT i jeszcze bardziej pozbawia go zdolności do spłaczania dłużu technicznego.

W efekcie cykle dostawy produktów stają się coraz wolniejsze, realizowanych jest coraz mniej projektów, a te które są realizowane, są mniej ambitne. Co więcej, opinie na temat pracy wszystkich docierają wolniej i jest ich mniej — dotyczy to zwłaszcza

sygnałów pochodzących od klientów. Bez względu na to, co staramy się robić, wszystko wydaje się coraz gorsze — już nie jesteśmy w stanie szybko reagować na zmieniający się krajobraz konkurencji ani nie jesteśmy w stanie zapewnić stabilnych i niezawodnych usług naszym klientom. W rezultacie ostatecznie tracimy pozycję na rynku.

Raz po raz dowiadujemy się, że gdy upada dział IT, upada cała organizacja. Jak zauważył Steven J. Spear w swojej książce *The High-Velocity Edge*: „Niezależnie od tego, czy zniszczenia pojawiają się powoli, jak podczas wyniszczającej choroby, czy też gwałtownie, jak podczas wypadku... destrukcja może być również kompletna”.

DLACZEGO TĘ SPIRALĘ DEGRADACJI OBSERWUJE SIĘ WSZĘDZIE?

Przez ponad dekadę autorzy tej książki obserwowali omawianą spiralę degradacji w licznych organizacjach wszystkich typów i rozmiarów. Rozumiemy lepiej niż kiedykolwiek, dlaczego ta spirala występuje i dlaczego wymaga zastosowania zasad DevOps w celu złagodzenia jej skutków. Po pierwsze, zgodnie z tym, co opisano wcześniej, każda organizacja IT ma dwa przeciwwstawne cele, i po drugie, każda firma IT jest przedsiębiorstwem technologicznym — niezależnie od tego, czy zdajemy sobie z tego sprawę, czy nie.

Jak powiedział Christopher Little, menedżer wytwarzania oprogramowania i jeden z pierwszych kronikarzy DevOps: „Każda firma jest firmą technologiczną, niezależnie od tego, w jakiej branży się lokalizuje. Bank to po prostu firma IT z licencją bankową”.

Aby przekonać się, że tak jest, zwróćmy uwagę, że zdecydowana większość projektów inwestycyjnych w jakimś stopniu polega na IT. Powszechnie uważa się, że „jest praktycznie niemożliwe, aby podjąć jakąkolwiek decyzję biznesową, która nie spowodowałaby co najmniej jednej zmiany związanej z IT”.

W kontekście biznesowym i finansowym projekty mają kluczowe znaczenie, ponieważ odgrywają rolę podstawowego mechanizmu zmian wewnętrz organizacji. Dla projektów zarząd zwykle musi zatwierdzić budżet i muszą być one możliwe do rozliczenia. Dlatego są one mechanizmem osiągania celów i aspiracji organizacji — niezależnie od tego, czy chodzi o jej rozwój, czy nawet o redukcję[†].

Projekty są zazwyczaj finansowane przez nakłady kapitałowe (tzn. fabryki, wyposażenie i główne projekty są kapitalizowane po osiągnięciu zwrotu, co trwa wiele lat), z których 50% jest związanych z technologią. Dotyczy to nawet „mało informatycznych” branż przemysłowych, takich jak energetyka, przetwórstwo metali, wydobycie surowców, przemysł motoryzacyjny czy budowlany. Innymi słowy, liderzy biznesowi

* W 2013 r. europejski bank HSBC zatrudniał więcej programistów niż Google.

† Na razie zawiesimy dyskusję na temat tego, czy oprogramowanie powinno być finansowane jako „projekt”, czy jako „produkt”. Zostanie to omówione w dalszej części książki.

są znacznie bardziej uzależnieni od efektywnego zarządzania IT w osiąganiu swoich celów, niż im się wydaje*.

KOSZTY — LUDZKIE I EKONOMICZNE

Gdy pracownicy firmy są uwięzieni w spirali degradacji od lat — zwłaszcza pracownicy działów zależnych od działu rozwoju — często czują się zablokowani w systemie, który „wzywa” do awarii i nie pozwala na poprawę wyników. Skutkiem bezsilności jest często wypalenie powiązane z uczuciem zmęczenia, cynizmem, a nawet beznadejną i rozpaczą.

Wielu psychologów uważa, że tworzenie systemów, które powodują uczucie bezsilności, jest jedną z najbardziej szkodliwych rzeczy, jakie możemy zrobić innym ludziom. Pozbawiamy ich zdolności do kontroli własnych wyników, a nawet tworzymy środowisko, w którym ludzie boją się zrobić coś dobrze z powodu strachu przed karą, awarią lub zagrożeniem dla własnej egzystencji. Może to stwarzać warunki **wyuczonej bezradności**, gdzie ludzie stają się niechętni do działania lub pozbawieni możliwości działania w sposób, który w przyszłości pozwala uniknąć powstania tego samego problemu.

Oznacza to długie godziny pracy w weekendy i obniżenie jakości życia nie tylko dla pracowników, ale i dla każdego, kto od nich zależy, w tym rodziny i przyjaciół. Nic więc dziwnego, że gdy to nastąpi, tracimy najlepszych ludzi (z wyjątkiem tych, którzy czują, że nie mogą odejść z poczucia obowiązku lub przywiązania).

Oprócz ludzkiego cierpienia, które wynika z bieżącego sposobu pracy, koszt wartości, którą moglibyśmy wytworzyć, jest oszałamiający — autorzy uważają, że możemy tracić rocznie około 2,6 biliona dolarów — co w momencie powstawania tego tekstu było równoważne rocznej produkcji gospodarczej Francji, szóstej co do wielkości gospodarki na świecie.

Weźmy pod uwagę poniższe obliczenia — zarówno IDC, jak i Gartner szacują, że w 2011 roku około 5% produktu krajowego brutto państw na całym świecie (3,1 biliona dolarów) wydano na IT (sprzęt, usługi i łączna telekomunikacyjne). Jeśli szacujemy, że 50% z tych 3,1 biliona dolarów wydano na koszty eksploatacji i utrzymania istniejących

* Na przykład dr Vernon Richardson i jego koledzy opublikowali zdumiewające odkrycie. Poddali analizie raport 10-K 184 publicznych korporacji i podzieliли je na trzy grupy:

A) przedsiębiorstwa ze słabościami materiałymi i brakami związanymi z IT, B) firmy ze słabościami materiałymi bez braków związanych z IT i C) przedsiębiorstwa „czyste” — bez słabości materiałowych. W firmach z grupy A dyrektorzy zarządzający zmieniali się osiem razy częściej niż w firmach z grupy C. Dyrektorzy finansowi w firmach z grupy A zmieniali się czterokrotnie częściej niż w grupie C. Wyraźnie widać, że dział IT może mieć większe znaczenie, niż zazwyczaj uważamy.

systemów i że jedną trzecią tych 50% wydano na pilne i nieplanowane prace lub przeróbki, to zmarnowano około 520 miliardów dolarów.

Gdyby przyjęcie zasad DevOps umożliwiło — poprzez lepsze zarządzanie i zwiększenie jakości działań operacyjnych — zmniejszenie strat o połowę i użycie potencjału ludzkiego do czegoś innego, to można by osiągnąć zysk równy pięciokrotności straty, tzn. produkt wyceniany na 2,6 miliardów dolarów rocznie.

ETYKA DEVOPS — ISTNIEJE LEPSZY SPOSÓB

W poprzednich punktach opisaliśmy problemy i negatywne konsekwencje istniejącego stanu, wynikające z podstawowych, przewlekłych konfliktów, braku możliwości osiągnięcia celów organizacyjnych, oraz wskazaliśmy szkody, jakie to może spowodować. Dzięki rozwiązyaniu tych problemów DevOps w zadziwiający sposób umożliwia także poprawienie wydajności organizacji, pozwala osiągnąć cele jednostek pełniących różne funkcje techniczne (np. działów rozwoju, walidacji, operacji IT, bezpieczeństwa informacji) oraz poprawia warunki pracy.

Ta ciekawa i rzadka kombinacja pozytywnych efektów może wyjaśnić, dlaczego metodyka DevOps zyskała tak wiele entuzjazmu w tak krótkim czasie wśród liderów technicznych, inżynierów oraz w dużej części ekosystemu oprogramowania, w którym przebywamy.

ZŁAMANIE SPIRALI DEGRADACJI DZIĘKI ZASTOSOWANIU DEVOPS

W warunkach idealnych niewielkie zespoły programistów niezależnie implementują swoje funkcje, walidują ich poprawność w środowiskach przypominających produkcyjne i wdrażają kod do produkcji — szybko, bezpiecznie i pewnie. Instalacje kodu są rutynowe i przewidywalne. Nie muszą już rozpoczynać się w piątek o północy i trwać przez cały weekend. Teraz mogą być wykonywane przez cały dzień roboczy, gdy wszyscy są już w biurze, a klienci nawet tego nie zauważają — z wyjątkiem chwil, gdy obserwują nowe funkcje i poprawki błędów, co zwykle wprowadza ich w zachwyt. Dzięki instalowaniu kodu w ciągu dnia roboczego po raz pierwszy od dziesięcioleci pracownicy działu operacji IT pracują w normalnych godzinach pracy — tak jak wszyscy inni.

Dzięki tworzeniu szybkich pętli sprzężeń zwrotnych na każdym etapie procesu każdy może natychmiast zobaczyć efekty swoich działań. Za każdym razem, gdy do systemu kontroli wersji zostaną zapisane zmiany, wykonywane są szybkie automatyczne testy w środowisku zbliżonym do produkcyjnego. To daje ciągłe poczucie pewności, że kod i środowisko pracy działają zgodnie z przeznaczeniem. Zawsze są bezpieczne i gotowe do zainstalowania.

Automatyczne testowanie pomaga deweloperom wykryć popełnione błędy szybko (zazwyczaj w ciągu kilku minut), co umożliwia szybkie wprowadzanie poprawek, a także prawdziwą naukę. Taka nauka nie jest możliwa w przypadku, gdy błędy zostaną wykryte sześć miesięcy później, podczas testów integracyjnych — gdy dawno zanikła pamięć na temat związków pomiędzy przyczynami a skutkami. Zamiast powiększać dług techniczny, problemy rozwiązuje się natychmiast po ich pojawienniu się. Jeśli jest taka potrzeba, to można zmobilizować całą organizację, ponieważ cele globalne są ważniejsze od celów lokalnych.

Wszechobecna telemetria produkcyjna w środowisku zarówno kodu, jak i docelowym środowisku pracy dają pewność szybkiego wykrywania i rozwiązywania problemów. W ten sposób zyskujemy potwierdzenie, że wszystko działa zgodnie z przeznaczeniem, a klienci zyskują wartość z tworzonego oprogramowania.

W tym scenariuszu wszyscy czują się produktywnie — architektura pozwala niewielkim zespołom pracować bezpiecznie i zapewnia oddzielenie ich pracy od pracy innych zespołów, korzystających z platform samoobsługowych bazujących na zbiorowych doświadczeniach działań operacji i bezpieczeństwa informacji. Nikt nie musi ciągle czekać na wykonanie dużych ilości pilnych prac. Zamiast tego zespoły pracują niezależnie i wydajnie w małych partiach i często dostarczają klientom nowe wartości.

Dzięki zastosowaniu technik *dark launch* nawet wydania złożonych produktów i funkcji stają się rutyną. Na długo przed datą wydania całość wymaganego kodu nowej cechy funkcjonalnej jest przekazywana do produkcji w sposób niewidoczny dla nikogo, z wyjątkiem wewnętrznych pracowników i małych grup rzeczywistych użytkowników. Pozwala to na testowanie produktu i jego ewolucyjne rozwijanie do momentu, w którym zostanie osiągnięty pożądany cel biznesowy.

I zamiast „gasić pożary” przez wiele dni lub tygodni, aby nowe funkcje zaczęły działać, zmieniamy jedynie przełącznik lub ustawienie konfiguracji funkcji. Dzięki niewielkim zmianom nowe funkcje widoczne dla coraz większych segmentów klientów mogą być automatycznie wycofane, jeśli coś pójdzie nie tak. W rezultacie nasze wydania są kontrolowane, przewidywalne, odwracalne i nie powodują niepotrzebnego stresu.

Nie chodzi tylko o to, że publikacje nowych cech funkcjonalnych przebiegają spokojniej — wszelkiego rodzaju problemy są znajdowane i rozwiązywane wcześniej, kiedy są mniejsze, tańsze i łatwiejsze do rozwiązania. Wraz z każdą poprawką organizacja uczy się. To pozwala uniknąć wystąpienia problemu ponownie i umożliwia szybsze wykrywanie i rozwiązywanie podobnych problemów w przyszłości.

Ponadto uczą się wszyscy pracownicy. Promowana jest kultura inspiracji hipotezą, gdzie do zadbania o to, aby nic nie było przyjmowane za pewnik, stosowane są metody naukowe — nie robimy nic bez pomiarów, a rozwój produktu i usprawnianie procesu są traktowane jako eksperymenty.

Ponieważ cenimy czas wszystkich, nie poświęcamy wielu lat na rozwijanie funkcji, których nasi klienci nie chcą,wdrażanie kodu, który nie działa, lub naprawianie rzeczy, które w istocie nie są rzeczywistą przyczyną problemu.

Ponieważ dbamy o osiąganie celów, tworzone są długoterminowe zespoły odpowiedzialne za ich osiągnięcie. Zamiast zespołów projektowych, do których są przydzielani deweloperzy, a następnie mieszani po każdym wydaniu bez otrzymania informacji zwrotnej na temat ich pracy, utrzymujemy zespoły nienaruszone. Dzięki temu mogą one iteracyjnie uczyć się i łatwiej osiągać swoje cele. Dotyczy to również zespołów utrzymania produktu, odpowiedzialnych za rozwiązywanie problemów zewnętrznych klientów, jak również wewnętrznych zespołów utrzymania platform, które pomagają innym zespołom w osiągnięciu lepszej wydajności, pewności i bezpieczeństwa.

Zamiast kultury strachu mamy kulturę wysokiego zaufania i współpracy, gdzie ludzie są nagradzani za podejmowanie ryzyka. Potrafią odważnie mówić o problemach, zamiast je ukrywać lub odsuwać na dalszy plan — ostatecznie, by umieć rozwiązywać problemy, trzeba je widzieć.

Ponieważ wszyscy są w pełni odpowiedzialni za jakość swojej pracy, to wszyscy codziennie korzystają z testów automatycznych i wzajemnie przeglądają swoją pracę. Dzięki temu można zyskać pewność, że problemy zostaną rozwiązane, na długo zanim będą mogły wywarzyć wpływ na klienta. Te procesy wzajemnej kontroli, w przeciwieństwie do zatwierdzania przez odległych decydentów, łagodzą zagrożenia. Pozwalają dostarczać wartość szybko, niezawodnie i bezpiecznie, a jednocześnie udowadniają sceptycznym audytorom, że istniejący system kontroli wewnętrznej jest skuteczny.

A kiedy coś pójdzie nie tak, przeprowadzamy analizę problemu — nie po to, by kogoś ukarać, ale aby lepiej zrozumieć, co spowodowało problem i jak można mu zapobiec. Ten rytuał wzmacnia kulturę uczenia się. Prowadzone są również wewnętrzne konferencje techniczne, które pozwalają podnieść poziom naszych umiejętności i dają pewność, że wszystkie osoby zawsze nauczają innych oraz same się uczą.

Ponieważ dbamy o jakość, to możemy nawet wstrzykiwać usterki do naszego środowiska produkcji. Dzięki temu możemy się nauczyć w zaplanowany sposób, jak nasz system ulega awariom. Przeprowadzamy zaplanowane ćwiczenia mające na celu sprawdzenie w praktyce awarii na szeroką skalę. Losowo zabijamy procesy i serwery obliczeniowe w produkcji oraz wstrzykujemy opóźnienia sieciowe i inne błędy — po to, aby się upewnić, że stajemy się coraz bardziej elastyczni. W ten sposób pozwalamy na większą odporność, jak również umożliwiamy uczenie się i poprawianie całej organizacji.

W tym świecie wszyscy są właścicielami swojej pracy, bez względu na spełnianą funkcję w organizacji technicznej. Dzięki temu mają pewność, że ich praca ma znaczenie i że znaczco przyczyniają się do osiągnięcia celów organizacyjnych. Dowodem na skuteczność ich działania jest bezstresowe środowisko pracy oraz sukces organizacji na rynku.

WARTOŚĆ BIZNESOWA TECHNIK DEVOPS

Istnieją przekonujące dowody wartości biznesowej metodyki DevOps. W latach 2013 – 2016 w ramach raportu *State of DevOps* publikowanego przez Puppet Labs (w jego tworzeniu uczestniczyli autorzy tej książki: Jez Humble i Gene Kim) w celu lepszego zrozumienia kondycji i nawyków organizacji na wszystkich etapach wdrażania DevOps zebrałyśmy dane od ponad 25 000 specjalistów technicznych.

Pierwszą niespodzianką, jaką ujawniły te dane, była przewaga wydajności organizacji stosujących praktyki DevOps w stosunku do firm, które ich nie wykorzystywały. Przewagę zaobserwowano w następujących obszarach:

- parametry wydajności;
- wdrożenia kodu i zmian (30 razy częściej);
- czas realizacji wdrożeń kodu i zmian (200 razy szybciej);
- wskaźniki niezawodności;
- wdrożenia produkcyjne (60 razy wyższy wskaźnik sukcesu zmian);
- średni czas do przywrócenia usługi (168 razy szybciej);
- wskaźniki wydajności organizacyjnej;
- cele produktywności, udziałów w rynku i rentowności (dwa razy bardziej prawdopodobne przekroczenie);
- wzrost kapitalizacji rynku (50-procentowy wzrost w okresie trzech lat).

Innymi słowy, firmy o wysokim poziomie wydajności były zarówno bardziej zwinne, jak i bardziej niezawodne. Badanie dostarczyło empirycznych dowodów, że DevOps pozwala przełamać podstawowy, przewlekły konflikt. Firmy z DevOps wdrażały kod 30 razy częściej, a czas potrzebny do przejścia od „kodu przesłanego do repositorym” do „kodu pomyślnie wdrożonego w produkcji” był 200 razy krótszy. Czas realizacji wdrożenia w firmach DevOps był mierzony w minutach lub godzinach, natomiast w pozostałych — w tygodniach, miesiącach lub nawet kwartałach.

Ponadto firmy stosujące DevOps dwa razy częściej przekraczały cele rentowności, udziału w rynku i wydajności. Wśród firm obecnych na giełdzie organizacje stosujące DevOps w okresie trzech lat zanotowały 50-procentowy wzrost. Zanotowały także wyższe zadowolenie z pracy pracowników, niższy współczynnik wypalenia zawodowego, ich pracownicy 2,2 razy częściej polecali swoje organizacje znajomym jako świetne miejsce do pracy*. Firmy z DevOps miały również lepsze wyniki w dziedzinie bezpie-

* Zgodnie z pomiarem eNPS (*employee Net Promoter Score*). Jest to znaczące odkrycie, ponieważ badania wykazały, że „firmom z wysoce zaangażowanymi pracownikami przychody rosły dwa i pół razy bardziej w porównaniu z firmami, w których pracownicy wykazywali niskie

czeństwa informacji. Dzięki zintegrowaniu celów bezpieczeństwa na wszystkich etapach procesów rozwoju i operacji poświęcili 50% mniej czasu na eliminowanie skutków problemów bezpieczeństwa.

DEVOPS POMAGA SKALOWAĆ WYDAJNOŚĆ PRACY DEWELOPERÓW

Gdy zwiększymy liczbę deweloperów, wydajność indywidualnego dewelopera często znacznie się zmniejsza ze względu na narzut związany z komunikacją, integracją i testowaniem. Problem ten został wskazany w słynnej książce Fredericka Brooka, *The Mythical Man-Month*. Autor wyjaśnia, że dodanie do spóźnionego projektu dodatkowych programistów nie tylko zmniejsza wydajność indywidualnego programisty, ale również zmniejsza ogólną wydajność pracy.

Z drugiej strony DevOps pokazuje nam, że gdy architektura jest właściwa, stosowane są właściwe techniczne praktyki i odpowiednie normy kulturowe, to małe zespoły programistów są w stanie szybko, bezpiecznie i niezależnie rozwijać, integrować, testować i wdrażać zmiany do produkcji. Jak zaobserwował Randy Shoup, były dyrektor inżynierii w Google, dzięki DevOps duże organizacje „zatrudniają tysiące programistów, ale ich architektura i stosowane praktyki pozwalają zachować wydajność małych zespołów na takim poziomie, jaki mają nowo powstające przedsiębiorstwa”.

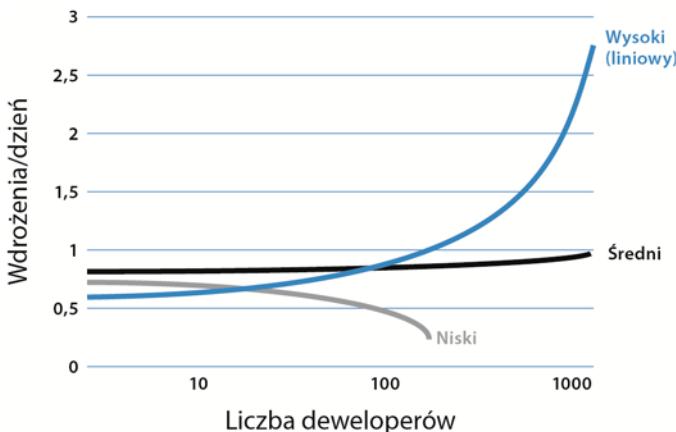
W raporcie *State of DevOps Report* z 2015 roku badano nie tylko „liczbę wdrożeń na dzień”, ale również „liczbę wdrożeń dziennie na jednego dewelopera”. Można postawić hipotezę, że firmy stosujące DevOps potrafiły skalować liczbę wdrożeń wraz ze wzrostem wielkości zespołu.

Na rysunku 1 wyraźnie widać, że w organizacjach o niskiej wydajności liczba wdrożeń dziennie maleje wraz ze wzrostem liczby zespołu, w przypadku firm o średniej wydajności pozostaje na stałym poziomie, a w przypadku organizacji o wysokiej wydajności wzrasta liniowo.

Innymi słowy, organizacje stosujące DevOps są w stanie liniowo zwiększać liczbę wdrożeń dziennie wraz ze wzrostem liczby deweloperów. Tę tezę udowodniła praktyka takich firm, jak Google, Amazon i Netflix*.

zaangażowanie. Obrót (publiczny) akcjami firm mających środowisko pracy wysokiego zaufania w latach 1997 – 2001 trzykrotnie przekroczył wskaźniki rynkowe”.

* Innym, bardziej skrajnym przykładem jest Amazon. W 2011 r. firma Amazon wykonywała około 7000 wdrożeń dziennie. W 2015 roku wykonywała ich około 130 000.



Rysunek 1. Liczba wdrożeń na dzień a liczba deweloperów
(źródło: Puppet Labs, State Of DevOps Report 2015)*

UNIWERSALNOŚĆ ROZWIĄZANIA

Jedną z najbardziej wpływowych książek dotyczących ruchu Lean jest pozycja dr. Eliyahu M. Goldratta z 1984 roku *The Goal: A Process of Ongoing Improvement*[†]. Książka ta wywarła wpływ na całe pokolenie profesjonalnych menedżerów fabryk na całym świecie. Była to powieść o kierowniku fabryki, który musiał rozwiązać problemy dotyczące kosztów i terminów dostarczania produktów w ciągu 90 dni. W przypadku niepowodzenia jego fabryka została zamknięta.

W późniejszym czasie kariery dr Goldratt opublikował listy, jakie otrzymał w odpowiedzi na wspomnianą książkę. Listy te zwykle brzmiały następująco: „Oczywiście opisujesz moją fabrykę, bo opisałeś dokładnie moje życie [jako kierownika zakładu]...”. Co najważniejsze, wykazały one, że menedżerowie byli w stanie replikować przełom w wydajności, opisany w książce, w swoich środowiskach pracy.

Książka Gene'a Kima, Kevina Behra i George'a Spafforda *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win*, napisana w 2013 roku, była dokładnie modelowana na książce Goldratta. Bohaterem tej powieści jest lider IT stawiający czoła wszystkim typowym problemom powszechnie występującym w organizacjach IT: realizuje projekt z przekroczonym budżetem, spóźniony, który musi trafić na rynek, aby firma mogła przetrwać. Doświadcza katastrof podczas wdrożeń, problemów z dostępnością, bezpieczeństwem, zgodnością z przepisami itd.

* Uwzględniono tylko te organizacje, które realizują co najmniej jedno wdrożenie dziennie.

† Wydanie polskie: *Cel I. Doskonałość w produkcji*, Mint Books 2013.

Ostatecznie on i jego zespół zaczynają stosować zasady i praktyki DevOps w celu pokonania tych wyzwań — aby pomóc organizacjom zwyciężyć na rynku. Dodatkowo powieść pokazuje, że stosowanie praktyk DevOps poprawiło środowisko pracy zespołu dzięki mniejszemu stresowi i większemu zadowoleniu wynikającemu z zaangażowania w cały proces.

Podobnie jak w przypadku celu, istnieją dowody uniwersalności problemów i rozwiązań opisanych w *Projektie Feniks*. Rozważmy niektóre zdania pochodzące z opinii w serwisie Amazon: „Wśród postaci opisanych w *Projektie Feniks* znalazłem swoich znajomych... Prawdopodobnie spotkałem większość z nich w trakcie mojej kariery”, „Jeśli kiedykolwiek pracowałaś w dowolnej z dziedzin IT, DevOps lub Infosec, to z pewnością odnajdziesz się w tej książce” albo „W *Projektie Feniks* nie ma postaci, której nie potrafiłbym zidentyfikować ze sobą lub kimś, kogo znam w swoim życiu... nie wspominając już o problemach, jakim stawiają czoła i jakie rozwiązuje te postacie”.

W dalszej części tej książki opiszemy, w jaki sposób replikować transformacje opisane w *Projektie Feniks*, a także zaprezentujemy wiele studiów przypadków opisujących sposoby stosowania przez organizacje zasad i praktyk DevOps w celu replikowania tych wyników.

PODRĘCZNIK DEVOPS — NIEZBĘDNY PRZEWODNIK

Celem książki *DevOps. Światowej klasy zwinność, niezawodność i bezpieczeństwo w Twojej organizacji* jest zaprezentowanie teorii, zasad i praktyk niezbędnych do skutecznego rozpoczęcia inicjatywy DevOps i osiągnięcia pożądanych rezultatów. Zaprezentowany przewodnik bazuje na wielu dekadach solidnej teorii zarządzania, badaniach organizacji technicznych o wysokiej wydajności, pracy wykonanej w celu pomocy organizacjom w transformacji oraz badaniach, które weryfikują skuteczność opisanych praktyk DevOps. Uwzględniono także wywiady z odpowiednimi ekspertami i analizy blisko 100 studiów przypadku zaprezentowanych na konferencji DevOps Enterprise Summit.

Książkę podzielono na sześć części. Opisuje ona teorie i zasady DevOps za pomocą techniki trzech dróg — specyficznego sposobu prezentacji teorii przedstawionego po raz pierwszy w *Projektie Feniks*. Książka *DevOps. Światowej klasy zwinność, niezawodność i bezpieczeństwo w Twojej organizacji* jest przeznaczona dla każdego, kto pracuje lub wpływa na pracę w strumieniu wartości technicznych (który zwykle obejmuje zarządzanie produktem, rozwój, waliadcję, operacje IT i bezpieczeństwo informacji), a także dla menedżerów biznesowych i marketingowych, od których rozpoczyna się większość inicjatyw technologicznych.

Czytelnik nie musi mieć rozległej wiedzy na temat żadnej z tych dziedzin. Nie musi też niczego wiedzieć na temat DevOps, Agile, ITIL, Lean lub usprawniania procesu. Każdy z tych tematów jest wprowadzony i objaśniony w książce, zanim jego znajomość stanie się niezbędna.

Naszym zamiarem jest stworzenie praktycznego źródła wiedzy na temat kluczowych pojęć w każdej z tych dziedzin — zarówno po to, aby służyło jako fundament, jak i po to, by wprowadzić język niezbędny do pomocy praktykującym tę technikę i umożliwić współpracę ze wszystkimi uczestnikami strumienia wartości IT, aby realizować wspólne cele.

Niniejsza książka będzie cenna dla liderów biznesowych i interesariuszy, którzy dążąc do osiągnięcia swoich celów, są coraz bardziej uzależnieni od organizacji technicznej.

Ponadto jest ona przeznaczona dla Czytelników pracujących w organizacjach, w których nie występują wszystkie spośród opisanych problemów (np. długie terminy realizacji wdrożeń lub wdrożenia stwarzające problemy). Nawet ci Czytelnicy, którzy są w tym szczęśliwym położeniu, skorzystają na zrozumieniu zasad DevOps — szczególnie tych, które odnoszą się do wspólnych celów, opinii i ciągłego uczenia się.

W części I zaprezentowano krótką historię DevOps. Przedstawiono podstawową teorię i kluczowe zagadnienia z obszarów zebranej przez dziesięciolecia wiedzy. Następnie omawiamy wysokopoziomowe zasady trzech dróg: **przepływu** (ang. *flow*), **sprzężenia zwrotnego** (ang. *feedback*) oraz **ciągłego uczenia się i eksperymentowania**.

W części II opisano, jak i od czego zacząć. Zaprezentowano takie pojęcia, jak strumienie wartości, zasady i wzorce projektowania, wzorce wdrażania i studia przypadków.

W części III wskazano możliwości przyspieszenia **przepływu** dzięki zbudowaniu podstaw potoku wdrożeń: szybkiego i skutecznego testowania automatycznego, ciągłej integracji, ciągłego dostarczania oraz tworzenia architektury wydań obarczonych niskim ryzykiem.

W części IV wyjaśniono, w jaki sposób przyspieszyć i wzmocnić **sprzężenie zwrotne** poprzez: tworzenie skutecznej telemetrii produkcji w celu obserwowania i rozwiązywania problemów oraz lepszego ich przewidywania; zbieranie opinii tak, aby działa Dev i Ops mogły bezpiecznie wdrażać zmiany, integrować testy A/B w codziennej pracy oraz tworzyć procesy przeglądania i koordynacji prowadzące do poprawienia jakości pracy.

W części V opisano, w jaki sposób można przyspieszyć **ciągłe uczenie się** poprzez ustanowienie **kultury sprawiedliwego traktowania** (ang. *just culture*), konwersję lokalnych odkryć na globalne usprawnienia oraz jak właściwie zarezerwować czas w celu stworzenia w organizacji systemu uczenia się i usprawniania.

Na koniec w części VI pokazujemy, jak w naszej codziennej pracy poprawnie zintegrować zasady zabezpieczeń oraz zgodności z przepisami. Omawiamy stosowanie zapobiegawczych mechanizmów kontroli repozytoriów kodu i usług, wykorzystanie zabezpieczeń w potoku wdrożeń, usprawnienia telemetrii w celu lepszego wykrywania i odzyskiwania, ochronę potoku wdrażania oraz osiąganie celów zarządzania zmianami.

Mamy nadzieję, że dzięki tej kodyfikacji przyczynimy się do szybszego przyjęcia praktyk DevOps, ułatwimy osiąganie sukcesów inicjatyw DevOps i obniżymy energię aktywacji wymaganą do przeprowadzenia transformacji DevOps.

Część I
Trzy drogi

W części I podręcznika *DevOps* opowiemy o konwergencji kilku ważnych ruchów w zarządzaniu i technologii, które ustawiły scenę dla kierunku DevOps. Opisujemy strumienie wartości, DevOps jako wynik zastosowania zasad metodyki Lean do strumienia wartości technologii i **trzy drogi**: przepływ, sprężenie zwrotne oraz ciągłe uczenie się i eksperymentowanie.

W tym rozdziale skoncentrujemy się przede wszystkim na następujących zagadnieniach:

- zasadach **przepływu**, które przyspieszają dostarczanie klientom wyników pracy działów rozwoju i operacji;
- zasadach **sprzężeń zwrotnych**, które pozwalają tworzyć coraz bezpieczniejsze systemy pracy;
- zasadach **ciągłego uczenia się i eksperymentowania**, które promują kulturę wysokiego zaufania oraz naukowe podejście do podejmowania ryzyka w codziennej pracy w celu poprawy działania organizacji.

KRÓTKA HISTORIA

DevOps i wynikające z niego praktyki kulturowe, techniczne i architektoniczne reprezentują połączenie wielu nurtów w filozofii i zarządzaniu. O ile wiele organizacji rozwinięło te zasady niezależnie, o tyle zrozumienie, że DevOps jest rezultatem połączenia szeregu ruchów — zjawiska opisanego przez Johna Willisa (jednego ze współautorów tej książki) jako „konwergencja DevOps” — pokazuje niesamowite postępy w myśleniu i nieprawdopodobne związki. Do stworzenia praktyk DevOps w takiej postaci, jaką znamy dziś, przyczyniły się dziesiątki lat doświadczeń wyniesionych z przedsiębiorstw produkcyjnych, firm wysokiej niezawodności, modeli zarządzania z wysokim poziomem zaufania oraz wielu innych czynników.

DevOps jest wynikiem zastosowania do strumienia wartości IT najbardziej sprawdzonych zasad z dziedziny fizycznej produkcji i zarządzania. DevOps bazuje na wiedzy opisanej jako metodyka Lean, na teorii ograniczeń, systemie produkcji w firmie Toyota (Toyota Production System), inżynierii odporności organizacyjnej, organizacjach uczących się, kulturze bezpieczeństwa, czynnikach ludzkich itp. Do innych cennych kontekstów, z których czerpie DevOps, należą kultury zarządzania wysokiego zaufania, przywództwo służebne oraz zarządzanie zmianami organizacyjnymi. W efekcie zyskujemy światowej klasy jakość, niezawodność, stabilność i bezpieczeństwo

coraz mniejszym kosztem i przy coraz mniejszym wysiłku. Jednocześnie zyskujemy przyspieszony przepływ i większą niezawodność w całym strumieniu wartości technologii, włącznie z zarządzaniem produktem, rozwojem, walidacją, operacjami IT i bezpieczeństwem informacji.

O ile powstanie DevOps może być postrzegane jako pochodna metodyki Lean, teorii ograniczeń i ruchu Toyota Kata, o tyle wiele osób uważa DevOps za logiczną kontynuację rozpoczętej w 2001 roku podróży z oprogramowaniem Agile.

NURT LEAN

Takie techniki, jak mapowanie strumienia wartości (ang. *Value Stream Mapping*), tablice kanban i całkowite produktywne utrzymanie ruchu (ang. *Total Productive Maintenance*), zostały skodyfikowane w systemie Toyota Production System w latach 80. W 1997 roku Lean Enterprise Institute rozpoczął badania nad zastosowaniami metodyki Lean do innych strumieni wartości, takich jak sektor usług i opieki zdrowotnej.

Dwa główne założenia metodyki Lean obejmują głęboko zakorzenione przekonanie, że **czas wymagany do przekształcenia surowców** na wyroby gotowe był najlepszym wskaźnikiem jakości, satysfakcji klienta i zadowolenia pracownika oraz że jednym z najlepszych predyktorów krótkich terminów realizacji była produkcja niewielkich partii.

Zasady metodyki Lean koncentrują się na tworzeniu wartości dla klienta za pośrednictwem systemów myślenia preferujących stałość celów, promujących myślenie naukowe, tworzenie przepływu, stosowanie zasad *pull* (dosł. „ciagnij”, w przeciwieństwie do *push* — dosł. „pchaj”), jakości u źródła (ang. *quality at the source*) oraz zarządzania z poszanowaniem każdego człowieka (ang. *leading with humility*).

MANIFEST AGILE

Manifest Agile został ogłoszony w 2001 roku przez 17 wiodących myślicieli w rozwoju oprogramowania. Osoby te dążyły do stworzenia prostego zbioru wartości i zasad w opozycji do stosowania złożonych procesów rozwoju oprogramowania, takich jak projektowanie kaskadowe lub metoda RUP (*Rational Unified Process*).

Jedną z kluczowych zasad było „dostarczanie działającego oprogramowania często — od kilku tygodni do kilku miesięcy, z preferencją krótszych ram czasowych”. Podkreślano znaczenie dążenia do partii o niewielkich rozmiarach oraz wydań przyrostowych zamiast złożonych wydań kaskadowych. Inne zasady podkreślały potrzebę tworzenia niewielkich, samomotywujących się zespołów pracujących w modelu zarządzania wysokiego zaufania.

Metodyce Agile przypisuje się zasługi znacznego zwiększenia produktywności wielu organizacji zajmujących się wytwarzaniem oprogramowania. Co ciekawe, zgod-

nie z tym, co opisano poniżej, wiele z kluczowych momentów w historii DevOps wystąpiło również w społeczności Agile lub na konferencjach poświęconych Agile.

INFRASTRUKTURA AGILE I RUCH NA RZECZ TEMPA WDROŻEŃ

Na konferencji Agile w 2008 roku w Toronto w Kanadzie Patrick Debois i Andrew Schafer przeprowadzili nieformalną sesję na temat stosowania zasad Agile do infrastruktury zamiast do kodu aplikacji. Mimo że byli jedynymi osobami prezentującymi tę ideę, szybko zyskali zwolenników wśród osób myślących w podobny sposób. Jedną z nich był współautor tej książki, John Willis.

Później, w 2009 roku, na konferencji Velocity Conference, John Allspaw i Paul Hammond mieli słynną prezentację „10 Deploys per Day: Dev and Ops Cooperation at Flickr”, w której opisali tworzenie wspólnych celów pomiędzy Dev i Ops oraz stosowanie praktyk ciągłej integracji po to, by instalacje stały się częścią codziennej pracy. Według osób uczestniczących w prezentacji wszyscy tam wiedzieli, że odbywa się coś ważnego, o historycznym znaczeniu.

Patricka Deboisa nie było na prezentacji, ale był tak podekscytowany pomysłem Allspawa i Hammonda, że w 2009 roku zorganizował w Belgii (gdzie mieszkał) pierwszą konferencję DevOpsDays. Na tej konferencji wymyślono termin „DevOps”.

NURT CIĄGŁEGO DOSTARCZANIA

Opierając się na koncepcji ciągłego budowania, testowania i integracji w rozwoju aplikacji, Jez Humble i David Farley rozszerzyli to pojęcie do ciągłego dostarczania, w którym zdefiniowano rolę „potoku wdrażania” w celu zagwarantowania stanu, w którym kod i infrastruktura są w stałej gotowości do wdrożenia, a każdy kod przesyłany do repozytorium może być bezpiecznie wdrożony do produkcji. Koncepcję tę zaprezentowano po raz pierwszy na konferencji Agile w 2006 roku. Dodatkowo została ona niezależnie opisana w 2009 roku przez Tim'a Fitz'a w poście na blogu zatytułowanym „Continuous Deployment”*.

* DevOps rozszerza również ideę infrastruktury jako kodu, której pionierami byli dr Mark Burgess, Luke Kanies i Adam Jacob. Koncepcja „infrastruktura jako kod” zakłada, że praca działu operacji jest zautomatyzowana i traktowana tak jak kod aplikacji, dzięki czemu można stosować nowoczesne praktyki rozwoju w całym stremieniu rozwoju. To stwarza warunki dla przepływu szybkich wdrożeń, obejmującego ciągłą integrację (jej pionierem był Grady Booch, włączając tę zasadę do 12 praktyk programowania ekstremalnego), ciągłe dostarczanie (pionierami byli Jez Humble i David Farley) oraz ciągłe wdrażanie (pionierskie były prace Etsy'ego, Wealthfronta i Erica Riesa w IMVU).

TOYOTA KATA

W 2009 Mike Rother napisał *Toyota Kata: Managing People for Improvement, Adaptiveness and Superior Results*, co było podsumowaniem jego 20-letniej pracy w celu zrozumienia i skodyfikowania systemu Toyota Production System. Mike Rother był jednym z praktykantów, którzy polecieli razem z kierownictwem firmy GM na odwiedziny w fabrykach firmy Toyota, i pomagał przy opracowaniu zbioru narzędzi Lean. Mike ze zdziwieniem zauważył, że żadna z firm, które zastosowały te praktyki, nie uzyskała wydajności obserwowanej w zakładach Toyota.

Doszedł do wniosku, że społeczność Lean pominęła najważniejszą praktykę, którą nazwał „kata usprawnień” (ang. *improvement kata*). Wyjaśnił, że każda organizacja ma procedury pracy, a kata usprawnień wymaga stworzenia struktury dla codziennych, zwyczajowych praktyk usprawniania pracy, ponieważ to codzienne praktyki poprawiają wyniki. Postęp w firmie Toyota był napędzany przez nieustanny cykl ustalania pożądanych przyszłych stanów, określanie tygodniowych docelowych wyników oraz ciągłe doskonalenie codziennej pracy.

Powyżej opisano historię DevOps oraz nurtów, z których DevOps czerpie inspiracje. Dalej w części I przyjrzymy się strumieniom wartości, sposobom stosowania zasad metodyki Lean do strumienia wartości technologii oraz trzem drogom: przepływowi, sprzężeniem zwrotnym oraz ciąglemu uczeniu się i eksperymentowaniu.

1

Agile, ciągłe dostarczanie i trzy drogi

W tym rozdziale zaprezentowano wstęp do teorii produkcji Lean, a także **trzy drogi** — zasady, na podstawie których można wywnioskować wszystkie obserwowane zachowania DevOps.

Skoncentrujemy się tutaj przede wszystkim na teorii i zasadach opisujących wiele dziesięcioleci doświadczeń firm produkcyjnych, organizacji wysokiej niezawodności, modeli zarządzania bazujących na wysokim zaufaniu i innych czynników i nurtów, z których wywodzą się praktyki DevOps. W pozostałych rozdziałach książki przedstawiono wynikowe konkretne zasady i wzorce oraz ich praktyczne zastosowanie do strumienia wartości technologii.

STRUMIEŃ WARTOŚCI PRODUKCJI

Jedną z podstawowych koncepcji Lean jest strumień wartości. Zdefiniujemy go najpierw w kontekście produkcji, a następnie dokonamy jego ekstrapolacji w celu zastosowania go do DevOps oraz strumienia wartości technologii.

Karen Martin i Mike Osterling w swojej książce *Value Stream Mapping: How to Visualize Work and Align Leadership for Organizational Transformation* zdefiniowali strumień wartości jako „sekwencję działań podejmowanych przez organizację w celu realizacji zlecenia klienta” lub „sekwencję działań wymaganych do zaprojektowania, wyprodukowania i dostarczenia towaru lub usługi do klienta z uwzględnieniem przepływów informacji i materiałów”.

W działalności produkcyjnej strumień wartości często jest łatwy do zaobserwowania: zaczyna się po otrzymaniu zamówienia klienta i dostarczeniu surowców do hali produkcyjnej. Aby umożliwić szybkie i przewidywalne terminy realizacji w każdym strumieniu wartości, zwykle kładzie się ciągły nacisk na tworzenie bezproblemowego i równomiernego przepływu pracy przy użyciu takich technik, jak niewielkie partie materiałów, zmniejszenie produkcji niezakończonej (ang. *Work In Process — WIP*), przeciwdziałanie przeróbkom, aby mieć pewność, że nie przekazujemy defektów do centrów pracy w dole strumienia, oraz ciągłe optymalizowanie systemu pod kątem osiągania globalnych celów.

STRUMIEŃ WARTOŚCI TECHNOLOGII

Te same zasady i wzorce, które umożliwiły szybki przepływ pracy w procesach fizycznych, odnoszą się również do prac nad technologią (oraz do wszystkich rodzajów pracy umysłowej). W infrastrukturze DevOps strumień wartości technologii zwykle określamy jako proces niezbędny do przekształcenia hipotezy biznesowej na korzystającą z technologii usługę, która dostarcza wartości dla klienta.

Dane wejściowe dla procesu to sformułowanie celu biznesowego, koncepcji, pomysłu lub hipotezy. Proces zaczyna się w chwili, gdy zaakceptujemy prace w rozwoju poprzez dodanie ich do zbioru zadań do wykonania (ang. *backlog*).

Od tego momentu zespoły deweloperskie stosujące typowe podejście Agile lub proces iteracyjny najczęściej przekształcają tę koncepcję na historyjki użytkowników oraz jakąś postać specyfikacji funkcji, które następnie są implementowane w kodzie aplikacji albo tworzonej usłudze. Następnie kod jest przekazywany do repozytorium kontroli wersji, gdzie każda zmiana jest testowana i integrowana z pozostałą częścią systemu oprogramowania.

Ponieważ wartość jest tworzona tylko wtedy, kiedy usługi są uruchamiane w produkcji, to musimy zadbać nie tylko o szybki przepływ, ale także o to, aby wdrożenie nie spowodowało chaosu i zakłóceń, takich jak przerwy w dostawie usług, słaba jakość usług czy też błędy w zabezpieczeniach lub zgodności z przepisami.

KONCENTRACJA NA CZASIE REALIZACJI WDRAŻANIA

W pozostałej części książki skoncentrujemy uwagę na czasie realizacji wdrożenia — podzbiorze strumienia wartości opisanego powyżej. Ten strumień wartości zaczyna się w chwili, gdy dowolny inżynier* w strumieniu wartości (który obejmuje dział rozwoju, validacji, operacji IT i bezpieczeństwa informacji) prześle zmianę do repozyto-

* Idąc dalej, określenie „inżynier” odnosi się do wszystkich inżynierów pracujących w strumieniu wartości, nie tylko deweloperów.

rium kontroli wersji, a kończy, gdy ta zmiana zostanie pomyślnie zastosowana w produkcji, zapewni wartość dla klienta oraz wygeneruje przydatne sprzężenie zwrotne i dane telemetryczne.

Pierwszy etap prac, który obejmuje projektowanie i rozwój, jest zbliżony do fazy rozwoju produktu w Lean (ang. *Lean Product Development*). Jest bardzo zróżnicowany i wysoce niepewny, często wymaga wysokiego stopnia kreatywności i wykonania pracy, która może nigdy nie zostać wykorzystana ponownie. W związku z tym czasy przetwarzania są na tym etapie bardzo zmienne. Dla odróżnienia drugi etap prac, który obejmuje testowanie i uruchamianie, jest zbliżony do produkcji Lean (ang. *Lean Manufacturing*). Wymaga on kreatywności i wiedzy oraz dążenia do tego, aby był przewidywalny i mechanistyczny. Celem jest uzyskanie efektów pracy przy jak najmniejszej zmienności (np. krótkie i przewidywalne czasy realizacji, prawie zerowa liczba defektów).

Zamiast dużych partii pracy przetwarzanych sekwencyjnie przez strumień wartości projektowania i rozwoju, a następnie strumień wartości testowania i uruchamiania (tak jak w dużym projekcie realizowanym według metodologii kaskadowej lub podczas rozwijania długowiecznych gałęzi funkcji) naszym celem jest realizowanie fazy testowania i uruchamiania w tym samym czasie, w którym są realizowane projektowanie i rozwój. Takie działania umożliwiają szybki przepływ i zapewniają wysoką jakość. Metoda kończy się sukcesem, gdy pracujemy niewielkimi partiami, budując jakość we wszystkich częściach strumienia wartości^{*}.

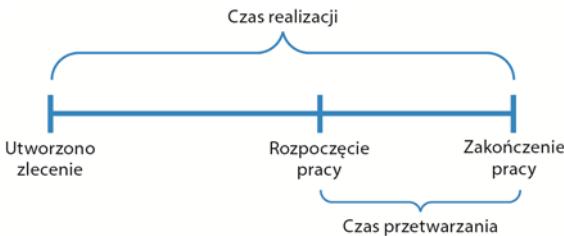
DEFINIOWANIE CZASU REALIZACJI A CZAS PRZETWARZANIA

W społeczności Lean czas realizacji (ang. *lead time*) jest jednym z dwóch parametrów powszechnie stosowanych do pomiaru wydajności w strumieniu wartości. Drugim jest czas przetwarzania — ang. *processing time* (czasami znany jako *czas zadania* — ang. *task time*)[†].

O ile czas realizacji zaczyna się w momencie złożenia zamówienia, a kończy się, gdy zamówienie zostanie zrealizowane, o tyle czas przetwarzania rozpoczyna się dopiero wtedy, gdy zaczynamy pracę nad zleceniem klienta — w szczególności pomijany jest czas, gdy zlecenie oczekuje w kolejce na przetworzenie (rysunek 2).

* W rzeczywistości w przypadku stosowania takich technik jak TDD testowanie jest wykonywane przed napisaniem jakiejkolwiek linijki kodu.

† Będziemy w tej książce faworyzowali termin „czas przetwarzania” z tych samych powodów, jakie wymienili Karen Martin i Mike Osterling: „Aby zminimalizować zamieszanie, będziemy unikali używania terminu *czas cyklu*, ponieważ istnieje dla niego kilka definicji równoznacznych, np. *czas przetwarzania* lub *częstotliwość wyników*”.

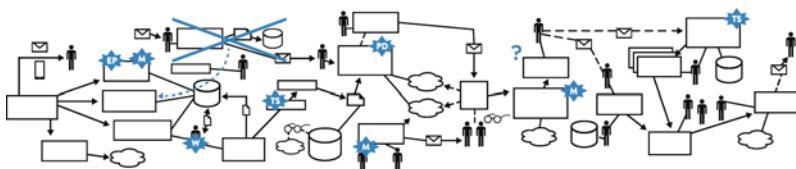


Rysunek 2. Czas realizacji a czas przetwarzania w operacji wdrażania

Ponieważ klient odczuwa czas realizacji, to zwykle na nim, a nie na czasie przetwarzania koncentrujemy wysiłek zmierzający do usprawniania procesu. Jednak proporcje pomiędzy czasem przetwarzania a czasem realizacji odgrywają rolę ważnego wskaźnika wydajności — uzyskanie szybkiego przepływu w krótkim czasie prawie zawsze wymaga skrócenia czasu oczekiwania pracy w kolejce.

TYPOWY SCENARIUSZ. CZASY REALIZACJI WDRAŻANIA WYMAGAJĄCE MIESIĘCY

W biznesie czasami występują sytuacje, gdy czasy realizacji wdrożeń wymagają miesięcy. Jest to szczególnie powszechnne w dużych, złożonych organizacjach, pracujących ze ścisłe powiązanymi, monolitycznymi aplikacjami, często z ograniczonymi środowiskami testów integracyjnych, długimi czasami wykonania testów i wdrożeń w środowisku produkcyjnym, wysokim stopniem uzależnienia od ręcznego testowania oraz wielu wymaganych procesów zatwierdzania. W takim przypadku strumień wartości może wyglądać tak, jak pokazano na rysunku 3:



Rysunek 3. Strumień wartości technologii z czasem wdrażania wynoszącym trzy miesiące
(źródło: Damon Edwards, „DevOps Kaizen”, 2015)

W przypadku długich czasów realizacji wdrożeń na prawie każdym etapie strumienia wartości wymagana jest „heroiczna” praca. Może się zdarzyć, że pod koniec projektu, gdy zachodzi potrzeba scalenia zmian wszystkich zespołów rozwojowych, okaże się, że nic nie działa. W efekcie uzyskujemy kod, który się nie buduje, i nie przechodzą żadne testy. Rozwiązywanie każdego problemu wymaga wielu dni lub tygodni prowadzenia dochodzenia w celu ustalenia, co spowodowało awarię i w jaki sposób trzeba ją naprawić. Dodatkowym efektem jest słaby poziom obsługi klientów.

IDEALNE ŚRODOWISKO DEVOPS. CZAS REALIZACJI WDRAŻANIA RZĘDU MINUT

W idealnym środowisku DevOps deweloperzy otrzymują szybkie i stałe sprzężenie zwrotne dotyczące ich pracy, co pozwala im szybko i samodzielnie implementować, integrować i weryfikować swój kod oraz wdrażać go w środowisku produkcyjnym (samodzielnie albo przez innych).

Można to osiągnąć przez iteracyjne wprowadzanie do repozytorium kontroli wersji niewielkich zmian, wykonywanie dla nich zautomatyzowanych testów badawczych oraz wdrażanie ich do produkcji. Dzięki temu można uzyskać wysoki stopień zaufania, że wprowadzone zmiany będą działać w środowisku produkcyjnym zgodnie z przeznaczeniem, a wszelkie problemy zostaną szybko wykryte i rozwiążane.

Najłatwiej to osiągnąć, gdy mamy architekturę, która jest modułowa, dobrze zhermetyzowana i preferuje luźne powiązania. Dzięki temu niewielkie zespoły mogą pracować w warunkach wysokiego stopnia autonomii, a awarie są niewielkie, dotyczą zamkniętego fragmentu i nie powodują globalnych zakłóceń.

W takim scenariuszu czas realizacji wdrożenia jest mierzony w minutach lub w najgorszym przypadku w godzinach. Uzyskana mapa strumienia wartości powinna mieć postać podobną do przedstawionej na rysunku 4:



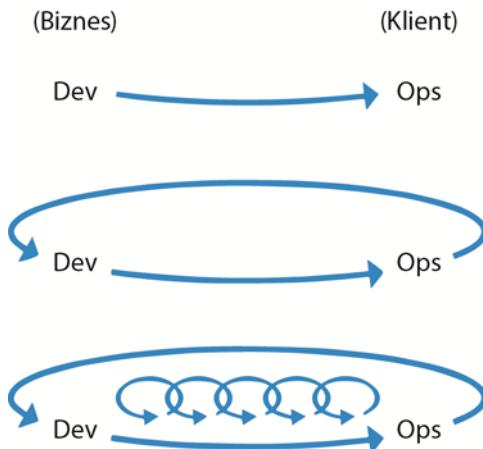
Rysunek 4. Strumień wartości technologii z czasem realizacji rzędu minut

OBSERWACJA WSKAŹNIKA %C/A JAKO PARAMETRU KONIECZNOŚCI WYKONYWANIA PRZERÓBEK

Trzecim kluczowym parametrem w strumieniu wartości technologii oprócz czasów realizacji i przetwarzania jest wskaźnik procentu C/A (ang. *complete and accurate* — dosł. „gotowe i dokładne”). Parametr ten odzwierciedla jakość wyników każdego etapu strumienia wartości. Karen Martin i Mike Osterling opisali ten parametr w następujący sposób: „Wartość wskaźnika % C/A można uzyskać, zadając klientom dalszego etapu strumienia pytanie o procent przypadków, gdy otrzymują pracę w postaci gotowej do wykorzystania, co oznacza, że mogą oni wykonywać swoją pracę bez konieczności korygowania dostarczonych informacji, dodawania brakujących danych lub wyjaśniania sformułowań, które mogłyby i powinny być czytelniejsze”.

TRZY DROGI. ZASADY LEŻĄCE U PODSTAW DEVOPS

W Projekcie Feniks zaprezentowano trzy drogi jako zbiór podstawowych zasad, z których wywodzą się wszystkie obserwowane zachowania i wzorce DevOps (rysunek 5).



Rysunek 5. Trzy drogi (źródło: Gene Kim, „The Three Ways: The Principles Underpinning DevOps”, blog IT Revolution Press, 9 sierpnia 2016, <http://itrevolution.com/the-three-ways-principles-underpinning-devops/>)

Pierwsza droga umożliwia szybki przepływ pracy od lewej do prawej, od działu rozwoju, poprzez dział operacji, do klienta. Aby zmaksymalizować przepływ, należy spowodować, aby praca była widoczna, zmniejszyć rozmiary partii i interwały pracy, tworzyć jakość poprzez przeciwdziałanie przekazywania defektów do centrów pracy w dole strumienia oraz stale optymalizować pracę, by osiągać globalne cele.

Dzięki przyspieszeniu przepływu przez strumień wartości technologii możemy skrócić czas realizacji zleceń wewnętrznych lub żądań klientów — zwłaszcza czas potrzebny do wdrożenia kodu w środowisku produkcyjnym. Dzięki temu podnosimy jakość pracy, jak również przepustowość i zwiększamy szanse na pokonanie konkurencji.

Praktyki stosowane do osiągnięcia tego celu obejmują procesy ciągłego budowania, integracji, testowania i wdrażania, tworzenie środowisk na żądanie, ograniczanie pracy w toku (WIP) oraz budowanie systemów i organizacji, które zapewniają bezpieczeństwo zmian.

Druga droga umożliwia szybki i stały przepływ informacji zwrotnych od prawej do lewej na wszystkich etapach strumienia wartości. Wzmocnienie sprzężenia zwrotnego jest potrzebne do przeciwdziałania wystąpieniu podobnych problemów w przyszłości lub wspomożenia procesu ich wykrywania oraz eliminowania skutków. W ten sposób tworzymy jakość u źródła i generujemy albo „osadzamy” wiedzę tam, gdzie jest ona potrzebna. To pozwala tworzyć coraz bezpieczniejsze systemy pracy, gdzie

problemy są znajdowane i rozwiązywane wcześnie, zanim wywołają katastrofalne skutki.

Dzięki dostrzeganiu problemów natychmiast po ich wystąpieniu oraz dzięki ich gromadzeniu do czasu znalezienia skutecznych środków zaradczych stale skracamy i wzmacniamy pętle sprzężeń zwrotnych — najważniejszy mechanizm prawie wszystkich nowoczesnych metodologii usprawniania procesów. To maksymalizuje możliwości uczenia się i doskonalenia organizacji.

Trzecia droga polega na stworzeniu generatywnej kultury wysokiego zaufania, która pozwala na dynamiczne, zdyscyplinowane i naukowe podejście do eksperymentowania i podejmowania ryzyka. To wspomaga proces czerpania nauki — zarówno z sukcesów, jak i porażek. Ponadto dzięki ciągłemu skracaniu i wzmacnianiu pętli sprzężenia zwrotnego tworzymy coraz bezpieczniejsze systemy pracy. Jesteśmy lepiej przygotowani na podejmowanie ryzyka i eksperymentowanie, co pomaga uczyć się szybciej niż konkurencja i zwyciężać na rynku.

W ramach trzeciej drogi projektujemy również nasz system pracy w taki sposób, aby można było pomnożyć efekty nowej wiedzy — przekształcić lokalne odkrycia na globalne usprawnienia. Bez względu na to, gdzie ktoś wykonuje swoją pracę, robi to z wykorzystaniem coraz większego, kolektywnego doświadczenia wszystkich osób w organizacji.

PODSUMOWANIE

W tym rozdziale opisaliśmy pojęcia strumieni wartości oraz czasu realizacji jako jedne z kluczowych parametrów skuteczności zarówno w strumieniu wartości produkcji, jak i technologii. Zapoznaliśmy się także z wysokopoziomowymi pojęciami każdej z trzech dróg — zasad, które leżą u podstaw DevOps.

W kolejnych rozdziałach opiszemy te zasady szczegółowo. Pierwszą jest **przepływ**. Należy dążyć do stworzenia szybkiego przepływu pracy w każdym strumieniu wartości — niezależnie od tego, czy dotyczy to produkcji, czy technologii. Praktyki umożliwiające szybki przepływ zostaną opisane w części III.

2

Pierwsza droga Zasady przepływu

W strumieniu wartości technologii praca zazwyczaj przepływa od działu rozwoju do działu operacji — pomiędzy obszarami funkcjonalnymi naszego biznesu a klientami. **Pierwsza droga** wymaga szybkiego i sprawnego przepływu pracy od działu rozwoju do operacji po to, aby szybko dostarczyć wartość dla klientów. Działania optymalizujemy pod kątem celu globalnego, a nie pod kątem takich celów lokalnych, jak współczynnik realizacji funkcji działu rozwoju, współczynniki wyszukiwania (poprawek) w dziale testów czy też parametry dostępności działu operacji.

Przepływ zwiększymy dzięki uwidocznieniu pracy, zmniejszeniu wielkości partii i przedziałów pracy oraz zbudowaniu jakości, a także przez zapobieganie przekazywaniu defektów do centrów pracy w dole strumienia. Poprzez przyspieszenie przepływu przez strumień wartości technologii można skrócić czas wymagany do zrealizowania żądań klientów wewnętrznych i zewnętrznych, co jeszcze bardziej poprawia jakość pracy, a jednocześnie czyni firmę bardziej zwinną i pozwala pokonać konkurencję.

Celem jest skrócenie czasu wymaganego do wdrożenia zmian w produkcji oraz poprawienie niezawodności i jakości usług. Wskazówek na temat tego, jak można to zrobić w strumieniu wartości technologii, można szukać w sposobie zastosowania zasad metodyki Lean do strumienia wartości produkcji.

SPRAW, ABY PRACA BYŁA WIDOCZNA

Znacząca różnica pomiędzy strumieniami wartości technologii i produkcji polega na tym, że praca w strumieniu wartości technologii jest niewidoczna. W odróżnieniu od procesów fizycznych w strumieniu wartości technologii nie można łatwo zobaczyć miejsc, gdzie jest utrudniony przepływ lub gdzie praca spiętrza się przed ograniczonymi centrami. Przenoszenie pracy pomiędzy centrami zwykle jest bardzo widoczne i powolne, ponieważ trzeba fizycznie przenieść zapasy.

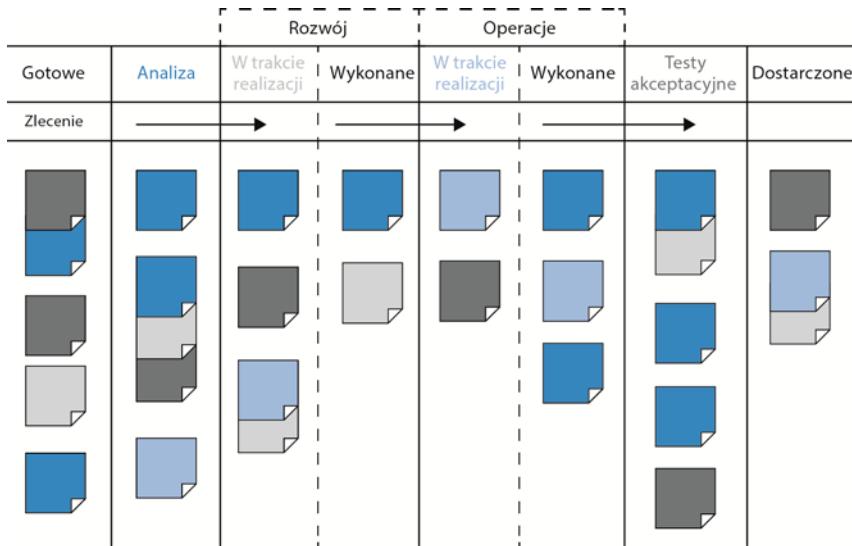
Jednak w pracy nad technologią przemieszczenie może odbywać się za jednym kliknięciem przycisku — na przykład poprzez wystawienie zlecenia pracy (ang. *ticket*) innemu zespołowi. Ponieważ jest to takie łatwe, to ze względu na niepełne informacje pracę można odbijać pomiędzy zespołami bez końca lub przekazywać do centrów pracy w dole strumienia z wadami pozostałymi całkowicie niewidocznymi do czasu dostarczenia obiecanych efektów do klienta lub kiedy nastąpi awaria aplikacji w środowisku produkcyjnym.

Aby ułatwić obserwację, czy praca przepływa sprawnie, oraz aby dostrzec miejsca, w których tworzą się kolejki lub martwe punkty, trzeba zadbać o zwiększenie widoczności pracy. Jedną z najlepszych metod osiągnięcia tego celu jest wykorzystanie wizualnych pulpitów, takich jak tablice kanban lub tablice planowania sprintu, gdzie możemy reprezentować wykonywane zadania za pomocą fizycznych albo elektronicznych kart. Praca wpływa z lewej strony (często jest pobierana z rejestru *backlog*), następnie jest przekazywana pomiędzy centrami pracy (reprezentowanymi przez kolumny) i kończy się po osiągnięciu prawej strony tablicy — zwykle w kolumnie oznaczonej „gotowe” lub „w produkcji”.

Dzięki tablicom kanban praca nie tylko staje się widoczna, ale także możemy nią zarządzać w taki sposób, aby jak najszybciej przepływała od lewej do prawej. Ponadto możemy zmierzyć czas realizacji — od momentu umieszczenia karty na tablicy do chwili, gdy zostanie przeniesiona do kolumny „Gotowe”.

W idealnej sytuacji tablica kanban powinna obejmować cały strumień wartości, a praca powinna być określana jako ukończona tylko wtedy, gdy dotrze do prawej strony tablicy (rysunek 6). Praca nie kończy się w momencie, gdy dział rozwoju zakończy implementację cechy funkcjonalnej. Zamiast tego kończy się w chwili, gdy zostanie pomyślnie przekazana do produkcji, dostarczając wartość dla klienta.

Dzięki umieszczeniu zadań każdego ośrodka w kolejkach oraz ich uwidoczenieniu wszyscy interesariusze mogą łatwiej określić priorytety w kontekście celów globalnych. Dzięki temu każdy ośrodek roboczy może koncentrować się na pracy o najwyższym priorytecie do czasu jej ukończenia, co zwiększa przepustowość.



Rysunek 6. Przykład tablicy kanban, obejmującej fazy wymagań, rozwoju, walidacji, testowania i w produkcji (źródło: David J. Andersen i Dominica DeGrandis, Kanban for ITOps, materiały na warsztaty szkoleniowe, 2012)

OGRANICZENIE PRACY W TOKU

W produkcji codzienna praca jest zazwyczaj dyktowana generowanym regularnie (np. codziennie, cotygodniowo) harmonogramem produkcji. Zadania do wykonania są wybierane na podstawie zamówień klientów, planowanych dat realizacji, dostępnych surowców itp.

W technologii praca jest zazwyczaj znacznie bardziej dynamiczna — jest to szczególnie widoczne w przypadku usług udostępnionych, gdzie zespoły muszą spełniać wymagania wielu różnych interesariuszy. W rezultacie codzienna praca jest zdominowana przez zadania o najwyższym priorytecie. Często żądania wykonania pilnych prac spływają przez wszystkie możliwe mechanizmy komunikacji, w tym system rejestracji zleceń, zgłoszenia awarii, wiadomości e-mail, połączenia telefoniczne, komunikatory oraz różne szczeble zarządzania.

Zakłócenia w produkcji są również bardzo widoczne i kosztowne. Często wymagają przerwania bieżącego zadania i pozostawienia niedokończonej pracy po to, aby rozpoczęć nową pracę. Ze względu na tak wysoki poziom wysiłku częste przerwy są bardzo niekorzystne.

Jednak przerywanie pracownikom technicznym jest łatwe, ponieważ konsekwencje są niewidoczne niemal dla nikogo, pomimo że negatywny wpływ na produktywność może być znacznie większy niż w produkcji. Na przykład inżynierowie przypisani do wielu projektów muszą przełączać się pomiędzy zadaniami, co wiąże się z kosztami ponownego ustanowienia kontekstu, jak również zmiany poznawczych zasad i celów.

Badania wykazały, że czas realizacji nawet tak prostych zadań, jak sortowanie figur geometrycznych, podczas pracy wielozadaniowej znacznie się wydłuża. Ponieważ prace w strumieniu wartości technologii są pod względem poznawczym znacznie bardziej skomplikowane niż sortowanie figur geometrycznych, to ujemny wpływ pracy wielozadaniowej na czas procesu jest znacznie większy.

Wielozadaniowość możemy ograniczyć, używając tablicy kanban do zarządzania pracą. Można skodyfikować i wymusić limity pracy w toku (ang. *Work In Progress — WIP*) dla każdej kolumny lub ośrodka pracy poprzez ograniczenie liczby kart, które mogą występować w jednej kolumnie.

Na przykład możemy ustawić limit WIP dla testowania na trzy karty. Gdy w kolumnie testowania są już trzy karty, nie można dodawać do niej nowych kart do czasu zrealizowania pracy reprezentowanej przez kartę lub do momentu, gdy karta zostanie usunięta z kolumny „w realizacji” i umieszczona z powrotem w kolejce (tzn. przesunięta do kolumny po lewej). Nie można wykonać żadnej pracy, zanim nie stworzy się jej reprezentacji w postaci karty pracy. To wzmacnia nacisk na to, aby wszystkie prace stały się widoczne.

Dominica DeGrandis, jeden z czołowych ekspertów w dziedzinie stosowania tablic kanban w strumieniu wartości DevOps zauważa, że „sterowanie rozmiarem kolejki [WIP] jest potężnym narzędziem zarządzania, ponieważ jest jednym z kilku wiodących wskaźników czasu realizacji — a w przypadku większości elementów pracy nie wiemy, jak długo potrwa ich realizacja, aż do czasu ukończenia”.

Ograniczanie WIP ułatwia również dostrzeżenie problemów uniemożliwiających ukończenie pracy*. Dzięki ograniczaniu WIP możemy odkryć, że nie mamy nic do zrobienia, ponieważ czekamy na kogoś innego. Chociaż możemy odczuwać pokusę do podjęcia nowej pracy (zgodnie z zasadą „lepiej robić coś niż nic”), to znacznie lepszym działaniem jest ustalenie przyczyny opóźnienia i próba rozwiązania tego problemu. Zła wielozadaniowość często występuje w sytuacji, gdy pracownicy są przypisywani do wielu projektów, co powoduje wiele problemów z priorytetyzacją.

Innymi słowy, jak zażartował David J. Andersen, autor książki *Kanban: Successful Evolutionary Change for Your Technology Business* — „przestań zaczynać, zacznij kończyć”.

ZMIEJSZENIE WIELKOŚCI PARTII

Innym kluczowym czynnikiem stworzenia sprawnego i szybkiego przepływu jest realizacja pracy w małych partiach. Przed rewolucją Lean w produkcji powszechną praktyką była produkcja dużych partii (albo wielu rozmiarów) — zwłaszcza w zadaniach,

* Taiichi Ohno porównał wymuszanie limitów WIP do spuszczania wody z rzeki zapasów w celu ujawnienia wszystkich przeszkoł, które uniemożliwiają szybki przepływ.

w których konfiguracja lub przełączanie pomiędzy zadaniami było czasochłonne i kosztowne. Na przykład produkcja dużych paneli nadwozia samochodowego wymaga ustawienia dużych i ciężkich matryc pras tłoczących — ten proces może trwać wiele dni. Ze względu na tak wysoki koszt przełączania naturalną tendencją jest wytłoczenie jak największej liczby paneli. Wytworzenie dużej partii ma zmniejszyć liczbę przełączeń.

Jednak produkcja dużych partii skutkuje olbrzymim poziomem WIP i wysokim poziomem zmienności przepływu, który narasta kaskadowo w całej fabryce. W efekcie uzyskuje się długi czas realizacji i niską jakość. W przypadku znalezienia problemu w nadwoziu całą partię trzeba zezłomować.

Oto jedna z kluczowych lekcji z Lean: w celu skrócenia czasu realizacji i zwiększenia jakości trzeba dążyć do ciągłego zmniejszania wielkości partii. Teoretycznym dolnym limitem rozmiaru partii jest przepływ pojedynczego elementu, gdzie każda operacja jest wykonywana na jednym urządzeniu naraz*.

Olbrzymie różnice pomiędzy stosowaniem dużych i małych partii można zobaczyć w prostej symulacji rozsyłania newslettera, opisanej w książce Jamesa P. Womacka i Daniela T. Jonesa *Lean Thinking: Banish Waste and Create Wealth in Your Corporation*.

Przypuśćmy, że mamy do wysłania 10 broszur i że wysyłka każdej broszury wymaga czterech kroków: złożenie listu, włożenie listu do koperty, zapieczętowanie koperty i naklejenie na niej znaczków.

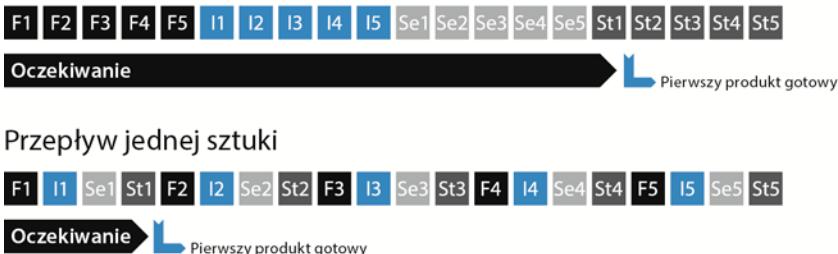
W przypadku stosowania strategii dużych partii (tzn. „produkcji masowej”) praca polegałaby na kolejnym wykonywaniu jednej operacji na każdej z 10 broszur. Innymi słowy, chcielibyśmy najpierw złożyć wszystkie 10 arkuszy papieru, następnie włożyć każdy z nich do koperty, następnie zapieczętować wszystkie 10 kopert i na koniec nakleić na nie znaczków.

W przypadku strategii małych partii (tzn. „przepływu jednej sztuki”) wszystkie czynności wymagane do zakończenia pracy z każdą broszurą są wykonywane sekwencyjnie, przed rozpoczęciem pracy nad kolejną broszurą. Inaczej mówiąc, składamy jeden arkusz papieru, wkładamy go do koperty, zaklejamy i naklejamy znaczek. Dopiero wtedy rozpoczynamy proces dla następnego arkusza papieru.

Różnica pomiędzy stosowaniem dużych i małych partii jest olbrzymia (rysunek 7). Założymy, że każda z czterech operacji trwa 10 sekund dla każdej z 10 kopert. W przypadku strategii partii dużych rozmiarów pierwsza ukończona koperta zostanie wyprodukowana dopiero po 310 sekundach.

* Termin rozmiaru partii i limitu WIP wynoszącego jeden jest również znany jako „partia o rozmiarze jeden” lub „przepływ 1 x 1”.

Duże partie



Rysunek 7. Symulacja „gry w koperty” (składanie, wkładanie, pieczętowanie i naklejanie znaczków) (źródło: Stefan Luyten, „Single Piece Flow: Why mass production isn't the most efficient way of doing «stuff»”, Medium.com, 8 sierpnia 2014, <https://medium.com/@stefanluyten/single-piece-flow-5d2c2bec845b#.9o7sn74ns>)

Co gorsza, założymy, że podczas operacji zaklejania koperty odkryjemy, że popełniliśmy błąd w pierwszym kroku — składaniu. W tym przypadku problem najwcześniej zostanie wykryty za 200 sekund i powstanie konieczność ponownego składania i wkładania do kopert całej partii 10 broszur.

W przypadku stosowania strategii małych partii pierwsza opatrzona znaczkiem koperta zostanie wyprodukowana po 40 sekundach, osiem razy szybciej niż w przypadku strategii dużych partii. Co więcej, jeśli popełniliśmy błąd w pierwszym kroku, to musimy powtórzyć pracę tylko dla jednej broszury w naszej partii. Stosowanie małych partii skutkuje mniejszym współczynnikiem pracy w toku, krótszym czasem realizacji, szybszym wykrywaniem błędów oraz mniejszą liczbą koniecznych przeróbek.

Negatywne skutki stosowania partii o dużych rozmiarach są tak samo istotne dla strumienia wartości technologii, jak dla produkcji. Rozważmy sytuację, gdy mamy roczny harmonogram wydań oprogramowania — sytuację, kiedy do produkcji wdrażana jest zmiana, nad którą działał rozwój pracował cały rok.

Podobnie jak w produkcji, to duże wydanie nagle tworzy wysoki poziom pracy w toku i ogromne zakłócenia we wszystkich ośrodkach pracy w dole strumienia. Skutkiem jest słaby przepływ i niska jakość wyników. To potwierdza nasze wspólne doświadczenie, że im większe zmiany wdrażamy do produkcji, tym trudniej zdiagnozować i naprawić błędy w produkcji oraz dłużej zajmuje wyeliminowanie powstałych problemów.

W poście w witrynie *Startup Lessons Learned* Eric Ries stwierdził, że „wielkość partii to jednostka używana do zmierzenia wielkości zbioru produktów pracy zamieszczanych pomiędzy poszczególnymi fazami rozwoju (albo etapami procesu DevOps). Dla oprogramowania wielkość partii najprościej zaobserwować w kodzie. Za każdym razem, gdy inżynier odda kod do repozytorium, przesyła partię określonej ilości pracy. Istnieje wiele technik kontrolowania tych partii, począwszy od niewielkich partii wymaganych do ciągłego wdrażania, do bardziej tradycyjnego rozwoju bazującego na gałęziach, kiedy cały kod, będący wynikiem pracy wielu deweloperów pracujących przez tygodnie lub miesiące, jest przetwarzany i integrowany”.

Odpowiednik przepływu jednej sztuki w strumieniu wartości technologii jest realizowany za pomocą ciągłego wdrażania, gdzie każda zmiana przesyłana do repozytorium kontroli wersji jest integrowana, testowana i wdrażana do produkcji. Praktyki, które to umożliwiają, zostały opisane w części IV.

ZMNIĘJSZENIE LICZBY PRZEŁĄCZEŃ

Jeśli w strumieniu wartości technologii czasy realizacji wdrożeń są rzędu miesięcy, to często przyczyną tego stanu są setki (a nawet tysiące) operacji wymaganych do przekazania kodu z repozytorium kontroli wersji do środowiska produkcyjnego. Przekazanie kodu przez strumień wartości wymaga pracy wielu działów nad różnymi zadaniami, takimi jak testy funkcjonalne, testy integracyjne, konfigurowanie środowiska, administracja serwerem, administracja pamięcią masową, obsługa sieci, równoważenie obciążenia i zapewnienie bezpieczeństwa informacji.

Za każdym razem, gdy praca jest przekazywana pomiędzy zespołami, potrzebne jest stosowanie różnych mechanizmów komunikacji: żądanie, specyfikowanie, sygnalizowanie, koordynacja, a często także określanie priorytetów, harmonogramowanie, usuwanie konfliktów, testowanie i weryfikowanie. Może to wymagać stosowania różnych systemów rejestracji zleceń lub zarządzania projektami; pisania dokumentów specyfikacji technicznych; komunikacji poprzez spotkania, wiadomości e-mail lub rozmowy telefoniczne, a także wykorzystywania udziałów systemu plików, serwerów FTP i stron wiki.

Jeśli będziemy polegali na zasobach współdzielonych pomiędzy różnymi strumieniami wartości (np. operacje skoncentrowane), to na każdym z tych etapów może się utworzyć kolejka zadań do wykonania. Czasy realizacji dla tych żądań są często tak długie, że występują problemy z wykonaniem pracy w wymaganym terminie.

Nawet w najlepszych okolicznościach każde przełączenie wiąże się z nieuchronnymi stratami wiedzy. Przy odpowiedniej liczbie przełączeń może dojść do całkowitej utraty kontekstu rozwiązywanego problemu lub świadomości obsługiwanych celów organizacyjnych. Na przykład administrator serwera może znaleźć zlecenie utworzenia kont użytkowników bez wiedzy na temat tego, do jakiej aplikacji lub usługi są one potrzebne, dlaczego należy je stworzyć, jakie są wszystkie zależności lub czy to rzeczywiście jest praca cykliczna.

Aby złagodzić te rodzaje problemów, staramy się zmniejszyć liczbę przełączeń, automatyzując znaczną część pracy lub reorganizując zespoły w taki sposób, aby mogły dostarczać wartości bezpośrednio do klienta zamiast być stale zależnymi od innych. W rezultacie zwiększymy przepływ poprzez skrócenie czasu, przez jaki praca musi cześć w kolejce, a także skrócenie czasu, który nie przynosi wartości dodanej (dodatek 4.).

STAŁE IDENTYFIKOWANIE I ELIMINOWANIE OGRANICZEŃ

Aby skrócić czasy realizacji i zwiększyć przepustowość, należy stale identyfikować ograniczenia systemu i dążyć do poprawy wydajności pracy. Dr Goldratt w książce *Beyond the Goal* stwierdził: „W każdym strumieniu wartości zawsze istnieje kierunek przepływu i zawsze jest jedno i tylko jedno ograniczenie; wszelkie usprawnienia, które nie dotyczą tego ograniczenia, są złudzeniem”. Jeśli usprawnimy pracę ośrodka, który jest umieszczony przed ograniczeniem, to praca będzie się spiętrzała przed wąskim gardłem jeszcze szybciej.

Z drugiej strony, jeśli usprawnimy pracę ośrodka zlokalizowanego za wąskim gardłem, to będzie on pozostawał bez pracy przez długie okresy oczekiwania. Jako rozwiązanie dr Goldratt zdefiniował „pięć kroków ogniskowania”:

- zidentyfikowanie ograniczenia systemu;
- podjęcie decyzji o sposobie wykorzystania ograniczenia systemu;
- podporządkowanie wszystkich pozostałych elementów decyzjom wymienionym powyżej;
- próba usunięcia ograniczenia systemu poprzez wzmacnienie ośrodka będącego wąskim gardłem;
- jeśli w poprzednich krokach ograniczenie zostało wyeliminowane, to wracamy do punktu pierwszego.

W typowych transformacjach DevOps, gdy przechodzimy od czasów realizacji mierzonych w miesiącach lub kwartałach do czasów realizacji mierzonych w minutach, ograniczenia zazwyczaj występują kolejno w następujących elementach:

- **Tworzenie środowiska** — nie możemy osiągnąć stanu instalacji na żądanie, jeśli zawsze musimy czekać kilka tygodni lub miesięcy na środowisko produkcyjne lub testowe. Środkiem zaradczym jest stworzenie środowisk na żądanie i całkowicie samoobsługowych, tak aby były zawsze dostępne, gdy ich potrzebujemy.
- **Instalacje kodu** — nie możemy osiągnąć stanu instalacji na żądanie, jeśli każde z wdrożeń produkcyjnych zajmuje kilka tygodni lub miesięcy (czyli każda instalacja wymaga ręcznego wykonania 1300 kroków, w których można popełnić błędy, i absorbuje 300 inżynierów). Środkiem zaradczym jest jak największe zautomatyzowanie instalacji. Celem powinna być całkowita automatyzacja, tak aby każda instalacja mogła być przeprowadzona samodzielnie przez każdego dewelopera.
- **Konfiguracja i uruchamianie testów** — nie można osiągnąć stanu instalacji na żądanie, jeśli każda operacja instalacji wymaga dwóch tygodni na skonfigurowanie środowiska testowego i zestawów danych oraz kolejnych czterech tygodni

na ręczne uruchomienie wszystkich testów regresji. Środkiem zaradczym jest zautomatyzowanie testów, tak aby można było przeprowadzić instalację bezpiecznie oraz współbieżnie. Dzięki temu tempo testowania może dorównać tempu pisania kodu.

- **Zbyt ścisła architektura** — nie możemy osiągnąć stanu instalacji na żądanie, jeśli z powodu zbyt ścisłej architektury każdorazowe dążenie do wprowadzenia zmian w kodzie zmusza inżynierów do udziału w spotkaniach komisji w celu uzyskania prawa do wprowadzenia zmian. Środkiem zaradczym jest stworzenie luźnej, sprężonej architektury, tak aby można było wprowadzać zmiany bezpiecznie i z większą autonomią, co zwiększa wydajność pracy deweloperów.

Po wyeliminowaniu wymienionych ograniczeń pozostałą ograniczenia związane z czasem tworzenia kodu lub właścicielami produktu. Ponieważ naszym celem jest umożliwienie małym zespołom programistów niezależnego tworzenia, testowania i wdrażania wartości u klientów szybko i niezawodnie, to właśnie ten element powinien być naszym ograniczeniem. Inżynierowie o wysokiej wydajności, bez względu na to, czy pracują w dziale rozwoju, validacji, operacji, czy bezpieczeństwa informacji, twierdzą, że ich celem jest dążenie do zmaksymalizowania produktywności deweloperów.

Gdy to jest naszym ograniczeniem, to jedynym limitem, jaki pozostaje, jest liczba tworzonych dobrych hipotez biznesowych oraz zdolność do tworzenia kodu niezbędnego do przetestowania tych hipotez z rzeczywistymi klientami.

Progresja ograniczeń wymieniona powyżej jest uogólnieniem typowego przekształcenia. Techniki identyfikowania ograniczeń w rzeczywistych strumieniach wartości, na przykład poprzez mapowanie strumienia wartości i wykonywanie odpowiednich pomiarów, zostały opisane w dalszej części tej książki.

ELIMINOWANIE TRUDNOŚCI I ODPADÓW W STRUMIENIU WARTOŚCI

Shigeo Shingo, jeden z pionierów systemu Toyota Production System, twierdził, że odpady stanowią największe zagrożenie dla rentowności przedsiębiorstwa. Powszechnie stosowana definicja odpadu w metodyce Lean brzmi: „odpad to wykorzystanie jakichkolwiek materiałów lub zasobów poza tym, czego wymaga klient i za co jest gotów zapłacić”. Shingo zdefiniował siedem głównych rodzajów odpadów w produkcji: zapasy, nadprodukcja, nadmiarowe przetwarzanie, transport, oczekiwanie, przemieszczanie i wady.

W bardziej nowoczesnych interpretacjach Lean zauważono, że „eliminowanie odpadów” może mieć kontekst poniżający i antyhumanistyczny. Zamiast niego celem powinno być zmniejszenie trudności i uciążliwości codziennej pracy przez ciągłe uczenie się w celu osiągnięcia celów organizacji. W pozostałej części tej książki termin

„odpady” będzie rozumiany zgodnie z tą bardziej nowoczesną definicją, ponieważ bardziej pasuje do ideałów DevOps i pożądanych rezultatów.

Mary i Tom Poppendieckowie w książce *Lean Software Development: From Concept to Cash* opisują odpady i trudności w strumieniu rozwoju oprogramowania jako wszystko to, co powoduje opóźnienia odczuwane przez klienta — na przykład działania, które mogą być pominięte bez wpływu na wynik.

W książce wymieniono następujące kategorie odpadów i trudności:

- **Prace wykonane częściowo** — oznacza to wszelkie prace w strumieniu wartości, które nie zostały dokonane (np. dokumenty wymagań lub żądania zmian, które jeszcze nie zostały zweryfikowane), oraz prace oczekujące w kolejce (np. czekające na weryfikację przez dział walidacji lub obsługę zlecenia przez administratora serwera). Praca wykonana częściowo z biegiem czasu staje się przestarczała i traci wartość.
- **Dodatkowe procesy** — wszelkie prace wykonywane w procesie, które nie dodają wartości dla klienta. Może to obejmować dokumentację nieużywaną w centrach pracy w dole strumienia albo weryfikacje lub zgody, które nie dodają wartości do wyniku. Dodatkowe procesy wiążą się z dodatkowym wysiłkiem i wydłużają czas realizacji.
- **Dodatkowe funkcje** — wbudowane w usługę funkcje, które nie są wymagane przez organizację lub klienta (tzw. „pozłacanie”). Dodatkowe funkcje zwiększą złożoność testowania i utrudniają zarządzanie.
- **Przełączanie pomiędzy zadaniami** — sytuacja, kiedy pracownicy są przypisani do wielu projektów i strumieni wartości, co zmusza ich do przełączania kontekstu i zarządzania zależnościami. To wprowadza do strumienia wartości dodatkowy wysiłek i czas.
- **Oczekiwanie** — wszelkie opóźnienia w pracy wymagające oczekiwania na możliwość ukończenia bieżącego zadania. Opóźnienia wydłużają czas cyklu i uniemożliwiają klientom uzyskanie wartości.
- **Ruch** — nakłady pracy niezbędne do przeniesienia informacji lub materiałów z jednego ośrodka roboczego do innego. Odpad spowodowany ruchem może być tworzony w sytuacji, gdy ludzie, którzy muszą się często komunikować, nie pracują w tej samej lokalizacji. Przekazywanie pracy także powoduje odpady związane z ruchem i często wymaga dodatkowej komunikacji w celu rozpoznania niejasności.
- **Defekty** — niepoprawne, niedostępne lub niejasne informacje, materiały lub produkty tworzą odpady, ponieważ wymagają wysiłku w celu rozwiązania problemów. Im dłuższy czas pomiędzy powstaniem defektu a jego wykryciem, tym bardziej skomplikowane jest jego wyeliminowanie.

- **Prace niestandardowe lub ręczne** — korzystanie z rozwiązań niestandardowych lub pracy ręcznej innych osób — na przykład korzystanie z wymagających ręcznej interwencji serwerów, środowisk testowych i konfiguracji. W idealnej sytuacji wszelkie zależności od działu operacji powinny być zautomatyzowane, samoobsługowe i dostępne na żądanie.
- **Heroizm** — aby organizacja mogła osiągnąć swoje cele, pojedyncze osoby i zespoły są stawiane w sytuacji, w której muszą realizować nadmierne obciążenia (np. konieczność pracy w nocy i problemy podczas każdej instalacji oprogramowania generujące setki zleceń pracy)*.

Naszym celem jest, aby te odpady i trudności — gdziekolwiek heroizm staje się niezbędny — stały się widoczne oraz aby systematycznie robić to, co jest potrzebne do złagodzenia lub wyeliminowania tych obciążzeń, i osiągnąć szybki przepływ.

PODSUMOWANIE

Poprawienie przepływu w strumieniu wartości technologii jest niezbędne do osiągnięcia pozytywnych wyników DevOps. Robimy to poprzez uwidocznienie pracy, ograniczenie nakładów pracy w toku, zmniejszenie wielkości partii i liczby przełączeń, stałą identyfikację i eliminowanie ograniczeń oraz usuwanie trudności w codziennej pracy.

Konkretnie praktyki umożliwiające szybki przepływ w strumieniu wartości DevOps zaprezentowano w części IV. W następnym rozdziale przedstawimy drugą drogę: zasady sprzężeń zwrotnych.

* Mimo, że „heroizmu” nie wymieniono na liście kategorii odpadów Poppendiecka, to wymieniamy go w tej książce ze względu na powszechność występowania tej sytuacji — zwłaszcza w odniesieniu do współdzielonych usług działu operacji.

3

Druga droga Zasady sprzężenia zwrotnego

Podczas gdy **pierwsza droga** opisuje zasady umożliwiające szybki przepływ pracy od lewej do prawej, **druga droga** opisuje zasady, które wspomagają szybkie uzyskiwanie stałego strumienia informacji zwrotnych — od prawej do lewej — na wszystkich etapach strumienia wartości. Celem jest stworzenie bezpieczniejszego i bardziej odpornego systemu pracy.

Jest to szczególnie ważne podczas pracy w złożonych systemach, gdy najwcześniejsza okazja do wykrycia i skorygowania błędów zazwyczaj występuje dopiero w chwili, gdy nastąpi zdarzenie o katastrofalnych skutkach — np. dojdzie do zranienia pracownika produkcji lub awarii reaktora jądrowego.

W firmach technicznych praca prawie w całości odbywa się w złożonych systemach, z wysokim ryzykiem katastrofalnych konsekwencji. Podobnie jak w produkcji, często odkrywamy problemy dopiero przy okazji wystąpienia dużych awarii — np. masowa przerwa w produkcji lub naruszenie zasad bezpieczeństwa skutkujące kradzieżą danych klienta.

System pracy stanie się bezpieczniejszy dzięki utworzeniu szybkiego, częstego przepływu informacji wysokiej jakości przez cały strumień wartości organizacji. Strumień ten powinien obejmować pętle sprzężenia zwrotnego i sprzężenia w przód. Dzięki temu możliwe jest wykrywanie i łagodzenie problemów wtedy, gdy mają mniejszą skalę, są tańsze i łatwiejsze do rozwiązania. W ten sposób można uniknąć problemów, zanim spowodują katastrofę, i utworzyć w organizacji system edukacji,

który można zintegrować w przyszłej pracy. Jeśli wystąpią awarie i wypadki, traktujemy je jako możliwość nauki, a nie jako powód do wskazywania winnych i ich karania. Zanim opowiemy, jak osiągnąć taki stan, przyjrzyjmy się naturze złożonych systemów i zastanówmy się, w jaki sposób uczynić je bezpieczniejszymi.

BEZPIECZNA PRACA W ZŁOŻONYCH SYSTEMACH

Jedną z charakterystycznych cech złożonego systemu jest brak możliwości dostrzeżenia go przez pojedynczą osobę jako całości i zrozumienia, jak działają wszystkie jego części. Złożone systemy zazwyczaj cechuje duża liczba wewnętrznie połączonych i ściśle ze sobą sprężonych komponentów, a zachowania systemu nie można wyjaśnić jedynie w kategoriach zachowania jego składników.

Dr Charles Perrow, który badał awarię w elektrowni jądrowej Three Mile Island, zauważył, że było niemożliwe, aby jakakolwiek osoba mogła zdawać sobie sprawę ze sposobu zachowania reaktora jądrowego we wszystkich możliwych okolicznościach i tego, jak może dojść do jego awarii. Gdy występował problem w jednym komponencie, to odizolowanie go od innych komponentów było trudne, a problem szybko rozprzestrzeniał się po ścieżkach najmniejszego oporu.

Dr Sidney Dekker, który skodyfikował również niektóre kluczowe elementy układu zabezpieczeń, zaobserwował inną cechę złożonych systemów: wykonywanie tej samej rzeczy dwukrotnie nie doprowadzi do tych samych albo dających się przewidzieć rezultatów. Jest to cecha, która sprawia, że statyczne listy kontrolne i najlepsze praktyki, chociaż są cenne, to są niewystarczające do tego, by zapobiec wystąpieniu katastrofy (dodatek 5.).

Ze względu na to, że w złożonych systemach, zarówno w produkcji, jak i technologii, awarie są nieuniknione, należy zaprojektować bezpieczny system pracy, tak aby można było pracować bez obaw i z przekonaniem, że wszelkie błędy zostaną wykryte szybko, na długo zanim spowodują katastrofalne skutki lub wywrą negatywny wpływ na klienta — na przykład nastąpi zranienie pracownika lub ujawnią się wady produktu.

Po zdekodowaniu w swojej pracy doktorskiej w Harvard Business School mechanizmów przyczynowych systemu Toyota Product System dr Steven Spear stwierdził, że zaprojektowanie całkowicie bezpiecznych systemów prawdopodobnie wykracza poza nasze umiejętności, ale możemy przyczynić się do bezpieczniejszego ich funkcjonowania w złożonych organizacjach, gdy zostaną spełnione następujące cztery warunki^{*}:

- Złożone prace są zarządzane tak, aby były ujawniane problemy w projektowaniu i działaniu.

* Dr Spear rozszerzył swoją pracę o objaśnienie długotrwałych sukcesów innych organizacji, takich jak sieci dostawców Toyoty Alcoa oraz program generowania energii jądrowej w US Navy.

- Problemy są wykrywane i natychmiast rozwiązywane, co sprzyja szybkiemu tworzeniu nowej wiedzy.
- Nowa wiedza lokalna jest wykorzystywana globalnie w całej organizacji.
- Liderzy wyznaczają nowych liderów, którzy nieustannie rozwijają nowe możliwości.

Złożony system musi spełniać wszystkie wymienione powyżej warunki. W następnych podrozdziałach opiszemy dwie pierwsze cechy oraz ich znaczenie. Opowiem też, w jaki sposób je osiągnięto w innych dziedzinach oraz jakie praktyki ułatwiają ich włączenie do strumienia wartości technologii (trzecią i czwartą cechę opisano w rozdziale 4.).

ZAUWAŻ PROBLEMY NATYCHMIAST PO ICH WYSTĄPIENIU

W bezpiecznym systemie pracy trzeba stale testować przyjęte założenia projektowania i eksploatacji. Celem jest zwiększenie w systemie przepływu informacji z jak największej liczby obszarów. Informacje powinny docierać wcześniej, płynąć szybciej, taniej oraz zawierać możliwie jak najwięcej objaśnień związków pomiędzy przyczynami a skutkami. Im więcej założeń uda nam się obalić, tym szybciej znajdziemy i rozwiążemy problemy, co przyczyni się do poprawy elastyczności, zwinności oraz zdolności do uczenia się i wprowadzania innowacji.

Aby to osiągnąć, należy utworzyć pętle sprzężenia zwrotnego i sprzężenia w przód. Dr Peter Senge w swojej książce *The Fifth Discipline: The Art & Practice of the Learning Organization* określił sprzężenia zwrotne jako kluczowy element uczenia się organizacji i myślenia systemowego. Pętle sprzężenia zwrotnego i sprzężenia w przód pozwalają wzmacnić komponenty w ramach systemu i ułatwiają ich współdziałanie.

Brak skutecznych informacji zwrotnych w produkcji często przyczynia się do poważnych problemów jakości i bezpieczeństwa. W jednym z dobrze udokumentowanych przypadków w fabryce General Motors Fremont nie było żadnych skutecznych procedur wykrywania problemów podczas procesu montażu. Nie istniały także jawne procedury postępowania w przypadku wykrycia problemów. W rezultacie zdarzały się przypadki odwrotnego podłączania silników, montowania samochodów bez kierownic lub opon. Zdarzało się nawet, że samochody musiały być holowane z linii montażowej, ponieważ nie można było ich uruchomić.

Natomiast w trybie produkcji wysokiej wydajności występuje szybki, częsty przepływ informacji przez cały strumień wartości — każda operacja robocza jest mierzona i monitorowana, a wszelkie wady lub znaczące odchylenia są szybko znajdowane i są realizowane działania zaradcze. To są podstawy osiągania wysokiej jakości, bezpieczeństwa, ciągłego kształcenia się i doskonalenia.

W strumieniu wartości technologii często osiągamy słabe wyniki z powodu braku szybkich informacji zwrotnych. Na przykład w projekcie oprogramowania realizowanym metodą kaskadową kod może być rozwijany nawet przez cały rok bez uzyskania opinii na temat jakości. Taki stan może trwać aż do rozpoczęcia fazy testowania — lub co gorsza, do chwili, gdy oprogramowanie zostanie opublikowane. Gdy informacje zwrotne napływają z opóźnieniem i rzadko, to uniknięcie niepożądanych efektów jest znacznie trudniejsze.

Należy dążyć do stworzenia szybkich pętli sprzężenia zwrotnego i sprzężenia w przód wszędzie tam, gdzie jest wykonywana praca, na wszystkich etapach strumienia wartości technologii obejmującego zarządzanie produktem, rozwój, validację, bezpieczeństwo informacji i działania operacyjne. Działania te obejmują tworzenie procesów zautomatyzowanej komplikacji, integracji i testowania, tak aby można było natychmiast wykryć stan wprowadzenia takiej zmiany, która wytrąca produkt ze stanu produktu prawidłowo funkcjonującego i gotowego do wdrożenia.

Tworzymy również całościowe mechanizmy telemetrii, dzięki którym można zobaczyć, jak działają wszystkie komponenty systemu w środowisku produkcyjnym. W ten sposób możemy szybko wykryć stan, kiedy przestają działać zgodnie z oczekiwaniemi. Telemetria pozwala również zmierzyć, czy są osiągane pożądane cele. W idealnej sytuacji promieniuje na cały strumień wartości, więc można zobaczyć, jak podejmowane działania wpływają na inne części systemu jako całości.

Pętle sprzężenia zwrotnego nie tylko umożliwiają szybkie wykrywanie i korygowanie wad, ale również pokazują, jak zapobiegać występowaniu podobnych problemów w przyszłości. W ten sposób zwiększa się jakość i bezpieczeństwo systemu pracy, a w organizacji tworzy się klimat sprzyjający uczeniu się.

Jak powiedziała Elisabeth Hendrickson, wiceprezes firmy Engineering Pivotal Software, Inc. oraz autorka książki *Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing*: „Odkąd zaczeliśmy stosować zasady inżynierii jakości, mogę opisać swoją pracę jako »tworzenie cykli sprzężeń«. Sprzężenia zwrotne mają kluczowe znaczenie, ponieważ to one pozwalają nam kierować. Musimy stale weryfikować zgodność pomiędzy potrzebami klientów, naszymi zamiarami a implementacjami. Testowanie to tylko jeden z rodzajów sprzężeń”.

SWARMING. ROZWIAZYWANIE PROBLEMÓW W CELU BUDOWANIA NOWEJ WIEDZY

Oczywiście samo wykrycie nieoczekiwanych sytuacji nie wystarcza. Gdy pojawią się problemy, trzeba zastosować technikę zwaną **swarmingiem** (od ang. *swarm* — „rój”), polegającą na zmobilizowaniu wszystkich osób, które są potrzebne do zażegnania kryzysu.

Według dr. Speara celem swarmingu jest opanowanie problemów, zanim zdążyła się rozprzestrzenić, oraz zdiagnozowanie i naprawienie złej sytuacji, tak by nie mogła się powtórzyć. Mówi, że „tak budowana jest coraz głębsza wiedza na temat sposobu zarządzania systemami w celu realizacji naszych zadań. Następuje konwersja nieuniknionej początkowej niewiedzy w wiedzę”.

Wzorcem stosowania tej zasady jest mechanizm *linki Andon* montowanej w fabrykach firmy Toyota. W każdym centrum produkcyjnym jest linka, za którą może pociągnąć każdy pracownik i menedżer, gdy coś idzie nie tak, jak powinno — na przykład kiedy jakaś część jest wadliwa, kiedy potrzebne części są niedostępne lub nawet wtedy, gdy praca trwa dłużej, niż zostało to udokumentowane*.

Gdy ktoś pociągnie za linkę Andon, powiadamiany jest lider odpowiedzialnego zespołu i natychmiast rozpoczyna pracę w celu rozwiązania problemu. Jeśli problemu nie można rozwiązać w określonym czasie (np. w ciągu 55 sekund), następuje zatrzymanie całej linii produkcyjnej. Dzięki temu można zmobilizować wszystkich pracowników do udziału w rozwiązywaniu problemu do czasu zastosowania skutecznego środka zaradczego.

Zamiast opracowywania obejścia problemu lub planowania wprowadzenia poprawki, „kiedy będzie więcej czasu”, tworzymy „rój”, aby rozwiązać problem natychmiast — jest to zachowanie niemal całkowicie odwrotne do opisanego wcześniej zachowania w fabryce GM Fremont. Rozwiązywanie problemu za pomocą techniki swarmingu jest niezbędne z następujących powodów:

- Zapobiega rozprzestrzenianiu się problemu w dół strumienia, gdzie koszty i wysiłek wymagany do jego rozwiązania zwiększą się wykładniczo, a dług techniczny może stawać się coraz większy.
- Zapobiega rozpoczęciu pracy nad nowym zadaniem, co prawdopodobnie spowodowałoby wprowadzenie do systemu nowych błędów.
- Jeśli problem nie zostanie rozwiązany, ośrodek pracy może mieć ten sam problem w następnej operacji (np. 55 sekund później), co będzie wymagało wprowadzenia dodatkowych poprawek i dodatkowej pracy (dodatek 6.).

Praktyka „tworzenia roju” może wydawać się sprzeczna z powszechną praktyką zarządzania, ponieważ rozmyślnie pozwalamy na to, aby lokalny problem zakłócał operacje globalne. Jednakże swarming umożliwia uczenie się. Zapobiega utracie kluczowych informacji ze względu na zatarcie pamięci lub zmianę okoliczności. Jest to szczególnie istotne w złożonych systemach, gdzie wiele problemów pojawia się z powodu pewnych nieoczekiwanych, osobliwych interakcji ludzi, procesów, produktów, miejsc i okoliczności — w miarę upływu czasu staje się niemożliwe odtworzenie dokładnie takiego stanu, jaki był w czasie wystąpienia problemu.

* W niektórych fabrykach Toyoty zamiast linki Andon stosowany jest przycisk Andon.

Jak zauważa dr Spear, swarming jest częścią „zdyscyplinowanego cyklu rozpoznawania, diagnozowania i rozwiązywania problemów w czasie rzeczywistym (w języku związanym z produkcją określa się to jako środki zaradcze lub korygujące). Jest to rodzaj cyklu Shewharta — planowanie, działanie, sprawdzanie, zastosowanie (ang. *PDCA* — *plan, do, check, act*) — spopularyzowanego przez Williama Edwardsa Deminga, ale w wersji bardzo przyspieszonej”.

Tylko dzięki technice swarmingu, pozwalającej wykryć niewielkie problemy we wczesnej fazie cyklu, można odwrócić skutki błędów, zanim dojdzie do katastrofy. Innymi słowy, gdy reaktor jądrowy zaczyna się topić, to jest już za późno na przeciwdziałanie najgorszemu.

Aby umożliwić szybkie sprzężenia zwrotne w strumieniu wartości technologii, trzeba opracować odpowiedniki linki Andon oraz reakcji za pomocą swarmingu. Wymaga to stworzenia kultury, w której pociąganie za linkę Andon w sytuacji, gdy wystąpi incydent w produkcji albo zostaną popełnione błędy we wczesnej fazie strumienia wartości (ktoś wprowadzi zmianę, która spowoduje awarię procesu ciągłej komplikacji lub testowania), jest bezpieczne, a nawet pożądane.

Gdy powstaną warunki, w których ktoś pociąga za linkę Andon, gromadzimy się w celu rozwiązyania problemu i nie przystępujemy do realizacji nowych zadań, dopóki problem nie zostanie rozwiązany*. Zapewnia to szybkie informacje zwrotne dla wszystkich osób w całym strumieniu wartości (zwłaszcza dla tej osoby, która spowodowała awarię systemu), umożliwia szybkie wyizolowanie i zdiagnozowanie problemu oraz przeciwdziała dalszym komplikacjom, które mogą przyczynić się do zacierania związku pomiędzy przyczyną a skutkiem.

Powstrzymanie się od realizacji nowych zadań umożliwia ciągłą integrację i wdrażanie, które stanowią podstawę przepływu jednej sztuki w strumieniu wartości technologii. Wszystkie zmiany, które przechodzą przez testy ciągłej komplikacji i budowania, są wdrażane do produkcji, a jakiekolwiek zmiany powodujące niepowodzenie testów inicjują pociągnięcie za linkę Andon i zastosowanie swarmingu w celu rozwiązywania problemu.

JAKOŚĆ BLIŻEJ ŹRÓDŁA

Sposób reagowania na awarie i incydenty może nieumyślnie przyczyniać się do promowania systemów pracy stwarzających zagrożenia. W złożonych systemach wprowadzenie dodatkowych czynności kontrolnych i procesów zatwierdzania w rzeczywistości

* Zdziwiające jest to, że zmniejszała się liczba przypadków wykorzystania linki Andon, menedżerowie fabryk łagodzili warunki jej użycia, aby na nowo uzyskać wzrost liczby jej zastosowań i kontynuować trend uczenia się i usprawniania procesu oraz wykrywania jeszcze słabszych sygnałów awarii.

zwiększa prawdopodobieństwo przyszłych awarii. Wraz ze wzrostem odległości ośrodka podejmowania decyzji od miejsca, w którym jest wykonywana praca, maleje skuteczność procesów zatwierdzania. To nie tylko obniża jakość podejmowanych decyzji, ale także wydłuża czas cyklu, a tym samym osłabia siłę sprzężenia zwrotnego pomiędzy przyczynami a skutkami oraz zmniejsza zdolność do uczenia się na podstawie sukcesów i porażek^{*}.

Właśnie tę można zaobserwować nawet w mniejszych i mniej skomplikowanych systemach. Biurokratyczne systemy zarządzania i kontroli z góry na dół zwykle okazują się nieskuteczne ze względu na to, że rozbieżność pomiędzy tym, „któ powinien coś zrobić”, a tym, „któ rzeczywiście coś robi”, jest zbyt duża ze względu na niewystarczającą przejrzystość i terminowość.

Oto kilka przykładów nieskutecznych mechanizmów kontroli jakości:

- Wymaganie od innego zespołu ręcznego wykonywania żmudnych, stwarzających możliwości popełnienia błędu zadań, które mogłyby być łatwo zautomatyzowane i zrealizowane zgodnie z potrzebami przez zespół, który potrzebuje wykonania tej pracy.
- Wymaganie zatwierdzania od zapracowanych osób, które pracują w odległej lokalizacji; zmuszanie ich do podejmowania decyzji bez odpowiedniej wiedzy na temat zadania lub potencjalnych jego implikacji lub tylko do zatwierdzenia decyzji poprzez postawienie pieczętki.
- Tworzenie dużych ilości dokumentacji zawierającej wątpliwe szczegóły, które wkrótce po napisaniu stają się przestarzałe.
- Powierzanie dużych partii pracy zespołom i specjalnym komisjom w celu ich zatwierdzenia i przetwarzania, a następnie oczekiwanie na odpowiedź.

Powinniśmy dążyć do tego, aby wszystkie osoby w strumieniu wartości wyszukiwały i rozwiązywały problemy w obszarze swoich kompetencji w ramach swojej codziennej pracy. W ten sposób zamiast oddalać odpowiedzialność za jakość i bezpieczeństwo oraz podejmowanie decyzji, przekazujemy ją do miejsca, w którym jest wykonywana praca.

Stosujemy wzajemne przeglądanie proponowanych zmian w celu uzyskania potrzebnego potwierdzenia, że wprowadzone zmiany będą działały zgodnie z przeznaczeniem.

* W XVIII wieku rząd brytyjski zastosował spektakularny przykład biurokratycznego zarządzania z góry na dół, które okazało się bardzo nieskuteczne. W tym czasie Georgia nadal była kolonią. Pomimo faktu, że brytyjski rząd był oddalony o trzy tysiące mil i brakowało mu wiedzy z pierwszej ręki na temat lokalnych gruntów — składu chemicznego, skalistości, topografii, dostępu do wody i innych warunków — to próbował zaplanować tam całą gospodarkę rolną. Wyniki tej próby były marne. Spośród 13 kolonii Georgia miała najniższy poziom życia i najmniejszą populację.

W maksymalnym możliwym stopniu automatyzujemy testy zwykle wykonywane przez działы walidacji lub bezpieczeństwa informacji. Deweloperzy nie muszą zlecać lub planować uruchamiania testów. Testy mogą być wykonane na żądanie, dzięki czemu deweloperzy mogą szybko przetestować swój kod, a nawet samodzielnie wdrożyć swoje zmiany do produkcji.

W ten sposób odpowiedzialność za jakość staje się obowiązkiem wszystkich pracowników, a nie tylko wyodrębnionego działu. Bezpieczeństwo informacji nie jest tylko zadaniem działu bezpieczeństwa informacji, podobnie dostępność nie jest jedynie zadaniem działu operacyjnego.

Przekazanie deweloperom współodpowiedzialności za jakość systemów, które budują, nie tylko poprawia wyniki, ale także przyspiesza naukę. Jest to szczególnie ważne dla deweloperów, ponieważ ich zespoły są zazwyczaj najbardziej oddalone od klientów. Jak zauważył Gary Gruver: „Jest niemożliwością, aby deweloper się czegokolwiek nauczył, gdy ktoś krzyczy na niego z powodu czegoś, w czym popełnił błąd sześć miesięcy wcześniej — właśnie dlatego sprzężenie zwrotne powinno docierać maksymalnie jak najwcześniej, z opóźnieniem rzędu minut, a nie miesięcy”.

OPTYMALIZACJE DLA OŚRODKÓW PRACY W DOLE STRUMIENIA

W latach 80. w dziedzinie projektowania zasad produkcyjnych dążono do tworzenia części i projektowania procesów w taki sposób, aby gotowe produkty mogły być wytworzone jak najmniejszym kosztem, by miały jak największą jakość oraz by był zapewniony jak najlepszy przepływ. Przykładem takiego podejścia może być produkcja części bardzo asymetrycznych, co miało zapobiec możliwości ich odwrotnego łączenia, czy też produkowanie zacisków śrubowych tak, aby nie było możliwości dokręcenia ich zbyt mocno.

Było to odejście od standardowego sposobu projektowania, gdzie koncentrowano się na klientach zewnętrznych, ale pomijano interesariuszy wewnętrznych — na przykład pracowników produkcji.

W metodyce Lean wyróżnia się dwa rodzaje klientów, o których potrzeby należy zadbać w projektach: zewnętrznych (którzy najczęściej płacą za dostarczaną usługę) oraz wewnętrznych (którzy odbierają i przetwarzają pracę bezpośrednio po nas). Według Lean najważniejszymi klientami są klienci występujący bezpośrednio po nas w dole strumienia. Optymalizacja pracy pod kątem ich potrzeb wymaga empatii dla ich problemów. Pozwala to na lepsze identyfikowanie problemów projektowych uniemożliwiających szybki i płynny przepływ.

W strumieniu wartości technologii optymalizujemy pracę pod kątem potrzeb ośrodków roboczych w dole strumienia w taki sposób, aby wymagania niefunkcjonalne (np. architektura, wydajność, stabilność, testowalność, konfigurowalność i bezpieczeństwo) były traktowane z takim samym priorytetem jak funkcje przeznaczone dla użytkownika.

W ten sposób tworzymy jakość u źródła, co stwarza warunki do zdefiniowania zbioru kodyfikowanych wymagań niefunkcjonalnych, które można proaktywnie zintegrować z każdą usługą, którą wytwarzamy.

PODSUMOWANIE

Tworzenie szybkich sprzężeń zwrotnych ma kluczowe znaczenie dla osiągnięcia jakości, niezawodności i bezpieczeństwa w strumieniu wartości technologii. Robimy to poprzez dostrzeganie problemów w chwili, gdy występują, stosujemy technikę swarmingu i rozwiązujeemy problemy po to, by budować nową wiedzę. Przenosimy jakość bliżej źródła problemów i stale optymalizujemy pracę pod kątem potrzeb ośrodków roboczych w dole strumienia.

Konkretnie praktyki umożliwiające szybki przepływ w strumieniu wartości DevOps zaprezentowano w części IV. W następnym rozdziale omówimy trzecią drogę, zasady ciągłego uczenia się i eksperymentowania.

Trzecia droga Zasady ciągłego uczenia się i eksperymentowania

Podczas gdy **pierwsza droga** opisuje przepływ pracy od lewej do prawej, **druga droga** podkreśla ważność szybkiego i stałego sprzeżenia zwrotnego od prawej do lewej, to **trzecia droga** koncentruje się na stworzeniu kultury ciągłego uczenia się i eksperymentowania. Wszystkie te zasady pozwalają na stałe tworzenie indywidualnej wiedzy, która następnie przekształca się w wiedzę zespołu i organizacji.

W produkcji, gdzie występują systemowe problemy jakości i bezpieczeństwa, praca jest zazwyczaj sztywno zdefiniowana i wymuszona. Na przykład w zakładzie GM Fremont, opisany w poprzednim rozdziale, pracownicy mieli niewielkie możliwości integracji usprawnień i wiedzy w swojej codziennej pracy. Propozycje usprawnień zwykle trafiały na mur obojętności.

W tego rodzaju środowiskach często istnieje również kultura strachu i niskiego zaufania. Pracownicy, którzy popełniają błędy, są karani, a ci, którzy wskazują problemy lub sugerują poprawki, są postrzegani jako osoby siejące niepotrzebny ferment. Gdy ktoś zgłosi potrzebę usprawnień, menedżerowie aktywnie tłumią, a nawet karzą za uczenie się i doskonalenie, utrwalanie jakości i wdrażanie zasad bezpieczeństwa.

Dla odróżnienia w firmach produkcyjnych pracujących z wysoką wydajnością istnieje wymaganie i aktywna promocja ciągłego uczenia się — zamiast sztywno zdefiniowanych zadań jest dynamiczny system pracy. Pracownicy na linii produkcyjnej codziennie eksperymentują, dzięki czemu generują nowe usprawnienia. Sprzyja temu rygorystyczna standaryzacja procedur pracy i dokumentacja wyników.

W strumieniu wartości technologii celem jest stworzenie kultury wysokiego zaufania, umocnienie przekonania, że uczymy się przez całe życie i w codziennej pracy stale musimy podejmować ryzyko. Dzięki zastosowaniu naukowego podejścia zarówno do usprawnień procesu, jak i rozwoju produktu uczymy się na podstawie naszych sukcesów i niepowodzeń. Identyfikujemy te pomysły, które się nie sprawdzają, i wzmacniamy te, które okazały się skuteczne. Ponadto wszystkie przypadki lokalnego uczenia się są szybko zamieniane w globalne usprawnienia, dzięki czemu nowe techniki i praktyki mogą być stosowane w całej organizacji.

W codziennej pracy rezerwujemy czas na usprawnienia oraz na szybsze i łatwiejsze uczenie się. Konsekwentnie poddajemy systemy coraz większym obciążeniom, tak by wymusić ciągłe udoskonalenia. W celu zwiększenia odporności na awarie symulujemy je w usługach produkcyjnych, „wstrzykując błędy” w kontrolowanych warunkach.

Tworząc ten dynamiczny system ciągłego uczenia się, umożliwiamy zespołowi szybkie i automatyczne dostosowanie się do stale zmieniającego się środowiska, co ostatecznie pozwala zwyciężyć na rynku.

TWORZENIE W ORGANIZACJI KULTURY UCZENIA SIĘ I BEZPIECZEŃSTWA

Praca w złożonym systemie uniemożliwia z jego definicji dokładne przewidywanie wszystkich możliwych rezultatów wszystkich podejmowanych działań. To przyczynia się do nieoczekiwanych lub nawet katastrofalnych skutków i incydentów w codziennej pracy — nawet wtedy, gdy podejmujemy środki ostrożności i gdy podczas działania zachowujemy ostrożność.

Jeśli te wypadki wywierają negatywny wpływ na naszych klientów, staramy się zrozumieć, dlaczego tak się dzieje. Często jako główna przyczyna podawany jest błąd człowieka, a popularną odpowiedzią menedżerów jest „wskazanie, obarczenie winą i zdyskredytowanie” osoby, która spowodowała problem*. Ponadto — subtelnie albo jawnie — menedżerowie sugerują, że osoba winna popełnienia błędu zostanie ukarana. Następnie w celu zapobieżenia powtórzeniu podobnego błędu tworzą dodatkowe procesy i wprowadzają dodatkowe autoryzacje.

Dr Sidney Dekker, który skodyfikował niektóre kluczowe elementy kultury bezpieczeństwa i spopularyzował termin *just culture* (dosł. „kultura właściwego postępowania”), napisał: „Reakcje na incydenty i wypadki, które są postrzegane jako niewłaściwe, mogą utrudniać dochodzenia dotyczące zasad bezpieczeństwa i promować wśród pracowników wykonujących pracę o kluczowym dla bezpieczeństwa znaczeniu strach zamiast rozwagi. W ten sposób organizacja zamiast stawać się bardziej ostrożna

* Postępowanie „wskazać, obarczyć winą i zdyskredytować” (ang. *name, blame and shame*) jest elementem teorii „bad apple”, dokładnie opisanej i skrytykowanej przez dr. Sydneya w książce *The Field Guide to Understanding Human Error*.

staje się bardziej biurokratyczna i pielęgnuje zasady tajemnicy zawodowej, uchyłania się od odpowiedzialności i ochronę własnych interesów”.

Kwestie te mają szczególnie istotne znaczenie w strumieniu wartości technologii — nasza praca jest prawie zawsze wykonywana w złożonych systemach, a sposób reagowania kierownictwa na awarie i incydenty prowadzi do kultury strachu. W związku z tym później sygnały o problemach i awariach nawet nie są zgłoszane. W rezultacie problemy pozostają ukryte, aż w końcu dochodzi do katastrofy.

Dr Ron Westrum był jedną z pierwszych osób, która zaobserwowała istotę kultury organizacyjnej oraz jej wpływ na bezpieczeństwo i wydajność. Zauważył, że w organizacjach związanych ze służbą zdrowia jednym z najważniejszych predyktorów bezpieczeństwa pacjentów było istnienie kultury „generatywnej”. Cechy trzech typów organizacji zaprezentowano na rysunku 8.

- Organizacje patologiczne charakteryzują się dużym poziomem strachu i zagrożeń. Ludzie często chowają informacje przed innymi, przetrzymując je z powodów politycznych lub zniekształcając je, aby zaprezentować się w lepszym świetle. Awarie są często ukrywane.
- Organizacje biurokratyczne charakteryzują się obecnością reguł i procesów często opisujących zasady utrzymania pojedynczych działań. Awarie są przetwarzane przez system dochodzeń, których wynikiem jest ukaranie albo usprawiedliwienie i odstąpienie od kary.
- Organizacje generatywne charakteryzuje aktywne poszukiwanie i udostępnianie informacji w celu ułatwienia realizacji misji. Obowiązki są współdzielone w całym strumieniu wartości, a skutkiem awarii są refleksje i zwykłe dochodzenie przyczyn.

Patologiczne	Biurokratyczne	Generatywne
Informacje są ukrywane	Informacje mogą być ignorowane	Informacje są aktywnie poszukiwane
Posłańcy są „rozstrzeliani”	Posłańcy są tolerowani	Posłańcy są szkoleni
Występuje zjawisko uchyłania się od odpowiedzialności	Odpowiedzialność jest „zaszufladowana”	Odpowiedzialność jest wspólna
Mosty pomiędzy zespołami są niewskazane	Mosty pomiędzy zespołami są dozwolone, ale niezalecane	Mosty pomiędzy zespołami są zalecane i nagradzane
Awarie są ukrywane	Organizacja jest sprawiedliwa i wyrozumiała	Awarie są powodem do stawiania pytań
Nowe pomysły są tłumione	Nowe pomysły stwarzają problemy	Nowe pomysły są mile widziane

Rysunek 8. Model typologii organizacji Westrum: w jaki sposób organizacje przetwarzają informacje (źródło: Ron Westrum, „A typology of organisation culture”, „BMJ Quality & Safety”, vol. 13, nr 2 (2004), doi:10.1136/qshc.2003.009522)

Odkrycia dr. Westruma w organizacjach związanych ze służbą zdrowia, dotyczące generatywnej kultury wysokiego zaufania, mają zastosowanie także do wydajności w branży IT i strumieniu wartości technologii.

W strumieniu wartości technologii możemy ustanowić fundamenty kultury generatywnej poprzez dążenie do stworzenia bezpiecznego systemu pracy. Gdy występują błędy i awarie, to zamiast szukać błędu człowieka, zastanawiamy się, w jaki sposób zmienić projekt systemu tak, aby przeciwdziałać możliwości wystąpienia podobnej awarii w przyszłości.

Na przykład po każdym zdarzeniu możemy przeprowadzać analizę problemu. Jej celem nie ma być szukanie winnych, ale uzyskanie lepszego zrozumienia sytuacji i znalezienie najlepszych środków zaradczych pozwalających na usprawnienie systemu, a w idealnej sytuacji zapobieżenie ponownemu wystąpieniu problemu bądź umożliwienie jego szybszego wykrycia i naprawy.

W ten sposób tworzymy w organizacji system uczenia się. Jak powiedziała Macri Bethany, inżynier w firmie Etsy, która przewodziła tworzeniu narzędzia Morque wspomagającego rejestrowanie analiz post-mortem, „dzięki usunięciu elementu szukania winnych usuwamy strach; dzięki usunięciu strachu promujemy uczciwość, a uczciwość umożliwia zapobieganie problemom”.

Dr Spear zaobserwował, że w wyniku usunięcia szukania winnych i nadania odpowiedniej rangi uczeniu się „organizacje zyskują zdolność do samodiagnozowania i samonaprawiania oraz umiejętności wykrywania problemów [i] ich usuwania”.

Wiele z tych atrybutów zostało również opisanych przez dr. Senge'a jako atrybuty organizacji uczącej się. W książce *The Fifth Discipline* dr Senge napisał, że te kluczowe cechy pomagają zapewnić jakość dla klientów, stworzyć przewagę konkurencyjną oraz wykreować energiczną i zaangażowaną zespół pracowników, a także odkryć prawdę.

INSTYTUCJONALIZACJA USPRAWNIEŃ W CODZIENNEJ PRACY

Zespoły często nie są zdolne lub są negatywnie nastawione do usprawniania procesów, które wykorzystują w pracy. W rezultacie nie tylko cierpią z powodu swoich bieżących problemów, ale także ich cierpienia z czasem stają się coraz większe. Mike Rother w *Toyota Kata* zaobserwował, że w przypadku braku poprawy procesy nie pozostają takie same — z powodu chaosu i entropii procesy w rzeczywistości z czasem stają się coraz gorsze.

Gdy unikamy rozwiązywania problemów w strumieniu wartości technologii i bazujemy na codziennych obejściach, to problemy i dług techniczny gromadzą się aż do momentu, kiedy zajmujemy się wyłącznie wykonywaniem obejść i próbami uniknięcia katastrofy, przez co nie mamy wolnych cykli do wykonywania efektywnej pracy. Dlatego właśnie Mike Orzen, autor książki *Lean IT*, zauważył, że „jeszcze ważniejsze od codziennej pracy jest usprawnianie codziennej pracy”.

Codzienną pracę możemy poprawiać, jawnie rezerwując czas na spłacanie dłużu technicznego, usuwanie wad i refaktoryzację oraz poprawianie problematycznych obszarów kodu i środowisk. Robimy to, rezerwując cykle w każdym interwale rozwojowym albo planując tzw. **kaizen blitzes** — tzn. okresy, w których inżynierowie samodzielnie organizują się w zespoły w celu rozwiązyania wybranych problemów.

W wyniku stosowania tych praktyk wszyscy zawsze znajdują i rozwiązują problemy w swoim obszarze odpowiedzialności oraz w ramach swojej codziennej pracy. Kiedy ostatecznie uda się rozwiązać problemy, dla których przez wiele miesięcy (lub lat) stosowaliśmy obejścia, możemy zająć się eliminowaniem z systemu problemów mniej oczywistych. Dzięki wykrywaniu i reagowaniu na te coraz słabsze sygnały awarii możemy rozwiązywać problemy wtedy, gdy nie tylko jest to łatwiejsze i tańsze, ale również wtedy, gdy skutki są mniejsze.

Rozważmy opisany poniżej przykład poprawienia bezpieczeństwa miejsca pracy w firmie produkcyjnej z branży aluminium Alcoa, z 7,8 mld dolarów przychodów w 1987 roku. W 1987 roku statystyki wypadków w firmie Alcoa były przerażające — 2% spośród 90 tysięcy pracowników każdego roku ulegało wypadkom — oznaczało to siedem wypadków dziennie. Gdy Paul O'Neill rozpoczął pracę na stanowisku CEO w firmie Alcoa, jego pierwszym celem było zero wypadków wśród pracowników, kontrahentów i gości.

O'Neill chciał otrzymywać informację w ciągu 24 godzin od każdego wypadku w pracy — nie po to, aby ukarać winnych, ale po to, aby czerpać z tych wypadków naukę, promować uczenie się, a w efekcie tworzyć bezpieczniejsze miejsce pracy. W ciągu 10 lat Alcoa zmniejszyła wskaźnik wypadków o 95%.

Zmniejszenie liczby wypadków pozwoliło firmie Alcoa skoncentrować się na mniejszych problemach i słabszych sygnałach awarii — zamiast powiadamiać O'Neilla tylko o powstałych wypadkach, zaczęto raportować także sytuacje bliskie wypadkom*. Dzięki podjętym działaniom firma Alcoa w ciągu ostatnich 20 lat poprawiła bezpieczeństwo pracy do takiego stanu, że obecnie może się poszczycić jednymi z najlepszych statystyk bezpieczeństwa w branży.

Jak napisał dr Spear: „Pracownicy firmy Alcoa stopniowo przestali obchodzić trudności, niedogodności i przeszkody, których doświadczyli podczas pracy. Obejścia, »gaszenie pożarów« stopniowo zastępowały w całej organizacji dynamiką identyfikowania możliwości usprawniania procesu i produktu. W miarę identyfikowania tych możliwości i analizy problemów pokłady niewiedzy, z których te problemy wynikały, były zamieniane na zasoby wiedzy”. To pomogło firmie uzyskać większą przewagę konkurencyjną na rynku.

* Poziom przekonania i pasji Paula O'Neilla co do moralnej odpowiedzialności liderów za tworzenie bezpiecznego miejsca pracy jest zadziwiający, pouczający i poruszający.

Podobnie jest w strumieniu wartości technologii. W miarę poprawiania bezpieczeństwa systemu pracy znajdujemy i rozwiążujemy problemy identyfikowane coraz słabszymi sygnałami. Na przykład początkowo możemy wykonywać analizy post-mortem bez szukania winnych tylko dla incydentów mających wpływ na klienta. Z czasem możemy je wykonywać dla mniej znaczących incydentów wywierających wpływ na zespół, a także dla sytuacji zagrożeń incydentami.

PRZEKSZTAŁCENIE LOKALNYCH ODKRYĆ W GLOBALNE USPRAWNIENIA

Gdy lokalnie zdobywana jest nowa wiedza, to musi również istnieć jakiś mechanizm, dzięki któremu z tej wiedzy może skorzystać pozostała część organizacji. Innymi słowy, gdy zespoły lub indywidualni członkowie zespołów zdobędą doświadczenia, które tworzą wiedzę ekspercką, to naszym celem powinno być przekształcenie tej ukrytej wiedzy (tzn. wiedzy, którą trudno przenieść na inną osobę poprzez zapisanie w dokumencie lub słowne opisanie) na jawną, skodyfikowaną wiedzę, która w wyniku doświadczenia staje się dziedziną ekspercką kogoś innego.

W ten sposób zyskujemy gwarancję, że gdy ktoś będzie wykonywał podobną pracę, to będzie działał, wykorzystując skumulowane i zbiorowe doświadczenia wszystkich osób w organizacji, które kiedykolwiek wykonywały podobną pracę. Znakomitym przykładem przekształcenia wiedzy lokalnej w globalną jest Nuclear Power Propulsion Program z US Navy (znany również jako „NR”, od „Naval Reactors”), w którym odnotowano ponad 5700 reaktorolat działania bez wypadku albo nieplanowanej emisji promieniowania.

Program NR jest znany z olbrzymiego zaangażowania w tworzenie procedur skryptowych oraz standardów pracy. Incydenty wszelkich odstępstw od procedur lub normalnego działania — bez względu na to, jak słaby był sygnał odstępstwa — zawsze są raportowane w celu akumulacji wiedzy. W oparciu o tę wiedzę stale są aktualizowane procedury i projekt systemu.

W rezultacie, gdy nowa załoga wyrusza w swój pierwszy rejs na morze, to jej członkowie oraz oficerowie mogą skorzystać ze zbiorowej wiedzy 5700 wolnych od wypadków reaktorolat. Równie wielkie wrażenie robi to, że ich własne doświadczenia na morzu zostaną dodane do tej zbiorowej wiedzy, co pomoże przyszłym załogom bezpiecznie wypełnić ich misje.

W strumieniu wartości technologii należy stworzyć podobne mechanizmy globalnej wiedzy. Można na przykład udostępnić zespołom mechanizmy wyszukiwania w raportach post-mortem. Można także stworzyć wspólne, obejmujące całą organizację repozytoria kodu źródłowego. Za ich pomocą można łatwo wykorzystać wspólny kod, biblioteki i konfiguracje będące ucieleśnieniem najlepszej zbiorowej wiedzy całej organizacji. Wszystkie te mechanizmy pomagają przekształcić wiedzę indywidualnego pracownika w artefakty, z których mogą skorzystać pozostałe osoby w organizacji.

ZASTOSOWANIE WZORCÓW ODPORNOŚCI W CODZIENNEJ PRACY

Firmy produkcyjne charakteryzujące się niższą wydajnością tworzą na wiele sposobów bufor oddzielający je od zakłóceń — innymi słowy, zwiększają nadmiar. Na przykład w celu zmniejszenia zagrożenia bezczynności ośrodka pracy (ze względu na spóźnienia w dostarczaniu surowców, konieczność złomowania urządzeń itp.) menedżerowie mogą dążyć do gromadzenia większej ilości zapasów w każdym ośrodku roboczym. Jednak te bufory zapasów zwiększają również współczynnik pracy w toku (WIP), który zgodnie z tym, co powiedzieliśmy wcześniej, powoduje różnego rodzaju niepożądane wyniki.

Na podobnej zasadzie, aby zmniejszyć ryzyko wyłączenia ośrodka pracy z powodu awarii maszyn, menedżerowie mogą zwiększać możliwości poprzez zakup większej ilości sprzętu, zatrudnianie większej liczby osób lub nawet zwiększenie powierzchni pracy. Wszystkie te działania zwiększają koszty.

Dla odróżnienia — firmy charakteryzujące się wysoką wydajnością osiągają takie same wyniki (lub lepsze) dzięki usprawnieniu codziennej pracy, ciąglemu dążeniu do podnoszenia wydajności, a także wprowadzaniu większej elastyczności do swojego systemu.

Rozważmy typowy eksperyment w jednej z fabryk materaców Aisin Seiki Global — to jeden z największych dostawców Toyoty. Założymy, że w firmie były dwie linie produkcyjne — każda z nich była zdolna do produkcji stu jednostek dziennie. W luźniejsze dni całą produkcję kierowano na jedną linię i eksperymentowano ze sposobami zwiększania wydajności oraz próbowano identyfikować luki w procesie, wiedząc, że jeśli przeciążenie linii spowoduje awarię, to można będzie skierować całą produkcję na drugą linię.

Dzięki ciąglemu eksperymentowaniu w codziennej pracy udało się osiągnąć wzrost wydajności, często bez dodawania żadnych nowych urządzeń ani zatrudniania nowych osób. Wzorzec, który wynika z tego rodzaju usprawnień, nie tylko zwiększa wydajność, ale również elastyczność, ponieważ organizacja stale znajduje się w stanie napięcia i zmiany. Nassim Nicholas Taleb, autor książek z dziedziny analizy ryzyka, nazwał proces zastosowania napięcia w celu zwiększenia elastyczności **antykruchością** (ang. *antifragility*).

W strumieniu wartości technologii możemy wprowadzić taki sam napięcia do naszych systemów poprzez ciągłe dążenie do skrócenia czasu realizacji, zwiększenie zakresu pokrycia testami, zmniejszenie czasu wykonywania testów, a nawet poprzez ponowne ich zaprojektowanie w celu zwiększenia wydajności pracy deweloperów lub zwiększenia niezawodności.

Można również przeprowadzać ćwiczenia typu **dzień gry** (ang. *game day*), w których można ćwiczyć awarie na dużą skalę — na przykład wyłączenie całego centrum danych. W celu zapewnienia maksymalnej odporności można też wstrzykiwać do

środowiska produkcyjnego błędy o coraz większej skali (np. za pomocą mechanizmu „Chaos Monkey” stosowanego w firmie Netflix, który przypadkowo zabija procesy oraz produkcyjne serwery obliczeniowe).

LIDERZY WZMACNIAJĄ KULTURĘ UCZENIA SIĘ

Tradycyjnie od liderów oczekuje się ustalania celów, przydzielania zasobów potrzebnych do osiągnięcia tych celów oraz ustanawiania właściwej kombinacji zachęt. Liderzy ustalają także emocjonalny ton dla organizacji, której przewodzą. Innymi słowy, liderzy kierują poprzez „podejmowanie właściwych decyzji”.

Istnieją jednak wiarygodne dowody na to, że podejmowanie przez liderów właściwych decyzji nie powoduje osiągnięcia świetności. Rolą lidera powinno być raczej stworzenie warunków, dzięki którym zespół może odkryć świetność w swojej codziennej pracy. Innymi słowy, tworzenie świetności wymaga zarówno liderów, jak i pracowników. Każda z tych stron nawzajem od siebie zależy.

Jim Womack, autor książki *Gemba Walks*, wskazał na uzupełniające relacje w pracy i wzajemny szacunek, które muszą występować pomiędzy liderami a podległymi im pracownikami. Według Womacka ten związek jest konieczny, ponieważ żadna ze stron nie może rozwiązać problemów samodzielnie — liderzy nie są wystarczająco blisko pracy, co jest konieczne do rozwiązywania każdego problemu, natomiast podlegli im pracownicy nie znają szerszego kontekstu lub nie mają uprawnień do wprowadzania zmian poza swoim obszarem pracy*.

Liderzy muszą podkreślać wartość uczenia się i zdyscyplinowanego rozwiązywania problemów. Mike Rother sformalizował te metody we wzorcu, który określił jako **coaching kata**. Wynik odzwierciedla metodę naukową, gdzie wyraźnie formułujemy cele przysłowiowej „północy rzeczywistej”, takie jak „utrzymanie wypadków na zero-wym poziomie” w przypadku firmy Alcoa lub „podwojenie przepustowości w ciągu roku” w przypadku firmy Aisin.

Następnie te strategiczne cele tworzą podstawę do wyznaczenia iteracyjnych celów krótkoterminowych, które są przekazywane kaskadowo, a później wykonywane poprzez ustanowienie warunków docelowych na poziomie strumienia wartości lub ośrodku pracy (np. „w ciągu następnych dwóch tygodni skrócić czas realizacji o 10%”).

Wyznaczone warunki docelowe tworzą ramę eksperymentu naukowego: wyraźnie formułujemy problem, który staramy się rozwiązać, hipotezę na temat sposobu, w jaki zaproponowane przez nas środki zaradcze go rozwiązają, metody do testowania postawionej hipotezy, interpretację wyników oraz wykorzystanie nauki w celu stworzenia podstawy dla następnej iteracji.

* Liderzy są odpowiedzialni za projekt i działanie procesów na wyższym poziomie agregacji — tam, gdzie inni mają większą perspektywę i mniejsze uprawnienia.

Lider pomaga prowadzącemu eksperyment, zadając mu na przykład następujące pytania:

- Jaki był twój ostatni krok i co się wydarzyło?
- Czego się nauczyłeś?
- Jaki jest stan bieżący?
- Co jest kolejnym celem?
- Jakie przeszkody teraz starasz się usunąć?
- Jaki będzie twój następny krok?
- Jakie są oczekiwane rezultaty?
- Kiedy możemy sprawdzić wyniki?

Podejście do rozwiązywania problemów polegające na tym, że liderzy pomagają pracownikom dostrzec i rozwiązać problemy w ich codziennej pracy, jest podstawą Toyota Production System, zasadą obowiązującą w organizacjach uczących się, podstawą Improvement Kata oraz regułą w firmach o wysokiej niezawodności. Mike Rother powiedział, że postrzega Toyotę „jako organizację zdefiniowaną przede wszystkim przez unikatowe procedury zachowań, których przestrzegania nieustannie uczy wszystkich swoich członków”.

W strumieniu wartości technologii to naukowe podejście i metoda iteracyjna kierują wszystkimi procesami wewnętrznego doskonalenia. Ponadto wpływają na sposób przeprowadzania eksperymentów zmierzających do tego, aby wytwarzane produkty rzeczywiście pomagały klientom osiągać ich cele.

PODSUMOWANIE

Zasady trzeciej drogi podkreślają wartość uczenia się w organizacji, stworzenia warunków wysokiego zaufania i przekraczania granic pomiędzy funkcjami. Wyrażają akceptację występowania awarii w złożonych systemach oraz pozwalają mówić o problemach, dzięki czemu można stworzyć bezpieczny system pracy. Potrzebne jest również zinstytucjonalizowanie usprawnień w codziennej pracy, konwersja lokalnej nauki na wiedzę globalną, która może być wykorzystana w całej organizacji, jak również stałe „wstrzykiwanie” napięcia do codziennej pracy.

Mimo że promowanie kultury ciągłego kształcenia się i eksperimentowania jest zasadą trzeciej drogi, to zasada ta jest również wpleciona w pierwszą i drugą drogę. Innymi słowy, poprawa przepływu i sprzężenia zwrotne wymagają iteracyjnego i naukowego podejścia, które obejmuje określenie stanu docelowego, sformułowanie hipotezy, która pomaga osiągnąć ten stan, zaprojektowanie i prowadzenie eksperymentów oraz ocenę wyników.

W rezultacie nie tylko osiągamy lepszą wydajność, ale także zwiększoną odporność, większe zadowolenie z pracy oraz poprawę zdolności przystosowania organizacji do nowych warunków.

PODSUMOWANIE CZĘŚCI I

W części I książki *DevOps* przyjrzeliśmy się kilku ruchom w historii, które pomogły doprowadzić do rozwinięcia infrastruktury DevOps. Omówiliśmy również trzy główne zasady, które tworzą podstawy sukcesu organizacji korzystających z DevOps: przepływ, sprzężenie zwrotne oraz ciągłe uczenie się i eksperymentowanie. W części II opowiemy o tym, jak rozpocząć ruch DevOps w organizacji.

Część II
Od czego zacząć?

Część II *Wprowadzenie*

Jak podjąć decyzję, od czego zacząć transformacje DevOps w organizacji? Kto powinien brać udział w tej transformacji? Jak należy zorganizować nasze zespoły, ochronić ich możliwości realizacji pracy i zmaksymalizować szanse na osiągnięcie sukcesu? Są to pytania, na które postaramy się odpowiedzieć w części II podręcznika *DevOps*.

W kolejnych rozdziałach przeprowadzimy Czytelników przez proces inicjowania transformacji DevOps. Rozpoczniemy od oceny strumieni wartości w organizacji, zlokalizowania dobrego miejsca do rozpoczęcia i opracowania strategii stworzenia dedykowanego zespołu transformacji z określonymi, konkretnymi celami poprawy i ostateczną ekspansją. Dla każdego strumienia wartości podlegającego transformacji zidentyfikujemy wykonywaną pracę, a następnie przyjrzymy się strategiom projektowania w organizacji oraz archetypom organizacyjnym, które najlepiej wspierają cele transformacji.

W tych rozdziałach omówimy następujące zagadnienia:

- Wybór strumieni wartości, od których należy zacząć.
- Zrozumienie pracy realizowanej w strumieniach wartości będących kandydatami do przekształceń.
- Zaprojektowanie naszej organizacji i jej architektury z uwzględnieniem prawa Conwaya.
- Generowanie wyników bardziej zgodnych z potrzebami rynku dzięki skuteczniejszej współpracy pomiędzy funkcjami w całym strumieniu wartości.
- Ochrona i wyzwolenie naszych zespołów.

Rozpoczęcie każdej transformacji wiąże się z niepewnością — planujemy podróż do idealnego stanu końcowego, ale prawie wszystkie kroki pośrednie są nieznane. Celem kolejnych rozdziałów jest opisanie procesu myślenia sterującego podejmowaniem naszych decyzji, zaprezentowanie kroków, jakie można podjąć, oraz zilustrowanie ich przykładami w postaci studiów przypadków.

5

Wybór strumienia wartości, od których należy zacząć

Wybór strumienia wartości dla transformacji DevOps zasługuje na staranną analizę. Strumień wartości, który wybierzemy, nie tylko będzie dyktował trudności naszej transformacji, ale także będzie decydował o tym, kto będzie w nią zaangażowany. Wybór, którego dokonamy, wpłynie na sposób organizacji zespołów oraz na najlepszą metodę do stworzenia dobrych warunków pracy zespołów i należących do nich osób.

Jedno z wyzwań zaobserwował Michael Rembtsky, który jako dyrektor operacyjny w firmie Etsy w 2009 roku kierował transformacjami DevOps. Zauważyl on: „Projekty do transformacji należy wybierać starannie — gdy jesteśmy w tarapatach, nie mamy zbyt wielu dróg wyjścia. W związku z tym trzeba starannie wybrać, a następnie chronić te projekty naprawy, które najbardziej przyczyniają się do poprawy stanu naszej organizacji”.

Przyjrzyjmy się, w jaki sposób w 2013 roku inicjatywę transformacji DevOps rozpoczęł zespół firmy Nordstrom. Courtney Kissler, wiceprezes tej firmy ds. e-commerce i technologii sklepowych, opisała te przekształcenia na konferencji DevOps Enterprise Summit w 2014 i 2015 roku.

Firma Nordstrom została założona w 1901 roku. Jest czołowym sprzedawcą modnej odzieży, który koncentruje się na dostarczeniu swoim klientom jak najwygodniejszego sposobu robienia zakupów. W 2015 roku roczne przychody firmy Nordstrom sięgały 13,5 mld dolarów.

„Scena” dla przekształceń DevOps w firmie Nordstrom DevOps prawdopodobnie została ustawniona w 2011 roku podczas jednego z corocznych spotkań zarządu dyrektorów. W tamtym roku jednym ze strategicznych omawianych tematów była

potrzeba wzrostu dochodów ze sprzedaży online. Studiowano trudną sytuację takich firm, jak Blockbusters, Borders oraz Barnes & Nobles, pokazując tragiczne konsekwencje dla tradycyjnych detalistów opóźnień w tworzeniu alternatywnych możliwości e-commerce — firmom tym realnie groziła utrata pozycji na rynku lub nawet całkowite bankructwo*.

W tamtym czasie Courtney Kissler pełniła funkcję starszego dyrektora ds. systemów dostaw i technologii sprzedaży. Była odpowiedzialna za znaczną część organizacji technicznej, w tym za działanie systemów w sklepach oraz witryny e-commerce online. Jak opisała Kissler: „W roku 2011 firma techniczna Nordstrom była zoptymalizowana pod kątem kosztów — zlecaliśmy wiele funkcji technicznych zewnętrznym firmom, stosowaliśmy roczny cykl planowania z dużymi wydaniami projektów oprogramowania realizowanymi metodą kaskadową. Mimo że mieliśmy 97-procentowy wskaźnik sukcesu zgodności z celami harmonogramu czasowego, budżetu i zakresu, to nie byliśmy przygotowani na osiągnięcie celu pięcioletniej strategii biznesowej, który mówił, że firma Nordstrom powinna rozpocząć optymalizację pod kątem szybkości działania, a nie jedynie pod względem kosztów”.

Kissler i zespół zarządzania technologiami w firmie Nordstrom musieli zdecydować, od czego zacząć transformację. Nie chcieli wprowadzać przełomowych przemian w całym systemie. Zamiast tego chcieli skoncentrować się na bardzo konkretnych obszarach działalności, tak aby mogli eksperymentować i uczyć się. Ich celem było pokazanie pierwszych zwycięstw, które dałyby wszystkim pewność, że ulepszenia mogą być powielone w innych obszarach organizacji. Dokładny sposób osiągnięcia celu nadal był nieznany.

Skoncentrowano się na trzech obszarach: aplikacji mobilnej dla klientów, systemach obsługi restauracji w sklepie oraz ich właściwościach cyfrowych. W każdym z tych obszarów istniały cele biznesowe, które nie były spełniane. W związku z tym obszary te były bardziej otwarte na rozważenie innego sposobu działania. Historie dotyczące pierwszych dwóch opisano poniżej.

Start aplikacji mobilnej Nordstrom był niepomyślny. Jak powiedziała Kissler: „Nasi klienci byli bardzo niezadowoleni z produktu, a po umieszczeniu aplikacji w App Store mieliśmy wiele negatywnych opinii. Co gorsza, istniejące struktury i procesy (czyli tzw. »system«) były zaprojektowane w taki sposób, że aktualizacje mogły być publikowane tylko dwa razy w roku”. Innymi słowy, wszelkie poprawki w aplikacji musiały czekać wiele miesięcy, zanim dotarły do klienta.

Pierwszym celem było umożliwienie szybszego cyklu wydań lub wydań na żądanie, co pozwoliłoby na szybsze iteracje oraz zdolność do reagowania na opinie klientów. Zaczęto od stworzenia dedykowanego zespołu dla produktu. Zadaniem tego zespołu było utrzymanie aplikacji mobilnej oraz niezależne implementowanie, testowanie

* Firmy te czasami określano nazwą „Killer B’s, które umierają”.

i dostarczanie wartości dla klienta. Zespół nie był zależny od innych zespołów wewnątrz firmy Nordstrom i nie musiał z nimi koordynować swojej pracy. Ponadto zrezygnowano z planowania raz w roku na rzecz procesu ciągłego planowania. W rezultacie powstał pojedynczy zbiór zadań do wykonania (*backlog*) w aplikacji mobilnej z priorytetami ustalonymi na podstawie potrzeb klienta. Znikiły wszystkie sprzeczne priorytety z czasów, gdy zespół miał do obsługi wiele produktów.

W ciągu kolejnego roku wyeliminowano testowanie jako oddzielną fazę pracy, a zamiast tego zintegrowano je z codzienną pracą każdego członka zespołu*. W ten sposób podwojono liczbę funkcji dostarczanych miesięcznie i zmniejszono o połowę liczbę defektów, zyskując pomyślny rezultat.

Drugim obszarem zainteresowania były systemy wspierające działanie restauracji *Café Bistro* w sklepach. W przeciwnieństwie do strumienia wartości aplikacji mobilnej, gdzie potrzeba biznesowa polegała na skróceniu czasu dostarczenia produktu na rynek i zwiększeniu wydajności funkcji, tutaj potrzebą biznesową było zmniejszenie kosztów i poprawa jakości. W 2013 r. firma Nordstrom miała za sobą przeprowadzonych 11 tzw. „zmian koncepcji” restauracji. Wszystkie one wymagały zmian w aplikacjach w sklepie, powodując szereg incydentów mających wpływ na klienta. Niepokojące było to, że na rok 2014 zaplanowano 44 tego rodzaju operacje zmian koncepcji — cztery razy więcej niż w roku poprzednim.

Jak powiedziała Kissler: „Jeden z naszych liderów biznesowych zasugerował, aby w celu obsługi tych nowych wymagań potroić rozmiar naszego zespołu, ale zaproponowałam, że musimy przerwać przydzielanie coraz większej liczby osób do rozwiązywania problemu, a zamiast tego poprawić sposób, w jaki pracujemy”.

Zidentyfikowano obszary problemów — na przykład procesy zlecania pracy i wdrażania. Na nich skoncentrowano wysiłki poprawy. Usprawnienia w tych obszarach pozwoliły skrócić czas wdrażania kodu o 60% i zmniejszyć liczbę zdarzeń w produkcji do 60 – 90%.

Sukcesy te pozwoliły zespołom uwierzyć, że zasady i praktyki DevOps można zastosować do różnych strumieni wartości. W 2014 roku Kissler awansowała na stanowisko wiceprezesa ds. e-commerce i technologii sklepowych.

W 2015 Kissler powiedziała, że aby umożliwić organizacji technicznej, zajmującej się sprzedażą lub usługami dla klientów, osiągnięcie swoich celów, „trzeba zwiększyć produktywność we wszystkich strumieniach wartości technologii, a nie tylko w kilku. Na poziomie kierownictwa wyznaczyliśmy cel skrócenia czasu trwania cyklu o 20% dla wszystkich usług skierowanych do klientów”.

* Praktyka stosowania fazy stabilizacji lub wzmacniania produktu na końcu projektu często przynosi bardzo słabe rezultaty, ponieważ oznacza, że problemy nie są wykrywane i rozwiązywane w ramach codziennej pracy. W efekcie pozostają nierozwiążane i potencjalnie tworzą efekt „kuli śnieżnej”, prowadząc do większych problemów.

W dalszej części wypowiedzi kontynuowała: „To zuchwałe wyzwanie. W naszym obecnym stanie mamy wiele problemów — czasy procesu i cyklu w zespołach ani nie są konsekwentnie mierzone, ani nie są widoczne. Spełnienie pierwszego warunku docelowego wymaga udzielenia pomocy wszystkim zespołom w mierzeniu, poprawianiu widoczności i przeprowadzaniu eksperymentów zmierzających do skrócenia czasu przetwarzania — iteracja po iteracji”.

Kissler stwierdziła na koniec: „Z perspektywy wysokiego szczebla wierzymy, że zastosowanie takich technik, jak mapowanie strumienia wartości, zmniejszenie wielkości partii w celu osiągnięcia przepływu jednej sztuki, a także zastosowanie ciągłych dostaw i mikrousług doprowadzi nas do pożądanego stanu. Jednak mimo że wciąż się uczymy, jesteśmy przekonani, że zmierzamy we właściwym kierunku, a wszyscy wiedzą, że podejmowane wysiłki mają wsparcie od najwyższych poziomów zarządzania”.

W niniejszym rozdziale zaprezentowano różne modele — które pozwalają replikować procesy myślowe wykorzystane przez zespół w firmie Nordstrom — używane do podjęcia decyzji dotyczących strumieni wartości, od których należy zacząć. Dokonamy oceny strumieni wartości-kandydatów na wiele sposobów, w tym pod kątem tego, czy są one usługami *greenfield* lub *brownfield*, systemem **SoE** (ang. *system of engagement*), czy **SoR** (ang. *system of record*). Oszacujemy także bilans zagrożeń i zysków z transformacji oraz ocenimy prawdopodobny możliwy poziom oporu zespołów, z którymi będziemy pracować.

USŁUGI GREENFIELD A BROWNFIELD

Usługi bądź produkty oprogramowania często są przydzielane do kategorii *greenfield* bądź *brownfield*. Pojęcia te były pierwotnie używane do planowania urbanistycznego i projektów budowlanych. Projekt określamy jako greenfield, gdy budujemy na terenie wcześniej niezagospodarowanym. Projekt opisujemy jako brownfield, gdy budujemy na terenie, który wcześniej był używany do celów przemysłowych, jest potencjalnie skażony niebezpiecznymi odpadami lub zanieczyszczeniami. Projekty greenfield w urbanistyce z wielu powodów są prostsze niż projekty brownfield — nie istnieją żadne struktury, które muszą zostać rozebrane, ani nie ma toksycznych materiałów, które muszą być usunięte.

Projekt greenfield w technologii oznacza nowy projekt oprogramowania lub nową inicjatywę, najczęściej w początkowej fazie planowania lub implementacji, kiedy budujemy aplikację i jej infrastrukturę od nowa, z niewielką liczbą ograniczeń. Zaczynamy od projektu oprogramowania greenfield może być łatwiejsze, zwłaszcza jeśli projekt ma już finansowanie, a zespół albo jest tworzony, albo jest już gotowy do pracy. Ponadto ze względu na to, że zaczynamy od podstaw, nie musimy tak bardzo martwić się istniejącymi bazami kodu, procesami i zespołami.

Projekty DevOps typu greenfield są często realizowane jako przedsięwzięcia pilotażowe, które mają na celu wykazanie możliwości skorzystania z publicznych lub prywatnych chmur, prezentują automatyczne wdrażanie oraz użycie podobnych narzędzi. Przykładem projektu DevOps, który można zakwalifikować jako greenfield, jest produkt Hosted LabVIEW opracowany w 2009 roku w National Instruments, liczącej 30 lat organizacji z 5 tysiącami pracowników i 1 mld dolarów rocznych przychodów. Aby szybko wprowadzić ten produkt na rynek, stworzono nowy zespół. Pozwolono mu działać poza istniejącymi procesami IT oraz badać zastosowanie publicznej chmury. Pierwotnie zespół składał się z architekta aplikacji, architektów systemowych, dwóch programistów, dewelopera automatyzacji systemów, lidera operacyjnego oraz dwóch zewnętrznych pracowników działu operacji. Dzięki wykorzystaniu praktyk DevOps zespół zdołał dostarczyć produkt Hosted LabVIEW na rynek w ciągu połowy czasu w porównaniu z typowymi przedsięwzięciami tego rodzaju.

Na drugim końcu spektrum są projekty DevOps typu brownfield. Dotyczą one istniejących produktów lub usług, które już obsługują klientów i potencjalnie są w użytkowaniu od wielu lat lub nawet dziesięcioleci. Projekty brownfield często charakteryzują się znaczącym długiem technicznym — na przykład brakuje w nich automatyzacji testów lub działają na nieobsługiwanych platformach. W przykładzie firmy Nordstrom zaprezentowanym wcześniej w tym rozdziale, zarówno systemy restauracji, jak i systemy e-commerce były projektami brownfield.

Chociaż wiele osób uważa, że infrastruktura DevOps dotyczy przede wszystkim projektów greenfield, to zasady DevOps z sukcesem stosowano w celu transformacji różnego rodzaju projektów brownfield. W rzeczywistości ponad 60% opisów transformacji prezentowanych na konferencji DevOps Enterprise Summit w 2014 roku dotyczyło projektów brownfield. W tych przypadkach występowała olbrzymia luka wydajności pomiędzy tym, co było potrzebne klientom, a tym, co aktualnie dostarczały organizacje. Przekształcenia DevOps przyniosły tym firmom ogromne korzyści biznesowe.

Zgodnie z jednym z wniosków raportu *State of DevOps* z 2015 roku to nie wiek aplikacji jest istotnym predyktorem jej wydajności. O wydajności decyduje raczej to, czy aplikacja została zaprojektowana (lub można zmienić jej projekt) tak, aby zapewnić możliwości jej testowania i automatycznego wdrażania.

Zespoły wspierające projekty brownfield mogą być bardzo otwarte na eksperymentowanie z DevOps, szczególnie gdy istnieje powszechnie przekonanie, że tradycyjne metody są niewystarczające do osiągnięcia celów — zwłaszcza jeśli istnieje silne poczucie pilności potrzeby poprawy^{*}.

* Nie powinno być zaskoczeniem, że usługi, które mają największy potencjał korzyści biznesowych, są systemami brownfield. Ostatecznie są to systemy, które są najczęściej wykorzystywane, mają najwięcej istniejących klientów lub generują najwyższą kwotę przychodów.

Podczas przekształcania projektów brownfield możemy napotkać wiele znaczących utrudnień i problemów. Są one najbardziej dotkliwe szczególnie wtedy, gdy nie dysponujemy mechanizmami automatycznego testowania albo gdy mamy do czynienia ze ścisłe sprężoną architekturą, która nie pozwala małym zespołom na niezależne rozwijanie, testowanie i wdrażanie kodu. W tej książce opisano, w jaki sposób można rozwiązać te problemy.

Oto przykłady udanych transformacji brownfield:

- **CSG (2013).** W 2013 roku firma CSG International miała 747 mln dolarów przychodów i ponad 3500 pracowników. Świadczyła usługi dla ponad 90 000 agentów zajmujących się usługami rozliczania faktur i obsługą księgową dla ponad 50 milionów użytkowników korzystających z kanałów wideo, głosowych i danych. Łączna liczba przeprowadzanych przez nich transakcji wynosiła ponad 6 miliardów, a liczba drukowanych i wysyłanych pocztą papierowych faktur każdego miesiąca sięgała 70 milionów. Pierwszym obszarem usprawnień w firmie było drukowanie faktur — jedna z podstawowych jej działalności. Wykorzystywano aplikację mainframe w COBOL-u i 20 dodatkowych platform technologicznych. W ramach transformacji zaczęto realizować codziennie wdrożenia w środowisku zbliżonym do produkcyjnego oraz podwojono częstotliwość publikacji dla klienta z dwóch do czterech razy w roku. W rezultacie znaczco poprawiła się niezawodność aplikacji, a czasy realizacji wdrożenia skrócono z dwóch tygodni do mniej niż jednego dnia.
- **Etsy (2009).** W 2009 firma Etsy zatrudniała 35 pracowników i generowała 87 mln dolarów przychodów, ale po wakacyjnym okresie sprzedaży, który firma ledwo przeżyła, rozpoczęto przekształcanie praktycznie wszystkich aspektów organizacji pracy. Ostatecznie przekształcono firmę Etsy w jedną z najbardziej szanowanych organizacji DevOps oraz doprowadzono do udanego jej debiutu na giełdzie w 2015 roku.

SYSTEMY SOR I SOE

Firma badawcza Gartner niedawno spopularyzowała pojęcie bimodalnego IT, odnoszące się do szerokiego spektrum usług świadczonych przez typowe przedsiębiorstwa. W ramach bimodalnego IT można wyróżnić **systemy SoR** (ang. *systems of record*) — podobne do ERP systemy sterujące działaniem firmy (np. systemy MRP, HR, sprawozdawczości finansowej). W systemach SoR kluczowe znaczenie ma poprawność transakcji i danych. Drugim rodzajem są **systemy SoE** (ang. *systems of engagement*). Ich użytkownikami są klienci lub pracownicy — są to na przykład systemy e-commerce i aplikacje poprawiające wydajność pracowników.

Systemy SoR zwykle charakteryzują się wolniejszym tempem zmian. Często podlegają wymaganiom związanym ze zgodnością z przepisami (np. ustanowione ochronie danych osobowych). W firmie Gartner te rodzaje systemów określano jako „typu 1”. W firmie koncentrowano się na tym, aby obsługując zmiany w tych systemach, „robić to dobrze”.

Systemy SoE zazwyczaj charakteryzują się znacznie szybszym tempem zmian, ponieważ muszą obsługiwać szybkie sprzężenia zwrotne, które umożliwiają eksperymentowanie w celu jak najlepszego spełnienia potrzeb klientów. W firmie Gartner te rodzaje systemów określano jako „typu 2”. Koncentrowano się w niej na tym, aby obsługując zmiany w tych systemach, „robić to szybko”.

Podział systemów na te dwie kategorie jest dość wygodny. Wiemy jednak, że dzięki transformacji DevOps można przerwać podstawowy, przewlekły konflikt pomiędzy „robić to dobrze” a „robić to szybko”. Dane z raportów State of DevOps generowanych przez Puppet Labs — oraz wnioski wyciągnięte ze stosowania metodyki Lean w produkcji — dowodzą, że wysokowydajne organizacje są w stanie jednocześnie zapewnić wyższy poziom wydajności i niezawodności.

Ponadto ze względu na współzależności pomiędzy systemami zdolność wprowadzania zmian w dowolnych z tych systemów jest ograniczona przez system, który jest najtrudniejszy do bezpiecznej zmiany. Prawie zawsze dotyczy to systemu SoR.

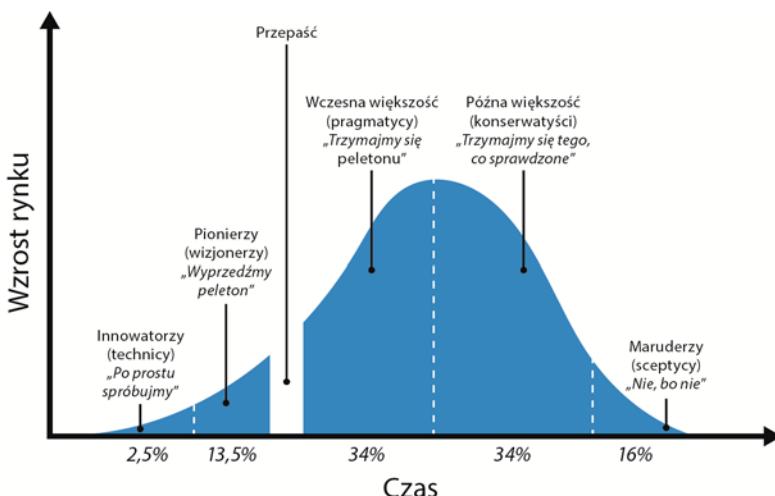
Scott Prugh, wiceprezes ds. rozwoju produktu w firmie CSG, zauważał: „Przyjęliśmy filozofię, która odrzuca bimodalny model IT, ponieważ wszyscy nasi klienci zasługują zarówno na szybkość, jak i na jakość. To oznacza, że potrzebujemy doskonałości technicznej niezależnie od tego, czy zespół obsługuje liczącą ponad 30 lat aplikację mainframe, aplikację Javy, czy aplikację mobilną”.

W związku z tym, gdy usprawniamy systemy brownfield, powinniśmy nie tylko dążyć do ograniczenia ich złożoności i poprawy niezawodności i stabilności, ale również powinniśmy starać się, aby można było zmieniać je szybciej, bezpieczniej i łatwiej. Nawet wtedy, gdy nowe funkcje są dodawane tylko do systemów SoE typu greenfield, to często wprowadzane zmiany powodują problemy niezawodności w systemach brownfield SoR, od których modyfikowane systemy zależą. Poprawiając bezpieczeństwo systemów w dole strumienia wartości, pomagamy w szybszym i bezpieczniejszym osiągnięciu celów całej organizacji.

ZACZNIJ OD ZESPOŁÓW NAJBARDZIEJ POZYTYWNIE NASTAWIONYCH NA INNOWACJE

W każdej organizacji są zespoły i pojedyncze osoby prezentujące różne postawy wobec przyjęcia nowych pomysłów. To spektrum po raz pierwszy zaprezentował Geoffrey A. Moore w książce *Przeskoczyć przepaść* jako cykl życia akceptacji technologii.

Przepaść w tytule reprezentuje klasyczne trudności dotarcia do grup innych niż **innowatorzy** (ang. *innovators*) i **pionierzy** (ang. *early adopters*) (rysunek 9).



Rysunek 9. Krzywa akceptacji technologii (źródło: G. Moore, „Preskoczyć przepaść”, s. 15)

Innymi słowy, nowe pomysły są często szybko akceptowane przez innowatorów i pionierów, natomiast grupy charakteryzujące się bardziej konserwatywnymi postawami (**postępowa większość, konserwatywna większość i maruderzy**) potrzebują więcej czasu. Naszym celem powinno być znalezienie tych zespołów, które wierzą w konieczność zastosowania zasad i praktyk DevOps oraz które wyraziły chęć oraz wykazały zdolność wprowadzania innowacji i usprawniania własnych procesów. Najlepiej byłoby, gdyby te grupy były entuzjastycznymi zwolennikami transformacji DevOps.

Nie należy poświęcać zbyt dużo czasu na próby przekonywania bardziej konserwatywnych grup, zwłaszcza w początkowych fazach transformacji. Lepiej skupić energię na osiągnięciu sukcesu z zespołami, które mniej boją się ryzyka, i wykorzystać je do stworzenia bazy (ten proces omówiono w następnym podrozdziale). Nawet wtedy, gdy możemy liczyć na najwyższy poziom poparcia kierownictwa, powinniśmy unikać podejścia **wchodzenia z hukiem** (tzn. rozpoczęmania transformacji wszędzie i przeprowadzania ich wszystkich naraz). Zamiast tego powinniśmy skoncentrować wysiłki na kilku obszarach organizacji, zadbać o to, aby te inicjatywy były udane, i na tej podstawie rozszerzać transformacje*.

* Transformacje „z hukiem” typu góra-dół są możliwe. Przykładem może być transformacja Agile w firmie PayPal, przeprowadzona w 2012 r. pod kierownictwem wiceprezesa firmy ds. technologii Kirsten Wolberg. Jednak podobnie jak w przypadku każdej zrównoważonej i udanej transformacji, wymagało to najwyższego poziomu wsparcia ze strony kierownictwa oraz nieustępnej, ciągłej koncentracji na uzyskaniu wymaganych wyników.

ROZWIJANIE METODYKI DEVOPS W ORGANIZACJI

Bez względu na zakres początkowych wysiłków trzeba zademonstrować wczesne zwycięstwa oraz rozpropagować informacje o osiągniętych sukcesach. Aby to zrobić, należy podzielić większe cele poprawy na małe, przyrostowe etapy. To pozwala nie tylko szybciej wprowadzać usprawnienia, ale również wykryć nieprawidłowy wybór strumienia wartości. Dzięki wczesnemu wykrywaniu popełnionych błędów można szybko się z nich wycofać i spróbować ponownie, podejmując różne decyzje na podstawie zdobytej nowej wiedzy.

Dzięki pierwszym sukcesom zdobywamy prawo do rozszerzenia zakresu inicjatywy DevOps. Należy postępować zgodnie z bezpieczną sekwencją, która pozwala metodycznie budować nasz poziom wiarygodności, wpływów i poparcia. Na poniższej liście, zaczerpniętej z kursu prowadzonego przez dr. Roberta Fernandeza, William F. Pounds, profesor zarządzania w MIT, opisuje idealne fazy stosowane przez agentów zmiany do budowania i rozwijania koalicji i bazy poparcia:

1. **Znajdź innowatorów i pionierów.** Na początek należy skoncentrować wysiłki na zespołach, które rzeczywiście chcą pomocy — będą one naszymi towarzyszami, którzy jako pierwi, na ochotnika, zgodzili się rozpocząć podróż DevOps. W idealnym świecie powinni to być ludzie, którzy cieszą się uznaniem oraz mają wysoki stopień wpływu na pozostałą część organizacji. Dzięki temu nasza inicjatywa zyskuje większą wiarygodność.
2. **Zbuduj masę krytyczną i milczącą większość.** W następnej fazie staramy się rozwinać praktyki DevOps dla kolejnych zespołów i strumieni wartości; celem tej fazy jest stworzenie stabilnej bazy poparcia. Dzięki pracy z zespołami otwartymi na nasze pomysły — nawet jeśli nie są to grupy najbardziej widoczne lub wpływowe — można rozwinać koalicję, wygenerować więcej sukcesów oraz stworzyć „efekt mody”, który jeszcze bardziej zwiększy nasze wpływy. W szczególności należy obchodzić wszelkie niebezpieczne walki polityczne, które mogłyby zagrozić naszej inicjatywie.
3. **Zidentyfikuj „hamulcowych”.** „Hamulcowi” to budzący respekt, wpływowi przeciwnicy, którzy z największym prawdopodobieństwem będą się sprzeciwiali wprowadzanym zmianom (a może nawet przeszkladzali). Ogólnie rzecz biorąc, tymi grupami należy się zająć dopiero po uzyskaniu milczącej większości, gdy osiągneliśmy wystarczająco dużo sukcesów, aby skutecznie obronić naszą inicjatywę.

Rozszerzanie inicjatywy DevOps w organizacji nie jest prostym zadaniem. Może stwarzać zagrożenie dla indywidualnych osób, działów oraz organizacji jako całości. Ale jak powiedział Ron van Kemenade, CIO firmy ING, który pomógł przekształcić organizację w jedną z najbardziej podziwianych organizacji technicznych: „Przewodzenie

zmianom wymaga odwagi, zwłaszcza w środowiskach korporacyjnych, gdzie ludzie boją się zmian i bronią się przed nimi. Ale jeśli zaczniemy od zmian na małą skalę, to naprawdę nie ma się czego obawiać. Każdy lider musi być wystarczająco odważny, by zlecić zespołom podjęcie ograniczonego ryzyka”.

PODSUMOWANIE

Jak mówi Peter Drucker, lider edukacji rozwoju zarządzania, „małe rybki w małych stawach uczą się, jak stać się wielkimi rybami”. Dzięki podjęciu uważnych decyzji o tym, gdzie i od czego zacząć, możemy eksperymentować i uczyć się w tych obszarach organizacji, które tworzą wartość, bez narażania reszty organizacji. W ten sposób budujemy naszą bazę poparcia, zdobywamy prawo do rozszerzania zastosowania DevOps w naszej organizacji oraz zyskujemy uznanie i wdzięczność coraz większego grona osób.

6

Zrozumienie pracy w strumieniu wartości, zapewnienie jej widoczności i rozszerzenie zrozumienia na całą organizację

Po zidentyfikowaniu strumienia wartości, do którego chcemy zastosować zasady i wzorce DevOps, następnym krokiem powinno być zrozumienie sposobu dostarczania wartości do klienta: jaka praca jest wykonywana i przez kogo oraz jakie kroki można podjąć w celu poprawy przepływu.

W poprzednim rozdziale omawialiśmy transformacje DevOps prowadzone przez Courtney Kissler oraz zespół firmy Nordstrom. Z biegiem lat w firmie przekonano się, że jednym z najskuteczniejszych sposobów rozpoczęcia poprawy dowolnego strumienia wartości jest prowadzenie warsztatów z wszystkimi najważniejszymi interesariuszami oraz przeprowadzanie ćwiczeń z mapowaniem strumienia wartości — procesu (opisanego w dalszej części tego rozdziału) zaprojektowanego w celu znalezienia wszystkich kroków wymaganych do stworzenia wartości.

Ulubionym podawanym przez Kissler przykładem nieoczekiwanych i cennych spostrzeżeń, które mogą pochodzić z mapowania strumienia wartości, są wnioski z prób poprawy długich okresów realizacji żądań z napisanej w COBOL-u aplikacji mainframe Cosmetics Business Office, która obsługiwała wszystkich menedżerów pięter i stoisk z kosmetykami.

Aplikacja ta pozwalała kierownikom działów rejestrować nowych sprzedawców dla różnych linii produktów sprzedawanych w podległych im sklepach. Dzięki temu można było monitorować prowizje od sprzedaży, realizować rabaty dostawców itd.

Kissler wyjaśniła:

Dobrze znałam tę aplikację mainframe — wcześniej w mojej karierze wspierałam ten zespół techniczny, więc wiem z pierwszej ręki, że przez prawie dekadę, podczas każdego rocznego cyklu planowania, debatowano o potrzebie przeniesienia tej aplikacji z platformy mainframe. Oczywiście, podobnie jak w większości organizacji, nawet gdyby było pełne wsparcie kierownictwa, nigdy nie wydawaliśmy się gotowi do migracji.

Mój zespół chciał przeprowadzić ćwiczenie mapowania strumienia wartości, aby stwierdzić, czy prawdziwym problemem była aplikacja w COBOL-u, czy może istniał większy problem, którym należało się zająć. Przeprowadziliśmy warsztaty, w których wzięły udział wszystkie osoby zaangażowane w dostarczanie wartości do naszych wewnętrznych klientów, w tym partnerzy biznesowi, zespół obsługi mainframe, zespoły współdzielonych usług itd.

Odkryto, że kiedy kierownicy działów przesyłali formularz „przydzielenia linii produktu”, wyświetlało się pytanie o numer pracownika, który nie był znany. W związku z tym albo pozostawiano to pole puste, albo wprowadzano w nim coś w stylu „Nie wiem”. Co gorsza, aby wypełnić formularz, kierownicy działów musieli opuścić sklep w celu skorzystania z komputera na zapleczu. W rezultacie tracono czas, a praca w procesie była przesyłana tam i z powrotem.

Uczestnicy warsztatów przeprowadzili kilka eksperymentów, w tym usuwanie pola numeru pracownika w formularzu oraz zlecanie innemu działowi pobrania tej informacji w dalszym kroku procesu. Eksperymenty te, prowadzone przy pomocy kierowników działów, pozwoliły skrócić czas przetwarzania o cztery dni. Później zespół zastąpił aplikację PC aplikacją na iPada, co umożliwiło menedżerom przesyłanie niezbędnych informacji bez opuszczania sklepu, co z kolei pozwoliło skrócić czas przetwarzania do kilku sekund.

Kissler z dumą stwierdziła: „Dzięki tym niesamowitym ulepszeniom zniknęły wszelkie wymagania przeniesienia aplikacji z platformy mainframe. Ponadto inni liderzy biznesowi zauważyl osiągnięte korzyści i zaczęli zwracać się do nas z całą listą dalszych eksperymentów, które chcieli prowadzić wraz z nami w swoich własnych organizacjach. Wszyscy członkowie zespołów biznesowych i technicznych byli podekscytowani wynikami, ponieważ rozwiązywały one rzeczywisty problem biznesowy i co najważniejsze, wprowadzenie zmian czegoś ich nauczyło”.

W pozostałej części tego rozdziału omówimy następujące tematy: zidentyfikowanie wszystkich zespołów wymaganych do stworzenia wartości dla klienta, utworzenie mapy strumienia wartości w celu uwidocznienia wszystkich potrzebnych prac oraz wyko-

rzystanie ich jako przewodnika dla zespołów, opisującego lepszy i szybszy sposób tworzenia wartości. Dzięki temu będziemy mogli powtórzyć niesamowite wyniki opisane w przykładzie firmy Nordstrom.

IDENTYFIKACJA ZESPOŁÓW UCZESTNICZĄCYCH W STRUMIENIU WARTOŚCI

Jak pokazuje przykład firmy Nordstrom, w strumieniach wartości o odpowiednio dużym poziomie złożoności żadna osoba nie ma świadomości wszystkich prac, które należy wykonać w celu stworzenia wartości dla klienta. Jest tak przede wszystkim dlatego, że prace muszą być wykonywane przez wiele różnych zespołów, często bardzo od siebie odległych na schematach organizacyjnych, geograficznie lub ze względu na różne motywy działania.

W rezultacie po wyborze aplikacji lub usługi do przeprowadzenia inicjatywy DevOps trzeba zidentyfikować wszystkich członków strumienia wartości odpowiedzialnych za wspólną pracę w celu stworzenia wartości dla obsługiwanych klientów. Ogólnie rzecz biorąc, są to następujące osoby:

- **Właściciel produktu** — menedżer, który definiuje następny zestaw funkcjonalności usługi.
- **Zespół deweloperów** — grupa osób odpowiedzialnych za rozwijanie funkcjonalności aplikacji w usłudze.
- **Zespół validacji (zapewnienia jakości — QA)** — grupa osób odpowiedzialnych za zapewnienie pętli sprzężenia zwrotnego po to, by funkcje działały zgodnie z projektem.
- **Zespół operacyjny** — grupa odpowiedzialna za utrzymanie środowiska produkcyjnego i pomoc w zapewnieniu wymaganego poziomu usług.
- **Zespół zabezpieczeń** — grupa odpowiedzialna za zabezpieczenie systemów i danych.
- **Menedżerowie wydania** — osoby odpowiedzialne za zarządzanie i koordynowanie procesów wdrażania i publikowania wersji produkcyjnych.
- **Menedżerowie techniczni lub menedżer strumienia wartości** — w literaturze Lean w ten sposób określa się osobę odpowiedzialną za „zadbanie o to, aby strumień wartości spełniał lub przekraczał wymagania klienta (i organizacji) dla całego strumienia wartości — od początku do końca”.

TWORZENIE MAPY STRUMIENIA WARTOŚCI W CELU ZAPEWNIENIA WIDOCZNOŚCI PRACY

Następnym krokiem po zidentyfikowaniu członków strumienia wartości jest uzyskanie wiedzy dotyczącej sposobu wykonywania pracy. Wiedza ta powinna być udokumentowana w formie mapy strumienia wartości. W strumieniu wartości praca najczęściej zaczyna się od właściciela produktu. Ma postać żądania klienta lub formuły biznesowej hipotezy. Jakiś czas później praca ta jest akceptowana przez zespół deweloperów, gdzie funkcje są implementowane w kodzie i składowane w repozytorium kontroli wersji. Kompilacje są następnie integrowane, testowane w środowisku zbliżonym do produkcyjnego i ostatecznie wdrażane w środowisku produkcyjnym, gdzie (w idealnej sytuacji) tworzą wartość dla klientów.

W wielu tradycyjnych organizacjach strumień wartości składa się z kilkuset, a czasem nawet kilku tysięcy etapów, które wymagają pracy setek ludzi. Ze względu na to, że udokumentowanie dowolnej mapy strumienia wartości o takim poziomie złożoności prawdopodobnie wymagałoby wielu dni, można przeprowadzić wielodniowe warsztaty, podczas których zostaną zmontowane wszystkie kluczowe składniki.

Celem nie jest udokumentowanie każdego etapu i powiązanych z nim drobnych szczegółów, ale zrozumienie tych obszarów strumienia wartości, które są zagrożeniem dla osiągnięcia celu szybkiego przepływu, krótkich terminów realizacji i wiarygodnych wyników dla klientów. W idealnej sytuacji należałoby zebrać osoby mające kompetencje do zmian w swojej części strumienia wartości.*

Damon Edwards, współgospodarz podcastu *DevOps Café*, zauważyl: „Z mojego doświadczenia wynika, że te rodzaje ćwiczeń mapowania strumienia wartości zawsze otwierają oczy. Często to jest pierwsza okazja do tego, by zobaczyć, jak wiele pracy potrzeba do dostarczenia wartości do klienta. Dla członków zespołu operacji może to być pierwsza okazja do zaobserwowania konsekwencji braku dostępu deweloperów do poprawnie skonfigurowanych środowisk — co przyczynia się do jeszcze bardziej szalonej pracy podczas wdrażania kodu. Dla zespołu deweloperów może to być pierwsza okazja, by dowiedzieć się, ile pracy muszą włożyć zespoły validacji i operacji — na długo po tym, gdy funkcja zostanie oznaczona jako »gotowa« — żeby wdrożyć ich kod do produkcji”.

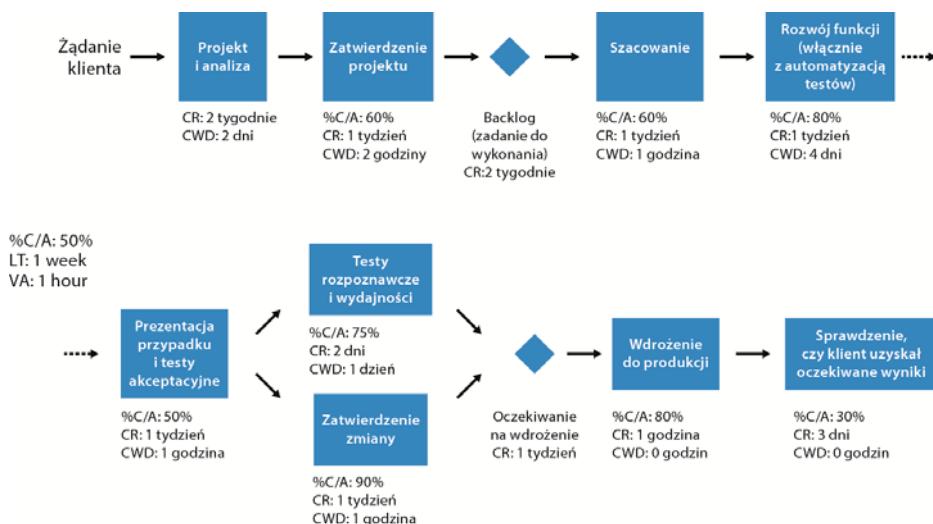
Używając całego zakresu wiedzy wnoszonej przez zespoły zaangażowane w strumień wartości, należy skoncentrować badania i zarządzanie na następujących obszarach:

- Miejscach, gdzie praca musi czekać wiele tygodni lub nawet miesięcy — przykładowo konfigurowanie środowisk przypominających produkcyjne, zmiana procesów zatwierdzania lub procesów weryfikowania zabezpieczeń.

* Dlatego bardzo ważne jest ograniczenie liczby zbieranych szczegółów — czas każdej osoby jest cenny.

- Miejscach, w których są generowane lub wykonywane znaczące przeróbki.

Pierwszy przebieg dokumentowania strumienia wartości powinien zawierać wyłącznie bloki procesu wysokiego poziomu. Zazwyczaj nawet w przypadku złożonych strumieni wartości, w ciągu kilku godzin grupy mogą utworzyć diagram zawierający od 5 do 15 bloków procesów. Przykład mapy strumienia wartości zaprezentowano na rysunku 10. Każdy blok procesu powinien zawierać czas realizacji i czas przetwarzania elementu pracy, jak również współczynnik %C/A, mierzony przez konsumentów danych wyjściowych w dole strumienia*.



Wartości łączne:

Całkowity czas realizacji (CR): 10 tygodni
 Czas tworzenia wartości dodanej (CWD): 7,5 dnia
 Procent kompletności i gotowości (C/A%): 8,6%

Rysunek 10. Przykład mapowania strumienia wartości (źródło: Humble, Molesky i O'Reilly, „Lean Enterprise”, s. 139)

Parametry z mapy strumienia wartości powinny być wykorzystywane jako przewodnik do usprawniania. W podawanej jako przykład firmie Nordstrom skoncentrowano się na niskich wartościach współczynnika %C/A na formularzu żądania przekazanym przez kierowników działów — z powodu niedoboru pracowników. W innych przypadkach mogą to być długie czasy realizacji lub niski współczynnik %C/A przy dostarczaniu poprawnie skonfigurowanych środowisk testowych do zespołów deweloperów

* Z drugiej strony istnieje wiele przykładów użycia narzędzi w sposób, który gwarantuje brak zmian zachowań. Na przykład organizacja zobowiązuje się do użycia narzędzi planowania Agile, a następnie konfiguruje planowanie zgodnie z procesem kaskadowym, co jedynie utrzymuje status quo.

albo długie okresy realizacji potrzebne do uruchomienia i doprowadzenia do przejścia wszystkich testów regresji przed wydaniem każdej wersji oprogramowania.

Po zidentyfikowaniu parametrów, które chcemy poprawić, należy przejść do następnego etapu obserwacji i pomiarów w celu lepszego zrozumienia problemu. Następnie należy skonstruować wyidealizowaną przyszłą mapę strumienia wartości, która służy jako warunek docelowy do osiągnięcia w określonym terminie (zwykle w perspektywie od 3 do 12 miesięcy).

Ten przyszły stan jest definiowany przez kierownictwo. Następnie menedżerowie umożliwiają zespołom przeprowadzenie burzy mózgów, aby wypracować hipotezy i środki zaradcze w celu osiągnięcia pożąданej poprawy tego stanu; wykonują eksperymenty zmierzające do testowania hipotez i interpretacji wyników, aby ustalić, czy hipotezy były prawidłowe. Zespoły przeprowadzają kolejne iteracje eksperymentów, używając nowej wiedzy jako danych wejściowych.

STWORZENIE DEDYKOWANEGO ZESPOŁU TRANSFORMACJI

Jedno z wyzwań związanych z takimi inicjatywami, jak transformacje DevOps, polega na tym, że są one niewątpliwie sprzeczne z bieżącymi działaniami biznesowymi. Po części jest to naturalny efekt ewolucji firm, które odniosły sukces. Dowolna organizacja, której udało się skutecznie działać przez dłuższy czas (kilka lat, dziesięcioleci lub nawet stuleci), stworzyła mechanizmy podtrzymywania praktyk, które pozwoliły jej odnieść sukces — takich jak rozwój produktu, zarządzanie zleceniami oraz obsługa łańcucha dostaw.

Dostępnych jest wiele technik służących podtrzymaniu i ochronie sposobu działania bieżących procesów pracy — przykładowo specjalizacja, koncentracja na efektywności i powtarzalności, biurokracja wymuszająca procesy zatwierdzania oraz kontrola w celu ochrony przed odchyleniami od przyjętych norm. W szczególności mechanizmy biurokratyczne są niezwykle odporne. Są zaprojektowane tak, by mogły przetrwać niekorzystne warunki — można usunąć połowę biurokratów, a proces będzie nadal funkcjonował.

O ile jest to dobre dla zachowania status quo, o tyle trzeba pamiętać, że zmiany często są nieuniknione — przykładowo trzeba zmienić sposób pracy w celu adaptacji do zmieniających się warunków na rynku. Przeprowadzanie tego rodzaju zmian wymaga zakłóceń i innowacji, co stawia nas w sytuacji konfliktu z grupami, które obecnie są odpowiedzialne za codzienne operacje, oraz z wewnętrzna biurokracją, która prawie zawsze wygrywa.

Dr Vijay Govindarajan i dr Chris Trimble, obaj pracownicy naukowi Tuck School of Business przy Dartmouth College, w książce *The Other Side of Innovation: Solving the Execution Challenge* opisali swoje badania na temat skutecznego wprowadzania

innovacji generujących zakłócenia wbrew tym potężnym siłom sterującym codzijnymi działańami. Udokumentowali oni skuteczne opracowanie i wprowadzenie na rynek ukierunkowanych na klienta produktów ubezpieczeniowych w firmie Allstate, stworzenie dochodowej cyfrowej działalności wydawniczej w „Wall Street Journal”, zaprojektowanie przełomowych butów do biegania w terenie w firmie Timberland oraz stworzenie pierwszego samochodu elektrycznego w BMW.

Na podstawie swoich badań doktorzy Govindarajan i Trimble twierdzą, że organizacje muszą wyłonić dedykowany zespół transformacji, który może działać oddzielnie od pozostałych zespołów w organizacji odpowiedzialnych za codzienne działania biznesowe. Ta grupa osób jest przez nich nazywana „zespołem dedykowanym” lub „silnikiem wydajności”.

Co najważniejsze, zespołowi dedykowanemu zostanie powierzone zadanie osiągnięcia czytelnie zdefiniowanego, wymiernego wyniku na poziomie systemu (np. „skrócenie czasu wdrożenia kodu — od momentu przesłania go do repozytorium kontroli wersji do pomyślnego wdrożenia do produkcji — o 50%”). W celu zrealizowania takiej inicjatywy należy wykonać następujące czynności:

- Przydzielić członkom zespołu dedykowanego zadania związane wyłącznie z transformacją DevOps (w przeciwieństwie do „utrzymania wszystkich bieżących obowiązków i dodatkowo poświęcenia 20% czasu na realizację zadań związanych z DevOps”).
- Wybrać do tego zespołu osoby, które mają przekrojową wiedzę i umiejętności z wielu dziedzin.
- Wybrać członków zespołu, którzy mają długieletnie i bazujące na wzajemnym szacunku relacje z resztą organizacji.
- Jeśli to możliwe, stworzyć oddzielną, fizyczną przestrzeń pracy dla tego zespołu. Pozwoli to zmaksymalizować przepływ komunikacji w ramach zespołu i odizoluje go od reszty organizacji.

Jeśli to możliwe, to członków zespołu dedykowanego należy zwolnić ze stosowania wielu reguł i zasad, które ograniczają resztę organizacji. Właśnie tak postąpiono w firmie National Instruments opisanej w poprzednim rozdziale. Ustanowione procesy są formą instytucjonalnej pamięci — potrzebujemy zespołu dedykowanego do stworzenia nowych procesów i wiedzy wymaganej do wygenerowania pożądanych rezultatów. Dzięki temu tworzymy nową pamięć instytucjonalną.

Tworzenie zespołu dedykowanego jest dobre nie tylko dla organizacji, ale również dla wydajności. Tworząc odrębny zespół, tworzymy przestrzeń do eksperymentowania z nowymi praktykami, a jednocześnie chronimy organizację przed potencjalnymi związanymi z tym zakłóceniami.

UZGODNIENIE WSPÓLNEGO CELU

Jednym z najważniejszych elementów każdej inicjatywy usprawniania jest określenie wymiernego celu, z jasno określonym terminem — od sześciu miesięcy do dwóch lat. Termin powinien być wyznaczony tak, aby jego dotrzymanie wymagało znaczącego wysiłku, ale by był osiągalny. Osiągnięcie celu powinno stworzyć oczywistą wartość dla organizacji jako całości oraz dla klientów.

Wyznaczone cele oraz ramy czasowe powinny być uzgodnione z kierownictwem i znane wszystkim osobom w organizacji. Należy również ograniczyć liczbę podobnych inicjatyw realizowanych równolegle, aby nie dopuścić do zbytniego obciążenia liderów i organizacji zadaniami zarządzania zmianami. Oto przykładowe cele poprawy:

- Zmniejszenie odsetka budżetu przeznaczonego na wsparcie produktu oraz niezaplanowanej pracy o 50%.
- Dla 95% zmian osiągnięcie czasu od przesłania ich do repozytorium kodu do wdrożenia do produkcji jednego tygodnia lub krótszego.
- Zapewnienie realizacji wydań zawsze w zwykłych godzinach pracy — bez przestojów.
- Zintegrowanie wszystkich wymaganych mechanizmów kontroli zabezpieczeń z potokiem wdrażania w celu spełnienia wszystkich potrzebnych wymagań zgodności.

Po czytelnym sformułowaniu wysokopoziomowego celu zespoły powinny opracować regularną kadencję zarządzania pracami związanymi z usprawnieniami. Należy dążyć do tego, aby prace dotyczące transformacji były wykonywane w sposób iteracyjny i przyrostowy, podobnie jak prace związane z rozwojem produktu. Typowa iteracja powinna trwać od dwóch do czterech tygodni. W każdej iteracji zespół powinien uzgodnić niewielki zestaw celów, które generują wartość, i zrobić postępy w kierunku osiągnięcia celu długoterminowego. Pod koniec każdej iteracji zespoły powinny dokonać przeglądu postępów i ustalić cele dla następnej iteracji.

ZACHOWANIE KRÓTKOTERMINOWEGO HORYZONTU PLANOWANIA POPRAWY

W każdym projekcie transformacji DevOps trzeba zadbać o utrzymanie krótkoterminowych horyzontów planowania — podobnie jak początkujące firmy pracują nad rozwojem produktu lub obsługą klientów. W podejmowanych inicjatywach należy dążyć do generowania wymiernej poprawy lub opracowania danych dających podstawę do działania w ciągu tygodni (lub w najgorszym przypadku miesięcy).

Utrzymanie krótkoterminowego horyzontu planowania i krótkich przedziałów iteracji pozwala osiągnąć następujące cele:

- Elastyczność, możliwość szybkiej zmiany priorytetów i planów.
- Zmniejszenie opóźnienia pomiędzy wydatkowaną pracą a zrealizowaną poprawą. To wzmacnia pętlę sprzężenia zwrotnego i zwiększa prawdopodobieństwo wzmocnienia pożądanych zachowań — sukces w inicjatywach poprawy zależy do nowych inwestycji.
- Szybsza nauka generowana od pierwszej iteracji, co oznacza szybszą integrację nowej wiedzy w następnej iteracji.
- Obniżenie energii aktywacji potrzebnej do uzyskania efektu poprawy.
- Szybsza realizacja usprawnień, która wprowadza znaczące różnice w codziennej pracy.
- Mniejsze ryzyko niepowodzenia projektu przed wygenerowaniem widocznych rezultatów.

REZERWACJA 20% CYKLI NA WYMAGANIA NIEFUNKCJONALNE I ZMNIEJSZENIE DŁUGU TECHNICZNEGO

Popularnym problemem wszelkiego rodzaju przedsięwzięć związanych z usprawnieniem jest prawidłowe ustalenie priorytetów. Zwykle te organizacje, które najbardziej potrzebują usprawnień, jednocześnie są tymi, które mają najmniej czasu na poprawę. Dotyczy to zwłaszcza organizacji technicznych, ze względu na dług techniczny.

Organizacje, które zmagają się z długiem finansowym, spłacając same odsetki i nigdy nie zmniejszają kapitału kredytu. W efekcie mogą znaleźć się w sytuacji, gdy dalsza obsługa spłat odsetek będzie niemożliwa. Na podobnej zasadzie organizacje, które nie spłacają dłużu technicznego, mogą być tak bardzo obciążone codziennymi obejściami nierozerwanych problemów, że nie starcza im czasu na realizację żadnej nowej pracy. Innymi słowy, teraz tylko spłacając odsetki od swojego zadłużenia technicznego.

Możemy aktywnie zarządzać dłużem technicznym, jeśli zainwestujemy co najmniej 20% wszystkich cykli operacji i rozwoju na refaktoryzację, zadania automatyzacji oraz realizację wymagań architektury i wymagań niefunkcjonalnych (ang. *non functional requirements* — **NFR**), nazywanych czasem „ilities”, takich jak możliwości utrzymywania, zarządzania, skalowalność, niezawodność, testowalność, możliwości wdrażania i bezpieczeństwo (rysunek 11).



Rysunek 11. Należy poświęcić 20% cykli na zadania, które tworzą wartość dodatnią niewidoczną dla użytkownika (źródło: „Machine Learning and Technical Debt with D. Sculley”, podcast Software Engineering Daily, 17 listopada 2015, <http://softwareengineeringdaily.com/2015/11/17/machine-learning-and-technical-debt-with-d-sculley/>)

Po „doświadczeniach bliskich śmierci” serwisu eBay pod koniec lat 90. Marty Cagan, autorka słynnej książki *Inspired: How To Create Products Customers Love*, poświęconej projektowaniu i zarządzaniu produktami, sformułowała następującą tezę:

Umowa pomiędzy właścicielami produktu a inżynierami powinna mieć następującą postać: zarządzanie produktem zabiera 20% potencjału zespołu i przekazuje go inżynierom, aby spożytkowali go tak, jak im się podoba. Mogą go wykorzystać na przepisywanie, zmianę architektury lub refaktoryzację tych części bazy kodu, które powodują problemy — wszystko, co ich zdaniem jest potrzebne do tego, aby nigdy nie okazało się konieczne stwierdzenie: „Musimy zatrzymać się i przepisać wszystko od nowa [cały kod]”. Jeśli kondycja kodu jest szczególnie zła, może być konieczne poświęcenie 30% zasobów albo jeszcze więcej. Zaczynam się jednak denerwować, gdy spotykam zespoły, które uważają, że mogą poświęcić na to znacznie mniej niż 20%.

Cagan zauważyła, że gdy organizacje nie płacą swojego „20-procentowego podatku”, to ich dług techniczny wzrasta do poziomu, w którym nieuchronnie poświęcają cały cykl na spłatę dłużu technicznego. W pewnym momencie usługi stają się tak kruche, że dostarczanie funkcji zatrzymuje się, a wszyscy inżynierowie pracują nad problemami związanymi z niezawodnością lub nad obejściemi problemów.

Poświęcenie 20% cykli na to, aby działały Dev i Ops mogły utworzyć trwałe środki przeciwdziałające problemom napotykany w codziennej pracy, pozwala zapewnić stan, w którym dług techniczny nie przeszkadza w zdolności szybkiego i bezpiecznego rozwijania i zapewnienia ciągłości działania usług w produkcji. Zmniejszanie dodatkowych napięć związanych z długiem technicznym pozwala także obniżyć poziom wypalenia zawodowego.

STUDIUM PRZYPADKU.

OPERACJA INVERSION W LINKEDIN (2011)

Operacja InVersion przeprowadzona w firmie LinkedIn to interesujące studium przypadku ilustrujące potrzebę spłacania dłużu technicznego w ramach codziennej pracy. Sześć miesięcy po pomyślnym debiucie na giełdzie w 2011 roku firma LinkedIn nadal zmagała się z problemami wdrożeń. Sprawiały one firmie tak wiele kłopotów, że postanowiono uruchomić operację InVersion. Polegała ona na zatrzymaniu rozwijania wszystkich funkcji przez okres dwóch miesięcy w celu przeprowadzenia naprawy środowisk obliczeniowych, wdrożeń i architektury.

Firma LinkedIn została utworzona w 2003 roku, aby pomóc użytkownikom „łączyć się w sieci zawodowej w celu wyszukiwania lepszych miejsc pracy”. Pod koniec pierwszego tygodnia pracy serwis skupiał 2700 użytkowników. Rok później było ich ponad milion i od tamtej pory liczba użytkowników wzrosła wykładniczo. W listopadzie 2015 roku LinkedIn miał ponad 350 milionów członków, którzy generowali dziesiątki tysięcy żądań na sekundę, skutkujących milionami zapytań na sekundę do systemów zaplecza LinkedIn.

Od samego początku firma LinkedIn wykorzystuje przede wszystkim własny system Leo — monolityczną aplikację w Javie, obsługującą wszystkie strony za pośrednictwem serwletów i zarządzanych połączzeń JDBC do różnych baz danych Oracle, działających na zapleczu. Jednak w celu obsługi rosnącego ruchu w początkowych latach działania wydzielono z aplikacji Leo dwie kluczowe usługi: pierwsza obsługuje zapytania grafu połączeń użytkowników całkowicie w pamięci, natomiast druga obsługuje wyszukiwanie użytkowników i bazuje na pierwszej.

Do 2010 roku większość nowych prac rozwojowych dotyczyła nowych usług — niemal sto usług działało poza aplikacją Leo. Problem polegał na tym, że aplikacja Leo była wdrażana raz na dwa tygodnie.

Josh Clemm, starszy menedżer inżynierii w firmie LinkedIn, wyjaśniał, że w 2010 r. firma miała znaczne problemy z aplikacją Leo. Pomimo pionowego skalowania aplikacji Leo przez dodawanie pamięci i procesorów „występowały częste awarie w produkcji, które były trudne do zdiagnozowania i wyeliminowania oraz występowały trudności z publikowaniem nowego kodu... Stało się jasne, że musimy »zabić Leo« i podzielić ją na wiele małych funkcjonalnych i bezstawnowych usług”.

W 2013 r. publicystka Ashlee Vance z Bloomberg napisała: „Gdyby LinkedIn spróbował jednocześnie dodać kilka nowych elementów naraz, witryna przyjęłaby postać jednego wielkiego bałaganu, a w celu rozwiązania problemów inżynierowie byliby zmuszeni do pracy w nocy”. Jesienią 2011 r. praca w nocy nie była już rytuałem przejścia do dalszego etapu wtajemniczenia lub zajęciem grupowym, ponieważ problemy stały się nie do zniesienia. Niektórzy z najlepszych inżynierów LinkedIn, włącznie z Kevinem Scottem, który dołączył do LinkedIn jako wiceprezes ds. inżynierii na trzy miesiące przed pierwszą ofertą publiczną, postanowił całkowicie zatrzymać prace inżynierów nad nowymi funkcjami i przydzielił

całemu działowi zadanie naprawy podstawowej infrastruktury witryny. Przedsięwzięcie nazwano operacją InVersion.

Scott rozpoczął operację InVersion jako metodę „wstrzygnięcia początków manifestu kulturalnego do kultury inżynierskiej swojego zespołu. Nie będzie rozwijania nowych funkcji, dopóki architektura obliczeniowa LinkedIn nie zostanie naprawiona — oto czego potrzebuje biznes i jego zespół”.

Scott opisał jeden minus: „Wchodzisz na giełdę, cały świat na ciebie patrzy, i wtedy mówisz kierownictwu, że nie zamierzasz dostarczyć niczego nowego, bo wszyscy inżynierowie będą pracowali nad tym projektem [InVersion] przez najbliższe dwa miesiące. To było przerażające”.

Vance opisała jednak globalnie pozytywne wyniki operacji InVersion. „Firma LinkedIn stworzyła pakiet oprogramowania i narzędzia ułatwiające tworzenie kodu witryny. Zamiast czekać wiele tygodni, aż nowe funkcje znajdą się na głównej stronie LinkedIn, inżynierowie mogliby opracować nową usługę, wykorzystać szereg automatycznych systemów w celu wykrycia w kodzie wszelkich błędów, dotyczących interakcji usługi z istniejącymi funkcjami, i uruchomić ją na produkcyjnej stronie LinkedIn... [Teraz] inżynierowie LinkedIn realizują poważne aktualizacje witryny trzy razy na dobę”. Dzięki zastosowaniu bezpieczniejszego systemu pracy tworzenie wartości wymagało mniej sesji w nocy i więcej czasu na opracowywanie nowych, innowacyjnych funkcji.

Zgodnie z tym, co napisał Josh Clemm w swoim artykule na temat skalowania LinkedIn, „skalowanie może być obserwowane w wielu wymiarach, w tym na poziomie organizacji... [Operacja InVersion] pozwoliła całej organizacji inżynierskiej skoncentrować się na poprawieniu oprzyrządowania, wdrażania, infrastruktury i wydajności programistów. Sukcesem było skorzystanie ze zwinności inżynierskiej, która była potrzebna do zbudowania nowych skalowalnych produktów, jakimi dysponujemy dzisiaj... [W] 2010 r. mieliśmy już ponad 150 oddzielnych usług. Dzisiaj mamy ich ponad 750”.

Kevin Scott stwierdził: „Zadaniem inżyniera i celem zespołu technicznego jest pomoc w osiągnięciu sukcesu organizacji. Jeśli jesteś liderem zespołu inżynierów, spróbuj przyjąć punkt widzenia dyrektora zarządzającego. Twoim zadaniem jest dowiedzieć się, czego potrzebują twoja firma, biznes, rynek oraz środowisko konkurencyjne. Aby twoja firma odniosła zwycięstwo, zastosuj tę wiedzę w swoim zespole inżynierów”.

Dzięki umożliwieniu firmie LinkedIn spłacania dłużu technicznego przez blisko dekadę, projekt InVersion wprowadził stabilność i bezpieczeństwo, a jednocześnie określił następny etap rozwoju firmy. Wymagało to jednak dwóch miesięcy całkowitego skupienia się na wymaganiach niefunkcjonalnych kosztem wszystkich funkcji, których wprowadzenie obiecano podczas debiutu na giełdzie. Dzięki wyszukiwaniu i rozwiązywaniu problemów w ramach codziennej pracy zarządzamy naszym długiem technicznym, co sprawia, że możemy uniknąć doświadczeń „bliskich śmierci”.

POPRAWA WIDOCZNOŚCI PRACY

Aby wiedzieć, czy zbliżamy się do celu, wszyscy pracownicy firmy powinni mieć możliwość zapoznania się z bieżącym stanem pracy. Istnieje wiele sposobów na to, by bieżący stan był widoczny. Najważniejsze jest jednak to, aby wyświetlane informacje były aktualne oraz aby stale weryfikować to, co mierzymy. W ten sposób możemy zyskać pewność, że robimy postęp w kierunku naszego aktualnego celu.

W kolejnym podrozdziale opisano wzorce, które mogą pomóc w zapewnieniu widoczności oraz wyrównaniu pomiędzy zespołami i funkcjami.

UŻYWAJ NARZĘDZI DO WZMOCNIENIA POŻĄDANYCH ZACHOWAŃ

Jak powiedział Christopher Little, menedżer oprogramowania oraz jeden z pierwszych kronikarzy DevOps, „antropolodzy opisują narzędzia jako artefakty kultury. Wszelkie dyskusje na temat kultury po wynalezieniu ognia muszą także dotyczyć narzędzi”. Na podobnej zasadzie w strumieniu wartości DevOps używamy narzędzi do wzmacniania kultury i przyspieszenia pożądanych zmian w zachowaniach.

Jednym z celów jest to, aby dzięki narzędziom działały Dev i Ops nie tylko miały wspólne cele, ale również wspólny rejestr zadań do wykonania, najlepiej przechowywany we wspólnym systemie pracy i zarządzany z wykorzystaniem wspólnej terminologii. Dzięki temu można w globalny sposób ustalić priorytety pracy.

W ten sposób działały Dev i Ops mogą stworzyć wspólną kolejkę pracy. To pozwala uniknąć sytuacji, w której każdy z silosów używa innej (np. dział Dev używa systemu JIRA, podczas gdy w dziale Ops wykorzystywany jest ServiceNow). Znaczącą korzyścią z prezentowania wydarzeń produkcji w tych samych systemach pracy, w których są prezentowane prace rozwojowe — zwłaszcza w przypadku korzystania z tablicy kanban — jest natychmiastowa widoczność przypadków, gdy bieżące zdarzenia powinny wstrzymać inne prace.

Inną zaletą wykorzystania wspólnych narzędzi przez działały Dev i Ops jest jednolity rejestr zadań do wykonania, w którym wszyscy określają priorytet dla projektów poprawy z perspektywy globalnej, wybierają prace o najwyższej wartości dla organizacji oraz które najbardziej ograniczają dług techniczny. W miarę identyfikowania dlułu technicznego, jeśli nie potrafimy „spłacić go” natychmiast, dodajemy zadania z nim związane do spriorytetyzowanego rejestru zaległości. W przypadku problemów, które pozostają nierozerwane, możemy użyć naszego „20-procentowego zapasu czasu na wymagania niefunkcjonalne”, aby nadrobić te elementy spośród zaległości, które mają najwyższy priorytet.

Do innych technologii, które wzmacniają wspólne cele, należą narzędzia typu *chat room*, takie jak kanały IRC, HipChat, Campfire, Slack, Flowdock i OpenFire. Narzędzia te umożliwiają szybkie współdzielenie informacji (w przeciwnieństwie do wypełniania formularzy, które są przetwarzane za pośrednictwem wstępnie zdefiniowanych przepływów pracy), zapraszanie innych osób według potrzeb oraz automatyczne zapisywanie logów historii, które mogą być analizowane podczas sesji post-mortem.

Cechą mechanizmu, który pozwala członkom zespołu szybko pomóc innym jego członkom lub nawet ludziom spoza niego, jest niesamowity dynamizm — czas wymagany do uzyskania informacji bądź wykonania potrzebnej pracy można skrócić z dni do minut. Ponadto dzięki temu, że wszystko jest zapisywane, unikamy konieczności proszania o pomoc w przeszłości — wystarczy poszukać w historii.

Środowisko szybkiej komunikacji dostępne za pośrednictwem kanałów chat może także być utrudnieniem. Jak powiedział Ryan Martens, założyciel i CTO RallySoftware, „jest całkowicie dopuszczalne, aby w sytuacji, gdy w konwersacji chat ktoś nie otrzyma odpowiedzi w ciągu kilku minut, spróbował zadać pytanie ponownie, do czasu, aż otrzyma to, czego potrzebuje”.

Oczekiwanie natychmiastowej odpowiedzi może oczywiście prowadzić do niepożądanych efektów. Ciągłe przerywanie i zadawanie pytań może utrudniać niektórym osobom wykonanie potrzebnej pracy. W rezultacie zespoły mogą zdecydować, że określone typy żądań powinny być kierowane za pośrednictwem bardziej strukturalnych, asynchronicznych narzędzi.

PODSUMOWANIE

W tym rozdziale pokazaliśmy, jak zidentyfikować wszystkie zespoły wspierające nasz strumień wartości, i zaprezentowaliśmy na mapie strumienia wartości, jakie prace są wymagane w celu dostarczenia wartości do klienta. Mapa strumienia wartości tworzy podstawę zrozumienia bieżącego stanu, włącznie z określeniem parametrów czasu realizacji i procentu wykonania (%C/A) dla obszarów problemów, a także dostarcza danych wejściowych w celu ustalenia stanu w przyszłości.

Dzięki temu dedykowane zespoły transformacji mogą przeprowadzać szybkie迭代 i eksperymentować, aby poprawić wydajność. Możemy również zadbać o przydzielanie odpowiedniej ilości czasu na usprawnienia, rozwiązywanie znanych problemów i usuwanie wad architektonicznych, a także obsługę wymagań niefunkcjonalnych. Studia przypadków firm Nordstrom i LinkedIn pokazały, jak wielką poprawę czasu realizacji i wydajności można uzyskać, gdy znajdujemy problemy w strumieniu wartości i spłacamy dług techniczny.

Projektowanie organizacji i jej architektury z uwzględnieniem prawa Conwaya

W poprzednich rozdziałach zidentyfikowaliśmy strumień wartości do rozpoczęcia transformacji DevOps oraz ustaliliśmy wspólne cele i praktyki umożliwiające dedykowanemu zespołowi transformacji poprawę sposobu dostarczania wartości dla klienta.

W tym rozdziale zaczniemy myśleć o tym, jak najlepiej zorganizować zespół, aby najłatwiej osiągnąć wyznaczone cele strumienia wartości. Ostatecznie sposób organizacji zespołów wpływa na to, jak wykonujemy pracę. W 1968 roku dr Melvin Conway wykonał słynny eksperyment z organizacją badawczą zatrudniającą osiem osób, którym zlecono opracowanie kompilatorów COBOL i ALGOL. Zauważył, że „po dokonaniu pierwszych oszacowań czasu i trudności zadania pięć osób przypisano do pracy nad COBOL-em, natomiast trzy do pracy nad ALGOL-em. Okazało się, że wynikowy kompilator COBOL-a był pięciofazowy, natomiast kompilator ALGOL-a używa trzech faz”.

Te obserwacje doprowadziły go do sformułowania tezy znanej dziś jako prawo Conwaya, które mówi, że „organizacje projektujące systemy... tworzą wzorce, które są kopiami struktur komunikacyjnych tych organizacji... Im większa organizacja, tym ma mniejszą elastyczność, a opisywane zjawisko jest wyraźniejsze”. Eric S. Raymond, autor książki *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, w opracowanym przez siebie *Jargon File* zaproponował uproszczoną (i obecnie bardziej znaną) wersję prawa Conwaya: „Organizacja

oprogramowania jest zgodna z organizacją zespołu programistów. Jeśli nad kompi-latorem pracują cztery zespoły, to powstanie czteroprzebiegowy kompilator”.

Innymi słowy, sposób, w jaki zorganizujemy nasze zespoły, ma potężny wpływ na oprogramowanie, które wytwarzamy, a także na tworzone struktury architektonicznej i rezultaty produkcji. Aby uzyskać szybki przepływ pracy od działu Dev do Ops, zapewnić wysoką jakość i dobre wyniki dla klientów, trzeba zorganizować zespoły i pracę tak, aby prawo Conwya działało na naszą korzyść. Przy złej organizacji prawo Conwya uniemożliwia zespołom bezpieczną i niezależną pracę. Zamiast tego zespoły są ze sobą ściśle powiązane, wzajemnie na siebie czekają, a nawet niewielkie zmiany mogą spowodować globalne, katastrofalne konsekwencje.

To, jak prawo Conwya może utrudniać lub ułatwiać osiąganie celów, można zobaczyć na przykładzie narzędzia Sprouter opracowanego w firmie Etsy. Transforma-cje DevOps w Etsy rozpoczęły się w 2009 roku. Jest to jedna z najbardziej podziwianych organizacji DevOps. Jej przychody w 2014 roku wyniosły prawie 200 milionów dolarów, a w 2015 roku firma udanie zadebiutowała na giełdzie.

Pierwotnie opracowane w 2007 roku narzędzie Sprouter łączyło osoby, procesy i technologie w sposób, który doprowadził do wielu niepożądanych efektów. Sprouter, skrót od *stored procedure router* (dosł. „router procedur składowanych”), został zapro-jektowany tak, aby ułatwić pracę zespołom programistów i inżynierów baz danych. Jak podczas prezentacji na konferencji Surge w 2011 roku powiedział Ross Snyder, starszy inżynier w firmie Etsy: „Sprouter został zaprojektowany po to, aby umożliwić zespołom deweloperów pisanie kodu aplikacji w PHP, administratorom baz danych pisanie instrukcji SQL w Postgresie. Dzięki Sprouterowi ci inżynierowie mieli spotkać się gdzieś pośrodku”.

Sprouter rezydował pomiędzy aplikacją frontend w PHP a bazą danych Postgres. Zapewniał centralizację dostępu do bazy danych i ukrywał implementację bazy przed warstwą aplikacji. Problem polegał na tym, że dodanie jakichkolwiek zmian do logiki biznesowej powodowało znaczne tarcia pomiędzy zespołami programistów a zespołami baz danych. Jak zaobserwował Snyder: „Prawie dla każdej nowej funkcjonalności witryny Sprouter wymagał od administratorów DBA napisania nowej procedury składowanej. W rezultacie za każdym razem, gdy deweloperzy chcieli dodać nowe funkcje, musieli o coś prosić administratorów DBA, co często wymagało od nich przedzierania się przez mnóstwo biurokracji”. Innymi słowy, deweloperzy tworzący nowe funkcje zależeli od zespołu DBA. Żądanie potrzebnych zmian wymagało określe-nia priorytetów, komunikacji i koordynacji. W efekcie praca była blokowana w kolej-kach, marnotrawiono czas na spotkaniach, czasy realizacji wydłużały się itd. Wszystko dlatego, że Sprouter stworzył ściśle powiązanie pomiędzy zespołami deweloperów a zespołami baz danych, uniemożliwiając programistom niezależne pisanie, testowanie i wdrażanie swojego kodu do produkcji.

Procedury składowane w bazach danych także były ściśle sprzężone ze Sprouterem — każda zmiana w procedurze składowanej wymagała zmian w Sprouterze. W efekcie Sprouter w coraz większym stopniu stawał się pojedynczym punktem awarii. Snyder wyjaśnił, że wszystko było tak ściśle ze sobą powiązane i wymagało tak wysokiego poziomu synchronizacji, że prawie każde wdrożenie powodowało miniawarię.

Oba problemy związane ze Sprouterem i ich ostateczne rozwiążanie można wyjaśnić za pomocą prawa Conwaya. W firmie Etsy początkowo były dwa zespoły: programistów i administratorów baz danych. Każdy z tych zespołów był odpowiedzialny za osobną warstwę usługi — odpowiednio warstwę logiki aplikacji oraz procedur składowanych. Tak jak przewiduje prawo Conwaya, dwa zespoły robocze pracowały nad dwiema warstwami. Sprouter miał ułatwić życie obu zespołom, ale nie działał zgodnie z oczekiwaniemi — gdy zmieniły się reguły biznesowe, to zamiast koniecznych zmian tylko w dwóch warstwach teraz trzeba było wprowadzać zmiany w trzech (aplikacji, procedurach składowanych oraz w Sprouterze). Wynikające stąd wyzwania koordynacji i priorytetyzacji pracy w trzech zespołach znacznie wydłużały czasy realizacji i spowodowały problemy z niezawodnością.

Na wiosnę 2009 roku w ramach przedsięwzięcia określonego przez Snydera jako „wielka transformacja kulturowa Etsy” do firmy dołączył Chad Dickerson jako nowy dyrektor techniczny. Dickerson zainicjował wiele przedsięwzięć, w tym ogromne inwestycje w stabilność witryny. Zlecił deweloperom realizację wdrożeń własnego kodu do produkcji, jak również rozpoczął dwuletnią podróż zmierzającą do wyeliminowania Sproutera.

Aby to zrobić, zespół postanowił przenieść całą logikę biznesową z warstwy bazy danych do warstwy aplikacji, co wyeliminowało potrzebę korzystania ze Sproutera. Stworzono mały zespół, który napisał warstwę PHP Object Relational Mapping^{*}, co umożliwiło programistom warstwy frontend korzystanie z wywołań bezpośrednio do bazy danych i pozwoliło zmniejszyć liczbę zespołów potrzebnych do modyfikacji logiki biznesowej z trzech do jednego.

Jak opisywał Snyder: „Zaczęliśmy używać ORM dla wszelkich nowych obszarów witryny i z czasem udało się nam dokonać migracji niewielkich części witryny ze Sproutera do ORM. Rezygnacja ze Sproutera dla całej witryny zajęła nam dwa lata. Przez cały czas, gdy Sprouter był wykorzystywany w produkcji, wszyscy na niego narzekali”.

Wyeliminowanie Sproutera pozwoliło także rozwiązać problemy związane z koniecznością koordynacji zmian w logice biznesowej przez wiele zespołów, zmniejszyło liczbę przełączeń pomiędzy zespołami i znacznie zwiększyło szybkość wdrożeń do

* ORM między innymi tworzy abstrakcję bazy danych, co pozwala programistom na wykonywanie operacji manipulowania danymi w taki sposób, jakby były one jedynie dodatkowym obiektem w języku programowania. Do popularnych mechanizmów ORM można zaliczyć Hibernate dla Javy, SQLAlchemy dla Pythona i ActiveRecord dla Ruby on Rails.

produkci, przez co poprawiła się stabilność witryny. Ponadto dzięki temu, że małe zespoły mogły samodzielnie rozwijać i wdrażać swój kod bez konieczności wprowadzania zmian w innych obszarach przez inne zespoły, wzrosła wydajność deweloperów.

Sprouter został ostatecznie usunięty z produkcji i repozytoriów kontroli wersji firmy Etsy na początku 2001 roku. Snyder powiedział: „Wow, wszyscy odetchnęli z ulgą”*.

Zgodnie z doświadczeniami Snydera i firmy Etsy sposób projektowania naszych organizacji decyduje o sposobie wykonywania pracy, a w związku z tym o osiąganych wynikach. W dalszej części niniejszego rozdziału pokażemy, jak prawo Conwya może negatywnie wpływać na wydajność strumienia wartości i co ważniejsze, w jaki sposób zorganizować zespoły, aby prawo Conwya działało na naszą korzyść.

ARCHETYPY ORGANIZACYJNE

W dziedzinie nauki o podejmowaniu decyzji wyróżnia się trzy podstawowe typy struktur organizacyjnych, które decydują o sposobie zaprojektowania strumieni wartości DevOps z uwzględnieniem prawa Conwya: **funkcjonalne, macierzowe i rynkowe**. Zostały one zdefiniowane przez dr. Roberta Fernandeza w następujący sposób:

- Organizacje **funkcjonalne** optymalizują swoje działania pod kątem wiedzy, podziału pracy lub redukcji kosztów. Organizacje te dążą do centralizacji wiedzy, co pomaga w rozwoju kariery i umiejętności. Często występują w nich wysokie hierarchiczne struktury organizacyjne. Sposób funkcjonalny był przeważającą metodą organizacji dla działu operacji (tzn. administratorzy serwerów, administratorzy sieci, baz danych itd. są podzieleni na osobne grupy).
- Organizacje **macierzowe** próbują łączyć metody organizacji funkcjonalną i rynkową. Jednak jak zaobserwowało wiele osób, które pracują w organizacjach macierzowych lub nimi zarządzają, w tych organizacjach często występują skomplikowane struktury organizacyjne. Przykładowo indywidualny pracownik jest podległy dwóm lub większej liczbie menedżerów. W związku z tym organizacjom macierzowym często nie udaje się osiągnąć żadnego z celów typowych dla organizacji funkcjonalnej oraz rynkowej.
- Organizacje **rynkowe** optymalizują swoje działania pod kątem szybkiego reagowania na potrzeby klientów. Organizacje te zwykle są płaskie, składają się z wielu współzależnych funkcjonalnie działów (np. marketing, inżynieria itp.), co często prowadzi do potencjalnych redundancji w całej organizacji. W ten sposób działa wiele czołowych organizacji, które zastosowały DevOps — w skrajnych

* Sprouter był jedną z wielu technologii wykorzystywanych w rozwoju i produkcji. Technologię tę firma Etsy wyeliminowała w ramach przeprowadzanych transformacji.

przykładach, takich jak Amazon i Netflix, zespół każdej z usług jest jednocześnie odpowiedzialny za dostarczanie funkcjonalności oraz wsparcie dla usługi^{*}.

Pamiętając o istnieniu tych trzech kategorii organizacji, przyjrzymy się, dlaczego orientacja zbyt funkcjonalna, zwłaszcza w dziale operacji, może — zgodnie z prognozą prawa Conwya — spowodować niepożądane efekty w strumieniu wartości technologii.

CZĘSTE PROBLEMY SPOWODOWANE ZBYT FUNKCJONALNĄ ORIENTACJĄ („OPTYMALIZACJA KOSZTÓW”)

W tradycyjnych organizacjach Operacji IT często stosuje się orientację funkcjonalną w celu zorganizowania zespołów zgodnie z ich specjalnością. Administratorzy baz danych tworzą jeden zespół, administratorzy sieci inny, administratorzy serwerów jeszcze inny itd. Jedną z najbardziej widocznych konsekwencji takiej organizacji są długie okresy realizacji. Dotyczy to zwłaszcza złożonych działań, takich jak skomplikowane wdrożenia, które wymagają generowania zleceń do wielu grup i koordynowania przekazywania pracy. W rezultacie w każdym kroku prace muszą czekać w długich kolejkach.

Na domiar złego osoba wykonująca pracę często nie wie lub nie rozumie, w jaki sposób wykonywana przez nią praca przyczynia się do osiągnięcia jakiegokolwiek celu w strumieniu wartości (np. „konfiguruje serwery, ponieważ otrzymałem takie zadanie”). W efekcie pracownicy pracują w próżni kreatywności i motywacji.

Problem pogarsza się, gdy każdy obszar funkcjonalny działu operacyjnego musi obsługiwać wiele strumieni wartości (np. wiele zespołów programistycznych) wzajemnie rywalizujących ze sobą o ograniczone cykle robocze. Aby zespoły programistów mogły wykonywać swoje zadania w sposób terminowy, często zachodzi potrzeba zgłaszania problemów kierownikowi lub dyrektorowi i ostatecznie komuś (zwykle menedżerowi wysokiego szczebla), kto w końcu nada priorytety pracy zgodnie z globalnymi celami zamiast celami pojedynczych funkcji. Decyzja ta musi być następnie przekazana kaskadowo w dół, do każdego z obszarów funkcjonalnych, aby zmienić lokalne priorytety, a to z kolei spowalnia pracę innych zespołów. Pomimo że każdy zespół stara się przyspieszyć swoją pracę, to ostatecznie postępy w każdym projekcie są osiągane tak samo wolno.

Oprócz długich kolejek i długich cykli realizacji sytuacja ta powoduje niepotrzebne przekazywanie pracy, dużą liczbę poprawek, problemy jakości, wąskie gardła i opóźnienia.

* Jednak jak wyjaśnimy później, również znane organizacje, takie jak Etsy i GitHub, mają organizację funkcjonalną.

Istniejący impas utrudnia osiągnięcie ważnych celów organizacji, które często są znacznie ważniejsze niż dążenie do obniżenia kosztów*.

Podobnie organizację funkcjonalną można rozpoznać w dążeniu do skoncentrowania funkcji walidacji i bezpieczeństwa informacji, co mogło się sprawdzać (lub przy najmniej dawać zadowalające efekty) w przypadku rzadszych wydań oprogramowania. Jednak w miarę wzrostu liczby zespołów programistycznych oraz zwiększenia częstotliwości instalacji i wydań oprogramowania większość organizacji funkcjonalnych będzie miała trudności z osiągnięciem zadowalających rezultatów, zwłaszcza gdy ich prace są wykonywane ręcznie. W dalszej części rozdziału opowiem o sposobie działania firm o organizacji rynkowej.

ZESPOŁY RYNKOWE („OPTYMALIZACJA SZYBKOŚCI”)

Ogólnie rzecz biorąc, aby osiągnąć pozytywne wyniki zastosowania DevOps, trzeba zminimalizować skutki orientacji funkcjonalnej („optymalizacji kosztów”) i wprowadzić orientację rynkową („optymalizację szybkości”). Taka optymalizacja przyczynia się do powstania wielu małych zespołów pracujących bezpiecznie, samodzielnie i szybko dostarczających wartość do klienta.

W sytuacji ekstremalnej zespoły zorientowane na potrzeby rynku są odpowiedzialne nie tylko za rozwój nowych funkcji, ale także za testowanie, zabezpieczanie, wdrażanie i wspieranie usługi w produkcji — od powstania koncepcji do wycofania produktu z rynku. Zespoły te są zaprojektowane tak, aby były wielofunkcyjne i niezależne. Powinny być zdolne do projektowania i przeprowadzania eksperymentów z użytkownikami, budowania i dostarczania nowych funkcji, wdrażania i uruchamiania usług w produkcji oraz naprawiania wszelkich wad bez zależności „ręcznych” od innych zespołów. Taki sposób organizacji zespołów przyczynia się do osiągania szybszych postępów. Model ten został przyjęty przez firmy Amazon i Netflix i jest reklamowany przez Amazon jako jeden z głównych czynników wpływających na zdolność firmy do osiągania szybkich postępów, nawet podczas wzrostu skali.

Aby osiągnąć zorientowanie na rynek, nie przeprowadza się dużych reorganizacji góra-dół, którym często towarzyszą zakłócenia, obawy i paraliż. Zamiast tego dąży się do wprowadzania inżynierów i umiejętności funkcjonalnych (np. Ops, QA, Infosec) do każdego zespołu usługowego lub dostarcza się ich możliwości do zespołów poprzez

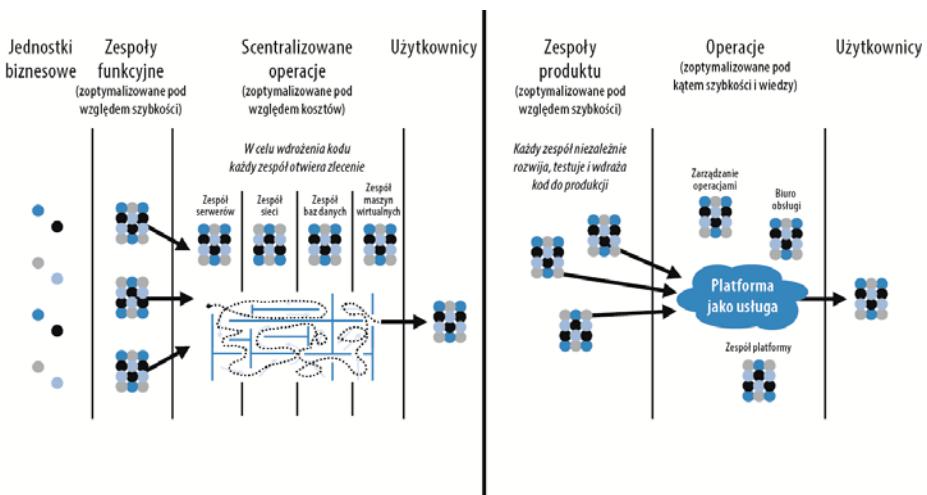
* Adrian Cockroft zauważył: „Firmy, które obecnie nawiązują pięcioletnie kontrakty outsourcingowe, działają tak, jakby zamrażały się podczas jednego z najbardziej uciążliwych okresów w technice”. Innymi słowy, outsourcing IT jest taktyką używaną do kontroli kosztów dzięki wymuszany przez kontrakt ograniczeniom — sztywnym cenom, które pozwalają na redukcję kosztów w skali roku. Często jednak skutkiem takich kontraktów jest brak możliwości reagowania na zmieniające się potrzeby biznesowe i powstające nowe technologie.

zautomatyzowane samoobsługowe platformy, które udostępniają środowiska podobne do produkcyjnych, inicjują automatyczne testy lub wykonują instalacje.

Dzięki temu każdy zespół usługowy może samodzielnie dostarczyć wartość do klienta bez konieczności generowania zleceń dla innych grup, takich jak działy operacji IT, validacji (QA) lub bezpieczeństwa informacji (Infosec)*.

ZAPEWNIENIE SKUTECZNOŚCI ORIENTACJI FUNKCJONALNEJ

W chwilę po zarekomendowaniu rynkowej orientacji zespołów warto zaznaczyć, że możliwe jest stworzenie skutecznych, wydajnych organizacji z funkcjonalną orientacją zespołów. Zespoły interdyscyplinarne i zorientowane na potrzeby rynku są jednym ze sposobów osiągnięcia szybkiego przepływu i niezawodności, ale nie jest to jedyny sposób. Pożądane rezultaty DevOps można również osiągnąć za pomocą orientacji funkcjonalnej, pod warunkiem że wszystkie osoby w strumieniu wartości bez względu na ich miejsce i rolę w organizacji postrzegają efekty działające na korzyść klienta i organizację jako wspólny cel. Orientację funkcjonalną i rynkową porównano na rysunku 12.



Rysunek 12. Orientacja funkcjonalna i rynkowa. Po lewej: orientacja funkcjonalna — wszystkie prace przepływają przez centralizowany dział operacji IT; po prawej: orientacja rynkowa — wszystkie zespoły produktu mogą wdrażać swoje luźno powiązane samoobsługowe komponenty do produkcji
(źródło: Humble, Molesky i O'Reilly, „Lean Enterprise”, wydanie Kindle, 4523 i 4592)

* W pozostałej części tej książki będziemy używać terminu „zespoł usługi” zamiennie z terminami „zespoł funkcji”, „zespoł produktu”, „zespoł rozwoju” oraz „zespoł dostaw”. Chcemy w ten sposób określić zespół, który przede wszystkim zajmuje się rozwijaniem, testowaniem i zabezpieczaniem kodu, by dostarczyć wartość do klienta.

Przykładowo możliwe jest osiągnięcie wysokiej wydajności z wykorzystaniem scentralizowanego i zorientowanego funkcjonalnie zespołu operacji, pod warunkiem że zespoły usług niezawodnie i szybko (najlepiej na żądanie) otrzymują to, czego potrzebują od działu operacji. Wiele z najbardziej podziwianych organizacji DevOps, w tym Etsy, Google i GitHub, zachowuje funkcjonalną orientację działu operacji.

Wspólna dla tych organizacji jest kultura wysokiego zaufania, która umożliwia wszystkim działom skutecną współpracę. Wszystkim działaniom są w przezroczysty sposób nadawane priorytety, a w systemie istnieje wystarczająco dużo swobody, aby zadania o wysokim priorytecie mogły być wykonane szybko. Po części jest to możliwe dzięki zastosowaniu zautomatyzowanych samoobsługowych platform, które dostarczają jakości dla budowanych przez wszystkich produktów.

Podczas ruchu Lean w firmach produkcyjnych w latach 80. wielu badaczy było zaskoczonych funkcjonalną orientacją Toyoty, która kłoci się z najlepszą praktyką interdyscyplinarnych i zorientowanych na rynek zespołów. Zdziwienie było tak wielkie, że zjawisko to nazwano „drugim paradoksem Toyoty”.

Jak napisał Mike Rother w *Toyota Kata*: „Choć wydaje się to kuszące, to nie da się zreorganizować przyjętego sposobu ciągłego doskonalenia się i adaptacji. Decyduje nie forma organizacji, ale sposób, w jaki jej członkowie działają i reagują. Korzenie sukcesu Toyoty leżą nie w jej strukturach organizacyjnych, lecz w możliwościach rozwoju i nawykach pracowników. Dla wielu osób zaskakujące jest to, że Toyota jest w dużej mierze zorganizowana w tradycyjnym stylu funkcjonalnych działów”. W kolejnych pododdziałach skoncentrujemy się na rozwoju nawyków i możliwości pracowników.

TESTOWANIE, OPERACJE I ZABEZPIECZENIA JAKO CODZIENNE ZADANIE KAŻDEGO Z NAS

W wysokowydajnych organizacjach wszyscy członkowie zespołu mają wspólny cel — jakość, dostępność i bezpieczeństwo nie należą do obowiązków poszczególnych działów, ale codziennie są częścią zadań każdego pracownika.

Oznacza to, że najpilniejszym problemem dnia może być praca nad wdrożeniem funkcji klienta lub nad wyeliminowaniem incydentu produkcji pierwszego stopnia. Alternatywnie dzień może wymagać przeglądu zmian wprowadzonych przez kolegę inżyniera, zainstalowania poprawek zabezpieczeń na serwerach produkcyjnych lub wprowadzenia usprawnień zwiększających produktywność współpracowników.

Podczas rozmów nad wspólnymi celami pomiędzy działami Dev i Ops Jody Mulkey, dyrektor techniczny w firmie Ticketmaster, powiedział: „Do opisywania działów Dev i Ops przez prawie 25 lat używałem metafory futbolu amerykańskiego. Wiecie, Ops to obrona, która stara się nie dopuścić do zdobycia punktu przez przeciwnika, a Dev to formacja ofensywna, która stara się zdobyć gole. I pewnego dnia zdałem

sobie sprawę, jak błędna była ta metafora, ponieważ oni nigdy nie grają na tym samym boisku i w tym samym czasie. W rzeczywistości oni nie są w tym samym zespole!”.

Jody Mulkey kontynuuje: „Teraz używam analogii, w której Ops są ofensywnymi liniowymi, a Dev to gracze na pozycjach specjalistycznych (np. rozgrywający i skrzydłowi), których zadaniem jest przemieszczanie piłki na boisku — zadaniem Ops jest zadbanie o to, aby Dev mieli wystarczająco dużo czasu do poprawnego wykonania swoich zagrań”.

Uderzającym przykładem tego, jak wspólne problemy mogą wzmacnić wspólne cele, jest okres burzliwego rozwoju Facebooka w 2009 roku. W firmie występowały znaczące problemy dotyczące wdrażania kodu — choć nie wszystkie miały ujemny wpływ na klienta, to często występowała konieczność „gaszenia pożarów” i długiego siedzenia w pracy. Pedro Canahuati, dyrektor inżynierii produkcji Facebooka, opisał spotkanie, w którym uczestniczyło wielu inżynierów Ops. Gdy ktoś poprosił, aby wszystkie osoby, które nie pracują nad incydentem, zamknęły swoje laptopy, nikt tego nie zrobił.

Jednym z najważniejszych przekształceń, których dokonano w celu poprawy wyników wdrożeń, było wprowadzenie wśród wszystkich inżynierów, menedżerów, inżynierów i architektów dyżurów przy obsłudze usług, które budowano. Dzięki temu wszyscy, którzy pracowali nad usługą, otrzymywali informacje zwrotne związane z podejmowanymi przez nich, w górze strumienia, decyzjami dotyczącymi kodowania i architektury. Miało to niezwykle pozytywny wpływ na wyniki w dole strumienia wartości.

POZWÓŁ KAŻDEMU CZŁONKOWI ZESPOŁU BYĆ GENERALISTĄ

W skrajnych przypadkach organizacji funkcjonalnej działu Ops mamy działa specjalistów, na przykład administratorzy sieci, administratorzy baz danych itd. Nadmierna specjalizacja działań powoduje powstawanie **silosów**. Dr Spear napisał, że jest to stan, w którym „praca poszczególnych działów przypomina działanie suwerennych państw”. Wszelkie złożone działania operacyjne wymagają wtedy wielokrotnego przekazywania pracy i kolejek pomiędzy różnymi obszarami infrastruktury, co prowadzi do dłuższych czasów realizacji (na przykład dlatego, że każda zmiana w sieci musi być wykonana przez kogoś z działu obsługi sieci).

Ponieważ wykorzystujemy coraz więcej technologii, musimy mieć inżynierów, którzy mają wiedzę specjalistyczną w tych obszarach technologii, których potrzebujemy. Nie chcemy jednak tworzyć specjalistów, którzy są „zamrożeni w czasie”, rozmierają oraz mogą dostarczać wartość tylko do tego jednego obszaru strumienia wartości.

Jednym ze środków zaradczych dla tego stanu jest umożliwienie bycia generalistą każdemu członkowi zespołu i zachęcanie do przyjęcia takiej postawy. Robimy to poprzez zapewnienie inżynierom możliwości nauczenia się wszystkich umiejętności

niezbędnych do tworzenia i uruchamiania systemów, za które są odpowiedzialni, oraz poprzez regularną rotację osób występujących w różnych rolach. Do opisania generalistów, którzy są zaznajomieni — przynajmniej na ogólnym poziomie — z całym stosem aplikacji (np. kod aplikacji, baza danych, systemy operacyjne, sieć, chmura), obecnie powszechnie używa się terminu **inżynier pełnego stosu** — ang. *full stack engineer*. Porównanie specjalistów z generalistami oraz personelem „w kształcie litery E” zaprezentowano w tabeli 2.

Tabela 2. Specjaliści kontra generaliści kontra personel „w kształcie litery E” (doświadczenie, wiedza, eksploracja i wykonanie)

„W kształcie litery I” (specjaliści)	„W kształcie litery T” (generaliści)	„W kształcie litery E”
Głęboka wiedza z jednej dziedziny	Głęboka wiedza z jednej dziedziny	Głęboka wiedza z kilku dziedzin
Niewielkie umiejętności lub doświadczenie w innych dziedzinach	Szerokie umiejętności w wielu dziedzinach	Doświadczenie w wielu dziedzinach
		Potwierdzone zdolności do praktycznego wykonywania zadań
		Zawsze nowatorscy
Szybko tworzą wąskie gardła	Potrafią usuwać wąskie gardła	Prawie nieograniczony potencjał
Niewrażliwi na marnotrawstwo i wywierający wpływ na pracowników w dole strumienia	Wrażliwi na marnotrawstwo i wywierający wpływ na pracowników w dole strumienia	
Przeszkadzają w elastyczności planowania oraz absorpcji zmian	Pomagają w uelastycznieniu i absorpcji zmian	

(Źródło: Scott Prugh, „Continuous Delivery”, ScaledAgileFramework.com, 14 lutego 2013, <http://scaledagileframework.com/continuous-delivery/>)

Scott Prugh opisał transformację w firmie CSG International, w której większość zasobów i obowiązków wymaganych do stworzenia i uruchomienia produktu (w tym analizę, architekturę, rozwój, testowanie i operacje) przekazano jednemu zespołowi. „Dzięki interdyscyplinarnemu szkoleniu i rozwojowi umiejętności inżynierskich generaliści potrafią wykonać o rząd wielkości więcej pracy w porównaniu ze swoimi odpowiednikami — specjalistami. Dodatkowo dzięki usunięciu kolejek i oczekiwania poprawił się ogólny przepływ pracy”. Takie podejście jest sprzeczne z tradycyjnymi praktykami zatrudniania, ale jak wyjaśnia Prugh, warto je stosować. „Tradycyjni menedżerowie często sprzeciwiają się zatrudnianiu inżynierów z umiejętnościami ogólnymi, argumentując, że są oni drożsi, i że »na miejsce każdego inżyniera operacji z wieloma umiejętnościami można zatrudnić dwóch administratorów serwerów«”. Jednak korzyści biznesowe wynikające z szybszego przepływu są ogromne. Ponadto, jak zauważył Prugh, „inwestycja w szkolenie interdyscyplinarne jest uzasadniona ze względu na rozwój kariery pracowników oraz sprawia, że praca przynosi więcej przyjemności”.

Gdy cenimy ludzi tylko za ich istniejące umiejętności lub wydajność w bieżącej roli, a nie za ich zdolność do nabywania i stosowania nowych umiejętności, to wzmacniamy (często przypadkowo) to, co dr Carol Dweck opisuje jako **podejście sztywne** (ang. *fixed mindset*), zgodnie z którym ludzie postrzegają swoją inteligencję i zdolności jako statyczne „dary”, które nie mogą być zmieniane w sensowny sposób.

Zamiast tego warto zachęcić do uczenia się, pomóc ludziom przezwyciężyć niechęć do nauki, pomóc im uwierzyć w to, że mają odpowiednie umiejętności i zdefiniowaną mapę drogową kariery itd. W ten sposób pomagamy wspierać inżynierów tzw. **podejście rozwojowe** (ang. *growth mindset*) — ostatecznie organizacje edukacyjne potrzebują ludzi, którzy chcą się uczyć. Zachęcając wszystkich do nauki oraz dostarczając szkoleń i wsparcia, tworzymy najbardziej zrównoważony i mniej kosztowny sposób tworzenia wielkości naszych zespołów — poprzez inwestycję w rozwój ludzi, których już mamy.

Jak opisuje Jason Cox, dyrektor inżynierii systemów w firmie Disney: „Musielimy zmienić praktyki zatrudniania w dziale operacji. Zaczeliśmy szukać osób, które miały »ciekawość, odwagę i szczerość«. Takich, które nie tylko były w stanie zostać generalistami, ale także renegatami... Chcemy promować pozytywną destrukcję, tak aby nasz biznes nie utknął w martwym punkcie i mógł się rozwijać zgodnie z wymogami przyszłości”. Jak przekonamy się w następnym podrozdziale, na osiągane wyniki wpływa także sposób finansowania zespołów.

FINANSOWANIE NIE PROJEKTÓW, LECZ PRODUKTÓW I USŁUG

Inną drogą umożliwiającą osiągnięcie wysokiej wydajności jest stworzenie stabilnych zespołów usług ze stałym finansowaniem pozwalającym realizować własną strategię i mapę drogową inicjatyw. Zespoły te mają dedykowanych inżynierów, niezbędnych do wypełniania konkretnych zobowiązań podejmowanych na rzecz klientów wewnętrznych i zewnętrznych. Do zadań tych należy tworzenie nowych cech funkcjonalnych, implementacja historyjek użytkownika i innych zadań.

Spróbujmy zestawić to z bardziej tradycyjnym modelem, w którym zespoły programistów (Dev) i testerów są przypisywani do „projektu”, a następnie, gdy skończą się fundusze, przydzielani do innego. Prowadzi to do różnego rodzaju niepożądanych efektów, w tym braku możliwości dostrzegania długoterminowych konsekwencji decyzji podejmowanych przez deweloperów (rodzaj sprzężenia zwrotnego) oraz modelu finansowania, w którym doceniane są tylko najwcześniejsze etapy cyklu życia oprogramowania — które na domiar złego są jednocześnie najmniej kosztowną fazą w cyklu życia udanych produktów lub usług.*

* Jak powiedział John Lauderbach, obecnie wiceprezes ds. technicznych w firmie Roche Bros Supermarkets: „Každa nowa aplikacja jest jak młody szczeniak. Nie ma znaczenia cena, którą za niego zapłacimy. Prawdziwe koszty to codzienna pielęgnacja i opieka”.

Celem modelu finansowania bazującego na produktach jest docenianie osiągnięcia celów organizacji i wyników klienta, takich jak przychody, komfort pracy klienta lub stopa akceptacji klienta — najlepiej przy minimalnych nakładach (np. wysiłku i czasu, linii kodu itp.). Spróbujmy zestawić to z typowym sposobem mierzenia projektów — na przykład czy projekt został zrealizowany w ramach planowanego budżetu, czasu i w planowanym zakresie.

PROJEKTOWANIE GRANIC ZESPOŁÓW ZGODNIE Z PRAWEM CONWAYA

W miarę jak organizacja się rozwija, jednym z największych wyzwań staje się utrzymanie skutecznej komunikacji i koordynacji pomiędzy ludźmi i zespołami. Bardzo często, gdy ludzie i zespoły pracują na innych piętrach, w innych budynkach lub w innej strefie czasowej, stworzenie i utrzymanie wspólnego zrozumienia i wzajemnego zaufania staje się trudniejsze i przeszkadza w skutecznej współpracy. Współpraca jest również utrudniona, gdy podstawowym mechanizmem komunikacji są zlecenia robocze, żądania zmian lub co gorsza, gdy zespoły są rozdzielone przez granice kontraktów — na przykład gdy pracę wykonuje firma zewnętrzna.

Jak widzieliśmy w przykładzie firmy Etsy na początku tego rozdziału, to, jak organizujemy zespoły, może być powodem słabych wyników — jest to skutek uboczny prawa Conwaya. Przykładem złej organizacji może być podział zespołów według funkcji (na przykład umieszczenie programistów i testerów w różnych lokalizacjach) lub według warstw architektonicznych (np. aplikacja, baza danych).

Takie konfiguracje wymagają znaczcej komunikacji i koordynacji pomiędzy zespołami, ale nadal powodują dużą liczbę przeróbek, nieporozumień dotyczących specyfikacji, niepotrzebne przekazywanie pracy oraz bezczynne oczekiwanie na zakończenie pracy przez kogoś innego.

W idealnej sytuacji architektura oprogramowania powinna umożliwiać małym zespołom uzyskanie niezależności i wydajności oraz odseparowanie zespołów od siebie na poziomie wystarczającym do realizacji pracy bez niepotrzebnej komunikacji i koordynacji.

TWORZENIE ARCHITEKTURY LUŻNYCH SPRZĘŻEŃ W CELU ZAPEWNENIA PRODUKTYWNOŚCI PROGRAMISTÓW I BEZPIECZEŃSTWA PRACY

W przypadku architektury ze ścisłymi sprzężeniami niewielkie zmiany mogą powodować błędy na dużą skalę. W rezultacie każdy, kto pracuje w jednej części systemu, musi stale koordynować swoją pracę z osobami pracującymi nad innymi częściami

systemu, na które wprowadzane zmiany mogą wpływać. Koordynacja ta może wymagać stosowania złożonych i biurokratycznych procesów zarządzania zmianami.

Ponadto przetestowanie, czy cały system działa, wymaga zintegrowania zmian ze zmianami setki lub nawet tysiący innych programistów, którzy z kolei mogą mieć zależności z dziesiątkami, setkami lub tysiącami połączonych ze sobą systemów. Testowanie odbywa się za pośrednictwem deficytowych integracyjnych środowisk testowania, do których uzyskanie dostępu i których skonfigurowanie często wymaga wielu tygodni. Skutkiem takiej organizacji są nie tylko długie terminy realizacji zmian (zazwyczaj mierzone w tygodniach lub miesiącach), ale także niska wydajność programistów i słabe wyniki wdrażania.

Natomiast gdy mamy architekturę, która pozwala małym zespołom programistów niezależnie implementować, testować i wdrażać kod do produkcji szybko i bezpiecznie, możemy zwiększyć wydajność deweloperów i poprawić wyniki wdrażania. Takie cechy można znaleźć w architekturach zorientowanych na usługi (ang. *service-oriented architectures* — **SOA**), które po raz pierwszy opisano w latach 90. W takiej architekturze usługi mogą być oddzielnie testowane i wdrażane. Kluczową cechą SOA jest to, że obejmuje ona **luźno sprzężone** usługi w ramach **kontekstów ograniczonych***.

Dzięki luźno sprzężonej architekturze usługi w produkcji mogą być aktualizowane niezależnie, bez konieczności aktualizacji innych usług. Usługi nie mogą zależeć od innych usług i, co równie ważne, od wspólnych baz danych (choć mogą dzielić usługę bazy danych, pod warunkiem że nie korzystają ze wspólnego schematu).

Koncepcja kontekstów ograniczonych została opisana w książce Erica J. Evansa *Domain Driven Design*. Chodzi o to, aby programiści mogli zrozumieć i zaktualizować kod usługi, nie wiedząc niczego o wewnętrznych mechanizmach działania innych usług. Usługi komunikują się z innymi usługami wyłącznie za pośrednictwem interfejsów API, a zatem nie mają wspólnych struktur danych, schematów baz danych lub innych wewnętrznych reprezentacji obiektów. Dzięki wykorzystaniu ograniczonych kontekstów usługi są rozdzielone i mają dobrze zdefiniowane interfejsy, co również ułatwia testowanie.

Randy Shoup, były dyrektor inżynierii w firmie Google App Engine, zauważał, że „organizacje stosujące architektury SOA, takie jak Google i Amazon, charakteryzuje niesamowita elastyczność i skalowalność. Organizacje te zatrudniają dziesiątki tysięcy programistów, a mimo to z powodzeniem zachowują wydajność małych zespołów”.

* Te właściwości mają również mikrousługi (ang. *microservices*), które bazują na zasadach SOA. Jednym z popularnych zbiorów wzorców dla nowoczesnej architektury webowej, bazujących na tych zasadach, jest „aplikacja dwunastu czynników”.

UTRZYMANIE MAŁEGO ROZMIARU ZESPOŁÓW (ZASADA „ZESPÓŁ NA DWIE PIZZE”)

Prawo Conwaya pomaga zaprojektować granice zespołów w kontekście pożądanych wzorców komunikacji, ale również zachęca do utrzymania niewielkiego rozmiaru zespołów, tak aby zmniejszyć skalę komunikacji pomiędzy zespołami i zachęcić do tego, aby zakres dziedziny każdego zespołu był mały i ograniczony.

W ramach inicjatywy transformacji od monolitycznej bazy kodu w 2002 r. w celu utrzymania niewielkich rozmiarów zespołów firma Amazon zastosowała regułę **dwie pizze** — rozmiar zespołu powinien być taki, aby na posiłek dla wszystkich wystarczyły dwie pizze — zazwyczaj od pięciu do dziesięciu osób.

Ograniczenie rozmiarów zespołów ma cztery istotne skutki:

1. Gwarantuje, że członkowie zespołu mają jasne, wspólne zrozumienie systemu, nad którym pracują. W miarę jak zespoły stają się większe, wymagana ilość komunikacji potrzebna do tego, aby wszyscy wiedzieli, co się dzieje, wzrasta wykładniczo.
2. Ogranicza tempo wzrostu produktu lub usługi podczas opracowywania. Po przez ograniczenie rozmiaru zespołu ograniczamy tempo, w jakim ewoluje system, nad którym zespół pracuje. To pomaga również zapewnić utrzymanie w zespole wspólnego zrozumienia systemu.
3. Decentralizuje kierownictwo i umożliwia autonomię. Każdy zespół dwóch pizz (2PT) zachowuje jak największą autonomię. Lider zespołu, współpracując z zespołem menedżerów, decyduje o kluczowych parametrach biznesowych, za które zespół jest odpowiedzialny. Jest to tzw. funkcja fitness, która staje się ogólnym kryterium oceny działania zespołu. Dzięki temu zespół może pracować autonomicznie w celu maksymalizacji tego parametru *.
4. Przewodzenie zespołowi 2PT jest sposobem na to, aby pracownicy mogli zdobyć doświadczenie w zarządzaniu, pracując w środowisku, w którym awaria nie powoduje katastrofalnych konsekwencji. Kluczowym elementem strategii firmy Amazon był związek pomiędzy strukturą organizacyjną 2PT i zastosowaniem architektury zorientowanej na usługi.

W 2005 roku CTO firmy Amazon Werner Vogels wyjaśnił zalety tej struktury Larry'emu Dignanowi z firmy Baseline.

„Małe zespoły działają szybko... i nie gręźną w sprawach administracyjnych (ang. *administrivia*)... Każdy zespół jest przypisany do konkretnego przedsię-

* W kulturze firmy Netflix jedna z siedmiu kluczowych wartości zespołów brzmi: „bardzo wyrównany, luźno powiązany”.

wzięcia i jest za nie całkowicie odpowiedzialny... Zespół definiuje zakres poprawki, projektuje ją, buduje, implementuje i monitoruje jej wykorzystanie. W ten sposób programiści technologii i architekci uzyskują bezpośrednie sprzężenie zwrotne od przedstawicieli biznesu, którzy korzystają z ich kodu lub aplikacji — podczas regularnych spotkań i nieformalnych konwersacji”.

Kolejnym przykładem tego, jak głęboko architektura może wpływać na poprawę produktywności, jest program API Enablement w firmie Target, Inc.

Studium przypadku.

API Enablement w firmie Target (2015)

Target jest szóstą co do wielkości siecią sprzedaży detalicznej w Stanach Zjednoczonych. Rocznie sprzedaje technologie za ponad 1 miliard dolarów. Heather Mickman, dyrektor rozwoju w tej firmie, opisała początki prowadzonych w niej transformacji DevOps: „W dawnych, złych czasach konfiguracja serwera w firmie Target wymagała zaangażowania dziesięciu różnych zespołów, a gdy nastąpiła awaria, aby uniknąć dalszych problemów, zwykle zatrzymywaliśmy wprowadzanie zmian, co oczywiście tylko pogarszało sprawę”.

Uciążliwości związane z konfigurowaniem środowisk i przeprowadzaniem instalacji, a także z uzyskiwaniem dostępu do potrzebnych danych stwarzały znaczące trudności zespołom programistów. Jak opisuje Mickman:

Problemem było to, że wiele z naszych podstawowych danych, takich jak informacje dotyczące zapasów, cen i sklepów, było zamkniętych w starszych systemach i platformie mainframe. Często mieliśmy wiele źródeł danych. Dotyczyło to zwłaszcza styku pomiędzy działem e-commerce a sklepami fizycznymi. Były za nie odpowiedzialne różne zespoły, które korzystały z różnych struktur danych i miały różne priorytety... W rezultacie, gdy nowy zespół programistów chciał coś zbudować dla użytkowników, to od trzech do sześciu miesięcy zajmowało zbudowanie integracji, aby uzyskać potrzebne dane. Na domiar złego ze względu na olbrzymią liczbę niestandardowych integracji punkt-punkt w ścisłe sprzężonym systemie kolejne trzy miesiące do sześciu miesięcy zajmowało wykonywanie ręcznych testów sprawdzających, czy żaden z kluczowych obszarów nie uległ awarii. Konieczność zarządzania interakcjami pomiędzy różnymi zespołami w liczbie od 20 do 30, wraz ze wszystkimi zależnościami, wymagała zatrudniania wielu menedżerów projektów. Oznaczało to, że działa programistów poświęcały cały swój czas na oczekивание w kolejach, zamiast dostarczać wyniki i realizować zadania.

Długi czas pobierania i tworzenia danych w systemach SOR był zagrożeniem dla istotnych celów biznesowych, takich jak integracja działania łańcucha dostaw fizycznych sklepów i ich witryny e-commerce firmy Target w celu dostarczania

towarów do sklepów i domów klientów. To uaktywniło łańcuch dostaw firmy Target daleko poza jego podstawowe przeznaczenie — tzn. ułatwienie przepływu towarów od dostawców do sklepów i centrów dystrybucyjnych.

W 2012 roku, próbując rozwiązać problem danych, Mickmana powołał zespół API Enablement, który miał za zadanie umożliwić zespołom programistów „dostarczanie nowych funkcjonalności w ciągu dni, a nie miesięcy”. Dążono do tego, aby każdy zespół inżynierów w firmie Target miał możliwość pobierania i zapisywania potrzebnych danych — na przykład informacji o produkach lub sklepach, w których są dostępne, w tym lokalizacji, godzin pracy, dostępności kawiarni Starbucks itd.

W doborze członków zespołu dużą rolę odegrały ograniczenia czasowe. Mickman wyjaśniła, że:

Ponieważ nasz zespół miał dostarczać funkcjonalności w ciągu kilku dni, a nie miesięcy, to potrzebowałem zespołu, który mógłby wykonać pracę, a nie powierzać ją kontrahentom — potrzebowaliśmy ludzi z umiejętnościami inżynierskimi, a nie takich, którzy potrafili zarządzać kontraktami. Aby mieć pewność, że nasza praca nie oczekuje w kolejce, musieliśmy być właścicielem całego stosu, co oznaczało, że przyjęliśmy także odpowiedzialność za wymagania działu Ops... Wprowadziliśmy wiele nowych narzędzi do wsparcia ciągłej integracji i ciągłych dostaw. Ponieważ wiedzieliśmy, że jeśli odniesiemy sukces, to będziemy zmuszeni do skalowania produktu do bardzo dużych rozmiarów, wprowadziliśmy nowe narzędzia, takie jak baza danych Cassandra i system komunikacji typu Message Broker Kafka. Gdy zapytaliśmy o zgodę na ich zakup, nie otrzymaliśmy jej, ale i tak zaopatrzyliśmy się w te narzędzia, ponieważ wiedzieliśmy, że są nam potrzebne.

W kolejnych dwóch latach zespół API Enablement dostarczył 53 nowe funkcjonalności biznesowe, tym usługi *Ship to Store* (dosł. „dostawa do sklepu”) oraz *Gift Registry* (dosł. „rejestr prezentów”), a także zrealizował integracje z systemami Instacart i Pinterest. Jak powiedziała Mickman: „Praca z systemem Pinterest nagle stała się bardzo łatwa, ponieważ dostarczyliśmy im nasze API”.

W 2014 roku zespół API Enablement obsługiwał ponad 1,5 miliarda wywołań API miesięcznie. Do 2015 roku liczba ta wzrosła do 17 miliardów wywołań miesięcznie, obejmujących 90 różnych interfejsów API. Do obsługi tej funkcjonalności rutynowo wykonywano 80 wdrożeń tygodniowo.

Wprowadzone zmiany przyniosły firmie Target wymierne korzyści — w okresie świątecznym 2014 roku cyfrowa sprzedaż wzrosła o 42%, a w ciągu drugiego kwartału zwiększyła się o kolejne 32%. Podczas weekendu z czarnym piątkiem w 2015 roku stworzono ponad 280 tysięcy zleceń odbioru zamówień w sklepie. Do 2015 roku celem firmy było wprowadzenie możliwości realizacji zamówień e-commerce w 450 z 1800 sklepów. W stanie początkowym takie możliwości zapewniało około 100 placówek.

„Zespół API Enablement pokazał, co może osiągnąć zespół pasjonatów-agentów zmiany”, powiedziała Mickman. „Pomógł nam ustawić scenę dla kolejnego etapu, którym jest rozwinięcie metodyki DevOps na całą organizację techniczną”.

PODSUMOWANIE

Na przykładzie studiów przypadków firm Etsy i Target możemy zobaczyć, w jaki sposób projekt architektury i organizacja zespołów mogą znacznie poprawić osiągane wyniki. Przy niewłaściwej architekturze i organizacji prawa Conwaya spowoduje złe wyniki, zablokuje bezpieczeństwo i zagrozi zwinności. Przy właściwej organizacji zespołów i architektury programiści zyskają możliwość niezależnego rozwijania, testowania i wdrażania wartości dla klientów.

Jak uzyskać świetne efekty poprzez zintegrowanie zadań działu Ops z codzienną pracą działu Dev?

Naszym celem jest osiągnięcie rynkowych wyników w konfiguracji, w której wiele małych zespołów może szybko i samodzielnie dostarczać wartość klientom. Wykonanie tego zadania może być wyzwaniem w sytuacji, gdy dział operacji jest decentralizowany i zorientowany na funkcję, zmuszony do spełniania wymagań wielu różnych zespołów programistów o potencjalnie bardzo różnych potrzebach. W efekcie czas realizacji pracy Ops często jest długi, wymaga ciągłej zmiany priorytetów i eskalacji, a wyniki wdrażania są niezadowalające.

Aby uzyskać lepsze wyniki rynkowe, należy lepiej zintegrować możliwości Ops z zadaniami zespołów Dev. Dzięki temu oba rodzaje zadań są realizowane skuteczniej i wydajniej. W tym rozdziale przeanalizujemy wiele sposobów osiągnięcia tego stanu — zarówno na poziomie instytucji, jak i poprzez codzienne praktyki. W ten sposób dział Ops może znacząco poprawić produktywność zespołów programistów w całej organizacji, a także zapewnić lepszą współpracę i wyniki organizacyjne.

W firmie Big Fish Games obsługującej setki gier na urządzenia mobilne i tysiące gier PC z przychodem w 2013 roku wynoszącym ponad 266 milionów dolarów Paul Farrall, wiceprezes działu operacji IT, był odpowiedzialny za decentralizowany dział Ops. Do jego obowiązków należało wsparcie wielu różnych jednostek biznesowych o dużej autonomii.

Każda z tych jednostek miała dedykowany zespół deweloperów, którzy często wybierali bardzo różne technologie. Gdy te grupy chcięły wdrożyć nową funkcjonalność,

były zmuszone rywalizować o współdzieloną pulę deficytowych zasobów Ops. Ponadto każda z tych grup borykała się z niewiarygodnymi środowiskami testowania i integracji, a także z niezwykle kłopotliwymi procesami wydawania oprogramowania.

Farrall pomyślał, że najlepszym sposobem na rozwiązywanie tych problemów będzie włączenie wiedzy Ops do zespołów programistów. Jak zauważał: „Gdy zespoły programistów miały problemy z testowaniem lub wdrażaniem, potrzebowały nie tylko technologii lub środowisk. Potrzebne im były także pomoc i nadzór merytoryczny. Początkowo staraliśmy się wprowadzić inżynierów i architektów Ops do każdego z zespołów programistów, ale po prostu nie było wystarczająco dużo inżynierów, aby zaspokoić potrzeby. Udało się pomóc większej liczbie zespołów dzięki zastosowaniu modelu łączników Ops, do którego wykorzystania potrzeba było znacznie mniej ludzi”.

Farrall zdefiniował dwa rodzaje łączników Ops: menedżera relacji biznesowych i dedykowanego inżyniera wydań. Menedżerowie relacji biznesowych pracowali z kierownikami produktu, właścicielami biznesowymi, kierownictwem projektu, kierownictwem zespołów Dev i programistami. Osoby te dokładnie poznaly motywacje biznesowe zespołów zajmujących się produktem i mapy drogowe produktu, występuowały jako przedstawiciele interesów właściciela produktu wewnętrz działu operacji i pomagały zespołowi obsługi produktu poruszać się w przestrzeni działu Ops w celu nadania priorytetów i usprawnienia obsługi zleceń roboczych.

Na podobnej zasadzie dedykowani inżynierowie wydania dokładnie znali problemy działów Dev i walidacji (QA) i pomagali im otrzymać od organizacji Ops to, co było potrzebne do osiągnięcia ich celów. Znali typowe żądania działów Dev i QA od działu Ops i często sami wykonywali potrzebne prace. W razie potrzeby zwracali się o pomoc do wyznaczonych inżynierów Ops (np. administratorów baz danych, specjalistów ds. Infosec, magazynów danych i inżynierów sieci) i pomagali w określeniu priorytetów tworzenia samoobsługowych narzędzi.

W ten sposób Farrall był w stanie pomóc zespołom programistów w całej organizacji stać się bardziej wydajnymi i osiągnąć cele. Ponadto pomógł zespołom wyznaczyć priorytety dotyczące globalnych ograniczeń Ops, doprowadził do zmniejszenia liczby niespodzianek odkrywanych w trakcie projektu i ostatecznie przyczynił się do ogólnej poprawy przepustowości projektu.

Farrall zauważył, że wskutek wprowadzonych zmian poprawiły się zarówno stosunki robocze z działem operacji, jak i tempo publikowania kodu. Jak stwierdził: „Model łączników Ops pozwolił nam osadzić wiedzę działu operacji IT w zespołach programistów i obsługi produktu bez zatrudniania nowych osób”.

Transformacja DevOps w firmie Big Fish Games pokazała sposób, dzięki któremu skonsolidowany zespół operacji potrafił osiągnąć wyniki zwykle związane z zespołami zorientowanymi na cele rynkowe. Możemy zastosować następujące trzy ogólne strategie:

- Tworzenie funkcji samoobsługowych pomagających poprawić wydajność programistom w zespołach usługowych.

- Wprowadzenie inżynierów Ops do zespołów usługowych.
- Przydzielenie łączników Ops do zespołów usługowych, gdy wprowadzenie do nich inżynierów Ops nie jest możliwe.

W kolejnym podrozdziale opiszemy, jak inżynierowie Ops mogą zintegrować się z praktykami stosowanymi na co dzień w pracy przez zespoły Dev, takimi jak codzienne spotkania *standup*, planowanie i retrospektywy.

TWORZENIE WSPÓLNYCH USŁUG W CELU ZWIĘKSZENIA WYDAJNOŚCI PRACY PROGRAMISTÓW

Jednym ze sposobów uzyskania rynkowych wyników jest stworzenie przez dział operacji zestawu scentralizowanych platform i narzędzi, które mogą być wykorzystywane przez zespoły Dev do poprawienia swojej wydajności. Można do nich zaliczyć środowiska zbliżone do produkcyjnych, potoki instalacji, narzędzi automatycznego testowania, pulpity telemetrii produkcji i tak dalej^{*}. W ten sposób stwarzamy zespołom programistów warunki dogodne do tego, aby mogły poświęcić więcej czasu na budowanie funkcjonalności dla swoich klientów w przeciwieństwie do dostarczenia całej infrastruktury wymaganej do opracowania i obsługi tej funkcji w produkcji.

Wszystkie platformy i usługi, które oferujemy, powinny (najlepiej) być zautomatyzowane i dostępne na żądanie i nie powinny wymagać od programisty wystawiania zlecenia roboczego i oczekiwania, aż ktoś ręcznie wykona pracę. Dzięki temu dział operacji nie stanie się wąskim gardłem dla swoich klientów (np.: „Otrzymaliśmy Twoje zgłoszenie. Ręczne skonfigurowanie tych środowisk testowych zajmie sześć tygodni”)[†].

W ten sposób umożliwiamy zespołom produktu uzyskać to, czego potrzebują, a także zmniejszyć potrzebę komunikacji i koordynacji. Jak zauważał Damon Edwards: „Bez samoobsługowych platform operacji chmurę można jedynie porównać do platformy Drogi Hosting 2.0”.

W prawie wszystkich przypadkach nie przesądzamy konieczności użycia tych platform i usług przez wewnętrzne zespoły — zespoły zajmujące się dostarczaniem platform będą musiały pozyskać i zaspokoić swoich wewnętrznych klientów, czasami nawet konkuruając z dostawcami z zewnątrz. Dzięki stworzeniu tego skutecznego wewnętrznego rynku możliwości pomagamy zapewnić stan, w którym tworzone platformy i usługi są najprostszym i najatrakcyjniejszym dostępnym mechanizmem (linia najmniejszego oporu).

^{*} W tej książce terminy „platforma”, „usługa współdzielona” i „łańcuch narzędzi” (ang. *toolchain*) będą używane zamiennie.

[†] Ernest Mueller zaobserwował: „W firmie Bazaarvoice ustalono, że zespoły obsługi platformy, które tworzą narzędzia, akceptują wymagania, ale nie pracę z innych zespołów”.

Na przykład możemy stworzyć platformę, która dostarcza wspólne repozytorium kontroli wersji z „namaszczonymi” bibliotekami zabezpieczeń, potokiem instalacji, który automatycznie uruchamia narzędzia validacji i skanowania naruszeń bezpieczeństwa i instaluje nasze aplikacje w **znanych, dobrych środowiskach**, już wyposażonych w narzędzia monitorowania produkcji. W idealnej sytuacji możemy tak bardzo ułatwić życie zespołom deweloperów, że w zdecydowanej większości uznają, iż korzystanie z naszej platformy jest najłatwiejszym, najbezpieczniejszym i zarazem naj pewnością sposobem wdrażania ich aplikacji do produkcji.

Budujemy na tych platformach skumulowaną i zbiorową wiedzę wszystkich osób w organizacji, w tym działów QA, Ops i Infosec, co pomaga nam tworzyć coraz większe bezpieczeństwo pracy. To zwiększa produktywność programistów i ułatwia zespołom produktów wykorzystanie wspólnych procesów, takich jak uruchamianie zautomatyzowanych testów czy też spełnienie wymogów dotyczących bezpieczeństwa i zgodności z przepisami.

Tworzenie i utrzymanie tych platform i narzędzi jest rzeczywistym procesem rozwoju produktu — klientami naszej platformy nie są zewnętrzni odbiorcy, ale wewnętrzne zespoły programistów. Podobnie jak w przypadku każdego dobrego produktu, zbudowanie doskonałej platformy, którą wszyscy uwielbiają, nie zdarza się przez przypadek. Zespół pracujący nad platformą wewnętrzną, który nie będzie wystarczającą koncentrował się na swoich klientach, z dużym prawdopodobieństwem stworzy narzędzia, których nikt nie będzie lubił, i szybko porzuci je na rzecz alternatyw — tworzonych przez inny wewnętrzny zespół albo zewnętrznego dostawcę.

Dianne Marsh, dyrektor działu narzędzi inżynierskich w Netflix, stwierdziła, że misją jej zespołu jest „wsparcie innowacyjności i szybkości działania zespołów inżynierskich. Nie budujemy, nie tworzymy, nie wdrażamy niczego dla tych zespołów ani nie zarządzamy ich konfiguracjami. Zamiast tego budujemy narzędzia umożliwiające samoobsługę. Nie mamy nic przeciwko temu, że ktoś uzależnia się od naszych narzędzi. Dla nas ważne jest to, że osoby te nie uzależniają się od nas”.

Często zespoły platform świadczą inne usługi pomagające klientom nauczyć się ich technologii, dokonywać migracji z innych technologii, a nawet zapewniać szkolenia i doradztwo w celu poprawy jakości praktyk stosowanych wewnętrz organizacji. Współzielone usługi ułatwiają również standaryzację, która pozwala inżynierom szybko uzyskać wydajność nawet wtedy, gdy zmieniają zespoły. Przykładowo, jeśli zespół każdego produktu wybierze inny zestaw narzędzi, to inżynierowie w celu wykonania swojej pracy będą musieli się nauczyć całkiem nowego zbioru technologii, przedkładając cele zespołu nad cele globalne.

W organizacjach, w których zespoły mogą stosować tylko zatwierdzone narzędzia, można rozpocząć od usunięcia wymagania zatwierdzenia dla kilku zespołów — na przykład zespołu transformacji. Dzięki temu możliwe staje się eksperymentowanie i odkrywanie nowych dróg, które czynią zespoły bardziej wydajnymi.

Wewnętrzne zespoły usług współdzielonych powinny nieustannie szukać wewnętrznych zestawów narzędzi, które są powszechnie wykorzystywane w organizacji, i decydować o tym, które z nich warto centralnie wspierać i udostępniać dla wszystkich. Ogólnie rzeczą biorąc, zastosowanie narzędzi, które już gdzieś się sprawdzają, i rozszerzenie ich użycia daje znacznie większe szanse powodzenia niż budowanie tych możliwości od podstaw^{*}.

WPROWADZANIE INŻYNIERÓW OPS DO ZESPOŁÓW USŁUGOWYCH

Innym sposobem osiągnięcia bardziej rynkowych wyników jest zapewnienie zespołom produktów większej samodzielności poprzez wprowadzenie do nich inżynierów Ops, a przez to osłabienie zależności tych zespołów od scentralizowanego działu operacji. Zespoły produktu mogą być również w pełni odpowiedzialne za dostarczanie usługi i jej wsparcie.

Dzięki wprowadzeniu inżynierów do zespołów Dev ich priorytetami stają się prawie w całości cele zespołów produktów, do których ci inżynierowie zostali wprowadzeni (w przeciwnieństwie do koncentrowania się inżynierów Ops wyłącznie na rozwiązywaniu własnych problemów). W rezultacie inżynierowie Ops stają się ściślej powiązani ze swoimi wewnętrznymi i zewnętrznymi klientami. Ponadto zespoły produktu często mają budżet pozwalający sfinansować zatrudnienie inżynierów Ops, choć w celu zapewnienia spójności i jakości personelu przeprowadzanie rozmów kwalifikacyjnych i podejmowanie decyzji rekrutacyjnych prawdopodobnie nadal będzie należało do scentralizowanego zespołu operacyjnego.

Jason Cox powiedział: „W wielu działach firmy Disney wprowadziliśmy inżynierów Ops (inżynierów systemowych) do zespołów produktu, a także do zespołów programistów, testów, a nawet bezpieczeństwa informacji. To całkowicie zmieniło dynamikę naszej pracy. Jako inżynierowie operacyjni tworzymy narzędzia i zapewniamy możliwości, które zmieniają sposób, w jaki ludzie pracują, a nawet w jaki myślą. W tradycyjnym modelu Ops jedynie prowadziliśmy pociąg zbudowany przez kogoś innego. Natomiast w nowoczesnej inżynierii operacji nie tylko pomagamy budować pociągi, ale także mosty, po których te pociągi się poruszają”.

W przypadku dużych nowych projektów rozwojowych można od razu wprowadzić inżynierów Ops do zespołów deweloperskich. Do ich obowiązków może należeć: wspomaganie podejmowania decyzji o tym, co budować i jak budować, wpływanie na architekturę produktu, wpływanie na decyzje dotyczące wyboru wewnętrznych i zewnętrznych technologii, wspomaganie tworzenia nowych możliwości wewnętrznych

* Jak wiadomo, projektowanie systemów do wielokrotnego wykorzystania „od góry” jest znanym i kosztownym błędem w wielu architekturach korporacyjnych.

platform, generowanie nowych możliwości operacyjnych. Po opublikowaniu produktu inżynierowie Ops pracujący wewnątrz zespołów programistów mogą pomagać w spełnianiu obowiązków produkcyjnych zespołu deweloperów.

Biorą oni udział we wszystkich „rytuałach” zespołu Dev, takich jak planowanie sprintów, codzienne standupy oraz demonstracje, podczas których zespół prezentuje nowe funkcje i decyduje o tym, które należy wdrożyć. Kiedy zapotrzebowanie na wiedzę i możliwości Ops zmniejszy się, inżynierowie Ops mogą przejść do innych projektów lub przedsięwzięć, zgodnie z ogólnym wzorcem, że skład zespołów produktu zmienia się w całym cyklu życia produktu.

Zastosowanie tego paradygmatu ma inną ważną zaletę: połączenie możliwości inżynierów Dev i Ops jest bardzo skutecznym sposobem włączenia wiedzy i doświadczenia zespołu operacji do zespołów usług. Może również przynosić duże korzyści z przekształcenia wiedzy operacyjnej w zautomatyzowany kod, który może być znacznie bardziej niezawodny i powszechnie używany.

PRZYDZIELENIE ŁĄCZNIKÓW OPS DO KAŻDEGO ZESPOŁU USŁUGOWEGO

Z różnych powodów, takich jak koszty czy niedobory kadrowe, możemy nie być w stanie wprowadzić inżynierów Ops do każdego zespołu produktu. Możemy jednakże uzyskać wiele takich samych korzyści, przypisując wyznaczonych łączników do każdego zespołu produktu.

W firmie Etsy taki model określa się „wyznaczonymi Ops” (ang. *designated Ops*). Ich scentralizowany zespół operacji w celu zapewnienia spójności działania nadal zarządza wszystkimi środowiskami — nie tylko środowiskami produkcyjnymi, ale także przedprodukcyjnymi. Wyznaczony inżynier Ops jest odpowiedzialny za:

- Zrozumienie, czym jest nowa funkcjonalność produktu i dlaczego jest budowana.
- Zrozumienie działania funkcjonalności pod względem operacyjności, skalowalności i możliwości obserwacji (zalecane jest tworzenie diagramów).
- Poznanie sposobów monitorowania i zbierania parametrów w celu zapewnienia postępów oraz obserwacji sukcesu lub niepowodzenia funkcjonalności.
- Poznanie wszelkich odstępstw w porównaniu z poprzednią architekturą i wzorcami oraz uzasadnienia dla tych odstępstw.
- Spełnianie wszelkich dodatkowych potrzeb w zakresie infrastruktury oraz rozumienie wpływu zastosowania funkcjonalności na możliwości infrastruktury.
- Przygotowanie planów uruchomienia funkcjonalności.

Ponadto podobnie jak w modelu inżynierów Ops wprowadzonych do zespołów, łącznik Ops uczestniczy w codziennych spotkaniach standup zespołu, integruje potrzeby zespołu z „mapą drogową” działu operacji oraz wykonuje wszelkie niezbędne zadania. Łączników wykorzystujemy do eskalowania problemów rywalizacji lub nadawania priorytetów. W ten sposób można zidentyfikować konflikty zasobów lub czasu, które powinny być ocenione i spriorytetyzowane w kontekście szerszych celów organizacyjnych.

Przypisanie łączników Ops pozwala wspierać więcej zespołów produktu w porównaniu z modelem inżynierów Ops wprowadzonych do zespołów. Celem jest zapewnienie stanu, w którym inżynierowie Ops nie są ograniczeniem dla zespołów produktu. W przypadku stwierdzenia, że łączników Ops jest zbyt mało i że ta sytuacja uniemożliwia zespołom produktów osiągnięcie wyznaczonych celów, należy albo zmniejszyć liczbę zespołów obsługiwanych przez każdego łącznika, albo tymczasowo wprowadzić inżyniera Ops do pojedynczych zespołów.

INTEGRÓWANIE INŻYNIERÓW OPS Z PRAKTYKAMI ZESPOŁÓW DEV

Gdy inżynierowie Ops są wprowadzeni lub przypisani jako łącznicy do zespołów produktów, można ich zintegrować ze stosowanymi praktykami przez zespoły Dev. W tym podrozdziale naszym celem jest umożliwienie inżynierom Ops i innym osobom niebędącym programistami lepiej zrozumieć istniejącą kulturę zespołów Dev i proaktywnie zintegrować ich ze wszystkimi aspektami planowania i codziennej pracy zespołów Dev. W rezultacie zespół operacji jest w stanie lepiej zaplanować i wyemitować wiedzę niezbędną dla zespołów produktów oraz wpływać na pracę zespołów programistów, zanim produkt trafi do produkcji. W kolejnych podrozdziałach opisano niektóre standardowe praktyki stosowane przez zespoły deweloperów korzystających z metodyki Agile oraz sposóbłączenia do tych praktyk inżynierów Ops. Stosowanie praktyk Agile w żadnym razie nie jest warunkiem wstępny dla tego kroku — celem inżynierów Ops powinno być rozpoznanie praktyk stosowanych przez zespoły produktów, zintegrowanie się z nimi i dodanie wartości*.

Jak zaobserwował Ernest Mueller: „Uważam, że DevOps sprawdza się znacznie lepiej, jeśli zespoły operacyjne przyjmą takie same praktyki Agile, jakie stosują zespoły

* Jeśli jednak odkryjemy, że w całej organizacji programiści tylko siedzą przy swoich biurkach przez cały dzień i nigdy ze sobą nie rozmawiają, to być może powinniśmy znaleźć inny sposób, aby ich zaktywizować. Można zafundować im obiad, założyć klub książek, zainicjować prezentacje w stylu „zjedz posiłek i naucz się czegoś” lub prowadzić rozmowy zmierzające do odkrycia największych problemów poszczególnych członków zespołu, tak by można było uczynić ich życie łatwiejszym.

programistów — odnieśliśmy fantastyczne sukcesy, rozwiązyując wiele problemów związanych z bolączkami Ops oraz zapewniając lepszą integrację inżynierów Ops z zespołami Dev”.

UDZIAŁ INŻYNIERÓW OPS W CODZIENNYCH SPOTKANIACH STANDUP ZESPOŁU DEV

Jednym z rytuałów Dev spopularyzowanych przez metodykę Scrum są codzienne spotkania na stojąco — tzw. standupy — krótkie spotkania, w których uczestniczą wszyscy członkowie zespołu i prezentują sobie nawzajem trzy rzeczy: co zrobili wczoraj, co będą robić dzisiaj i co przeszkadzi im w wykonaniu zadań*.

Celem tego rytuału jest rozpropagowanie informacji w zespole, zrozumienie wykonywanych zadań oraz zakomunikowanie, co będzie zrobione. Dzięki temu, że członkowie zespołu prezentują te informacje, dowiadujemy się o wykonywanych zadaniach, przeszkołach, które blokują ich realizację, oraz odkrywamy sposoby udzielania pomocy członkom zespołu, tak by poruszali się w kierunku wyznaczonego celu. Ponadto dzięki obecności menedżerów można szybko rozwiązać konflikty dotyczące priorytetyzacji i zasobów.

Częstym problemem jest to, że informacje prezentowane na spotkaniach standup są zamknięte wewnętrz zespołu programistów. Dzięki uczestnictwu inżynierów Ops w tych spotkaniach dział operacji zyskuje wiedzę o zadaniach realizowanych przez zespół programistów. To umożliwia lepsze planowanie i przygotowanie się do zadań — na przykład, jeśli dział operacji dowie się, że zespół produktu planuje wdrożenie złożonej funkcjonalności w ciągu dwóch tygodni, to może zadbać o przydzielenie do wsparcia realizacji tego zadania odpowiednich ludzi i odpowiednie zasoby. Można również wyróżnić obszary, w których potrzebna jest bliższa interakcja lub większe przygotowania (np. stworzenie więcej kontroli monitorujących lub automatycznych skryptów). W ten sposób można stworzyć warunki do tego, by dział operacji pomagał w rozwiązyaniu bieżących problemów zespołu (np. poprawa wydajności poprzez dostrojenie bazy danych zamiast optymalizowania kodu) lub przyszłych (np. stworzenie większej liczby środowisk testów integracyjnych w celu umożliwienia testowania wydajności), zanim zamienią się one w kryzys.

* Scrum to metodologia zwinnego wytwarzania oprogramowania, opisywana jako „elastyczna, holistyczna strategia rozwoju produktu, zgodnie z którą zespół programistów pracuje jako jednostka w kierunku osiągnięcia wspólnego celu”. Po raz pierwszy została kompleksowo opisana przez Kena Schwabera i Mike'a Beedle'a w książce *Agile Software Development with Scrum*. W niniejszej książce do opisania technik wykorzystywanych w różnych metodologiach, takich jak Agile i Scrum, używamy terminów „zwinne wytwarzanie oprogramowania” lub „iteracyjne wytwarzanie oprogramowania”.

UDZIAŁ INŻYNIERÓW OPS W RETROSPEKTYWACH ZESPOŁU DEV

Innym powszechnym rytuałem Agile są retrospektywy. Na koniec każdego interwału zespół omawia przedsięwzięcia, które zakończyły się pomyślnie, to, co należy poprawić, oraz to, jak uwzględnić sukcesy i ulepszenia w przyszłych iteracjach lub projektach. Zespół próbuje wyciągnąć wnioski zmierzające do rozwiązymania problemów oraz omawia doświadczenia z poprzedniej iteracji. Jest to jeden z głównych mechanizmów, za którego pośrednictwem odbywa się uczenie się w organizacji oraz opracowywanie środków zaradczych. Prace do wykonania są realizowane natychmiast albo są dodawane do rejestru zadań do zrealizowania.

Uczestnictwo inżynierów Ops w retrospektywach zespołu Dev umożliwia im osiąganie korzyści ze zdobywania nowej wiedzy. Co więcej, jeśli podczas wybranego interwału są zaplanowane wdrożenie lub publikacja, to dział operacji powinien zaprezentować wyniki tych przedsięwzięć oraz wnioski, z których może skorzystać zespół produktu. W ten sposób możemy poprawić sposób planowania i realizacji przyszłej pracy, co zapewnia poprawę osiąganych wyników. Oto przykłady informacji, które inżynierowie Ops mogą prezentować podczas retrospektyw:

- „Dwa tygodnie temu znaleźliśmy »martwy punkt« w monitorowaniu i zadecydowaliśmy o sposobie jego naprawy. Poprawka sprawdziła się. W ubiegły wtorek zdarzył się incydent. Udało się go szybko wykryć i skorygować, zanim odczuli go klienci”.
- „Wdrożenie w ubiegłym tygodniu należało do najtrudniejszych i najdłuższych w roku. Mamy kilka pomysłów, jak można to poprawić”.
- „Kampania promocyjna, którą przeprowadziliśmy w ubiegłym tygodniu, okazała się znacznie trudniejsza, niż myśleliśmy. Chyba nie powinniśmy wprowadzać takiej oferty ponownie. Oto kilka pomysłów na inne oferty, które mogą pomóc osiągnąć nasze cele”.
- „Podczas ostatniego wdrożenia największym problemem były reguły zapory firewall, które obecnie mają wiele tysięcy. Modyfikowanie ich jest bardzo trudne i ryzykowne. Musimy zmodyfikować architekturę blokowania nieuprawnionego ruchu sieciowego”.

Informacje z działu operacji pomagają zespołom produktu zauważać i zrozumieć wpływ ich decyzji na pracę osób w dole strumienia wartości. W przypadku niekorzystnych wyników można wprowadzić zmiany niezbędne do zapobieżenia problemom w przyszłości. Opinie z działu operacji pozwalają także zidentyfikować dodatkowe problemy oraz defekty, które należy naprawić — mogą nawet przyczynić się do wykrycia większych problemów architektonicznych.

Dodatkowe prace zidentyfikowane podczas retrospektyw zespołu projektowego można przypisać do szerokiej kategorii zadań usprawniania, takich jak poprawianie defektów, refaktoryzacja i automatyzowanie ręcznej pracy. Menedżerowie produktu i projektu czasem dążą do odkładania lub nadawania zadaniom usprawniania niskiego priorytetu w porównaniu z opracowywaniem funkcji dla klienta.

Trzeba jednak przypomnieć, że usprawnianie codziennej pracy jest ważniejsze niż samo jej wykonywanie oraz że wszystkie zespoły powinny przydzielać czas na tego rodzaju prace (na przykład zarezerwować 20% wszystkich cykli — tzn. jednego dnia w tygodniu lub jednego tygodnia w miesiącu). Bez tego wydajność zespołu będzie prawie na pewno maleć z powodu obciążen związań z „zadłużeniem” proceduralnym i technicznym.

PREZENTOWANIE PRAC INŻYNIERÓW OPS NA WSPÓLNEJ TABLICY KANBAN

Zespoły programistów często prezentują realizowane przez siebie zadania na tablicy projektu lub tablicy kanban. Jednak znacznie rzadziej na tablicach są prezentowane prace inżynierów Ops, które muszą być zrealizowane, aby aplikacja mogła być pomyślnie uruchomiona w produkcji — tzn. tam, gdzie jest w rzeczywistości tworzona wartość dla klienta. W rezultacie nie zdajemy sobie sprawy z koniecznych prac Ops do czasu, kiedy powstanie kryzys stwarzający zagrożenie dla terminów lub powodujący przestoje w produkcji.

Ponieważ zadania operacyjne są częścią strumienia wartości produktu, to zadania działu operacji ważne z punktu widzenia dostawy produktu powinny być prezentowane na wspólnej tablicy kanban. Umożliwia to wyraźne zaprezentowanie wszystkich zadań koniecznych do wdrożenia kodu do produkcji, a także śledzenie wszystkich zadań operacyjnych wymaganych do wsparcia produktu. Ponadto pozwala zaobserwować te miejsca, w których zadania Ops są zablokowane, gdzie praca wymaga eskalacji, oraz wyróżnić obszary wymagające poprawy.

Tablice kanban są idealnym narzędziem poprawy widoczności, a widoczność jest kluczowym elementem prawidłowego rozpoznawania i integrowania prac operacyjnych we wszystkie istotne strumienie wartości. Właściwe zastosowanie tablic kanban pozwala osiągnąć wyniki rynkowe niezależnie od tego, jak wyglądają schematy organizacyjne firmy.

PODSUMOWANIE

W całym rozdziale omówiliśmy sposoby integracji zadań działu operacji z codziennymi zadaniami działu programistów oraz wyjaśniliśmy, co zrobić, aby praca działu Dev była bardziej widoczna dla działu Ops. W tym celu zbadaliśmy trzy ogólne strategie: tworzenie mechanizmów samoobsługowych w celu zwiększenia wydajności programistów w zespołach usługowych, wprowadzenie inżynierów Ops do zespołów usługowych oraz przydzielenie łączników Ops do zespołów usługowych tam, gdzie przydział inżynierów Ops nie jest możliwy. Na koniec opisaliśmy sposób, w jaki inżynierowie Ops mogą zintegrować się z praktykami stosowanymi przez zespoły Dev w codziennej pracy, takimi jak codzienne spotkania standup, planowanie i retrospektywy.

PODSUMOWANIE CZĘŚCI II

W części II, „Od czego zacząć”, zbadaliśmy różne sposoby myślenia o przekształceniach DevOps. Wskazaliśmy obszary, od których zacząć, opisaliśmy ważne aspekty architektury i organizacji oraz wyjaśniliśmy, jak zorganizować zespoły. Opowiedzieliśmy również o tym, jak zintegrować działania Ops ze wszystkimi aspektami planowania i codziennej pracy działu Dev.

W części III, „Pierwsza droga. Techniczne praktyki przepływu”, zaczniemy omawiać sposoby wdrażania określonych praktyk technicznych w celu realizacji zasad przepływu, umożliwiających szybki przepływ pracy od działu Dev do działu Ops, bez powodowania chaosu i zakłóceń w dolnej części strumienia wartości.

Część III
Pierwsza droga
Techniczne praktyki przepływu

Część III Wprowadzenie

Celem części III jest zaprezentowanie praktyk technicznych i architektury potrzebnej do zapewnienia i utrzymania szybkiego przepływu pracy od działu Dev do Ops bez wywoływania chaosu i zakłóceń w środowisku produkcyjnym lub u klientów. Oznacza to konieczność zmniejszenia zagrożeń związanych z wdrażaniem i publikowaniem zmian do produkcji. Robimy to przez zastosowanie zbioru technicznych rozwiązań znanego jako **ciągłe dostarczanie** (ang. *continuous delivery*).

Ciągłe dostarczanie tworzy podstawy dla potoku zautomatyzowanego wdrażania, w którym na pewno istnieją zautomatyzowane testy, stale weryfikujące, czy oprogramowanie jest w stanie nadającym się do wdrożenia, programiści codziennie integrują kod z repozytorium oraz tworzą architekturę środowisk i kod w taki sposób, aby publikacje wiązały się z niskim ryzykiem. W tym rozdziale skoncentrujemy się przede wszystkim na następujących zagadnieniach:

- stworzenie podstaw dla potoku wdrażania;
- zapewnienie szybkiego i niezawodnego automatycznego testowania;
- umożliwienie i praktykowanie ciągłej integracji i testowania;
- automatyzacja;
- architektura zapewniająca minimalizację zagrożeń związanych z wydaniami.

Implementacja tych praktyk skraca czas potrzebny do stworzenia środowisk przy-pominających produkcyjne, umożliwia ciągłe testowanie, dostarczające wszystkim szybkich sprzężeń zwrotnych na temat pracy, pozwala niewielkim zespołom szybko i niezależnie projektować testy oraz wdrażać kod do produkcji i sprawia, że wdrożenia do produkcji i wydania stają się rutynową częścią codziennej pracy.

Ponadto integrowanie celów działań walidacji i operacji w codziennej pracy wszystkich osób pracujących w strumieniu wartości zmniejsza potrzebę „gaszenia pożarów oraz trudu i znoju”. Jednocześnie zwiększa produktywność kodu i zadowolenie z wykonywanej pracy. W ten sposób nie tylko poprawiamy wyniki, ale także zwiększamy szanse organizacji na odniesienie sukcesu na rynku.

Podstawy potoku wdrożeń

Aby utworzyć szybki i niezawodny przepływ od Dev do Ops, trzeba zadbać o to, by na każdym etapie strumienia wartości używać środowisk zbliżonych do produkcyjnych. Ponadto środowiska te powinny być stworzone w sposób zautomatyzowany, najlepiej na żądanie oraz z wykorzystaniem informacji o skryptach i konfiguracji przechowywanych w repozytorium kontroli wersji. Powinny być one całkowicie samoobsługowe, niewymagające wykonywania jakichkolwiek zadań działu operacji. Celem jest zapewnienie możliwości odtworzenia całego środowiska produkcyjnego na podstawie tego, co jest zapisane w repozytorium kontroli wersji.

Bardzo często o tym, jak aplikacja działa w środowisku przypominającym produkcyjne, możemy się przekonać dopiero po wdrożeniu jej do produkcji. Zwykle to o wiele za późno na poprawienie błędów w sposób, który nie wpłynie negatywnie na klienta. Obrazowym przykładem spektrum problemów, które mogą być spowodowane przez niespójnie budowane aplikacje i środowiska, jest prowadzony w 2009 roku w dużej australijskiej firmie telekomunikacyjnej przez Em Campbell-Pretty program Enterprise Data Warehouse. Campbell-Pretty została dyrektorem generalnym i sponsorem biznesowym programu o budżecie 200 milionów dolarów i przejęła odpowiedzialność za wszystkie strategiczne cele tej platformy.

W swojej prezentacji na szczytce DevOps Enterprise Summit 2014 Campbell-Pretty wyjaśniła: „W tamtym czasie mieliśmy dziesięć rozpoczętych strumieni pracy. We wszystkich stosowaliśmy procesy kaskadowe i wszystkie dziesięć strumieni miało znaczne opóźnienia. Tylko jeden pomyślnie i zgodnie z harmonogramem dotarł do

fazy testów akceptacyjnych użytkownika. Zakończenie UAT zajęło sześć kolejnych miesięcy, a uzyskane efekty okazały się dalekie od biznesowych oczekiwani. Słaba wydajność była głównym katalizatorem dla transformacji działu do metodyki Agile”.

Jednak po niemal roku stosowania Agile zanotowano tylko niewielkie usprawnienia, a uzyskiwane wyniki biznesowe nadal były dalekie od oczekiwanych. Campbell-Pretty, prowadząc retrospeptywy wśród uczestników całego programu, zadała pytanie: „Co możemy zrobić, uwzględniając wszystkie doświadczenia z poprzedniego wydania, aby podwoić naszą produktywność?”.

Przez cały projekt można było usłyszeć narzekania na „brak biznesowego zaangażowania”. Jednak podczas retrospeptywy na szczytce listy znalazła się „poprawa dostępności środowisk”. Z perspektywy czasu to było oczywiste — zespoły Dev do rozpoczęcia pracy potrzebowaly skonfigurowanych środowisk. Często musiały na nie czekać nawet do ósmego tygodni.

W firmie stworzono nowy zespół integracji i budowania, który był odpowiedzialny za „wprowadzanie jakości do procesów zamiast prób sprawdzania jakości po fakcie”. Początkowo składał się z administratorów baz danych (DBA) oraz specjalistów automatyzacji, którym powierzono zadania automatyzowania procesu tworzenia środowiska. Zespół szybko dokonał zaskakującego odkrycia: zaledwie 50% kodu źródłowego w środowiskach deweloperskich i testowych odpowiadało temu, co działało w środowiskach produkcyjnych.

Campbell-Pretty zaobserwowała: „Nagle zrozumieliśmy, dlaczego napotykaliśmy tak wiele defektów za każdym razem, gdy wdrażaliśmy nasz kod w nowych środowiskach. W każdym środowisku wprowadzaliśmy poprawki, ale wykonywane zmiany nie były wprowadzane do repozytorium kontroli wersji”.

Zespół starannie przeprowadził reverse-engineering wszystkich zmian, które zostały wprowadzone w różnych środowiskach, i umieścił je w repozytorium kontroli wersji. Zautomatyzowano również proces tworzenia środowisk, dzięki czemu można było to robić wielokrotnie i bezbłędnie.

Campbell-Pretty opisała wyniki, zauważając, że „czas stworzenia prawidłowego środowiska uległ skróceniu z ósmiu tygodni do jednego dnia. Była to jedna z kluczowych korekt, które pozwoliły nam osiągnąć cele dotyczące czasu realizacji, kosztów dostarczenia oraz liczby wad, które przedostały się do fazy produkcji”.

Historia Campbell-Pretty pokazuje szereg problemów, których przyczyny można przypisać niespójnie skonstruowanym środowiskom oraz brakowi systematycznego wprowadzania zmian do repozytorium kontroli wersji.

W pozostałej części tego rozdziału omówimy sposoby budowania mechanizmów pozwalających na tworzenie środowisk na żądanie, rozszerzania korzystania z repozytoriów kontroli wersji na wszystkie osoby w strumieniu wartości, przystosowania infrastruktury do tego, aby jej odbudowa była łatwiejsza niż naprawa, oraz zapewnienia uruchamiania przez programistów kodu w środowiskach przypominających produkcyjne we wszystkich fazach cyklu rozwoju oprogramowania.

TWORZENIE ŚRODOWISK PROGRAMISTYCZNYCH, TESTOWYCH I PRODUKCYJNYCH NA ŻĄDANIE

Jak widać na przedstawionym powyżej przykładzie hurtowni danych przedsiębiorstwa, jedną z głównych przyczyn chaotycznych, uciążliwych, a czasami nawet powodujących katastrofalne skutki publikacji oprogramowania jest to, że dopiero podczas publikacji mamy możliwość zaobserwowania tego, jak aplikacja się zachowuje w środowisku produkcyjnym, z realistycznymi obciążeniami i produkcyjnymi zestawami danych*. W wielu przypadkach zespoły deweloperskie proszą o stworzenie środowisk testowych w początkowych fazach projektu.

Jednak w przypadku, gdy cykle realizacji wymagane do dostarczenia środowisk testowych przez dział operacji są długie, to zespoły mogą nie otrzymać ich wystarczająco wcześnie, by przeprowadzić odpowiednie testy. Co gorsza, środowiska testowe często są nieprawidłowo skonfigurowane lub są tak różne od środowisk produkcyjnych, że zamiast wykonywać testy przed wdrożeniem, borykamy się z olbrzymimi problemami produkcji.

W tym kroku chcemy, aby programiści mogli uruchomić środowiska zbliżone do produkcyjnych na własnych stacjach roboczych, by były one utworzone na żądanie i samoobsługowe. Dzięki temu programiści mogą uruchamiać i testować kod w środowiskach produkcyjnych w ramach codziennej pracy oraz uzyskiwać wcześnie i ciągle informacje na temat jakości swojej pracy.

Zamiast ograniczać się do udokumentowania specyfikacji środowiska produkcyjnego w dokumencie lub na stronie wiki, tworzymy wspólny mechanizm budowy, który tworzy wszystkie potrzebne środowiska — na przykład programistyczne, testowe i produkcyjne. Dzięki temu każdy, kto chce stworzyć środowisko podobne do produkcyjnego, może to zrobić w ciągu kilku minut, bez otwierania zlecenia roboczego, nie mówiąc już o konieczności oczekiwania wielu tygodni†.

W tym celu trzeba zdefiniować i zautomatyzować tworzenie znanych, dobrych środowisk, które są stabilne, bezpieczne i stwarzające minimalne zagrożenia, zgodnie ze zbiorową wiedzą w całej organizacji. Wszystkie wymagania są osadzone nie w dokumentach lub w postaci wiedzy w czyjeś głowie, ale skodyfikowane w zautomatyzowanym procesie budowy środowiska.

* W tym kontekście środowisko jest rozumiane jako wszystko, co należy do stosu aplikacji, włącznie z bazami danych, systemami operacyjnymi, sieciami, wirtualizacją, a także odpowiednimi konfiguracjami.

† Większość programistów chce przetestować swój kod. Często robią oni wszystko, aby uzyskać potrzebne do tego środowiska testowe. Programiści posuwają się do używania starych środowisk testowych z poprzednich projektów (często wieloletnich) lub zwracają się o nie z prośbą do kogoś, kto potrafi takie środowisko stworzyć — nie proszą o nie tam, skąd ono pochodzi, bo zawsze komuś gdzieś brakuje serwera.

Zamiast zwracać się do działu operacji o ręczne stworzenie i skonfigurowanie środowiska, możemy użyć automatyzacji dla dowolnej lub dla wszystkich spośród następujących czynności:

- Skopiowanie zwirtualizowanego środowiska (np. obrazu VMware, działającego skryptu Vagrant, załadowanie pliku Amazon Machine Image w EC2).
- Zbudowanie zautomatyzowanego procesu tworzenia środowiska, rozpoczynającego się od „gołego metalu” (np. zainstalowanie PXE z obrazu bazowego).
- Skorzystanie z narzędzi zarządzania konfiguracją typu „infrastruktura jako kod” (np. Puppet, Chef, Ansible, Salt, CFEngine itp.).
- Wykorzystanie automatycznego narzędzia konfiguracji systemu operacyjnego (np. Jumpstart w systemie Solaris, Kickstart w Red Hat oraz preseed w Debianie).
- Stworzenie środowiska z zestawu wirtualnych obrazów lub kontenerów (np. Vagrant, Docker).
- Uruchomienie nowego środowiska w chmurze publicznej (np. Amazon Web Services, Google App Engine, Microsoft Azure), chmurze prywatnej lub innych systemów PaaS (platforma jako usługa), takich jak OpenStack, Cloud Foundry itp.

Dzięki starannemu zdefiniowaniuawczasu wszystkich aspektów środowiska nie tylko możemy stworzyć nowe środowiska szybko, ale również zapewnić, że środowiska te będą stabilne, niezawodne, spójne i bezpieczne. To przynosi korzyści wszystkim.

Dział operacji korzysta, ponieważ tworzy środowiska szybko, a automatyzacja procesu tworzenia środowisk zapewnia spójność i zmniejsza ilość żmudnej, ręcznej pracy, stwarzającej możliwość popełnienia błędów. Ponadto korzyści uzyskuje dział Dev, ponieważ może odtworzyć wszystkie niezbędne elementy środowiska produkcyjnego w celu budowania, uruchamiania i testowania kodu na swoich stacjach roboczych. W ten sposób zapewniamy programistom możliwość znalezienia i rozwiązania wielu problemów nawet w najwcześniejszych fazach projektu, a nie dopiero podczas testowania integracyjnego lub co gorsza, w produkcji.

Dzięki dostarczeniu programistom środowiska, które w pełni kontrolują, umożliwiamy im szybkie odtworzenie, zdiagnozowanie i naprawienie defektów w środowisku bezpiecznie odizolowanym od usług produkcyjnych oraz innych współdzielonych zasobów. Mają oni także możliwość eksperymentowania ze zmianami w środowiskach, a także w kodzie infrastruktury, która je utworzyła (np. skryptach zarządzania konfiguracją), co z kolei przyczynia się do tworzenia wspólnej wiedzy pomiędzy działami Dev a Ops*.

* W idealnej sytuacji błędy powinny być znajdowane przed testowaniem integracyjnym. Faza testów integracyjnych następuje zbyt późno, by można było tworzyć szybkie informacje zwrotne dla programistów. Jeśli nie możemy tego osiągnąć, to prawdopodobnie w architekturze występuje problem, który należy rozwiązać. Projektowanie systemów pod kątem możliwości

TWORZENIE JEDNEGO REPOZYTORIUM PRAWDY DLA CAŁEGO SYSTEMU

W poprzednim kroku pokazaliśmy sposób stworzenia na żądanie środowisk programistycznych, testowych i produkcyjnych. Teraz musimy zadbać o zmontowanie wszystkich części systemu oprogramowania.

Z biegiem dziesięcioleci upowszechniła się praktyka stosowania systemów kontroli wersji zarówno wśród indywidualnych programistów, jak i zespołów*. System kontroli wersji rejestruje zmiany w plikach lub zestawach plików zapisanych w systemie. Mogą to być kod źródłowy, zasoby lub inne dokumenty, które mogą być częścią projektu oprogramowania. Zmiany wprowadzamy w grupach zwanych **rewizjami** (ang. *commit*). Każda rewizja wraz z metadanymi opisującymi to, kto wprowadził zmianę i kiedy, jest przechowywana w systemie. Dzięki temu można wgrywać, porównywać, scalać i odtwarzać wcześniejsze rewizje. Stosowanie systemu kontroli wersji zmniejsza również zagrożenia dla projektu, ponieważ pozwala przywracać obiekty produkcyjne do wcześniejszych wersji (niżej wymienione terminy będą stosowane w tej książce zamienne: „wgranie do kontroli wersji”, „zatwierdzenie zmian w systemie kontroli wersji”, „wgranie rewizji”, „rewizja”).

Gdy programiści umieszczą wszystkie pliki źródłowe aplikacji i pliki konfiguracyjne w systemie kontroli wersji, staje się on pojedynczym repozytorium prawdy zawierającym precyzyjny, zamierzony stan systemu. Ponieważ jednak dostarczanie wartości do klienta wymaga zarówno kodu, jak i środowisk, w których ten kod działa, to środowiska również powinny być przechowywane w systemie kontroli wersji. Innymi słowy, system kontroli wersji powinien służyć wszystkim osobom w strumieniu wartości, włącznie z inżynierami weryfikacji, operacjami oraz Infosec. Dzięki umieszczeniu wszystkich artefaktów produkcji w repozytorium kontroli wersji możemy wielokrotnie i wiarygodnie odtwarzać wszystkie elementy działającego systemu oprogramowania — dotyczy to aplikacji i środowiska produkcji, a także wszystkich środowisk przedprodukcyjnych.

Aby zapewnić możliwość przywrócenia usług produkcyjnych w sposób wielokrotny i przewidywalny (a najlepiej szybki) nawet wtedy, gdy wystąpią katastrofalne zdarzenia, należy sprawdzać następujące zasoby współdzielonego repozytorium kontroli wersji:

testowania, tak by zapewnić możliwości wykrywania większości defektów za pomocą środowiska wirtualnego na stacji deweloperskiej, to kluczowy element architektury zapewniającej szybki przepływ i informacje zwrotne.

* Pierwszym systemem kontroli wersji był prawdopodobnie UPDATE na komputerze CDC6600 (1969). Później był system SCCS (1972), CMS na komputerach VMS (1978), RCS (1982) itd.

- Kod aplikacji wraz z zależnościami (tzn. bibliotekami, zawartością statyczną itp.).
- Skrypty używane do tworzenia schematów bazy danych, dane referencyjne aplikacji itp.
- Wszystkie narzędzia do tworzenia środowiska i artefakty opisane w poprzednim kroku (np. obrazy VMware lub AMI, receptury Puppet lub Chef itp.).
- Wszystkie pliki używane do tworzenia kontenerów (np. pliki definicji lub kompozycji Docker lub Rocket).
- Wszelkie pomocnicze skrypty testów automatycznych i ręcznych.
- Skrypty do tworzenia pakietów kodu, instalacji, migrowania bazy danych oraz tworzenia środowiska.
- Wszystkie artefakty projektu (np. dokumentacja wymagań, procedury wdrażania, informacje o wydaniu itp.).
- Pliki konfiguracyjne chmury (np. szablony AWS Cloudformation, pliki Microsoft Azure Stack DSC, OpenStack HEAT).
- Inne informacje w postaci skryptu lub konfiguracji potrzebne do stworzenia infrastruktury, która wspiera wiele usług (np. korporacyjne magistrale usług, systemy zarządzania bazami danych, pliki stref DNS, reguły konfiguracji zapór firewall i innych urządzeń sieciowych)*.

Możemy stworzyć wiele repozytoriów dla różnych typów obiektów i usług oznaczanych etykietami lub tagami i wykorzystywanych razem z kodem źródłowym. Przykładowo w repozytoriach artefaktów (np. Nexus, Artifactory) możemy przechowywać duże obrazy maszyn wirtualnych, pliki ISO, skompilowane binaria itd. Alternatywnie można umieścić je w magazynach typu blob (np. Amazon S3), umieścić obrazy Docker w rejestrach Docker itd.

Nie wystarczy zadbać o możliwość odtworzenia dowolnego poprzedniego stanu środowiska produkcyjnego. Trzeba również mieć możliwość ponownego wykonania wszystkich procesów przedprodukcyjnych i procesów budowania. W związku z tym w repozytorium kontroli wersji należy umieścić wszystko, co jest wykorzystywane w procesach budowania, włącznie z narzędziami (tzn. kompilatorami i narzędziami do testowania), a także środowiskami, od których one zależą[†].

* Można zauważyc, że repozytoria kontroli wersji dzięki inwentaryzacji wszystkiego, co jest potrzebne do odtworzenia środowiska produkcyjnego, mają niektóre cechy konstrukcji ITIL, takie jak biblioteka **DML** (ang. *Definitive Media Library*) oraz baza danych zarządzania konfiguracją (ang. *Configuration Management Database — CMDB*).

[†] Na kolejnych etapach transformacji będziemy umieszczać w repozytorium kontroli wersji także całą pomocniczą infrastrukturę — na przykład zestawy testów automatycznych oraz mechanizmy ciągłej integracji i potoku wdrażania.

W raporcie *State of DevOps Report* z 2014 roku wygenerowanym w Puppet Labs wskazano użycie systemów kontroli wersji przez działy Ops jako najwyższy wskaźnik zarówno wydajności działu IT, jak i całej organizacji. W rzeczywistości użycie repozytoriów kontroli wersji przez działy Ops wskazano jako wyższy predyktor dla wydajności IT i wydajności organizacyjnej niż użycie repozytoriów kontroli wersji przez działy Dev.

Wnioski z raportu *State of DevOps Report* z 2014 r. w firmie Puppet Labs podkreślają kluczową rolę repozytoriów kontroli wersji w procesie rozwoju oprogramowania. Teraz wiemy, że rejestrowanie w systemie kontroli wersji wszystkich zmian w aplikacji i środowisku nie tylko pozwala nam szybko dostrzec wszystkie modyfikacje, które mogły przyczynić się do powstania problemu, ale również zapewnia możliwość cofnięcia się do poprzedniego znanego, działającego stanu, co umożliwia szybsze zlikwidowanie skutków awarii.

Dlaczego jednak stosowanie repozytoriów kontroli wersji dla środowisk jest lepszym predyktorem wydajności IT i organizacji niż używanie ich w odniesieniu do kodu?

Ponieważ w prawie wszystkich przypadkach istnieje znacznie więcej konfigurowalnych ustawień w środowisku niż w kodzie. W związku z tym to środowisko najbardziej wymaga kontroli wersji*.

Repozytoria kontroli wersji są również mechanizmem komunikacji dla wszystkich pracujących w strumieniu wartości. Możliwość wzajemnej obserwacji zmian wprowadzanych przez programistów, inżynierów validacji i inżynierów operacyjnych pomaga zminimalizować niespodzianki, poprawia widoczność pracy innych oraz pomaga budować i wzmacniać zaufanie (dodatek 7.).

TWORZENIE INFRASTRUKTURY, KTÓRĄ ŁATWIEJ ODBUDOWAĆ, NIŻ NAPRAWIĆ

Jeśli zarówno aplikację, jak i środowisko można szybko odbudować i ponownie utworzyć na żądanie, to można je również szybko odbudować, zamiast naprawiać w przypadku awarii. Mimo że technika ta jest powszechnie stosowana w konfiguracjach webowych dużej skali (np. kilka tysięcy serwerów), to można zastosować ją nawet wtedy, gdy mamy tylko jeden serwer produkcyjny.

Bill Baker, uznany inżynier w firmie Microsoft, zażartował, że serwery dawniej były traktowane jak zwierzęta domowe: „Nadawano im imiona, a kiedy zachorowały,

* Każdy, kto przeprowadzał migrację kodu z systemu ERP (np. SAP, Oracle Financials itd.), być może rozpoznaje następującą sytuację: migracja kodu bardzo rzadko zawodzi z powodu błędu kodowania. Znacznie częściej przyczyną niepowodzenia są pewne różnice w środowiskach — na przykład różnice pomiędzy środowiskiem programistów a środowiskiem validacji albo pomiędzy środowiskiem validacji a produkcyjnym.

były poddawane kuracji, aby powróciły do zdrowia. [Obecnie] serwery są [traktowane] jak bydło. Nadajemy im numery, a gdy zachorują, zabijamy je”.

Dzięki dostępności powtarzalnego środowiska tworzenia systemów możemy łatwo zwiększyć możliwości poprzez dodanie większej liczby serwerów (tzw. skalowanie poziome). Możemy także uniknąć katastrofy, która jest nieuchronna w przypadku konieczności przywrócenia usługi po poważnej awarii infrastruktury, która nie pozwala się odtworzyć, a która powstawała przez lata nieudokumentowanych i wprowadzanych ręcznie zmian w produkcji.

Aby zapewnić spójność środowisk, wprowadzone zmiany produkcyjne (zmiany w konfiguracji, instalacji poprawek, uaktualnień itp.) zawsze powinny być replikowane we wszystkich środowiskach, zarówno produkcyjnych, jak i przedprodukcyjnych, a także we wszystkich środowiskach tworzonych na nowo.

Zamiast ręcznie logować się na serwerach i wprowadzać zmiany, należy wprowadzać je w sposób zapewniający ich automatyczną replikację oraz zadbać o to, aby wszystkie zmiany zostały wprowadzane w systemie kontroli wersji.

W celu zapewnienia spójności można skorzystać z automatycznych systemów konfiguracji (np. Puppet, Chef, Ansible, Salt, Bosh itp.). Możemy także stworzyć nowe maszyny wirtualne lub kontenery i wdrożyć je do produkcji za pomocą automatycznego mechanizmu budowania, nisząc stare maszyny lub poddając je rotacji*.

Drugi wzorzec jest znany jako infrastruktura niezmienna. Ręczne zmiany w środowisku produkcyjnym nie są już dozwolone. Jedynym sposobem wprowadzenia zmian produkcyjnych jest wprowadzenie zmian do repozytorium kontroli wersji, a następnie odtworzenie kodu i środowisk od podstaw. Dzięki temu nie ma sposobu, aby środowiska produkcyjne były zróżnicowane.

Aby zapobiec niekontrolowanym odchyleniom konfiguracji, możemy wyłączyć zdalne logowanie do serwerów produkcyjnych[†] lub rutynowo zabijać i zastępować egzemplarze w produkcji, tak by wprowadzone ręcznie zmiany na serwerach produkcyjnych były usuwane. Takie działania motywują wszystkich do prawidłowego wprowadzania swoich zmian za pośrednictwem systemu kontroli wersji. Dzięki zastosowaniu takich środków systematycznie eliminujemy możliwości istnienia odstępstw infrastruktury (np. różnice w konfiguracji, kruche artefakty, tzw. „dzieła sztuki”, „płatki śniegu” itd.).

Ponadto musimy zadbać o aktualność środowisk przedprodukcyjnych. W szczególności powinniśmy dbać o to, aby programiści korzystali z najbardziej aktualnego środowiska. Programiści często dążą do korzystania ze starszych środowisk, ponieważ

* W Netflix średni wiek egzemplarza Netflix AWS wynosi 24 dni, a 60% ma mniej niż tydzień.

† Można też zezwalać na takie zmiany tylko w nagłych przypadkach, jednocześnie dbając o to, aby kopia logu konsoli była automatycznie wysyłana pocztą elektroniczną do działu operacji.

obawiają się, że aktualizacje środowiska mogą doprowadzić do zakłócenia działania istniejących funkcji. Należy je jednak często aktualizować, tak aby problemy były znajdowane w jak najwcześniejszej fazie cyklu życia systemu^{*}.

ZMODYFIKUJ DEFINICJĘ STANU „GOTOWY”. UWZGLĘDNIJ W NIM URUCHOMIENIE W ŚRODOWISKU ZBLIŻONYM DO PRODUKCYJNEGO

Teraz, gdy możemy tworzyć środowiska na żądanie, a wszystkie zmiany są zaewidencjonowane w repozytorium kontroli wersji, naszym celem powinno być umożliwienie użycia tych środowisk w codziennej pracy działu Dev. Trzeba sprawdzić, czy aplikacja działa zgodnie z oczekiwaniami w środowisku produkcyjnym na długo przed zakończeniem projektu lub pierwszym wdrożeniem do produkcji.

Większość nowoczesnych metod wytwarzania oprogramowania zaleca krótkie i iteracyjne interwały rozwoju, w przeciwieństwie do podejścia **big bang** (tak jak w modelu kaskadowym). Ogólnie rzecz biorąc, im dłuższe interwały pomiędzy wdrożeniami, tym gorsze wyniki. Na przykład w metodologii Scrum stosowanym interwałem rozwoju jest **sprint** (zwykle trwający miesiąc lub mniej), w ramach którego trzeba uzyskać ostateczny efekt, powszechnie definiowany jako „kod działający, potencjalnie nadający się do publikacji”.

Celem powinno być zapewnienie stanu, w którym w trakcie trwania projektu działa Dev i QA rutynowo integrują kod w środowiskach przypominających produkcyjne w coraz częstszych interwałach[†]. Robimy to przez rozszerzenie definicji „gotowy”. Nie oznacza on już wyłącznie zaimplementowania funkcjonalności kodu (dodatek oznaczono pogrubieniem): na końcu każdego interwału mamy zintegrowany, przetestowany, działający i potencjalnie nadający się do publikacji kod **zademonstrowany w środowisku zbliżonym do produkcyjnego**.

Innymi słowy, akceptujemy zadania programistyczne jako wykonane tylko wtedy, gdy zostały pomyślnie zbudowane, wdrożone i gdy potwierdzono ich działanie zgodne z oczekiwaniami w środowisku zbliżonym do produkcyjnego, a nie wtedy, gdy programista uznaje kod za zaimplementowany. W idealnej sytuacji kod powinien działać pod obciążeniem zbliżonym do produkcyjnego, na produkcyjnym zestawie danych,

* Cały stos aplikacji i środowisko mogą być spakowane do kontenerów. Pozwala to zapewnić maksymalną prostotę instalacji i poprawić jej szybkość w całym potoku wdrażania.

† Termin „integracja” ma wiele różnych znaczeń wśród pracowników działów Dev i Ops. W dziale Dev integracja zwykle dotyczy kodu — tzn. jest to scalenie wielu gałęzi kodu w repozytorium kontroli wersji. W warunkach ciągłego dostarczania i DevOps testowanie integracyjne odnosi się do testowania aplikacji w środowisku zbliżonym do produkcyjnego lub w zintegrowanym środowisku testów.

na długo przed zakończeniem sprintu. Zapobiega to sytuacji, w której określona funkcjonalność jest nazywana „gotową” tylko z tego powodu, że programiście udało się ją pomyślnie uruchomić na swoim laptopie, ale nigdzie indziej.

Dzięki umożliwieniu programistom pisania, testowania i uruchamiania kodu w środowisku zbliżonym do produkcyjnego większość pracy potrzebnej do pomyślnego zintegrowania kodu i środowisk jest wykonywana podczas codziennej pracy, a nie na końcu wydania. Już na końcu pierwszego interwału można zademonstrować poprawne działanie aplikacji w środowisku przypominającym produkcyjne, z kodem i środowiskiem, które zostało zintegrowane już wiele razy. W idealnej sytuacji wszystkie kroki powinny być zautomatyzowane (bez konieczności ręcznej interwencji).

Co więcej, na końcu projektu kod jest pomyślnie wdrożony i uruchomiony w środowiskach zbliżonych do produkcyjnych setki lub nawet tysiące razy. To daje pewność, że większość problemów wykryto i rozwiązano.

W idealnej sytuacji w środowiskach przedprodukcyjnych powinny być używane te same narzędzia monitorowania, rejestrowania i wdrażania co w środowiskach produkcji. Dzięki temu zdobywamy wiedzę i doświadczenie, które pomogą nam sprawnie wdrożyć i uruchomić produkt, a także diagnozować i rozwiązywać problemy usługi, gdy już będzie w produkcji.

Poprzez umożliwienie działom Dev i Ops uzyskania wspólnej wiedzy na temat interakcji kodu i środowiska oraz dzięki wczesnemu i częstemu ćwiczeniu wdrożeń, możemy znacznie zmniejszyć ryzyko związane z wdrażaniem kodu produkcyjnego. Pozwala to nam również wyeliminować całą klasę defektów operacyjnych i wad zabezpieczeń, a także problemów architektonicznych, które zwykle są wykrywane podczas projektu, zbyt późno, aby można je było rozwiązać.

PODSUMOWANIE

Aby było możliwe osiągnięcie szybkiego przepływu pracy od działu Dev do Ops, trzeba zadbać o to, aby wszyscy mogli na żądanie uzyskać środowiska zbliżone do produkcyjnych. Przez umożliwienie programistom używania środowisk zbliżonych do produkcyjnych w najwcześniejjszych fazach projektu oprogramowania możemy znacznie zmniejszyć ryzyko wystąpienia problemów w fazie produkcji. Jest to jedna z wielu praktyk, które dowodzą poprawy wydajności programistów dzięki współpracy z działem operacji. Wymuszamy stosowanie przez programistów praktyki uruchamiania kodu w silnikach środowiskowych zbliżonych do produkcyjnych poprzez włączenie tej czynności do definicji pojęcia „kod gotowy”.

Ponadto poprzez umieszczenie wszystkich artefaktów produkcji w repozytorium kontroli wersji zyskujemy „pojedyncze źródło prawdy”, które pozwala nam odtworzyć całe środowisko produkcyjne w szybki, powtarzalny i udokumentowany sposób, stosując dla artefaktów działu Ops te same praktyki, jakie są stosowane dla zadań Dev.

Dzięki temu, że infrastruktura produkcji jest łatwiejsza do odbudowania niż naprawienia, rozwiązywanie problemów jest łatwiejsze i szybsze. Łatwiejsze jest również rozwijanie możliwości.

Stosowanie tych praktyk ustawia scenę dla kompleksowej automatyzacji testów. Tym tematem zajmiemy się w następnym rozdziale.

10

Szybkie i niezawodne testowanie automatyczne

W tym momencie działały Dev i QA używają środowisk zbliżonych do produkcyjnych w codziennej pracy. Udało się nam pomyślnie zintegrować i uruchomić kod w środowisku podobnym do produkcyjnego dla wszystkich zaakceptowanych funkcjonalności, a wszystkie zmiany zostały zaewidencjonowane w repozytorium kontroli wersji. Jeśli jednak wyszukiwaniem i naprawianiem błędów zajmiemy się w oddzielnej fazie testowania, realizowanej przez oddzielnego dział walidacji (QA) po zakończeniu wszystkich prac programistycznych, to prawdopodobnie uzyskamy niepożądane efekty. Co więcej, jeśli testowanie będzie wykonywane tylko kilka razy w roku, to programiści dowiedzą się o popełnionych błędach kilka miesięcy po wprowadzeniu zmiany, która spowodowała błąd. W tym momencie istnieje duże prawdopodobieństwo zatarcia się związku pomiędzy przyczyną a skutkiem. Rozwiążanie problemu wymaga „gaszenia pożarów” i „archeologii”, a co najgorsze, możliwości uczenia się na błędach i integracji zdobytej w ten sposób wiedzy do realizacji przyszłych zadań są znacznie mniejsze.

Testowanie automatyczne rozwiązuje także inny znaczący i niepokojący problem. Gary Gruver zauważył, że „bez testowania automatycznego, im więcej kodu piszemy, tym więcej czasu i pieniędzy potrzebujemy do przetestowania kodu — w większości przypadków, w dowolnej organizacji technicznej, jest to całkowicie nieskalowalny model biznesowy”.

Chociaż bez wątpienia dziś firma Google jest przykładem kultury, która promuje testy zautomatyzowane na dużą skalę, to nie zawsze tak było. W 2005 roku, kiedy

w Google rozpoczął pracę Mike Bland, instalacje w firmie często stwarzały problemy, zwłaszcza dla zespołu Google Web Server (GWS).

Jak wyjaśnia Bland: „W połowie pierwszego dziesięciolecia XXI w. zespół GWS znajął się w sytuacji, gdy wprowadzanie zmian na serwerze WWW, aplikacji C++ obsługującej wszystkie żądania do strony głównej serwisu Google, a także w wielu innych stronach internetowych serwisu było bardzo trudne. Choć zespół GWS był ważny i prominentny, to znalezienie się w nim nie było powodem do zazdrości — często zespół ten był »wysypiskiem« dla różnych zespołów opracowujących różne funkcje wyszukiwania. Wszystkie te zespoły pisały kod niezależnie od siebie. Występowały różne problemy. Kompilacje i testy zajmowały zbyt dużo czasu, kod był wdrażany do produkcji bez odpowiednich testów, a zespoły niezbyt często wgrywały do repozytoriów duże zmiany, które kolidowały ze zmianami wprowadzanymi przez inne zespoły”.

Konsekwencje tej sytuacji były poważne — zdarzało się, że wyniki wyszukiwania zawierały błędy albo działały bardzo wolno, co miało wpływ na tysiące kwerend wyszukiwania w witrynie Google.com. Mogło to prowadzić nie tylko do utraty przychodów, ale także zaufania klientów.

Bland opisuje, w jaki sposób sytuacja ta wpływała na programistów wprowadzających zmiany: „Strach stał się zabójcą myślenia. Nowi członkowie zespołu bali się cokolwiek zmieniać, ponieważ nie rozumieli systemu. Podobne obawy przed zmianami mieli także doświadczeni członkowie zespołu, ponieważ rozumieli system zbyt dobrze*”. Bland należał do grupy osób zdeterminowanych do podjęcia walki z występującym problemem.

Lider zespołu GWS, Bharat Mediratta, uważały, że problemy mogą zostać rozwiązane dzięki wprowadzeniu testów automatycznych. Jak opisuje Bland: „Stworzono twardy wymóg — żadne zmiany w kodzie zespołu GWS nie będą zaakceptowane, jeśli nie będą im towarzyszyć testy automatyczne. Skonfigurowano środowisko ciąglej komplikacji i religijnie dążono do tego, by wszystkie testy przechodziły. Wprowadzono monitorowanie pokrycia testami i dbano o to, aby z czasem poziom pokrycia testami się podniósł. Napisano przewodniki i strategie testowania i wymagano, aby były przestrzegane zarówno przez programistów należących do zespołu, jak i spoza zespołu”.

Wyniki były zaskakujące. Jak zauważał Bland: „GWS szybko stał się jednym z najbardziej produktywnych zespołów w firmie. Co tydzień integrowano wiele zmian z różnych zespołów, a jednocześnie utrzymywano harmonogram częstych wydań.

* Bland pisał, że jedną z konsekwencji dysponowania tak wieloma utalentowanymi programistami w Google był tzw. „syndrom oszusta” — termin ukuty przez psychologów, nieformalnie opisujący ludzi, którzy nie wierzą we własne osiągnięcia. Według Wikipedii „pomimo zewnętrznych dowodów własnej kompetencji osoby cierpiące z powodu tego syndromu pozostają przekonane, że są oszustami i nie zasługują na sukces, który osiągnęły. Przyczyn sukcesu upatrują w szczęściu, sprzyjających okolicznościach bądź w rezultacie bycia postrzeganym jako osoba bardziej inteligentna i kompetentna niż w rzeczywistości”.

Dzięki dobremu pokryciu kodu testami i dobremu stanowi kodu nowi członkowie zespołu potrafili skutecznie wprowadzać zmiany w tym skomplikowanym systemie. Ostatecznie stosowanie tej radykalnej polityki pozwoliło stronie głównej serwisu Google.com szybko rozszerzyć swoje możliwości i rozwijać się w niezwykle dynamicznym i konkurencyjnym krajobrazie technologii”.

Ale GWS nadal pozostawał stosunkowo małym zespołem w dużej, rozwijającej się firmie i chciał rozszerzyć nowe praktyki na całą organizację. W związku z tym powstał zespół Testing Grouplet — nieformalna grupa inżynierów, którzy chcieli promować praktyki testów automatycznych na całą organizację. W ciągu najbliższych pięciu lat udało się im replikować kulturę testowania automatycznego w całej firmie Google^{*}.

Teraz, gdy jakiś programista Google ewidencjonuje kod w repozytorium kontroli wersji, automatycznie uruchamiany jest zestaw kilkuset spośród wielu tysięcy testów automatycznych. Jeżeli testy przechodzą, kod jest automatycznie scalany z pniem repozytorium (ang. *trunk*) i jest gotowy do wdrożenia do produkcji. Wiele właściwości Google jest budowanych co kilka godzin lub codziennie. Następnie wybierane są komplikacje do wydania. W innych przypadkach stosowana jest filozofia dostarczania „push on green” (dosł. „pchaj, jeśli zielono”[†]).

Stawka jest wyższa niż kiedykolwiek — błąd wdrażania kodu w Google może spowodować awarię wszystkich właściwości w tym samym czasie (tak jak w przypadku zmiany globalnej infrastruktury lub gdy powstanie defekt w jednej z głównych bibliotek, od której zależą wszystkie właściwości).

Eran Messeri, inżynier w grupie Google Developer Infrastructure, zauważyl: „Duże awarie od czasu do czasu się zdarzają. Dostajesz mnóstwo wiadomości na komunikatorze, a inżynierowie pukają do Twoich drzwi. [Gdy potok wdrożeń ulegnie awarii], trzeba natychmiast usunąć problem, ponieważ programiści nie mogą ewidencjonować kodu w repozytorium. W związku z tym trzeba dbać o to, aby zmiany można było łatwo wycofać”.

W Google ten system może działać dzięki profesjonalizmowi inżynierów i kulturze wysokiego zaufania, która zakłada, że wszyscy chcą robić dobrą robotę. Ważna jest także zdolność szybkiego wykrywania i korygowania problemów. Messeri wyjaśnia: „W Google nie ma twardych reguł w rodzaju: »Jeśli spowodujesz awarię w produkcji dla więcej niż dziesięciu projektów, musisz spełnić kontrakt SLA — rozwiązać problem w ciągu dziesięciu minut«. Zamiast tego jest wzajemny szacunek między zespołami

* Stworzono programy szkoleniowe, zaczęto publikować słynny cykl Testing w newsletterze „Toilet” (umieszczanym w łazienkach). Stworzono mapę drogową i program certyfikacji Test Certified oraz zorganizowano wiele kampanii „fix-it” (tzn. kampanii wprowadzania poprawek), co pomogło zespołom usprawnić automatyczne procesy testowania. Dzięki temu udało się replikować doskonałe wyniki zespołu GWS.

† Chodzi o ewidencjonowanie kodu w repozytorium tylko wtedy, gdy przejdzie testy — przyp. tłum.

i niejawne porozumienie, że każdy robi wszystko, co może, aby zachować działanie potoku wdrożeń. Wiem, że pewnego dnia być może przypadkowo uszkodzę czyjś projekt. Następnego dnia ktoś może uszkodzić mój projekt”.

Dzięki osiągnięciom Mike'a Blanda i zespołu Testing Grouplet firma Google stała się jedną z najbardziej wydajnych organizacji technologicznych na świecie. Do roku 2013 testowanie automatyczne i ciągła integracja w firmie Google pozwoliła wspólnie pracować i zachować wydajność ponad czterem tysiącom małych zespołów, które jednocześnie zajmowały się rozwijaniem, integracją, testowaniem i wdrażaniem swojego kodu w środowisku produkcyjnym. Cały ich kod jest zapisany w jednym, wspólnym repozytorium, składa się z miliardów plików. Wszystkie są ciągle budowane i integrowane, a każdego miesiąca zmiany są wprowadzane w 50% kodu. Oto niektóre inne imponujące statystyki wydajnościowe:

- 40 000 operacji rejestracji rewizji kodu w repozytorium dziennie;
- 50 000 komplikacji dziennie (w dni powszednie liczba ta może przekraczać 90 000);
- 120 000 zestawów testów automatycznych;
- 75 milionów przypadków testowych uruchomionych dziennie;
- ponad 100 inżynierów (0,5% wszystkich pracowników R&D) pracujących nad inżynierią testów, ciągłą integracją i narzędziami inżynierii publikowania w celu zwiększenia wydajności programistów.

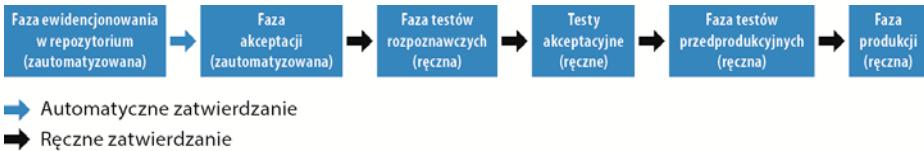
W dalszej części tego rozdziału omówimy praktyki ciągłej integracji potrzebne do replikacji tych wyników.

CIĄGŁE BUDOWANIE, TESTOWANIE I INTEGROWANIE KODU ORAZ ŚRODOWISK

Naszym celem powinno być budowanie jakości produktu od najwcześniejszych faz jego cyklu życia. Aby to było możliwe, należy zapewnić programistom możliwość korzystania z testów automatycznych w ramach ich codziennej pracy. Tworzy to pętlę szybkiego sprzężenia zwrotnego, co pomaga w szybkim znajdowaniu i rozwiązywaniu problemów — gdy istnieje najmniej ograniczeń (np. czasu i zasobów).

W tym kroku pokażemy sposób tworzenia testów automatycznych zwiększających częstotliwość operacji integracji i testowania kodu — z okresowych do wykonywanych w trybie ciągłym. Możemy to zrobić poprzez budowę potoku wdrożeń, którego zadaniem będzie integracja kodu i środowisk oraz zainicjowanie serii testów za każdym razem, gdy w repozytorium kontroli wersji zostaną zaewidencjonowane nowe zmiany* (rysunek 13).

* Dla działu Dev ciągła integracja często dotyczy scalania wielu gałęzi kodu do pnia i zapewnienie przechodzenia testów jednostkowych. Jednak w kontekście ciągłego dostarczania i infrastruktur-



Rysunek 13. Potok wdrożeń (źródło: Humble i Farley, „Continuous Delivery”, s. 3)

Potok wdrożeń (ang. *deployment pipeline*) po raz pierwszy zdefiniowany przez Jęza Humble'a i Davida Farleya w książce *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, ma zapewnić, aby każdy kod zaewidencjonowany w repozytorium kontroli wersji był automatycznie zbudowany i przetestowany w środowisku zbliżonym do produkcyjnego. Stosując tę praktykę, wszelkiego rodzaju błędy komplikacji, testowania lub integracji można znaleźć natychmiast po zaewidencjonowaniu zmiany w repozytorium, co pozwala od razu je naprawić. Dzięki temu można zawsze mieć pewność, że aplikacja jest w stanie gotowym do opublikowania.

Aby to osiągnąć, trzeba stworzyć procedury automatycznej komplikacji i testowania działające w dedykowanych środowiskach. Ma to kluczowe znaczenie z następujących powodów:

- Procesy budowania i testowania mogą działać przez cały czas, niezależnie od nawyków pojedynczych inżynierów.
- Oddzielne procesy komplikacji i testowania dają pewność zrozumienia wszystkich zależności wymaganych do budowania, tworzenia pakietów instalacyjnych, uruchamiania i testowania kodu (tym samym można usunąć problem typu: „działało na laptopie programisty, a przestało działać w produkcji”).
- Można stworzyć pakiet instalacyjny aplikacji, pozwalający na powtarzalne instalowanie kodu i konfiguracji w środowisku produkcyjnym (np. za pomocą narzędzi RPM, yum, npm w systemie Linux lub OneGet w systemie Windows). Alternatywnie można skorzystać z systemów pakietowania specyficznych dla określonego framework'a — na przykład pliki EAR i WAR dla Javy, gemy dla Ruby itp.).
- Zamiast tworzyć pakiety instalacyjne kodu, można skorzystać z możliwości tworzenia kontenerów (np. Docker, Rkt, LXD, AMI).

tury DevOps ciągła integracja powinna uwzględniać także uruchomienie aplikacji w środowiskach zbliżonych do produkcyjnych i uruchomienie testów akceptacyjnych i integracyjnych. Jez Humble i David Farley odróżniają te dwa pojęcia dzięki nadaniu temu drugiemu procesowi nazwy CI+. W tej książce termin „ciągła integracja” będzie zawsze dotyczył praktyk CI+.

- Środowiska można bardziej upodobnić do produkcyjnych. Dzięki temu są spójne i powtarzalne (np. można usunąć ze środowiska kompilatory, wyłączyć flagi debugowania itp.).

Po wprowadzeniu dowolnej zmiany potok wdrożeń weryfikuje, czy kod pomyślnie integruje się w środowisku zbliżonym do produkcyjnego. Staje się on platformą, dzięki której testerzy żądają i poświadczają komplikacje podczas testów akceptacyjnych i testów użyteczności. Dodatkowo uruchamia automatyczne weryfikacje wydajności i zabezpieczeń.

Ponadto potok wdrożeń obsługuje samoobsługowe komplikacje dla środowisk testów UAT (testów akceptacyjnych użytkownika) oraz testów integracji i bezpieczeństwa. Na kolejnych etapach, po rozwinięciu potoku wdrożeń, będzie on służył również do zarządzania wszystkimi działaniami niezbędnymi do przeniesienia zmian z repozytorium kontroli wersji do wdrożenia.

Funkcjonalność potoku wdrożeń jest dostępna za pośrednictwem wielu narzędzi. Duża część z nich ma status open source (np. Jenkins, Go firmy ThoughtWorks, Concourse, Bamboo, Microsoft Team Foundation Server, TeamCity, Gitlab CI). Dostępne są również rozwiązania w chmurze (np. Travis CI i Snap)^{*}.

Potok wdrożeń zaczyna się od fazy ewidencjonowania kodu w repozytorium, w której inicjowana jest komplikacja i tworzenie pakietu oprogramowania, uruchamiane są automatyczne testy jednostkowe oraz wykonywane są dodatkowe weryfikacje — na przykład analiza statycznego kodu, analiza występowania duplikatów i pokrycia testami oraz sprawdzanie stylu[†]. Po pomyślnym zakończeniu tej fazy inicjowana jest faza akceptacji, w której pakiety utworzone w fazie ewidencjonowania są instalowane w środowisku zbliżonym do produkcyjnego oraz uruchamiane są automatyczne testy akceptacyjne.

Po zaakceptowaniu zmian w repozytorium kontroli wersji w całym potoku wdrożeń korzystamy z tych samych pakietów instalacyjnych (ponieważ chcemy, aby tworzenie

* Jeśli w potoku wdrażania utworzymy kontenery i dysponujemy architekturą podobną do mikrousług, to możemy umożliwić każdemu programiście tworzenie niezmienialnych artefaktów — tzn. zapewnić środowisko, w którym programista może stworzyć i uruchomić wszystkie komponenty usługi w środowisku identycznym z produkcyjnym na swojej stacji roboczej. To pozwala programistom tworzyć i uruchamiać więcej testów na swoich stacjach roboczych zamiast na serwerach testowych, co umożliwia jeszcze szybsze sprzężenie zwrotne dotyczące wykonywanej przez nich pracy.

† Możemy nawet wymagać, aby te narzędzia zostały uruchomione przed zaakceptowaniem zmian w repozytorium kontroli wersji (np. za pomocą tzw. haków precommit). Możemy również uruchomić te narzędzia w zintegrowanym środowisku programisty (ang. *integrated development environment* — IDE; w którym autor edytuje, komplikuje i uruchamia kod). W ten sposób sprzężenie zwrotne jest jeszcze szybsze.

pakietów instalacyjnych było wykonywane tylko raz). Dzięki temu kod w zintegrowanym środowisku testowania i środowisku przedprodukcyjnym (ang. *staging*) jest instalowany w taki sam sposób, w jaki jest instalowany w środowisku produkcyjnym. Powoduje to zmniejszenie różnic (np. różnych kompilatorów, flag kompilatorów, wersji bibliotek lub konfiguracji), dzięki czemu można uniknąć trudnych do zdiagnozowania błędów na dalszych etapach strumienia wartości*.

Celem zastosowania potoku wdrożeń jest zapewnienie wszystkim uczestnikom strumienia wartości, zwłaszcza programistom, jak najszybszej informacji o tym, że zmiana spowodowała naruszenie gotowości do wdrożenia. Może to być zmiana w kodzie, zmiana w dowolnym środowisku, testach automatycznych lub nawet w infrastrukturze potoku (np. ustawieniach konfiguracji Jenkins).

W rezultacie infrastruktura potoku wdrożeń staje się równie podstawowa dla procesów rozwoju produktu, jak infrastruktura kontroli wersji. Potok wdrożeń przechowuje również historię wszystkich komplikacji kodu. Są w nim zapisane informacje o testach uruchomionych na każdej komplikacji, przyporządkowaniu komplikacji do środowisk oraz wynikach testów. W połączeniu z informacjami w historii kontroli wersji można szybko ustalić, co spowodowało awarię potoku wdrożeń, oraz prawdopodobnie również to, jak należy naprawić błąd.

Te informacje pomagają także spełnić wymagania dostarczania dowodów dla celów inspekcji i zgodności z przepisami, ponieważ dowody są automatycznie generowane w ramach codziennej pracy.

Teraz, gdy dysponujemy działającą infrastrukturą potoku wdrożeń, trzeba stworzyć praktyki **ciągłej integracji**. Wymagają one zapewnienia trzech możliwości:

- kompleksowego i niezawodnego zbioru automatycznych testów weryfikujących, czy produkt jest w stanie gotowym do wdrożenia;
- zwyczaju „zatrzymania całej linii produkcyjnej” w sytuacji, gdy testy walidacji nie przejdą;
- programistów pracujących „małymi partiami” na pniu repozytorium zamiast utrzymywania przez długi czas gałęzi, w których są rozwijane pojedyncze funkcje.

W kolejnym podrozdziale napiszemy, dlaczego jest potrzebne szybkie i niezawodne testowanie automatyczne oraz w jaki sposób je budować.

* Jako mechanizmu tworzenia pakietów instalacyjnych można również użyć kontenerów takich jak Docker. Kontenery zapewniają możliwość „napisz raz, uruchom gdziekolwiek”. Kontenery są tworzone jako część procesu komplikacji. Mogą być szybko zainstalowane i uruchomione w dowolnym środowisku. Dzięki temu, że w każdym środowisku jest uruchamiany ten sam kontener, łatwiej jest wymusić spójność wszystkich artefaktów komplikacji.

TWORZENIE ZESTAWU SZYBKICH I NIEZAWODNYCH AUTOMATYCZNYCH TESTÓW WERYFIKACYJNYCH

W poprzednim kroku zaczęliśmy tworzenie infrastruktury testowania automatycznego, która potwierdza, że mamy **zieloną komplikację** (tzn. wszystko, co zostało zaewidencjonowane w repozytorium kontroli wersji, buduje się i nadaje się do wdrożenia). Aby podkreślić ważność wykonywania kroku integracji i testowania, zastanówmy się, co by się stało, gdybyśmy wykonywali go tylko okresowo — na przykład podczas procesu komplikacji wykonywanego w nocy.

Przypuśćmy, że mamy zespół dziesięciu programistów. Każdy codziennie ewidencjonuje kod w repozytorium kontroli wersji. Programista wprowadza zmianę, która powoduje uszkodzenie nocnej komplikacji, i testy przestają przechodzić. Jeśli w tym scenariuszu następnego dnia odkryjemy, że nie mamy już „zielonej komplikacji”, to ustalenie, jakie zmiany spowodowały problem, kto je wprowadził i jak je naprawić, zajmie zespołowi programistów kilka minut lub częściej kilka godzin.

Rozważmy gorszy scenariusz. Założymy, że problemu nie spowodowała zmiana w kodzie, ale usterka w środowisku testowym (np. niepoprawne ustawienie konfiguracji). Zespółowi programistów może się wydawać, że problem został rozwiązyany, ponieważ wszystkie testy jednostkowe przechodzą. Tymczasem po następnej nocy okazuje się, że wyniki testów dalej są negatywne.

Problem staje się jeszcze bardziej skomplikowany, ponieważ zespół zaewidencjonował tego dnia w repozytorium kontroli wersji dziesięć dodatkowych zmian. Każda z tych zmian może wprowadzić więcej błędów, które mogą spowodować, że testy automatyczne przestaną przechodzić. To jeszcze bardziej utrudnia skuteczną diagnozę i rozwiązanie problemu.

Krótko mówiąc, powolne i okresowe sprzężenia zwrotne stwarzają wiele kłopotów. Jest to szczególnie uciążliwe w przypadku większych zespołów programistów. Problem staje się jeszcze bardziej uciążliwy, gdy mamy dziesiątki, setki, a nawet tysiące programistów ewidencjonujących każdego dnia swoje zmiany w systemie kontroli wersji. W rezultacie do komplikacji często trafiają błędy, a testy automatyczne nie przechodzą. Zdarza się nawet, że programiści przestają ewidencjonować kod w systemie kontroli wersji („Po co zadawać sobie trud, skoro komplikacje zawierają wady, a testy ciągle nie przechodzą?”). Zamiast tego czekają z integracją swojego kodu do końca projektu. W efekcie dotykają ich wszystkie niepożądane skutki dużych rozmiarów partii, integracji i wdrożeń produkcyjnych w stylu „big bang”*.

W celu zapobieżenia tej sytuacji potrzebujemy szybkich testów automatycznych, które są uruchamiane w środowisku komplikacji i testowania przy zaewidencjonowaniu każdej zmiany w repozytorium kontroli wersji. W ten sposób możemy znaleźć i roz-

* Występowanie właśnie tych problemów doprowadziło do powstania praktyk ciągłej integracji.

wiązać wszelkie problemy natychmiast — tak jak pokazaliśmy na przykładzie zespołu Google Web Server. Dzięki temu możemy zapewnić utrzymanie niewielkich rozmiarów partii oraz stanu ciągłej gotowości do wdrożenia.

Ogólnie rzecz biorąc, testy automatyczne można przypisać do jednej z poniższych kategorii — od najszybszych do najwolniejszych:

- **Testy jednostkowe** — zwykle testują pojedynczą metodę, klasę lub funkcję w izolacji, dając autorowi pewność, że jego kod działa zgodnie z projektem. Z wielu powodów, w tym ze względu na potrzebę utrzymania testów szybkich i bezstanowych, w testach jednostkowych często są „zaślepiane” bazy danych i inne zewnętrzne zależności (np. funkcje zamiast odwoływania się do rzeczywistej bazy danych zwracają statyczne, predefiniowane wartości)*.
- **Testy akceptacyjne** — zazwyczaj testują aplikację jako całość w celu uzyskania pewności, że bardziej wysokopoziomowe funkcjonalności działają zgodnie z przeznaczeniem (np. biznesowe kryteria akceptacji dla historyjek użytkownika, poprawność interfejsu API) oraz że nie zostały wprowadzone błędy regresji (tzn. przestała działać funkcjonalność, która wcześniej działała poprawnie). Humble i Farley definiują różnicę pomiędzy testowaniem jednostkowym a akceptacyjnym w następujący sposób: „Celem testu jednostkowego jest pokazanie, że pojedyncza część aplikacji działa zgodnie z zamierzeniami programisty... Celem testów akceptacyjnych jest udowodnienie, że aplikacja działa w sposób oczekiwany przez klienta, a nie że działa w sposób, w jaki powinna działać zdaniem programisty”. Kiedy komplikacja przejdzie przez testy jednostkowe, potok wdrożenia kieruje ją do uruchomienia testów akceptacyjnych. Następnie każda komplikacja, dla której przejdą testy akceptacyjne, jest zazwyczaj udostępniana do testów ręcznych (np. testy rozpoznawcze, testy interfejsu użytkownika itp.), jak również do testowania integracyjnego.
- **Testy integracji** — testowanie integracyjne jest fazą, w której sprawdzamy, czy aplikacja prawidłowo współdziała z innymi produkcyjnymi aplikacjami i usługami (w przeciwieństwie do wywoływanego „zaślepionych” interfejsów). Jak zaobserwowali Humble i Farley: „Znaczną część pracy w środowisku SIT tworzy wdrażanie nowych wersji poszczególnych aplikacji do czasu, aż wszystkie zaczyną współpracować. W tej sytuacji testem dymnym[†] jest zwykle pełny zestaw

* Istnieje szeroka gama kategorii technik architektonicznych i testowych stosowana w celu obsługi problemu konieczności wejścia do testów z zewnętrznych punktów integracji włącznie z „namiastkami” (ang. *stub*), makietami (ang. *mock*), wirtualizacją usług itp. Problem ten staje się jeszcze ważniejszy w przypadku testów akceptacyjnych i integracyjnych, które znacznie bardziej zależą od stanów zewnętrznych.

† Test dymny (ang. *smoke test*) to zbiór przypadków testowych pokrywających główne funkcjonalności produktu — przyp. tłum.

testów akceptacyjnych uruchomionych dla całej aplikacji". Testy integracyjne są wykonywane dla komplikacji, które przeszły testy jednostkowe i akceptacyjne. Ponieważ testy integracyjne często są kruche, należy dążyć do zminimalizowania ich liczby oraz starać się, aby znaleźć jak najwięcej defektów podczas testów jednostkowych i akceptacyjnych. Możliwość używania podczas testów akceptacyjnych wirtualnych lub symulowanych wersji usług staje się podstawowym wymogiem architektonicznym.

W warunkach presji terminu programiści mogą zaprzestać praktyki pisania testów jednostkowych w swojej codziennej pracy, niezależnie od tego, w jaki sposób zdefiniowaliśmy znaczenie określenia kod „gotowy”. W celu wykrycia tej sytuacji można dokonać pomiaru i uwidoczyć wyniki pokrycia testami (w zależności od liczby klas, wierszy kodu, permutacji itp.). W przypadku, gdy poziom pokrycia testami jednostkowymi spadnie poniżej określonego poziomu (np. gdy testy jednostkowe ma mniej niż 80% wszystkich klas), możemy nawet uznać wynik całego testu weryfikacji za negatywny*.

Martin Fowler zauważał, że ogólnie rzecz biorąc, „dziesięciominutowa komplikacja [wraz z procesem testowania] jest w granicach rozsądku... [Wcześniej] wykonujemy komplikację i uruchamiamy testy, które są bardziej zlokalizowanymi testami jednostkowymi, z całkowicie odciętą bazą danych. Takie testy działają bardzo szybko, bez trudu mieszczą się w limicie dziesięciu minut. Jednak uruchomienie testów jednostkowych nie pozwala znaleźć błędów związanych z interakcjami w wyższej skali — w szczególności dotyczących interakcji z bazą danych. W drugiej fazie komplikacji uruchamiany jest inny zestaw testów [testów akceptacyjnych], które uwzględniają rzeczywistą bazę danych oraz zachowania od końca do końca. Uruchomienie takiego zestawu testów może zajmować do kilku godzin”.

WYKRYWANIE BŁĘDÓW W JAK NAJWCZEŚNIEJSZEJ FAZIE TESTOWANIA ZAUTOMATYZOWANEGO

Specyficznym celem zestawu testów zautomatyzowanych jest znalezienie błędów w jak najwcześniejszej fazie testowania. To dlatego uruchamiamy szybsze testy automatyczne (tzn. testy jednostkowe) przed testami automatycznymi działającymi wolniej (np. testami akceptacyjnymi i integracyjnymi), a oba te rodzaje testów uruchamiamy przed testami ręcznymi.

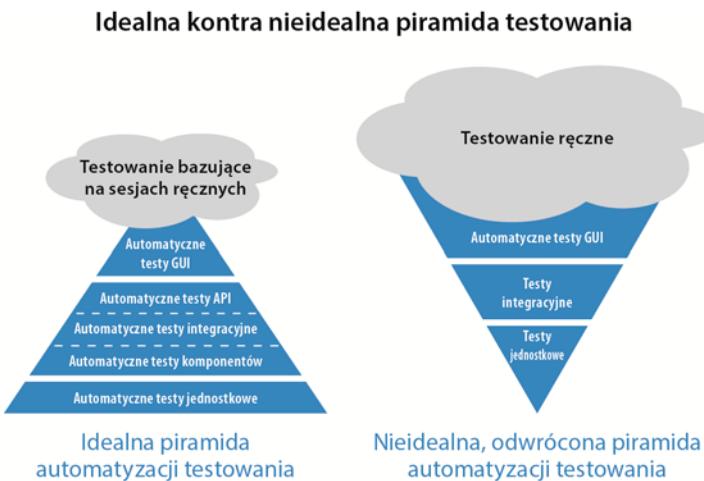
Inną konsekwencją tej zasady jest to, że błędy powinny być znajdowane za pomocą możliwie najszybszej kategorii testowania. Jeśli większość błędów znajdziemy za pomocą testów akceptacyjnych i integracyjnych, to sprzężenie zwrotne, które dotrze do programistów, będzie znacznie wolniejsze niż przy użyciu testów jednostkowych.

* Można to zrobić tylko wtedy, gdy zespoły korzystają w pełni z testowania automatycznego — ten typ parametru jest często wykorzystywany przez programistów i menedżerów.

Ponadto testy integracyjne wymagają użycia limitowanych i złożonych środowisk testowych, które mogą być używane tylko przez jeden zespół naraz, co jeszcze bardziej opóźnia sprzężenie zwrotne.

Podczas testowania integracyjnego wykrywanie błędów jest trudne, a reprodukowanie ich przez programistów czasochłonne. Uciążliwe jest nawet zweryfikowanie, że błędy zostały naprawione (tzn. programista wprowadza poprawkę, a następnie musi czekać kilka godzin, żeby dowiedzieć się, czy testy integracyjne zaczęły przechodzić).

W związku z tym zawsze, gdy znajdziemy błąd za pomocą testu akceptacyjnego lub integracyjnego, powinniśmy stworzyć test jednostkowy, za pomocą którego błąd można znaleźć szybciej, wcześniej i taniej. Martin Fowler opisał pojęcie „idealnej piramidy testowania”, która zapewnia wykrycie większości błędów za pomocą testów jednostkowych (rysunek 14). W przeciwnieństwie do tego stanu w wielu programach testowania jest odwrotnie — najwięcej uwagi poświęca się testom ręcznym i integracyjnym.

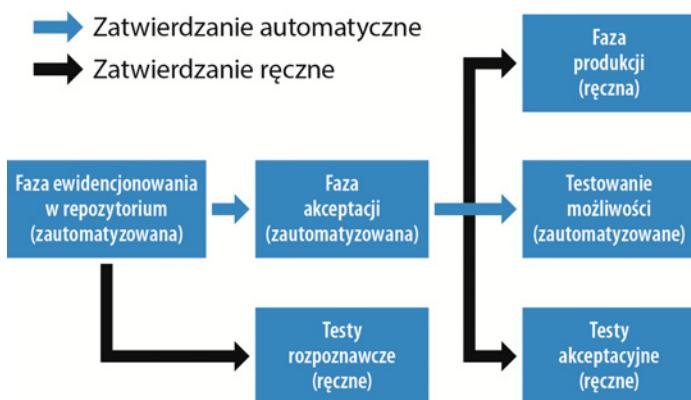


Rysunek 14. Piramidy testowania automatycznego: idealna i nieidealna (źródło: Martin Fowler, „TestPyramid”)

W przypadku stwierdzenia, że testy jednostkowe lub akceptacyjne są zbyt trudne i zbyt kosztowne do pisania i utrzymywania, istnieje prawdopodobieństwo, że architektura jest zbyt ściśle sprzężona, a silna separacja pomiędzy granicami modułów nie istnieje (lub być może nigdy nie istniała). W takim przypadku trzeba zadbać o stworzenie luźniej sprzężonego systemu, w którym moduły mogą być niezależnie testowane bez użycia środowisk integracji. Możliwe jest stworzenie zestawu testów akceptacyjnych, działających w ciągu kilku minut nawet dla najbardziej złożonych aplikacji.

ZADBAJ O SZYBKIE DZIAŁANIE TESTÓW (JEŻELI TO KONIECZNE, WYKORZYSTAJ WSPÓŁBIEŻNOŚĆ)

Ponieważ chcemy, aby testy działały szybko, powinniśmy zaprojektować je do działania współbieżnego, potencjalnie na wielu serwerach. W niektórych przypadkach warto również zadbać o równoległe uruchomienie różnych kategorii testów. Na przykład, kiedy dla komplikacji przejdą testy akceptacyjne, możemy uruchomić testy wydajności równolegle z testami bezpieczeństwa, tak jak pokazano na rysunku 15. Możemy też wykonywać ręczne testy rozpoznawcze równolegle z integracyjnymi, choć w niektórych przypadkach warto poczekać, aż pomyślnie przejdą wszystkie testy automatyczne. Wykonywanie testów ręcznych równolegle z integracyjnymi zapewnia szybsze sprzężenie zwrotne, ale może prowadzić do sytuacji, w której ręcznie testujemy komplikacje obarczone błędami.



Rysunek 15. Współbieżne uruchamianie testów automatycznych i ręcznych (źródło: Humble i Farley, „Continuous Delivery”, wydanie Kindle, lokalizacja 3868)

Kompilacje, które pomyślnie przejdą przez wszystkie testy automatyczne, można udostępnić do ręcznych testów rozpoznawczych, a także do innych rodzajów testów ręcznych oraz takich, które wymagają wielu zasobów (np. testy wydajności). Chcemy, aby wszystkie testy były uruchamiane jak najczęściej, zawsze gdy to możliwe i praktyczne — stale lub zgodnie z harmonogramem.

Każdy tester (w tym wszyscy programiści) powinien używać najnowszej komplikacji, która pomyślnie przeszła wszystkie testy automatyczne, w przeciwieństwie do oczekiwania, aż programiści oznamią określoną komplikację jako gotową do testowania. W ten sposób zapewniamy uruchamianie testów w jak najwcześniejszej fazie procesu.

PISANIE TESTÓW ZAUTOMATYZOWANYCH PRZED ROZPOCZĘCIEM PISANIA KODU (WYTWARZANIE OPROGRAMOWANIA STEROWANE TESTAMI)

Jednym z najskuteczniejszych sposobów uzyskania gwarancji, że dysponujemy wiarygodnymi testami zautomatyzowanymi, jest pisanie tych testów w codziennej pracy. Techniki te są znane jako **wytwarzanie oprogramowania sterowane testami** (ang. *test-driven development — TDD*) oraz **wytwarzanie oprogramowania sterowane testami akceptacyjnymi** (*acceptance test-driven development — ATDD*). Zgodnie z nimi każdą zmianę w systemie zaczynamy od napisania testu automatycznego, który początkowo **nie przechodzi**. Dopiero po napisaniu testu piszemy kod, który sprawia, że test zaczyna przechodzić.

Technika ta została opracowana przez Kenta Becka w latach 90. jako element programowania ekstremalnego. Składają się na nią następujące trzy etapy:

1. Upewnij się, że testy nie przechodzą. „Napisz testy dla fragmentu funkcjonalności, którą chcesz dodać”. Wgraj rewizję kodu do repozytorium.
2. Upewnij się, że testy przechodzą. „Napisz kod funkcjonalności, aż testy zaczną przechodzić”. Wgraj rewizję kodu do repozytorium.
3. „Zrefaktoryzuj zarówno stary, jak i nowy kod, aby uzyskał właściwą strukturę”. Zadbaj o to, aby testy zaczęły przechodzić. Ponownie wgraj rewizję kodu do repozytorium.

Zestawy testów automatycznych są ewidencjonowane w repozytorium kontroli wersji razem z kodem, co zapewnia istnienie dynamicznej, aktualnej specyfikacji systemu. Programiści, którzy chcą zrozumieć, w jaki sposób korzystać z systemu, mogą przeanalizować ten zestaw testów, aby znaleźć działające przykłady użycia interfejsu API systemu*.

ZAUTOMATYZUJ JAK NAJWIĘCEJ TESTÓW RĘCZNYCH

Naszym celem powinno być znalezienie jak największej błędu w kodzie za pośrednictwem zestawów testów automatycznych, co zmniejsza naszą zależność od ręcznego testowania. Elisabeth Hendrickson na konferencji Flowcon 2013 w swojej prezentacji pod tytułem *On the Care and Feeding of Feedback Cycles* zauważała: „Choć można zautomatyzować testy, nie da się zautomatyzować tworzenia jakości. Powierzanie

* Nachi Nagappan, E. Michael Maximilien i Laurie Williams (odpowiednio z firm Microsoft Research, IBM Almaden Labs i North Carolina State University) przeprowadzili badania, które dowiodły, że zespoły używające TDD wytwarzają kod o 60 – 90% lepszy pod względem występowania defektów w porównaniu z zespołami niestosującymi tej techniki, poświęcając na pisanie kodu tylko 15 – 35% więcej czasu.

ludziom uruchamiania testów, które powinny być zautomatyzowane, jest stratą ludzkiego potencjału”.

Jeśli zautomatyzujemy testy, to wszyscy testerzy w strumieniu wartości (w tej liczbie są oczywiście również programiści) będą poświęcali swój czas na wartościowe działania, których nie da się zautomatyzować — takie, jak testy rozpoznawcze lub usprawnienia samego procesu testowania.

Jednak samo zautomatyzowanie wszystkich testów ręcznych może przynosić niepożądane efekty — nie chcemy testów automatycznych, które są niewiarygodne lub generują fałszywe alarmy (np. testy, które powinny przechodzić, ponieważ kod jest funkcjonalnie poprawny, ale nie przechodzą z powodu takich problemów, jak niska wydajność, przekraczanie limitu czasu, niekontrolowany stan początkowy lub niezamierzony stan spowodowany użyciem zaślepeka bazy danych lub współdzielonych środowisk testowych).

Niewiarygodne testy, które generują fałszywe alarmy, stwarzają istotne problemy — powodują marnotrawstwo cennego czasu (np. poprzez zmuszanie programistów do ponownego uruchomienia testu, aby ustalić, czy problem faktycznie występuje), zwiększenie ogólnego nakładu pracy związanego z uruchamianiem i interpretacją wyników testów oraz często wywołuje niepotrzebny stres u programistów, którzy całkowicie ignorują wyniki testów bądź wyłączają testy automatyczne i koncentrują się na tworzeniu kodu.

Efekt zawsze jest ten sam: wykrywamy problemy później, problemy są trudniejsze do rozwiązania, a klienci mają gorsze wyniki, co z kolei powoduje stres w całym strumieniu wartości.

Aby złagodzić tego rodzaju sytuacje, powinniśmy pamiętać, że niewielka liczba wiarygodnych testów automatycznych prawie zawsze jest lepsza od dużej liczby niewiarygodnych testów automatycznych. W związku z tym należy skoncentrować się na automatyzacji tylko tych testów, które rzeczywiście weryfikują cele biznesowe. Jeśli rezygnacja z testów powoduje defekty w produkcji, należy dodać te testy ponownie do zestawu testów ręcznych, a w idealnej sytuacji ostatecznie ponownie je zautomatyzować.

Jak powiedział Gary Gruver, były wiceprezes ds. inżynierii jakości, inżynierii wydań i zadań operacyjnych w Macys.com: „Dla dużej witryny e-commerce przeszliśmy od uruchamiania 1300 testów ręcznych co dziesięć dni do uruchamiania zaledwie dziesięciu testów automatycznych, które uruchamiamy przy każdym ewidencjonowaniu kodu w repozytorium — znacznie lepiej uruchomić kilka testów, którym ufamy, niż wiele testów, które są niewiarygodne. Z biegiem czasu zestaw testów automatycznych rozrosł się do liczby kilkuset tysięcy”.

Innymi słowy, należy zacząć od niewielkiej liczby wiarygodnych testów automatycznych i z czasem je dodawać. Trzeba stale dążyć do uzyskiwania coraz większej pewności, że wszelkie zmiany w systemie, które sprawią, że produkt przestanie być gotowy do wdrożenia, zostaną szybko wykryte.

WŁĄCZENIE TESTOWANIA WYDAJNOŚCI DO ZESTAWU TESTÓW

Często zdarza się, że o niskiej wydajności aplikacji dowiadujemy się dopiero podczas testów integracyjnych albo po wdrożeniu aplikacji do produkcji. Problemy z wydajnością są często trudne do wykrycia. Nieraz aplikacja z czasem spowalnia działanie w sposób trudny do zauważenia, a gdy to zauważymy, jest już zbyt późno (np. jeśli kwerenda bazy danych działa bez indeksu). Wiele problemów jest trudnych do rozwiązania — zwłaszcza gdy są spowodowane podjętymi wcześniej decyzjami architektonicznymi albo nieprzewidzianymi ograniczeniami sieci, bazy danych, pamięci masowej lub innych systemów.

Celem jest napisanie i uruchamianie automatycznych testów wydajności, które pozwolą zweryfikować wydajność działania całego stosu aplikacji (kodu, bazy danych, pamięci masowej, sieci, wirtualizacji itd.) w ramach potoku wdrożeń, tak aby można było wykryć problemy wcześnie — gdy poprawki są najtańsze i gdy można je wprowadzić najszybciej.

Dzięki zrozumieniu zachowania aplikacji i środowisk pod obciążeniem zbliżonym do produkcyjnego możemy znacznie lepiej zaplanować możliwości aplikacji, a także wykryć następujące sytuacje:

- Czasy obsługi kwerend do bazy danych rosną nieliniowo (np. zapomnieliśmy włączyć indeksowanie bazy danych i czas ładowania strony wzrósł z 30 sekund do kilkuset minut).
- Kiedy zmiana w kodzie powoduje kilkakrotne zwiększenie liczby odwołań do bazy danych, wykorzystania pamięci masowej lub ruchu sieciowego.

Gdy mamy testy akceptacyjne, które mogą być uruchamiane równolegle, możemy użyć ich jako podstawy do testów wydajności. Założymy, że obsługujemy witrynę e-commerce. Zidentyfikowaliśmy dwie wartościowe operacje „szukaj” i „do kasy”, które powinny wydajnie działać pod obciążeniem. Aby to sprawdzić, możemy jednocześnie przeprowadzić tysiące testów akceptacyjnych wyszukiwania równolegle z tysiącami testów płatności w kasie.

Ze względu na dużą moc obliczeniową i wiele operacji wejścia-wyjścia wymaganych do uruchomienia testów wydajności stworzenie środowiska testowania wydajności może być bardziej skomplikowane niż stworzenie środowiska produkcyjnego dla samej aplikacji. Z tego powodu warto zadbać o stworzenie środowiska testowania wydajności na początku każdego projektu i zadbać o przydzielenie zasobów niezbędnych do tego, by zostało ono zbudowane odpowiednio wcześnie i prawidłowo.

Aby szybko znaleźć problemy wydajności, należy rejestrować wyniki wydajności i porównywać każde uruchomienie testów wydajności z poprzednim. Można na przykład ustalić, że wynik testu wydajności jest negatywny, jeśli wydajność spada o więcej niż 2% w porównaniu z poprzednim uruchomieniem.

WŁĄCZENIE TESTOWANIA WYMAGAŃ NIEFUNKCJONALNYCH DO ZESTAWU TESTÓW

Oprócz sprawdzania, czy kod funkcjonuje zgodnie z projektem oraz działa wydajnie pod obciążeniem zbliżonym do produkcyjnego, należy także weryfikować wszystkie inne atrybuty obsługiwanego systemu. Często są one nazywane wymaganiami niefunkcjonalnymi i obejmują dostępność, skalowalność, wydajność, pojemność, bezpieczeństwo itd.

Wiele z tych wymagań jest spełnionych dzięki poprawnej konfiguracji środowiska, zatem trzeba również stworzyć testy automatyczne, które weryfikują, czy środowiska zostały prawidłowo zbudowane i skonfigurowane. Na przykład chcemy wymusić spójność i poprawność poniższych ustawień, na których bazuje wiele wymagań niefunkcjonalnych (np. bezpieczeństwa, wydajności, dostępności):

- pomocnicze aplikacje, bazy danych, biblioteki itp.;
- interpreterы языка, компиляторы itp.;
- системы операционные (np. включенные регистрация инспекции itp.);
- все зависимости.

Kiedy używamy narzędzi zarządzania konfiguracją w środowisku IaC (ang. *Infrastructure as a code* — dosł. „infrastruktura jako kod”) (np. Puppet, Chef, Ansible, Salt, Bosh), to możemy użyć tych samych frameworków testowych, których używamy do testowania kodu, aby sprawdzić, czy środowiska są poprawnie skonfigurowane i właściwie działają (np. możemy zakodować środowiska testowe do testów cucumber lub gherkin).

Ponadto na podobnej zasadzie, na jakiej uruchamiamy narzędzia analizy aplikacji w potoku wdrożeń (np. analizę statycznego kodu, analizę pokrycia testami), powinniśmy także uruchamiać narzędzia analizujące kod, który tworzy nasze środowiska (np. Foodcritic dla systemu Chef, puppet-lint dla systemu Puppet). W ramach testów automatycznych powinniśmy także uruchomić wszelkie testy hartowania zabezpieczeń (np. za pomocą narzędzia server-spec), tak aby sprawdzić, czy wszystko zostało skonfigurowane prawidłowo i zgodnie z zasadami zabezpieczeń.

Testy automatyczne powinny mieć możliwość zweryfikowania w dowolnym momencie, że mamy „zieloną komplikację” i że produkt jest w stanie gotowym do wdrożenia. Trzeba teraz stworzyć linkę Andon, aby w sytuacji, gdy ktoś uszkodzi potok wdrożeń, można było podjąć wszelkie niezbędne kroki w celu przywrócenia stanu „zielonej komplikacji”.

W PRZYPADKU AWARII POTOKU WDROŻEŃ POCIĄGNIJ ZA LINKĘ ANDON

Gdy w potoku wdrożeń mamy zieloną komplikację, to możemy zaufać, że po wdrożeniu zmian do produkcji kod i środowisko będą działać zgodnie z założeniami.

W celu utrzymania potoku wdrożeń w stanie zielonym utworzymy wirtualną linkę Andon przypominającą linkę fizyczną w systemie Toyota Production System. Za każdym razem, gdy ktoś wprowadzi zmianę, w wyniku której komplikacja lub testy automatyczne przestaną działać, do systemu nie będą mogły być wprowadzone żadne nowe prace tak długo, aż problem zostanie rozwiązany. Jeśli ktoś potrzebuje pomocy przy rozwiązyaniu problemu, może poprosić o dowolną pomoc, jakiej potrzebuje, podobnie jak w przykładzie firmy Google na początku tego rozdziału.

Gdy potok wdrożeń ulegnie uszkodzeniu, to w planie minimum informujemy cały zespół o awarii, tak aby każdy mógł albo rozwiązać problem, albo wycofać zmiany. Można nawet skonfigurować system kontroli wersji w taki sposób, aby nie pozwolić na dalsze ewidencjonowanie rewizji do czasu, aż pierwsza faza potoku wdrożeń (tzn. komplikacje i testy jednostkowe) będą w stanie zielonym. Jeśli problem spowodował test automatyczny, który wszczął fałszywy alarm, to należy taki test przepisać albo usunąć*. Do wycofania rewizji w celu przywrócenia komplikacji do stanu zielonego powinien być uprawniony każdy członek zespołu.

O znaczeniu przywracania potoku wdrożeń do stanu zielonego pisał Randy Shoup, były dyrektor inżynierii w Google App Engine: „Cele zespołu stawiamy nad celami indywidualnymi — zawsze, kiedy pomagamy komuś ruszyć z miejsca, pomagamy całemu zespołowi. Dotyczy to zarówno udzielenia komuś pomocy przy naprawieniu komplikacji, testu automatycznego, jak i wykonania przeglądu czyjegoś kodu. Oczywiście każdy członek zespołu ma świadomość, że inni udzielają mu pomocy, kiedy będzie tego potrzebował. Taki system działał bez zbędnych formalności lub dodatkowych reguł — wszyscy wiedzieli, że naszym zadaniem jest nie tylko »napisać kod«, ale »uruchomić usługę«. Dlatego właśnie nadaliśmy najwyższy priorytet wszystkim problemom jakości, zwłaszcza dotyczącym niezawodności i skalowania, i zaczęliśmy traktować je jako problemy o priorytecie 0. Z punktu widzenia systemów stosowanie tych praktyk pozwala nam uniknąć cofania się”.

Gdy zawiodą późniejsze fazy potoku wdrożeń — na przykład testy akceptacyjne lub testy wydajności — to zamiast zatrzymywać wszystkie nowe prace, możemy skorzystać z pomocy programistów i testerów odpowiedzialnych za natychmiastowe rozwiązanie powstałych problemów. Powinni oni również stworzyć nowe testy uruchamiane na

* Jeśli proces wycofywania kodu z repozytorium nie jest dobrze znany, to potencjalnym środkiem zaradczym jest zaplanowanie wycofywania w trybie programowania w parach, tak aby było ono lepiej udokumentowane.

wcześniejszym etapie potoku wdrożeń, tak aby w przyszłości móc wykryć regresję. Na przykład, jeśli odkryjemy defekt testów akceptacyjnych, powinniśmy napisać test jednostkowy, który wykrywa problem. Podobnie jeśli wykryjemy defekt w teście rozpoznawczym, powinniśmy napisać test jednostkowy lub akceptacyjny.

Aby zwiększyć widoczność awarii testów automatycznych, powinniśmy utworzyć bardzo widoczne wskaźniki, tak aby cały zespół mógł zobaczyć, kiedy komplikacja lub test automatyczny kończą się niepowodzeniem. Wiele zespołów umieszcza na widocznych miejscach na ścianie wskaźniki, które pokazują bieżący status komplikacji, lub stosuje inne ciekawe sposoby informowania zespołu o tym, że komplikacja jest uszkodzona. Należą do nich lampy lawowe, próbki głosowe lub utwory muzyczne, klaksony, światła itd.

Pod wieloma względami realizacja tego kroku jest trudniejsza niż tworzenie komplikacji i serwerów testów — to były czynności czysto techniczne, podczas gdy ten krok wymaga zmiany zachowań ludzkich i motywatorów. Jednak wdrożenie technik ciągłej integracji i ciągłego dostarczania wymaga wprowadzenia tych zmian. Opowiem o tym w następnym podrozdziale.

DLACZEGO TRZEBA POCIĄGAĆ ZA LINKĘ ANDON?

Konsekwencją rezygnacji z pociągnięcia za linkę Andon i natychmiastowego przystąpienia do naprawy potoku wdrożeń jest dobrze znany problem — sytuacja, w której przywrócenie aplikacji i środowiska do stanu gotowości do wdrożenia staje się jeszcze trudniejsze. Rozważmy następującą sytuację:

- Ktoś ewidencjonuje rewizję kodu, która powoduje awarię komplikacji lub testów automatycznych, ale nikt nie rozwiązuje problemu.
- Ktoś inny wprowadza kolejną zmianę do niedziałającej komplikacji, po której testy automatyczne również nie przechodzą — ale nikt nie zauważa negatywnych wyników testów, które pozwoliłyby wykryć nowy defekt, nie mówiąc już o jego naprawieniu.
- Istniejące testy nie działają wiarygodnie, dlatego zachodzi małe prawdopodobieństwo, że będziemy budować nowe testy. (Po co się trudzić? Przecież nawet te testy, które mamy, nie działają).

W takim przypadku wdrożenia w dowolnym środowisku stają się równie niewiarygodne, jak wtedy, gdy nie stosowaliśmy testów automatycznych lub gdy korzystaliśmy z metody kaskadowej, a większość problemów była wykrywana w fazie produkcji. Nieuniknionym efektem tego błędnego koła jest powrót do punktu wyjścia — z nieprzewidywalną „fazą stabilizacji”, która trwa wiele tygodni bądź miesięcy, gdzie

cały zespół jest pogrążony w kryzysie, starając się doprowadzić testy do stanu, w którym zaczną przechodzić, wykonuje obejścia ze względu na presję terminów i przyczynia się do powiększania dłużu technicznego*.

PODSUMOWANIE

W tym rozdziale stworzyliśmy kompleksowy zbiór testów automatycznych, których zadaniem jest zagwarantowanie „zielonego” stanu komplikacji, dla której przechodzą wszystkie testy i która nadaje się do wdrożenia. Zestawy testów oraz działania związane z testowaniem zorganizowaliśmy w formie potoku wdrożeń. Stworzyliśmy również kulturę, która wymaga od nas, by w przypadku, gdy ktoś z zespołu wprowadzi zmianę powodującą ujemny wynik testów automatycznych, robić wszystko, co jest możliwe, aby powrócić do stanu „zielonego”.

W ten sposób ustawiliśmy scenę dla implementacji techniki ciągłej integracji, która pozwala wielu małym zespołom niezależnie i bezpiecznie rozwijać, testować i wdrażać kod do produkcji, jednocześnie dostarczając wartość dla klientów.

* Czasami tę sytuację określa się jako antywzorzec water-Scrum-fall. Dotyczy on sytuacji, w której organizacja twierdzi, że stosuje praktyki przypominające Agile, ale w rzeczywistości wszystkie testy i naprawy defektów są wykonywane na końcu projektu.

Wdrożenie i stosowanie praktyk ciągłej integracji

W poprzednim rozdziale zaprezentowaliśmy praktykę stosowania testów automatycznych, która miała zapewnić programistom szybkie sprzężenia zwrotne na temat jakości ich pracy. Staje się to jeszcze ważniejsze w przypadku zwiększenia liczby programistów oraz liczby gałęzi ewidencjonowanych w repozytorium kontroli wersji.

Zdolność tworzenia „gałęzi” (ang. *branch*) w systemach kontroli wersji została stworzona przede wszystkim po to, aby umożliwić programistom równoległą pracę nad różnymi częściami systemu oprogramowania bez ryzyka, że zmiany zaewidencjonowane przez jednego programistę zdestabilizują „pień” (ang. *trunk*) repozytorium (nazywanego też rewizją *master* lub *mainline*) albo wprowadzą do niego błędy*.

Jednak im dłużej pozwalamy programistom na pracę nad swoimi gałęziami w izolacji, tym trudniejsza staje się integracja i scalanie zmian wprowadzonych przez poszczególnych programistów z rewizją *master*. Trudności z integracją wzrastają wykładniczo wraz ze wzrostem liczby gałęzi oraz liczb zmian w każdej z gałęzi kodu.

Problemy integracji skutkują koniecznością wprowadzania wielu przeróbek niezbędnych do doprowadzenia produktu z powrotem do stanu gotowości do wdrożenia. Dodatkowe problemy stwarzają zmiany kolidujące ze sobą, które trzeba scalać ręcznie, albo takie scalenia, które łamią testy ręczne lub automatyczne. Ich skuteczne naprawienie

* Gałęzie w systemach kontroli wersji stosuje się na wiele sposobów, ale zazwyczaj używany ich w celu podziału pracy pomiędzy członków zespołu według wydania, promocji, zadania, komponentu, platform technologicznych itp.

zwykle wymaga współpracy wielu programistów. Ponieważ integracja w tradycyjnych systemach zwykle była realizowana pod koniec projektu, a czas jej trwania był znacznie dłuższy od oczekiwanej, to często zachodziła konieczność „chodzenia na skróty”, aby dotrzymać terminu wydania.

To powoduje kolejną „spiralę w dół”: gdy scalanie kodu sprawia kłopoty, to zwykle jest wykonywane rzadziej, a to sprawia, że kolejne scalenia stają się jeszcze bardziej trudne. W celu rozwiązania tego problemu zaprojektowano ciągłą integrację — praktykę scalania zmian z rewizją master w ramach codziennej pracy.

Gary Gruver, dyrektor inżynierii działu HP LaserJet Firmware, zajmującego się tworzeniem oprogramowania firmware dla wszystkich skanerów, drukarek i urządzeń wielofunkcyjnych produkowanych przez HP, zaprezentował rozwiązania ciągłej integracji stosowane w firmie, wskazując zaskakującą wiele problemów, które rozwiązało wdrożenie tych praktyk.

Zespół składał się z 400 programistów rozproszonych w Stanach Zjednoczonych, Brazylii i Indiach. Pomimo rozmiarów zespołu postępy były zbyt wolne. Przez wiele lat firma nie była w stanie dostarczać nowych funkcjonalności dostatecznie szybko, aby zaspokoić potrzeby biznesu.

Gruver opisał ten problem w następujący sposób: „Ludzie z marketingu przychodzą do nas z milionem pomysłów na olśnienie naszych klientów, a my musimy im odpowiadać: »Wybierzcie z tej listy dwie rzeczy, które chcecie zrealizować w ciągu najbliższych sześciu do dwunastu miesięcy«”.

Zespół był w stanie zrealizować zaledwie dwa wydania oprogramowania firmware rocznie. Większość czasu poświęcano na przenoszenie kodu na nowe platformy w celu wsparcia nowych produktów. Gruver oszacował, że zaledwie 5% czasu poświęcano na tworzenie nowych funkcjonalności — pozostały czas przeznaczano na nieproduktywne prace związane z ich technicznym długiem — takie jak zarządzanie wieloma gałęziami kodu i ręczne testowanie. Oto podział tych zadań:

- 20% na szczegółowe planowanie (niską wydajność i długi czas realizacji zadań błędnie przypisywano złemu szacowaniu pracochłonności zadań, dlatego w nadziei na uzyskanie dokładniejszej odpowiedzi żądano bardziej szczegółowego szacowania pracy).
- 25% zajmowało przenoszenie kodu, który w całości był utrzymywany na oddzielnych gałęziach.
- 10% poświęcano na integrowanie kodu pomiędzy gałęziami poszczególnych programistów.
- 15% zajmowało ręczne testowanie.

Gruver wraz z zespołem postawił sobie za cel dziesięciokrotne wydłużenie czasu na innowacje i nowe funkcjonalności. Sądzono, że ten cel można osiągnąć poprzez:

- Wprowadzenie praktyk ciągłej integracji i rozwoju oprogramowania bazującego na rewizji master (ang. *trunk-based development — TBD*).
- Znaczące inwestycje w automatyzację testów.
- Utworzenie symulatorów sprzętu, tak aby testy mogły być uruchamiane na wirtualnych platformach.
- Reprodukowanie błędów w testach na stacjach roboczych programistów.
- Nową architekturę do obsługi wszystkich drukarek za pośrednictwem wspólnej komplikacji i wydania.

Wcześniej każda linia produktu wymagała nowej gałęzi kodu. Dla każdego modelu utrzymywano unikatową komplikację oprogramowania firmware, a funkcjonalności definiowano w fazie komplikacji*. W nowej architekturze wszyscy programiści mieli pracować na wspólnej bazie kodu i korzystać z jednego wydania oprogramowania firmware obsługującego wszystkie modele LaserJet, budowanego na podstawie rewizji master. Funkcjonalności drukarki były włączane w fazie działania aplikacji, za pomocą pliku konfiguracyjnego XML.

Cztery lata później w całej firmie korzystano z jednej bazy kodu, obsługującej wszystkie 24 linie produktów HP LaserJet, które były opracowywane na podstawie rewizji master. Gruver przyznaje, że tworzenie oprogramowania bazujące na rewizji master wymagało wielkich zmian w sposobie myślenia. Inżynierowie sądzili, że stosowanie modelu TBD nigdy się nie sprawdzi, ale kiedy zaczęto go stosować, nie potrafili sobie wyobrazić, że można się z tego wycofać. Przez lata inżynierowie, którzy odchodziли z HP, dzwoniли do mnie i mówili, jak zafanowane były modele wytwarzania oprogramowania w ich nowych firmach. Podkreślali przy tym, jak trudno jest być skutecznymi wydawać dobry kod, gdy nie ma sprzężenia zwrotnego za pośrednictwem ciągłej integracji.

Stosowanie modelu TBD wymagało jednak budowania skuteczniejszych testów automatycznych. Gruver zauważał: „Bez testowania automatycznego ciągła integracja jest najszybszym sposobem uzyskania olbrzymiego stresu śmieci, który nigdy się nie skompiluje ani nie będzie działać poprawnie”. Początkowo w pełni ręczny cykl testowania wymagał sześciu tygodni.

Aby wszystkie komplikacje oprogramowania firmware były automatycznie przetwarzane, zainwestowano duże środki w symulatory drukarek i w sześć tygodni stworzono farmę testową. W ciągu kilku lat uzyskano stan, w którym na sześciu półkach serwerowych było uruchomionych dwa tysiące symulatorów drukarek, które ładowały komplikacje oprogramowania firmware z potoku wdrożeń. Ich system ciągłej integracji

* Do włączania i wyłączania funkcjonalności drukowania, obsługiwanych rozmiarów papieru itp. wykorzystywano flagi kompilatora (`#define` i `#ifdef`).

(CI) zajmował się uruchamianiem całego zbioru automatycznych testów jednostkowych, akceptacyjnych i integracyjnych, bazując na komplikacjach z rewizji master, tak jak opisano w poprzednim rozdziale. Ponadto stworzono kulturę, w której zatrzymywano wszystkie prace za każdym razem, gdy jakiś programista spowodował awarię potoku wdrożeń, i dążono do szybkiego przywrócenia systemu do stanu „zielony”.

Testy automatyczne dostarczyły szybkiego sprzężenia zwrotnego, co pozwoliło programistom błyskawicznie uzyskać potwierdzenie, że kod zaewidencjonowany w repozytorium faktycznie działa. Uruchomienie testów jednostkowych na stacjach roboczych zajmowało kilka minut, każde zaewidencjonowanie rewizji w repozytorium inicjowało trzy poziomy testów automatycznych, które dodatkowo były uruchamiane co 2 – 4 godziny. Ostateczne pełne testy regresji były wykonywane co 24 godziny. Podczas tego procesu:

- Zmniejszono liczbę komplikacji do jednej dziennie. Docelowo wykonywano 10 – 15 komplikacji na dzień.
- Liczbę około 20 operacji ewidencjonowania rewizji realizowanych przez „bossa budowania” zwiększo do ponad 100 dziennie, wykonywanych przez pojedynczych programistów.
- Umożliwiono programistom modyfikowanie lub dodawanie 75 000 – 100 000 wierszy kodu każdego dnia.
- Skrócenie czasu wykonywania testów regresji z sześciu tygodni do jednego dnia.

Takiego poziomu wydajności nigdy nie osiągnięto w czasach przed przyjęciem ciągłej integracji. Wtedy samo doprowadzenie komplikacji do stanu „zielony” wymagało wielu dni „heroicznej” pracy. Uzyskane korzyści biznesowe były zadziwiające:

- Czas poświęcony na innowacyjność i pisanie nowych funkcjonalności zwiększył się z 5% czasu programisty do 40%.
- Ogólne koszty wytwarzania oprogramowania zostały zmniejszone o mniej więcej 40%.
- O mniej więcej 140% zmniejszyła się liczba „programów w budowie”.
- Koszty wytwarzania oprogramowania spadły o 78%.

Doświadczenia Gruvera pokazują, że po kompleksowym wykorzystywaniu repozytoriów kontroli wersji stosowanie technik ciągłej integracji jest jedną z najważniejszych praktyk umożliwiających szybki przepływ pracy w strumieniu wartości. Ich stosowanie pozwala wielu zespołom programistów niezależnie rozwijać, testować i dostarczać wartość. Niemniej jednak praktyki ciągłej integracji stwarzają kontrowersje. W pozostałej części tego rozdziału opisano praktyki wymagane do implementacji ciągłej integracji, a także pokazano, w jaki sposób pokonać najczęstsze zastrzeżenia.

WYTWARZANIE OPROGRAMOWANIA MAŁYMI PARTIAMI. CO SIĘ DZIEJE, GDY RZADKO EWIDENCJONUJEMY KOD DO REWIZJI MASTER?

Jak napisano w poprzednich rozdziałach, w każdym przypadku, gdy do repozytorium kontroli wersji zostaną wprowadzone zmiany, które spowodują awarię potoku wdrożeń, natychmiast stosujemy swarming, aby jak najszybciej przywrócić potok wdrożeń do stanu „zielony”. Jednak gdy programiści pracują z „długowiecznymi” prywatnymi gałęziami (nazywanymi również „gałęziami funkcjonalności”) i tylko sporadycznie scalają je z rewizją master — przez co wprowadzanych jest wiele zmian — powstają istotne problemy. Jak opisano w przykładzie HP LaserJet, w rezultacie powstaje znaczący chaos, a doprowadzenie kodu do stanu gotowości do wdrożenia wymaga wielu przeróbek.

Jeff Atwood, założyciel witryny Stack Overflow i autor bloga Coding Horror, zauważał, że chociaż istnieje wiele strategii tworzenia gałęzi, to wszystkie je można podzielić na następujące kategorie:

- **Optymalizowane pod kątem indywidualnej wydajności** — każda osoba w projekcie wykorzystuje własną, prywatną gałąź. Każdy pracuje niezależnie i nikt nie może zakłócić pracy innej osoby, ale scalanie kodu staje się koszmarem. Współpraca staje się niemal komicznie trudna — aby zobaczyć choćby najmniejszą część kompletnego systemu, praca każdego programisty musi być scalona w uciążliwym procesie z pracą pozostałych.
- **Optymalizacja pod kątem wydajności zespołu** — każdy pracuje w tym samym, wspólnym obszarze. Nie ma gałęzi — jest jedynie dłuża, nieprzerwana prosta linia rozwoju. Nie trzeba niczego rozumieć, dlatego ewidencjonowanie rewizji jest proste. Jednak każda operacja ewidencjonowania może doprowadzić do awarii całego projektu i zatrzymać jakkolwiek postęp.

Obserwacje Atwooda są całkowicie poprawne — mówiąc dokładniej, wysiłek potrzebny do pomyślnego scalenia gałęzi zwiększa się wykładniczo wraz ze wzrostem liczby gałęzi. Problem tkwi nie tylko w liczbie przeróbek koniecznych z powodu tego „piekła scalania”, ale również w spóźnionym sprzężeniu zwrotnym otrzymywany z potoku wdrożeń. Na przykład zamiast testować wydajność w trybie ciągłym, z wykorzystaniem w pełni zintegrowanego systemu, naprawdopodobniej będzie to realizowane tylko pod koniec procesu.

Ponadto w miarę zwiększania tempa produkcji kodu i dodawania nowych programistów zwiększymy również prawdopodobieństwo, że wprowadzona zmiana wywierze wpływ na kod innych, przez co zwiększy się liczba programistów, na których będzie miała wpływ awaria potoku wdrożeń.

Oto jeden z ostatnich niepokojących skutków ubocznych scalania dużych partii: gdy scalanie jest trudne, mamy mniejsze możliwości i słabszą motywację do poprawiania i refaktoryzowania kodu, ponieważ operacja scalania z większym prawdopodobieństwem spowoduje konieczność wprowadzania przeróbek przez innych. Gdy tak się dzieje, odnosimy się niechętnie do modyfikowania kodu z zależnościami od innych części bazy kodu, czyli takiego, który jest dla nas najwartościowszy.

Ward Cunningham, programista pierwszej strony wiki, obserwując to zjawisko, po raz pierwszy opisał techniczny dług: gdy nie przeprowadzamy agresywnej refaktoryzacji bazy kodu, to z czasem wprowadzanie zmian i utrzymywanie kodu staje się coraz bardziej skomplikowane, a tempo dodawania nowych funkcjonalności maleje. Rozwiążanie tego problemu było jednym z podstawowych celów techniki ciągłej integracji i praktyk pracy na rewizji master. Wszystko po to, aby optymalizacja wydajności zespołu była traktowana priorytetowo w porównaniu z wydajnością indywidualnych programistów.

STOSOWANIE PRAKTYK PRACY NA REWIZJI MASTER

Środkiem zaradczym dla scalania dużych partii jest wykorzystanie praktyk ciągłej integracji oraz pracy na rewizji master, gdzie każdy programista co najmniej raz dziennie ewidencjonuje kod w rewizji master. Tak częste ewidencjonowanie kodu zmniejsza rozmiar partii do pracy wykonywanej przez cały zespół programistów w ciągu jednego dnia. Im częściej programiści ewidencjonują kod do rewizji master, tym mniejszy rozmiar partii i tym bliżej do osiągnięcia teoretycznego ideału przepływu jednej części.

Częste ewidencjonowanie kodu do rewizji master oznacza, że możemy uruchomić wszystkie zautomatyzowane testy systemu oprogramowania jako całości i otrzymywać alerty, gdy zmiana spowoduje awarię innych części aplikacji lub zakłóci pracę innego programisty. A ponieważ problemy scalania możemy wykryć łatwiej wtedy, gdy dotyczą niewielkich partii kodu, to możemy szybciej je poprawić.

Mogliśmy nawet skonfigurować potok wdrożeń w taki sposób, aby wszelkie rewizje (np. zmiany kodu lub środowiska), które powodują naruszenie stanu gotowości do wdrożenia, były odrzucane. Tę metodę określa się terminem **gated commits** (dosł. „rewizje ogrodzone”). Polega ona na tym, że potok wdrożeń przed scaleniem rewizji z masterem najpierw sprawdza, czy przesłana rewizja może być pomyślnie scalona, buduje się zgodnie z oczekiwaniami, a wszystkie testy automatyczne przechodzą. W przeciwnym razie zostaje wysłane powiadomienie do programisty, tak by mógł wprowadzić odpowiednie korekty bez wpływu na inne osoby w strumieniu wartości.

Dyscyplina codziennego przesyłania rewizji zmusza także do podziału pracy na mniejsze części przy jednoczesnym utrzymaniu rewizji master w stanie poprawnego działania i gotowości do wdrożenia.

Ponadto repozytorium kontroli wersji staje się integralnym mechanizmem, za pomocą którego komunikują się ze sobą członkowie zespołu — wszyscy mają lepsze

wspólne rozumienie systemu, są świadomi stanu potoku wdrożeń i mogą nawzajem sobie pomagać w przypadku jego awarii. W rezultacie osiągamy wyższą jakość i szybsze czasy realizacji wdrożenia.

Po stworzeniu tych praktyk możemy ponownie zmodyfikować naszą definicję określenia „kod gotowy” (dodatki oznaczono pogrubioną czcionką): „Na koniec każdego interwału rozwoju, musimy mieć kod zintegrowany, przetestowany, działający i potencjalnie gotowy do wdrożenia, zademonstrowany w środowisku zbliżonym do produkcyjnego oraz **zbudowany na podstawie rewizji master w procesie inicjowanym jednym kliknięciem oraz zweryfikowany za pomocą testów automatycznych**”.

Stosowanie się do tej poprawionej definicji terminu „kod gotowy” pomaga na bieżąco zapewnić stan testowalności i gotowości do wdrożenia wytwarzanego kodu. Dzięki utrzymywaniu kodu w stanie gotowości do wdrożenia możemy wyeliminować powszechną praktykę stosowania osobnej fazy testowania i stabilizacji na końcu projektu.

Studium przypadku

Ciągła integracja w firmie Bazaarvoice (2012)

Ernest Mueller, który pomógł przeprowadzić transformacje DevOps w firmie National Instruments, później, w 2012 roku, pomógł przekształcić procesy rozwoju i publikowania w firmie Bazaarvoice. Firma Bazaarvoice dostarcza treści generowane przez klientów (np. opinie, oceny) tysiącom sprzedawców detalicznych, takich jak Best Buy, Nike i Walmart.

W tamtym okresie firma Bazaarvoice miała 120 milionów dolarów rocznych przychodów i przygotowywała się do debiutu na giełdzie*. Głównym produktem biznesowym firmy Bazaarvoice była wtedy aplikacja Bazaarvoice Conversations — powstała w 2006 roku monolityczna aplikacja w Javie, zawierająca niemal pięć milionów wierszy kodu i obejmująca 15 000 plików. Usługa działała na 1200 serwerach rozmieszczenych w czterech centrach przetwarzania danych oraz u wielu dostawców usług w chmurze.

Częściowo ze względu na przełączenia się na proces wytwarzania oprogramowania Agile oraz stosowanie dwutygodniowych interwałów rozwoju oprogramowania było ogromne dążenie do zwiększenia częstotliwości publikowania w porównaniu z bieżącym 10-tygodniowym harmonogramem. Zaczęto również prace nad rozbijaniem monolitycznej aplikacji na mikrousługi.

Pierwszą próbę publikacji co dwa tygodnie podjęto w styczniu 2012 roku. Jak zaobserwował Mueller, „nie poszło dobrze”. Powstał ogromny chaos, klienci zgłosili 44 zdarzenia produkcyjne. Główną reakcją kierownictwa na tę sytuację było po prostu stwierdzenie: „Nie róbmy tego nigdy więcej”.

* Wydanie produkcyjne było opóźnione z powodu (udanego) debiutu na giełdzie.

Wkrótce po tym fakcie kierownictwo nad procesami publikowania przejął Mueller, który postawił sobie cel, aby realizować proces publikowania co dwa tygodnie w taki sposób, by nie było przestojów u klientów. Wśród celów biznesowych częstszego publikowania wymieniano umożliwienie szybszego testowania A/B (opisanego w kolejnych rozdziałach) oraz zwiększenie przepływu funkcjonalności do produkcji. Mueller wyróżnił trzy podstawowe problemy:

- Brak automatyzacji testów sprawiał, że testowanie na jakimkolwiek poziomie podczas dwutygodniowych interwałów nie było w stanie zapobiec awariom na dużą skalę.
- Strategia zarządzania gałęziami w repozytorium kontroli wersji pozwalała programistom ewidencjonować nowe rewizje kodu bezpośrednio do wydania produkcyjnego.
- Zespoły obsługujące mikrousługi realizowały także niezależne publikacje, co często powodowało problemy podczas publikowania monolitu. Zdarzały się również sytuacje odwrotne (gdy publikowanie monolitu zakłócało pracę mikrousług).

Mueller doszedł do wniosku, że proces wdrażania monolitycznej aplikacji musi być ustabilizowany, a to wymaga zastosowania ciągłej integracji. W ciągu kolejnych sześciu tygodni programiści zaprzestali pracy nad nowymi funkcjonalnościami, a zamiast tego skoncentrowali się na pisaniu zestawów testów automatycznych — jednostkowych w JUnit oraz testów regresji w Selenium. Ponadto skupiali się na utrzymaniu działania potoku wdrożeń, korzystając z systemu TeamCity. „Dzięki ciągłemu uruchamianiu testów cały czas czuliśmy, że możemy wprowadzać zmiany, zachowując pewien poziom bezpieczeństwa. A co najważniejsze, mogliśmy się natychmiast dowiedzieć, że jakaś zmiana spowodowała awarię, w przeciwieństwie do odkrywania problemów dopiero w produkcji”.

Przełączono się również do modelu publikowania trunk/branch (dosł. „master-gałąz”), gdzie co dwa tygodnie tworzona było nowa dedykowana gałąź wydania. Do tej gałęzi nie można było ewidencjonować żadnych rewizji, jeśli nie było sytuacji kryzysowej. Wszystkie zmiany były zatwierdzane na podstawie zlecenia przez wyznaczoną osobę albo wewnątrz zespołu, z wykorzystaniem wiki. Gałąź była poddawana procesowi walidacji, a następnie promowana do produkcji.

Poprawa w zakresie przewidywalności i jakości wydań była zaskakująca:

- Wydanie styczeń 2012: 44 incydenty zgłoszone przez klientów (początek wdrażania ciągłej integracji).
- Wydanie 6 marca 2012: pięć dni spóźnienia, pięć incydentów zgłoszonych przez klientów.
- Wydanie 22 marca 2012: zgodnie z planem, jedno zgłoszenie incydentu od klientów.
- Wydanie 5 kwietnia 2012: zgodnie z planem, zero zgłoszeń incydentów od klientów.

Mueller opisywał kolejne sukcesy tego przedsięwzięcia:

Publikacje co dwa tygodnie przyniosły tak wielki sukces, że przeszliśmy na wydania cotygodniowe niemal bez konieczności wprowadzania jakichkolwiek zmian w organizacji pracy zespołów inżynierskich. Ponieważ publikacje stały się rutyną, to skrócenie interwału publikacji do tygodnia sprowadziło się do podwojenia liczby wydań w kalendarzu i realizowania publikacji, kiedy przypadała właściwa data. W gruncie rzeczy nie było to żadne wydarzenie. Większość potrzebnych zmian dotyczyła zespołów obsługi klienta i marketingu, gdzie zachodziła konieczność zmiany procesów — na przykład zmiana harmonogramu cotygodniowych e-maili do klientów, aby poinformować ich o zamiarze wprowadzenia zmian. Następnie zaczęliśmy pracować w kierunku osiągnięcia następnych celów. Ostatecznie nasze działania doprowadziły do przyspieszenia czasu testowania z ponad trzech godzin do mniej niż jednej godziny, zmniejszenia liczby środowisk z czterech do trzech (rozwojowe, testowe, produkcyjne; wyeliminowano przedprodukcyjne — ang. *staging*) i przełączenia do pełnego modelu ciągłego dostarczania, w którym zapewniliśmy szybkie wdrożenia (w trybie *oneclick*).

PODSUMOWANIE

Wytwarzanie oprogramowania bazujące na rewizji master jest prawdopodobnie najbardziej kontrowersyjną praktyką omawianą w tej książce. Wielu inżynierów nie wierzy, że jej stosowanie jest możliwe. Wątpią nawet ci programiści, którzy wolą pracować bez przeszkód na prywatnej gałęzi, nie przejmując się innymi programistami. Jednak dane z raportu *Puppet Labs State of DevOps Report 2015* są czytelne: wytwarzanie oprogramowania bazujące na rewizji master pozwala prognozować wyższą wydajność, lepszą stabilność i większą satysfakcję z pracy oraz niższe wskaźniki wypalenia zawodowego.

O ile przekonanie programistów do stosowania tej metody początkowo może być trudne, o tyle gdy zobaczą na własne oczy korzyści, jakie z niej wynikają, to prawdopodobnie staną się jej gorącymi zwolennikami. Dowodzą tego przykłady firm HP LaserJet i Bazaarvoice. Praktyki ciągłej integracji ustawnią scenę dla następnego kroku — automatyzacji procesu wdrażania i publikowania obarczonego niskim ryzykiem.

12

Automatyzacja i zapewnienie wydań niskiego ryzyka

Chuck Rossi jest dyrektorem inżynierii wydań w firmie Facebook. Jednym z jego obowiązków jest nadzorowanie codziennego publikowania kodu. W 2012 roku Rossi opisywał ten proces w następujący sposób: „Mniej więcej o 13.00 przełączam się do »trybu operacyjnego« i pracuję wraz z moim zespołem, aby przygotować się do przesłania zmian, które zostaną opublikowane tego dnia w systemie Facebook.com. To jest bardziej stresująca część pracy, która w dużym stopniu bazuje na ocenie mojego zespołu oraz wcześniejszych doświadczeniach. Pracujemy nad tym, aby wszyscy, których zmiany są zaplanowane do wydania, zostali uwzględnieni oraz aby aktywnie testowali i wspomagali swoje zmiany”.

Tuż przed wykonaniem produkcyjnej operacji push muszą być obecni wszyscy programiści, których zmiany są wydawane. Muszą oni zalogować się na swoim kanale IRC. Jeśli jakiś programista się nie zaloguje, to jego zmiany są automatycznie usuwane z publikowanego pakietu. Dalej Rossi pisał: „Jeśli wszystko wygląda dobrze, a pulpity testowe i testy kanarkowe* przechodzą, to naciskamy duży czerwony przycisk, który powoduje, że na całej flocie serwerów Facebook.com zostaje opublikowany nowy kod.

* Test kanarkowy wydania polega na wdrożeniu oprogramowania na małej grupie serwerów produkcyjnych. Celem testu jest sprawdzenie, czy oprogramowanie nie ulegnie awarii pod realnym obciążeniem.

W ciągu dwudziestu minut na tysiącach maszyn jest instalowany nowy kod, bez widocznego wpływu na osoby korzystające z serwisu”*.

Później w tym samym roku Rossi podwoił częstotliwość publikowania oprogramowania do dwóch razy na dobę. Wyjaśnił, że drugie wydanie dało inżynierom spoza Zachodniego Wybrzeża Stanów Zjednoczonych możliwość „publikowania kodu w takim samym tempie, w jakim publikowali inni inżynierowie w firmie”. Dodatkowo wszyscy każdego dnia mieli dwie szanse publikowania kodu i uruchamiania nowych funkcjonalności. Rosnącą liczbę aktywnych programistów publikujących kod w Facebook.com pokazano na rysunku 16.



Rysunek 16. Liczba programistów wdrażających co tydzień kod w firmie Facebook
(źródło: Chuck Rossi, „Ship early and ship twice as often”)

Kent Beck, twórca metodyki programowania ekstremalnego, jeden z czołowych zwolenników techniki TDD (ang. *Test Driven Development* — dosł. „programowanie sterowane testami”) i techniczny trener w firmie Facebook, w artykule umieszczonym na swojej facebookowej stronie skomentował strategię wydań stosowaną przez Facebook: „Chuck Rossi zaobserwował, że w jednym wdrożeniu Facebook jest w stanie obsłużyć stałą liczbę zmian. Jeśli chcemy więcej zmian, potrzebujemy więcej wdrożeń. Doprowadziło to do stałego wzrostu w tempie wdrażania w ciągu ostatnich pięciu lat. Zaczynaliśmy od trybu cotygodniowego, a doszliśmy do wdrożeń kodu PHP trzy razy dziennie. Zmieniliśmy również cykl wdrożeń aplikacji mobilnych z sześciu tygodni, poprzez cztery, aż do dwóch. To usprawnienie było sterowane głównie przez zespół inżynierów wydania”.

* Baza kodu warstwy frontend Facebooka jest napisana głównie w PHP. W celu poprawy wydajności witryny w 2010 roku kod PHP został przekształcony na C++ za pomocą opracowanego w firmie Facebook kompilatora HipHop. Po komplikacji program wykonywalny miał rozmiar 1,5 GB. Ten plik był następnie kopowany na serwery produkcyjne za pomocą mechanizmu BitTorrent. Dzięki temu wykonanie operacji mogło być zakończone w 15 minut.

Dzięki zastosowaniu ciągłej integracji i zmniejszeniu zagrożeń związanych z procesem wdrażania firma Facebook doprowadziła do tego, że wdrożenia kodu stały się częścią codziennej pracy, a wydajność programistów się ustabilizowała. Aby to osiągnąć, trzeba było zautomatyzować wdrożenia kodu tak, by proces wdrożenia był powtarzalny i przewidywalny. W praktykach opisanych dotychczas w tej książce, pomimo że kod i środowiska były testowane razem, nie stosowaliśmy zbyt częstych wdrożeń do produkcji. Powód był prosty: wdrożenia były realizowane ręcznie, były czasochłonne, kłopotliwe i stwarzały okazje do popełnienia błędów. Często również wymagały niewygodnego i niewiarygodnego przekazywania pracy pomiędzy działami Dev i Ops.

A ponieważ proces był kłopotliwy, to realizowaliśmy go coraz rzadziej. Efektem była kolejna, samonapędzająca się spirala „w dół”. Poprzez odraczanie wdrożeń do produkcji gromadzimy coraz większe różnice pomiędzy kodem, który ma być wdrożony, a tym, który działa w produkcji, a zatem zwiększymy rozmiar partii wdrożenia. Równolegle z rozmiarami partii wdrożenia rośnie ryzyko nieoczekiwanych efektów zmiany, a także trudność ich naprawienia.

W tym rozdziale postaramy się zmniejszyć tarcia związane z wdrożeniami produkcyjnymi, tak aby zagwarantować ich częste i łatwe wdrożenia — realizowane przez dział albo Ops, albo Dev. Zrobimy to poprzez rozszerzenie potoku wdrożeń.

Zamiast poprzesiąć na ciągłej integracji kodu w środowisku zbliżonym do produkcyjnego, pozwolimy na promocję do produkcji każdej komplikacji, która przechodzi automatyczny proces testów i walidacji — albo na żądanie (tzn. po wcięnięciu przycisku), albo automatycznie (tzn. automatycznie wdrażana będzie każda komplikacja, dla której przejdą wszystkie testy).

Ze względu na liczbę prezentowanych praktyk zamieszczono w tym rozdziale liczne przykłady i informacje dodatkowe, nie przerywając prezentowania nowych pojęć.

AUTOMATYZACJA PROCESU WDROŻEŃ

Osiągnięcie takich wyników, jakie osiągnął Facebook, wymaga zastosowania automatycznego mechanizmu wdrażającego kod do produkcji. Zwłaszcza w sytuacji, gdy potok wdrożeń istnieje od lat, trzeba w pełni udokumentować kroki procesu wdrażania — na przykład wykonując ćwiczenie mapowania strumienia wartości zorganizowane w formie warsztatów — albo dokumentować proces przyrostowo (np. na stronie wiki).

Gdy proces jest już udokumentowany, kolejnym celem powinno być uproszczenie i automatyzacja jak największej liczby ręcznych działań. Na przykład:

- tworzenie pakietów instalacyjnych kodu w sposób ułatwiający wdrażanie;
- tworzenie wstępnie skonfigurowanych obrazów maszyn wirtualnych lub kontenerów;
- automatyzacja wdrożeń i konfiguracja oprogramowania middleware;

- kopiowanie pakietów lub plików na serwery produkcyjne;
- restartowanie serwerów, aplikacji lub usług;
- generowanie plików konfiguracyjnych z szablonów;
- uruchamianie automatycznych testów dymnych (ang. *smoke tests*) w celu sprawdzenia, czy system jest poprawnie skonfigurowany i działa prawidłowo;
- uruchamianie procedur testowych;
- tworzenie skryptów i automatyzacja migracji bazy danych.

Jeśli to możliwe, można zmodyfikować architekturę, aby usunąć niektóre kroki — zwłaszcza te, których wykonanie zajmuje dużo czasu. Aby zredukować liczbę błędów i utratę wiedzy, warto również nie tylko skrócić czasy realizacji, ale także zdecydowanie zmniejszyć liczbę przełączeń odpowiedzialności.

Skierowanie uwagi programistów na automatyzację i optymalizację procesu wdrażania może prowadzić do znacznego usprawnienia przepływu. Na przykład można zadbać o to, aby niewielkie zmiany konfiguracji w aplikacji nie wymagały nowych wdrożeń lub nowych środowisk.

Wymaga to jednak ścisłej współpracy działu Dev z działem Ops. Taka współpraca powinna zagwarantować możliwość wielokrotnego wykorzystania wszystkich wspólnie tworzonych narzędzi i procesów, w przeciwieństwie do alienacji działu operacji lub „odkrywania koła na nowo”.

Wiele narzędzi zapewniających ciągłą integrację i testowanie daje również możliwość rozszerzania potoku wdrożeń. Dzięki temu zweryfikowane komplikacje mogą być promowane do produkcji. Zwykle dzieje się to po wykonaniu produkcyjnych testów akceptacyjnych (należą do nich na przykład wtyczka Jenkins Build Pipeline, ThoughtWorks Go.cd i Snap CI, Microsoft Visual Studio Team Services oraz Pivotal Concourse).

Do wymagań potoku wdrożeń można zaliczyć:

- **Wdrażanie w każdym środowisku w taki sam sposób** — dzięki użyciu tego samego mechanizmu wdrażania dla każdego środowiska (np. programistycznego, testowego i produkcyjnego) wdrożenia produkcyjne z większym prawdopodobieństwem zakończą się sukcesem, ponieważ wiemy, że przeprowadzono je pomyślnie już wiele razy we wcześniejszych fazach potoku wdrożeń.
- **Testy dymne wdrożeń** — podczas procesu wdrażania należy przetestować możliwość nawiązania połączenia z wszelkimi systemami pomocniczymi (np. bazami danych, magistralami komunikatorów, usługami zewnętrznymi), a następnie uruchomić pojedynczą transakcję testową w systemie, aby upewnić się, że system działa zgodnie z projektem. Jeśli którykolwiek z przeprowadzonych testów nie przejdzie, to będzie to oznaczało niepowodzenie wdrożenia.

- **Utrzymywanie spójnych środowisk** — w poprzednich krokach stworzyliśmy jednoetapowy proces budowania środowiska, dzięki któremu środowiska programistyczne, testowe i produkcyjne mają wspólny mechanizm komplikacji. Trzeba stale dbać o to, aby środowiska te pozostały zsynchronizowane.

Oczywiście gdy podczas wdrażania pojawią się jakiekolwiek problemy, możemy pociągnąć za linkę Andon i zastosować technikę swarmingu, by rozwiązać problem — w podobny sposób reagowaliśmy w przypadku awarii potoku wdrożeń w dowolnym z wcześniejszych etapów.

Studium przypadku

Codzienne wdrożenia w firmie CSG International (2013)

Firma CSG International to jedna z największych amerykańskich firm zajmujących się drukowaniem rozliczeń. Scott Prugh, ich główny architekt i wiceprezes ds. rozwoju, w celu poprawy przewidywalności i niezawodności wydań oprogramowania podwoił częstotliwość publikacji — z dwóch do czterech w ciągu roku (zmniejszając przy tym interwał wdrażania z 28 do 14 tygodni).

Chociaż zespoły programistów stosowały ciągłą integrację do codziennych wdrożeń kodu w środowiskach testowych, to wydania produkcyjne były realizowane przez zespół operacji. Prugh zaobserwował: „Było to tak, jakbyśmy mieli »zespół treningowy«, który trenuje codziennie (lub nawet częściej) w środowiskach testowych niskiego ryzyka, doskonalać procesy i narzędzia. Ale produkcyjny »zespół meczowy« ma bardzo niewiele okazji do treningu — zaledwie dwa razy w roku. Co gorsza, treningi są prowadzone w obarczonych wysokim ryzykiem środowiskach produkcyjnych, które często bardzo różnią się w stosunku do środowisk przedprodukcyjnych wieloma ograniczeniami — środowiskom programistycznym brakuje aktywów produkcyjnych, takich jak zabezpieczenia, zapory firewall, mechanizmy równoważenia obciążenia czy systemy sieci SAN”.

Aby rozwiązać ten problem, stworzono zespół operacji współdzielonych (ang. *Shared Operations Team* — **SOT**), któremu powierzono zadanie zarządzania wszystkimi środowiskami (programistycznym, testowym i produkcyjnym), realizującym codziennie wdrożenia w środowiskach deweloperskich i testowych, jak również realizującym wdrożenia produkcyjne i wydania co 14 tygodni. Ponieważ zespół SOT wykonywał wdrożenia codziennie, to wszystkie napotkane problemy, które pozostały nierozerwane, po prostu występowały ponownie następnego dnia. To stworzyło ogromną motywację do zautomatyzowania żmudnych i stwarzających okazje do popełnienia błędów ręcznych działań oraz do naprawienia wszelkich problemów, które mogły się powtórzyć. Ponieważ przed wdrożeniem produkcyjnym przeprowadzano wdrożenia prawie 100 razy, większość problemów można było znaleźć i rozwiązać dużo wcześniej.

W ten sposób ujawniano problemy, na które wcześniej napotykał wyłącznie zespół Ops, a później ich rozwiązaniem zajmowali się wszyscy należący do strumienia wartości. Codzienne wdrożenia dostarczały informacji zwrotnych na temat tych praktyk, które się sprawdzają, oraz tych, które się nie sprawdzają.

Dodatkowo koncentrowano się na tym, aby wszystkie środowiska wyglądały podobnie, włącznie z ograniczonymi prawami dostępu i mechanizmami równoważenia obciążenia. Prugh pisał: „Środowiska nieprodukcyjne upodobniliśmy maksymalnie do produkcyjnych oraz dążyliśmy do emulowania jak największej liczby ograniczeń produkcyjnych. Wczesne skonfrontowanie ze środowiskami produkcyjnymi spowodowało modyfikację projektów architektury, tak aby stały się bardziej przyjazne w różnych środowiskach obarczonych wieloma ograniczeniami. Dzięki zastosowaniu tego podejścia wszyscy stali się mądrzejsi”.

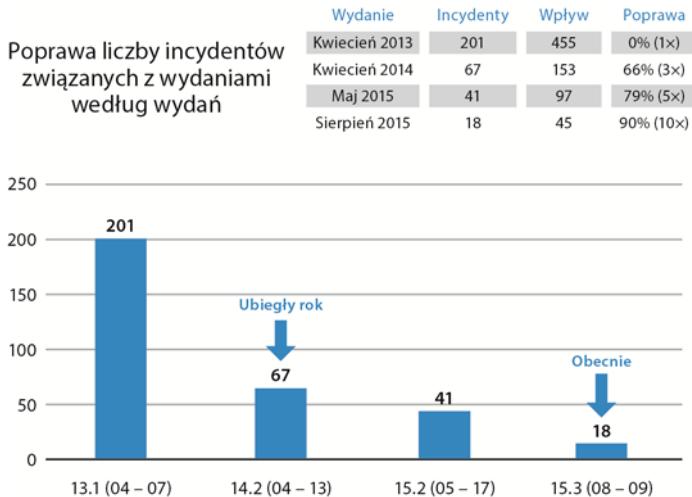
Prugh zauważał również:

„Zaobserwowałyśmy wiele przypadków, w których zmiany w schematach baz danych albo 1) były przekazywane do zespołów DBA z poleceniem »zobaczcie, co tam się dzieje«, albo 2) były poddawane automatycznym testom bazującym na nierealistycznie małych zestawach danych (czyli np. 100 MB zamiast 100 GB), co prowadziło do awarii w produkcji. W naszym starym sposobie pracy sytuacja ta sprowadzała się do wzajemnego przerzucania odpowiedzialności pomiędzy zespołami oraz podejmowania prób złagodzenia skutków chaosu. Stworzyliśmy proces rozwoju i wdrażania, w którym usunięto potrzebę przekazywania pracy do administratorów DBA. Osiągnięto to dzięki dodatkowemu szkoleniu programistów, automatyzacji zmian schematu i uruchamianiu automatycznych skryptów codziennie. Stworzyliśmy testy uruchamiane pod realistycznymi obciążeniami z wykorzystaniem okrojonych danych klientów, najlepiej z codziennymi migracjami. W ten sposób przed sprawdzeniem systemu w warunkach rzeczywistego ruchu produkcyjnego uruchamiamy usługi setki razy w realistycznych scenariuszach”*.

Uzyskane wyniki były zadziwiające. Dzięki stosowaniu codziennych wdrożeń i podwojeniu częstotliwości wydań produkcyjnych liczba zdarzeń w produkcji zmniejszyła się o 91%, wskaźnik MTTR spadł o 80%, a czas realizacji wdrożenia wymagany do tego, aby usługa działała w produkcji w stanie „w pełni bezobsługowym”, uległ skróceniu z 14 dni do jednego dnia (rysunek 17).

Jak informował Prugh, wdrożenia stały się tak rutynowe, że członkowie zespołu Ops po pierwszym dniu mogli grać w gry wideo. Oprócz płynniejszej realizacji wdrożeń przez zespoły Dev i Ops w 50% przypadków klienci otrzymywali wartość

* Eksperymenty udowodniły, że zespoły SOT odnosili sukces niezależnie od tego, czy były zarządzane przez dział Dev, czy Ops. Warunkiem koniecznym było obsadzenie zespołów SOT odpowiednimi ludźmi, konsekwentnie dającymi do sukcesu.



Rysunek 17. Codzienne wdrożenia i zwiększenie częstotliwości wydań spowodowały zmniejszenie liczby incydentów produkcji i wskaźnika MTTR (źródło: „DOES15 — Scott Prugh i Erica Morrison — Conway & Taylor Meet the Strangler (v2.0)”, nagranie wideo w serwisie YouTube, 29:39, opublikowane na konferencji DevOps Enterprise Summit, 5 listopada 2015, <https://www.youtube.com/watch?v=tKdIHCL0DUg>)

w ciągu połowy czasu. To jeszcze wyraźniej pokazuje, że częste wdrożenia mogą przynieść korzyści działom Dev, validacji, jak również klientom.

WDROŻENIA AUTOMATYCZNE I SAMOOBSŁUGOWE

Rozważmy poniższy cytat z wypowiedzi Tim'a Tischlera, dyrektora ds. automatyzacji operacji w firmie Nike, Inc., który opisuje typowe doświadczenia pokolenia programistów: „W mojej karierze programiści najbardziej satysfakcjonujące momenty były wtedy, gdy napisałem kod, wcisnąłem przycisk, aby go wdrożyć, gdy mogłem zobaczyć, jak wskaźniki produkcji potwierdzają, że kod rzeczywiście działa, oraz kiedy mogłem go naprawić, jeśli tak nie było”.

Zapewnienie programistom możliwości samodzielnego wdrażania kodu do produkcji, aby móc szybko zobaczyć zadowolonych klientów, gdy ich funkcja działa, albo szybko rozwiązać wszelkie problemy bez konieczności otwierania zlecenia pracy do działu operacji, straciło na znaczeniu w ostatnim dziesięcioleciu — częściowo z powodu potrzeby kontroli i nadzoru, której źródłem często są wymagania dotyczące bezpieczeństwa i zgodności z przepisami.

W rezultacie powszechną praktyką jest realizowanie wdrożeń kodu przez dział operacji. Podział obowiązków jest powszechnie akceptowaną praktyką realizowaną w celu zmniejszenia zagrożeń związanych z przestojami w produkcji lub oszustwami.

Jednak aby osiągnąć wyniki DevOps, trzeba postawić sobie cel zmiany mechanizmów kontroli, które mogą złagodzić wymienione zagrożenia w równy lub nawet bardziej skuteczny sposób. Mogą nimi być automatyczne testowanie, automatyczne wdrażanie i przeglądanie zmian.

Z raportu Puppet Labs *2013 State of DevOps Report*, do którego opracowania przeprowadzono ankietę wśród ponad 4000 specjalistów technicznych, wynika, że nie istnieją statystycznie znaczące różnice współczynnika sukcesu zmian pomiędzy firmami, w których kod był wdrażany przez dział Dev, a tymi, gdzie wdrożeniami zajmował się dział Ops.

Innymi słowy, gdy istnieją wspólne cele realizowane przez działy Dev i Ops oraz gdy istnieje przejrzystość, odpowiedzialność i możliwość rozliczania wyników wdrażania, nie ma znaczenia, kto wykonuje wdrożenia. Zadania wdrażania można nawet powierzyć osobom odgrywającym inne role. Na przykład testerom lub menedżerom produktu można zapewnić możliwość wdrażania do określonych środowisk po to, by mogli szybko wykonać takie prace, jak konfigurowanie demonstracji określonych funkcji w środowisku testów QA lub testów akceptacyjnych użytkownika.

Dla ułatwienia szybkiego przepływu potrzebny jest proces promocji kodu, który może być zrealizowany przez dział Dev albo Ops — idealnie, bez żadnych działań ręcznych i bez przekazywania pracy. Ma to wpływ na następujące etapy:

- **Kompilacja** — potok wdrożeń na podstawie repozytorium kontroli wersji powinien tworzyć pakiety, które można wdrożyć w każdym środowisku, w tym w środowisku produkcyjnym.
- **Testowanie** — każdy powinien mieć możliwość uruchamiania dowolnego testu lub wszystkich testów z zestawu testów automatycznych na swojej stacji roboczej lub na dedykowanych systemach testowych.
- **Wdrażanie** — każdy powinien mieć możliwość wdrażania tych pakietów w każdym środowisku, do którego ma dostęp. Powinny one być tam uruchamiane za pomocą skryptów, które razem z kodem powinny zostać zaewidencjonowane w repozytorium kontroli wersji.

Są to praktyki, które umożliwiają pomyślną realizację wdrożeń, niezależnie od tego, kto je realizuje.

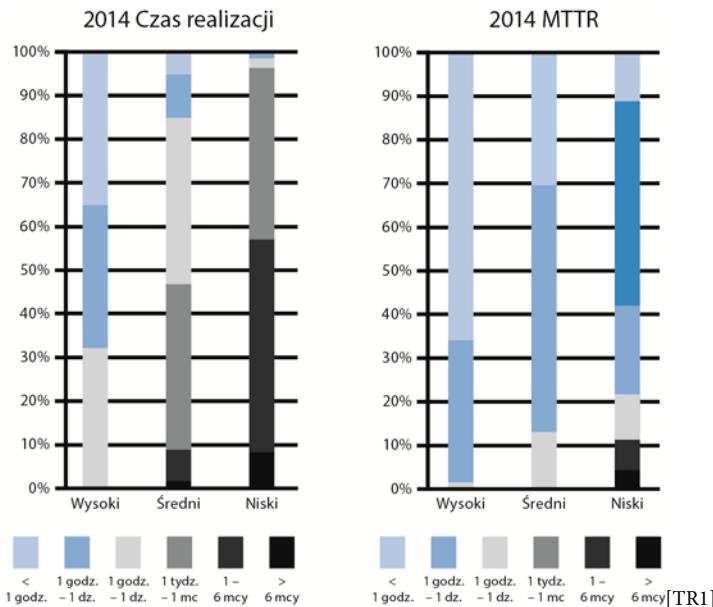
INTEGRACJA ZADAŃ WDRAŻANIA KODU Z POTOKIEM WDROŻEŃ

Gdy proces wdrażania kodu jest zautomatyzowany, to może być częścią potoku wdrożeń. W konsekwencji automatyzacja wdrażania powinna zapewniać następujące możliwości:

- gwarantować zdolność wdrożenia do produkcji pakietów stworzonych w procesie ciągłej integracji;
- błyskawiczną ocenę gotowości środowisk produkcyjnych;
- dostarczać „przycisk wdrażania” — samoobsługowe wdrażanie do produkcji dowolnej wersji kodu zapisanej w pakiecie instalacyjnym;
- automatyczny zapis — dla celów nadzoru i zgodności z przepisami — polecień uruchomionych na wskazanych maszynach wraz z informacjami o tym, kiedy zostały uruchomione, kto dokonał autoryzacji i jaki był ich wynik;
- uruchamianie testów dymnych, które powinny zweryfikować prawidłowe działanie systemu oraz poprawność ustawień konfiguracji, włącznie z takimi elementami, jak ciągi połączenia z bazami danych;
- zapewnienie szybkiej pętli sprzężenia zwrotnego dla osoby realizującej wdrożenie, tak by można było szybko sprawdzić, czy wdrożenie zakończyło się pomyślnie (np. czy wdrożenie się udało, czy aplikacja działa zgodnie z oczekiwaniemi w środowisku produkcyjnym itp.).

Celem jest zapewnienie szybkich wdrożeń — nie chcemy czekać wielu godzin, aby ustalić, czy wdrożenie kodu powiodło się, czy zakończyło porażką, a następnie potrzebować wielu godzin, aby wdrożyć potrzebne poprawki w kodzie. Obecnie, gdy dysponujemy takimi technologiami, jak kontenery, mamy możliwość realizacji nawet najbardziej złożonych wdrożeń w ciągu kilku sekund lub minut. Z raportu *State of DevOps Report 2014* firmy Puppet Labs wynika, że w firmach o wysokiej wydajności czasy realizacji wdrożeń były mierzone w minutach lub godzinach, a w firmach o najniższej wydajności ten czas mierzono w miesiącach (rysunek 18).

Dzięki stworzeniu tej możliwości będziemy mieli przycisk „wdrażanie kodu”, który pozwoli szybko i bezpiecznie wypromować zmiany w kodzie i środowisku do produkcji za pośrednictwem potoku wdrożeń.



Rysunek 18. Firmy o wysokiej wydajności mają znacznie szybsze czasy realizacji wdrażania i znacznie szybszy czas przywrócenia usług produkcyjnych po incydentach (źródło: Puppet Labs, State of DevOps Report)

Studium przypadku

Etsy — samoobsługowe wdrażanie przez programistów. Przykład techniki ciągłego wdrażania (2014)

Inaczej niż w firmie Facebook, gdzie wdrożenia były zarządzane przez inżynierów wydania, w firmie Etsy są wykonywane przez każdego, kto chce wdrożenia — może to być inżynier działu Dev, Ops lub specjalista Infosec. Proces wdrażania w firmie Etsy stał się tak bezpieczny i rutynowy, że nowi inżynierowie potrafią wykonywać wdrożenia w środowisku produkcyjnym w swoim pierwszym dniu pracy. Potrafią to zrobić nawet członkowie zarządu firmy Etsy, a nawet psy!

Jak napisał Noah Sussman, architekt testów w firmie Etsy: „Jeśli około 8 rano zaczyna się normalny dzień roboczy, to około 15.00 tego dnia ludzie i psy zaczynają ustawać się w kolejkę. Wszyscy mają nadzieję kolejtywnie wdrożyć przed końcem dnia do 25 zmian kodu”.

Inżynierowie, którzy chcą wdrożyć swój kod, najpierw logują się na czacie, gdzie samodzielnie dodają się do kolejki wdrożeń, obserwując zadania wdrażania w toku, patrząc, kto oprócz nich jest w kolejce, rozgłasza swoje aktywności i korzystając z pomocy innych inżynierów, gdy tego potrzebują. Gdy nadziejde kolej określonego inżyniera na wdrożenie jego kodu, otrzymuje on powiadomienie na kanale czatu.

W firmie Etsy wyznaczono cel, aby wdrożenia do produkcji były łatwe i bezpieczne, aby wymagały jak najmniej kroków i jak najmniej ceremonii. Prawdopodobnie jeszcze przed zaewidencjonowaniem kodu w repozytorium kontroli wersji programista uruchomił na swojej stacji roboczej wszystkie 4500 testów jednostkowych. Ich uruchomienie zajęło mniej niż minutę. Wszystkie wywołania do systemów zewnętrznych, na przykład baz danych, zostały w nich zaślepione.

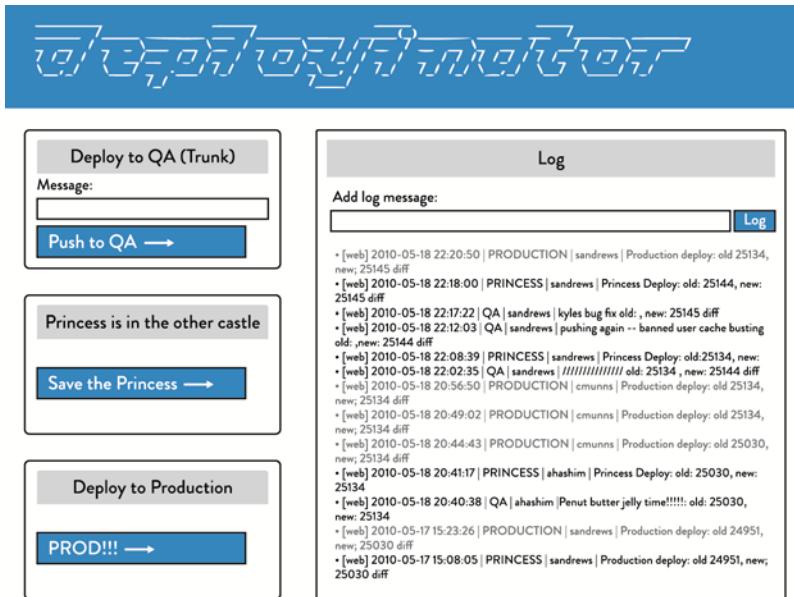
Po scaleniu zmian z rewizją master w repozytorium kontroli wersji uruchamianych jest natychmiast na serwerach ciągłej integracji (CI) ponad 7000 automatycznych testów rewizji master. Sussman pisze: „Metodą prób i błędów wyznaczyliśmy czas około 11 minut jako najdłuższy dopuszczalny czas uruchomienia automatycznych testów podczas wdrożenia. To pozostawia czas na ponowne uruchomienie testów podczas wdrażania [jeśli ktoś coś zepsuje i chce to naprawić], bez zbytniego wychodzenia poza 20-minutowy limit”.

Gdyby wszystkie testy były uruchamiane sekwencyjnie, to według Sussmana „uruchomienie 7000 testów rewizji master zajęłoby jakieś pół godziny. Z tego powodu dzielimy testy na mniejsze zbiory i rozdzielimy je na 10 maszyn w klasztrze Jenkins [CI]... Dzięki podzieleniu zestawu testów i uruchamianiu równolegle wielu testów uzyskaliśmy pożądany czas 11 minut”.

Kolejne do uruchomienia są testy dymne — testy systemowe, które korzystają z biblioteki cURL w celu uruchomienia przypadków testowych PHPUnit. Po tych testach uruchamiane są testy funkcjonalne. Są to testy „od końca do końca” korzystające z GUI oraz aktywnego serwera. Jest to serwer ze środowiskiem QA lub środowiskiem przedprodukcyjnym (o przydomku „Princess”), który faktycznie jest zapasowym serwerem produkcyjnym rotacyjnie wymienianym z innymi. To daje pewność, że serwer ten dokładnie imituje środowisko produkcyjne.

Kiedy nadchodzi kolej inżyniera na wdrożenie, wtedy — jak pisze Erik Kastner — „uruchamiasz Deployinator [wewnętrznie opracowane narzędzie, rysunek 19] i wciskasz przycisk, aby poddać kod weryfikacji QA. Stamtaąd kod idzie na serwer Princess... Następnie, gdy jest gotowy do wdrożenia do produkcji, możesz naciągnąć przycisk Prod. Wkrótce po tym twój kod jest wdrożony, a wszyscy na IRC [kanale czatu] dowiadują się o tym, kto wgrał ten kod, a także otrzymują łącze do różnic w porównaniu z rewizją master. Do wszystkich, którzy nie są na IRC, wysyłany jest e-mail zawierający te same informacje”.

W 2009 roku proces wdrażania w firmie Etsy był przyczyną stresu i strachu. Do 2011 roku stał się operacją rutynową, wykonywaną od 20 do 25 razy dziennie, która pomaga inżynierom w szybkim dostarczeniu kodu do produkcji oraz wartości dla klientów.



Rysunek 19. Konsola programu Deployinator w firmie Etsy (źródło: Erik Kastner, „Quantum of Deployment”, CodeasCraft.com, 20 maja 2010, <https://codeascraft.com/2010/05/20/quantum-of-deployment/>)

ODDZIENIE WDROŻEŃ OD WYDAŃ

W tradycyjnych projektach oprogramowania wydania są sterowane datą wydania wskazywaną przez dział marketingu. Poprzedniego wieczora wdrożyliśmy do produkcji kompletne oprogramowanie (lub maksymalnie zbliżone do kompletnego). Następnego dnia rano ogłosiliśmy nowe możliwości świata, zaczeliśmy przyjmowanie zamówień, dostarczanie nowej funkcjonalności do klientów itp.

Jednak bardzo często nie wszystko przebiega zgodnie z planem. Obciążenia produkcyjne mogą być wyższe, niż kiedykolwiek testowaliśmy lub uwzględnialiśmy w projekcie, co może być przyczyną spektakularnej awarii naszej usługi — zarówno po stronie klientów, jak i naszej firmy. Co gorsza, przywrócenie usługi może wymagać kłopotliwego procesu wycofywania lub równie ryzykownej operacji poprawki „w przód”, gdzie zmiany są wprowadzane bezpośrednio w produkcji. Sytuacje te mogą być źródłem wielu złych doświadczeń dla pracowników. Gdy w końcu wszystko zacznie działać, wszyscy oddychają z ulgą, zadowoleni z tego, że wdrożenia do produkcji i wydania nie są realizowane częściej.

Oczywiście wiemy, że aby osiągnąć wymarzony cel, jakim jest sprawny i szybki przepływ, trzeba realizować wdrożenia częściej, a nie rzadziej. Aby to było możliwe, trzeba oddzielić wdrożenia produkcyjne od publikacji funkcjonalności. W praktyce

terminy **wdrożenie** (ang. *deployment*) i **wydanie** (ang. *release*) często są używane zamiennie. Są to jednak dwa różne działania, które służą dwóm bardzo różnym celom:

- Wdrożenie to instalacja określonej wersji oprogramowania w danym środowisku (np. wdrożenie kodu w zintegrowanym środowisku testowym lub wdrożenie kodu do produkcji). W szczególności wdrożenie może, lecz nie musi, być powiązane z opublikowaniem funkcjonalności klientom.
- Wydanie jest procesem, za pomocą którego cecha funkcjonalna (lub zestaw cech funkcjonalnych) są udostępniane wszystkim klientom lub określonym segmentowi klientów (np. uaktywniamy cechę funkcjonalną, która będzie wykorzystywana przez 5% naszej bazy klientów). Kod i środowiska powinny być zaprojektowane w taki sposób, aby wydanie funkcjonalności nie wymagało zmiany w kodzie aplikacji*.

Innymi słowy, gdy pomieszamy wdrażanie z wydawaniem, to powierzenie odpowiedzialności za pomyślne wyniki może okazać się trudne. Rozdzielenie tych dwóch działań pozwala uczynić działa Dev i Ops odpowiedzialnymi za sukces szybkich i częstych wdrożeń, a jednocześnie umożliwia powierzenie właścicielom produktu odpowiedzialności za pomyślne wyniki biznesowe wydania (tzn. za udzielenie odpowiedzi na pytanie czy budowanie i uruchamianie cechy funkcjonalnej było warte poświęconego czasu).

Stosowanie praktyk opisanych dotychczas w tej książce pozwala zyskać pewność wykonywania szybkich i częstych wdrożeń produkcyjnych za pośrednictwem rozwoju cech funkcjonalnych. Celem tych działań jest zminimalizowanie zagrożeń i ujemnego wpływu błędów wdrażania. Pozostałe ryzyko jest związane z wydaniem — tzn. zadaniem o to, aby cechy funkcjonalne przekazane do produkcji pozwoliły osiągnąć pożądany wpływ na klientów oraz wyniki biznesowe.

Bardzo długie czasy realizacji wdrażania wpływają na niską częstotliwość publikowania nowych cech funkcjonalnych na rynku. Jednak jeśli uzyskamy zdolność wdrażania na żądanie, to tempo eksponowania nowych funkcjonalności klientom stanie się decyją biznesową i marketingową, a nie decyją techniczną. Istnieją dwie szerokie kategorie wzorców wydania, których możemy użyć:

- **Wzorce wydań bazujące na środowisku** — w tym modelu mamy dwa środowiska lub więcej środowisk, w których możemy realizować wdrożenia, ale tylko do jednego środowiska dociera „żywy” ruch od klientów (np. dzięki skonfigurowaniu mechanizmów równoważenia obciążenia). Nowy kod jest wdrażany do środowiska nieaktywnego, a wydanie jest realizowane poprzez przełączenie

* Za skuteczną metaforę może służyć operacja „Pustynna Tarcza”. Począwszy od 7 sierpnia 1990 roku, tysiące mężczyzn i materiałów bezpiecznie wdrożono na cztery miesiące do teatru produkcyjnego. Kulminacją było jedno, multidyscyplinarne, wysoce skoordynowane wydanie.

ruchu klientów do tego środowiska (uaktywnienie go). Wzorce te dają bardzo duże możliwości, ponieważ zwykle wymagają niewiele lub nie wymagają żadnych zmian w aplikacji. Do tego rodzaju wzorców można zaliczyć **wdrożenia niebieski-zielony, wydania kanarkowe** oraz systemy **CIS** (ang. *cluster immune systems* — dosł. „klastrowe systemy odpornościowe”), które omówimy wkrótce.

- **Wzorce wydań bazujące na aplikacjach** — polegają na modyfikowaniu aplikacji w taki sposób, aby można było selektywnie wydawać i eksponować specyficzne funkcjonalności aplikacji za pomocą niewielkich zmian w konfiguracji. Na przykład możemy zaimplementować flagi funkcjonalności, które stopniowo eksponują nowe funkcjonalności pracownikom działu Dev, wszystkim wewnętrznym pracownikom, 1% klientów lub gdy jesteśmy pewni, że wydanie będzie działać zgodnie z oczekiwaniami — całej bazie klientów. Jak wspomniano wcześniej, umożliwia to zastosowanie techniki zwanej **ślepym uruchomieniem** (ang. *dark launching*), gdzie przed wydaniem wszystkie funkcjonalności są poddawane procesowi określanemu jako **staging**. Na przykład możemy w niewidoczny sposób przetestować nowe funkcjonalności z obciążeniem produkcyjnym na wiele tygodni przed wydaniem. W ten sposób można wykryć problemy i rozwiązać je przed faktyczną publikacją.

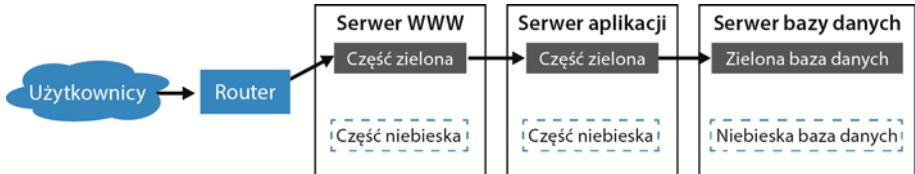
WZORCE WYDAŃ BAZUJĄCE NA ŚRODOWISKU

Oddzielenie wdrożeń od wydań dramatycznie zmienia sposób naszej pracy. Nie trzeba już przeprowadzać wdrożeń w środku nocy lub w weekendy, aby zmniejszyć ryzyko negatywnego wpływu wydania na klientów. Zamiast tego wdrożenia są realizowane w zwykłych godzinach pracy. Dzięki temu pracownicy działu Ops mogą mieć正常的 godziny pracy, podobnie jak wszyscy inni.

W tym podrozdziale skoncentrujemy się na wzorcach bazujących na środowisku, które nie wymagają wprowadzania zmian w kodzie aplikacji. W tym celu utrzymujemy wiele środowisk, do których wdrażamy kod, ale tylko do jednego z nich „na żywo” dociera ruch klientów. W ten sposób możemy znacznie zmniejszyć zagrożenia związane z wydaniami produkcyjnymi oraz skrócić czas realizacji wdrożeń.

WZORZEC WDRAŻANIA NIEBIESKIE-ZIELONE

Najprostszy spośród trzech wzorców z tej grupy nosi nazwę wdrożenia niebieskie-zielone. W tym wzorcu mamy dwa środowiska produkcyjne: niebieskie i zielone. W każdym momencie tylko jedno z nich obsługuje ruch klienta — na rysunku 20. aktywne jest środowisko zielone.



Rysunek 20. Wzorzec wdrożenia niebieskie-zielone (źródło: Humble i North, „Continuous Delivery”, s. 261)

W celu wydania nowej wersji usługi realizujemy wdrożenie do nieaktywnego środowiska, w którym możemy przeprowadzić testy bez przerywania świadczenia usługi klientom. Gdy jesteśmy pewni, że wszystko działa zgodnie z projektem, inicjujemy wydanie poprzez skierowanie ruchu do środowiska niebieskiego. W ten sposób środowisko niebieskie staje się aktywne, a zielone staje się tymczasowe. Wycofanie zmian polega na skierowaniu ruchu z powrotem do środowiska zielonego*. Wzorzec wdrażania niebieskie-zielone jest prosty oraz niezwykle łatwy do zastosowania w istniejących systemach. Ma również niesamowite zalety — na przykład pozwala zespołowi przeprowadzać wdrożenia w zwykłych godzinach pracy oraz wprowadzać proste zmiany (np. zmiana ustawień routera, zmiana dowiązania symbolicznego) w godzinach poza szczytem. Tylko te korzyści pozwalają dramatycznie poprawić warunki działania zespołu realizującego wdrożenie.

OBSŁUGA ZMIAN W BAZIE DANYCH

Posługiwanie się dwoma wersjami aplikacji w produkcji stwarza problemy w przypadku, gdy zależą one od wspólnej bazy danych. Jeśli wdrożenie wymaga wprowadzenia zmian w schemacie bazy danych albo dodania modyfikacji lub usunięcia tabel bądź kolumn, to baza danych nie jest w stanie obsłużyć obu wersji aplikacji. Istnieją dwa uniwersalne podejścia do rozwiązywania tego problemu:

* Istnieją także inne sposoby wykorzystania wzorca niebieskie-zielone. Można skonfigurować wiele serwerów Apache/NGINX nasłuchujących na różnych fizycznych lub wirtualnych interfejsach; wykorzystać wiele wirtualnych katalogów głównych na serwerach Windows IIS powiązanych z różnymi portami. W takiej konfiguracji każdy katalog jest przypisany do innej wersji systemu, a dowiązanie symboliczne określa, która z nich jest aktywna (np. tak jak za pomocą systemu Capistrano w środowisku Ruby on Rails). Można też równolegle uruchomić wiele wersji usługi lub oprogramowania middleware, gdzie każda wersja nasłuchuje na innym porcie. Można skorzystać z dwóch różnych centrów danych i przełączać pomiędzy nimi ruch, zamiast używać ich jedynie w roli części zamiennych „na gorąco” lub „na zimno” w celu odzyskiwania po awarii (nawiasem mówiąc, rutynowe korzystanie z obu środowisk potwierdza działanie procesu odzyskiwania zgodnie z założeniami). Jeszcze innym sposobem jest korzystanie z różnych stref dostępności w chmurze.

- **Utworzenie dwóch baz danych (czyli niebieskiej i zielonej)** — każda wersja aplikacji — niebieska (stara) i zielona (nowa) — ma własną bazę danych. Podczas wydania przełączamy niebieską bazę danych na tryb tylko do odczytu, wykonujemy jej kopię zapasową, przywracamy do stanu zielonej bazy danych i na koniec przełączamy ruch do zielonego środowiska. Problem z tym wzorcem polega na tym, że w przypadku konieczności cofnięcia do wersji niebieskiej możemy stracić transakcje, jeśli wcześniej nie przeprowadzimy ręcznej migracji z wersji zielonej.
- **Oddzielenie zmiany bazy danych od zmian w aplikacji** — zamiast obsługiwać dwie bazy danych, oddzielamy publikację zmian w bazie danych od wydania zmian w aplikacji, wykonując dwie rzeczy: po pierwsze, wprowadzamy do bazy danych wyłącznie zmiany przyrostowe; nigdy nie mutujemy istniejących obiektów bazy danych; i po drugie, przyjmujemy w aplikacji założenie dotyczące wersji bazy danych używanej w produkcji. Takie podejście bardzo różni się od tradycyjnej nauki o bazach danych, gdzie staramy się unikać dublowania danych. Proces oddzielenia zmian w bazie danych od zmian w aplikacji zastosowała (między innymi) firma IMVU około roku 2009. Dzięki temu stworzono możliwości realizacji 50 wdrożeń dziennie, spośród których niektóre wymagały zmian w bazie danych*.

Studium przypadku

Dixons Retail — wdrożenia niebieskie-zielone dla systemu POS (2008)

Dan North i Dave Farley, współautorzy książki *Continuous Delivery*, pracowali nad projektem w Dixons Retail — dużej brytyjskiej firmie handlowej, korzystającej z tysięcy systemów **POS** (ang. *point-of-sale*) zainstalowanych w setkach sklepów detalicznych i działających pod wieloma różnymi markami klienta.

* Ten wzorzec jest również powszechnie znany pod nazwą „rozwiń-zwiń”, który Timothy Fitz opisał słowami „nie zmieniamy (nie mutujemy) takich obiektów bazy danych, jak kolumny lub tabele. Zamiast tego najpierw je rozwijamy poprzez dodawanie nowych obiektów, a następnie zwijamy poprzez usunięcie starych”. Ponadto coraz częściej stosowane są technologie, które umożliwiają wirtualizację, wersjonowanie, etykietowanie i wycofywanie zmian w bazach danych, takie jak Redgate, Delphix, DBMaestro i Datical, a także narzędzia open source, takie jak DBDeploy, dzięki którym wprowadzanie zmian w bazach danych jest znacznie bezpieczniejsze i szybsze.

Chociaż wdrożenia niebieskie-zielone są najczęściej związane z usługami webowymi online, to North i Farley korzystał z tego wzorca w celu znacznego skrócenia czasu aktualizacji systemów POS oraz wyeliminowania związanych z tym zagrożeń.

W tradycyjnym podejściu aktualizacja systemów POS odbywała się zgodnie z modelem big bang, typowym dla kaskadowej metody wytwarzania oprogramowania — klienci POS i centralizowany serwer były aktualniane w tym samym czasie. Aby nowe oprogramowanie klienckie trafiło do wszystkich punktów detalicznych, potrzebne były długie przestoje (często trwające cały weekend), a także znacząca przepustowość sieci. Jakiekolwiek zakłócenia w stosunku do planu skutkowały problemami z działaniem sklepów.

Przy aktualizacji opisywanej w tym studium przypadku nie było wystarczającej przepustowości do jednoczesnego aktualnienia wszystkich systemów POS, dlatego zastosowanie tradycyjnej strategii było niemożliwe. W celu rozwiązania problemu skorzystano ze strategii niebieskie-zielone. Utworzono dwie wersje produkcyjne centralnego serwera oprogramowania, co pozwoliło na jednoczesne obsługiwanie starych i nowych wersji klientów POS. Następnie na kilka tygodni przed zaplanowaną aktualizacją POS zaczęto wysyłanie nowych wersji pakietów instalacyjnych oprogramowania POS do punktów detalicznych przez powolne łącza sieciowe. Dzięki temu wdrażano nowe oprogramowanie systemów POS w stanie nieaktywnym. W międzyczasie stara wersja działała tak jak zwykle.

Gdy dla wszystkich klientów POS było wszystko gotowe do przeprowadzenia aktualizacji (pomyślnie przeprowadzono wspólne testy aktualionego klienta i serwera, a nowe oprogramowanie klienckie zostało wdrożone do wszystkich klientów) i pozostało podjąć decyzję o wydaniu, to postanowiono, że zrobią to menedżerowie punktów sprzedaży.

W zależności od potrzeb biznesowych niektórzy menedżerowie chcieli natychmiast korzystać z nowych funkcji i przeprowadzili wydanie od razu, podczas gdy inni postanowili poczekać. W obu przypadkach — niezależnie od tego, czy nowe funkcjonalności były wydawane natychmiast, czy po pewnym czasie — było znacznie lepiej, gdy decyzję o wydaniu podejmowali menedżerowie, a nie centralny dział IT.

W rezultacie wydanie przebiegło znacznie łatwiej i szybciej, osiągnięto większe zadowolenie menedżerów punktów sprzedaży i odnotowano znacznie mniej zakłóceń w działalności tych punktów. Ponadto opisane powyżej zastosowanie wdrożenia niebieskie-zielone do aplikacji typu gruby klient PC pokazuje możliwość uniwersalnego zastosowania wzorców DevOps do różnych technologii — często w bardzo zaskakujący sposób, ale niekiedy z fantastycznymi wynikami.

WZORCE WYDAŃ KANARKOWYCH ORAZ SYSTEMY CIS

Wzorzec wydania niebieskie-zielone jest łatwy do zastosowania i może znacznie zwiększyć bezpieczeństwo wydań oprogramowania. Istnieją odmiany tego wzorca, które pozwalają jeszcze bardziej poprawić bezpieczeństwo i czasy realizacji wdrażania, ale potencjalnie wiążą się z dodatkową złożonością.

Wzorzec wydań kanarkowych automatyzuje proces wydań w zakresie promowania do coraz bardziej rozbudowanych i ważniejszych środowisk po potwierdzeniu, że kod działa zgodnie z przeznaczeniem.

Termin **wydanie kanarkowe** pochodzi z tradycji górników, którzy przywozili do kopalń kanarki w klatkach po to, by móc wcześniej wykrywać toksyczne poziomy tlenku węgla. Jeśli w szybie było zbyt duże stężenie gazu, zabijało kanarki, zanim zabiło górników. W ten sposób górnicy otrzymywali ostrzeżenie i mogli się ewakuować.

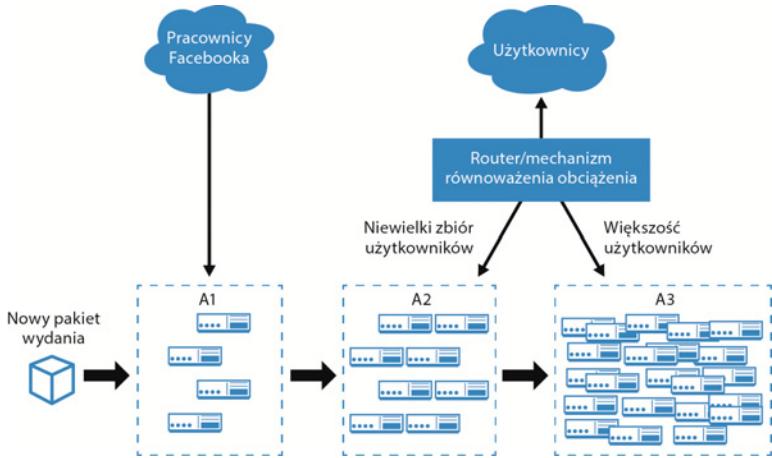
W tym wzorcu po przeprowadzeniu wydania monitorujemy, jak zachowuje się oprogramowanie w każdym środowisku. Gdy wydaje się, że coś dzieje się źle, można wycofać zmianę. W przeciwnym razie można przeprowadzić wdrożenie w kolejnym środowisku*.

Grupy środowisk stworzone w firmie Facebook w celu wsparcia tego wzorca wydania zaprezentowano na rysunku 21:

- **Grupa A1** — serwery produkcyjne, które obsługują wyłącznie wewnętrznych pracowników.
- **Grupa A2** — serwery produkcyjne, które obsługują tylko niewielki procent klientów i są wdrażane, gdy zostaną spełnione określone kryteria akceptacji (co jest weryfikowane automatycznie lub ręcznie).
- **Grupa A3** — pozostałe serwery produkcyjne, na których wdrożenie odbywa się wtedy, gdy oprogramowanie działające w klastrze A2 spełni określone kryteria akceptacyjne.

Wzorzec **CIS** (ang. *cluster immune system* — dosł. „układ odpornościowy w formie klastra”) jest rozszerzeniem kanarkowego wzorca wydań. W tym wzorcu z procesem wydawania jest powiązany system monitorowania produkcji. Dodatkowo wycofywanie publikacji jest zautomatyzowane i następuje, gdy wydajność systemu w produkcji pod obciążeniem odbiega od wstępnie zdefiniowanego zakresu — na przykład, gdy współczynnik konwersji dla nowych użytkowników spadnie poniżej historycznej normy 15 – 20%.

* Należy pamiętać, że stosowanie wzorca wydań kanarkowych wymaga równoczesnego uruchomienia w produkcji wielu wersji oprogramowania jednocześnie. Ponieważ jednak każda dodatkowa wersja w produkcji stwarza dodatkową złożoność i konieczność zarządzania nią, to należy dbać o utrzymanie liczby wersji na minimalnym poziomie. Może to wymagać skorzystania z opisanego wcześniej wzorca rozwijania-zwijania bazy danych.



Rysunek 21. Wzorzec wydania kanarkowego (źródło: Humble i Farley, „Continuous Delivery”, s. 263)

Są dwie znaczące korzyści ze stosowania tego rodzaju zabezpieczeń. Po pierwsze, zabezpieczamy się przed defektami, które są trudne do znalezienia za pomocą testów automatycznych — na przykład modyfikacja strony WWW, która powoduje, że jakiś kluczowy element strony stanie się niewidoczny (np. zmiana CSS). Po drugie, skracamy czas wymagany do wykrywania i reagowania na obniżoną wydajność spowodowaną przez wprowadzoną zmianę*.

WZORCE BAZUJĄCE NA APLIKACJI, POPRAWIAJĄCE BEZPIECZEŃSTWO WYDAŃ

W poprzednim podrozdziale omówiliśmy wzorce bazujące na środowisku, które pozwoliły na oddzielenie wdrożeń od wydań dzięki wykorzystaniu wielu środowisk oraz przełączaniu środowiska aktywnego. Wzorce te można w całości zaimplementować na poziomie infrastruktury.

W tym podrozdziale opisano wzorce wydań bazujące na aplikacji, które można zaimplementować w kodzie. Zapewnia to jeszcze większą elastyczność wdrażania nowych funkcjonalności dla klientów — często na poziomie pojedynczej funkcji. Ponieważ wzorce wydań bazujące na aplikacji są implementowane w aplikacji, to ich zastosowanie wymaga zaangażowania działu Dev.

* Wzorzec CIS został po udokumentowany raz pierwszy przez Erica Riesa, w okresie jego pracy w IMVU. Ta funkcjonalność jest również wspierana w firmie Etsy za pośrednictwem biblioteki Feature API, a także w firmie Netflix.

IMPLEMENTACJA PRZEŁĄCZNIKÓW FUNKCJI

Podstawowym sposobem zastosowania wzorców wdrożeń na bazie aplikacji jest implementacja **przełączników cech funkcjonalnych** (ang. *feature toggles*), które dostarczają mechanizmu selektywnego włączania lub wyłączania cech funkcjonalnych bez konieczności wdrażania kodu do produkcji. Wspomniane przełączniki pozwalają także zdecydować o tym, które cechy funkcjonalne są widoczne i dostępne dla określonych segmentów użytkowników (np. pracowników wewnętrznych lub wybranych segmentów klientów).

Przełączniki cech funkcjonalnych zazwyczaj są implementowane poprzez opakowanie logiki aplikacji lub elementów interfejsu użytkownika wewnętrz instrukcji warunkowych, a włączenie bądź wyłączenie cechy funkcjonalnej bazuje na zapisanym gdzieś ustawieniu konfiguracyjnym. Może to być prosty plik konfiguracyjny aplikacji (np. w formacie JSON lub XML), usługa katalogowa albo nawet usługa sieciowa specjalnie zaprojektowana do zarządzania włączaniem i wyłączeniem cech funkcjonalnych*.

Przełączniki cech funkcjonalnych umożliwiają również wykonywanie następujących operacji:

- **Łatwe wycofywanie** — cechy funkcjonalne, które stwarzają problemy lub zakłócenia w produkcji, mogą być szybko i bezpiecznie wyłączone za pośrednictwem prostej zmiany przełącznika funkcji. Jest to szczególnie przydatne, gdy wdrożenia są rzadkie — wyłączenie jednej konkretnej cechy funkcjonalnej jest zazwyczaj znacznie łatwiejsze niż wycofanie całego wydania.
- **Obniżanie wydajności „z wdziękiem”** — gdy obciążenie usługi jest bardzo wysokie — na tyle wysokie, że normalnie wymagałoby zwiększenia możliwości lub co gorsza, stwarzałoby zagrożenie awarii usługi w produkcji, możemy użyć przełączników cech funkcjonalnych w celu obniżenia jakości usługi. Innymi słowy, możemy zwiększyć liczbę obsługiwanych użytkowników poprzez obniżenie poziomu dostarczanych funkcjonalności (np. zmniejszenie liczby klientów, którzy mogą uzyskać dostęp do niektórych funkcji, wyłączenie funkcjonalności intensywnie korzystających z procesora, takich jak rekommendacje itp.).
- **Zwiększenie odporności dzięki wykorzystaniu architektury zorientowanej na usługi** — jeśli mamy cechę funkcjonalną bazującą na innej usłudze, która nie

* Zaawansowanym przykładem takiej usługi jest system Gatekeeper w firmie Facebook — opracowana wewnętrznie usługa, która dynamicznie wybiera funkcje widoczne dla określonych użytkowników na podstawie danych demograficznych, takich jak lokalizacja, typ przeglądarki, oraz danych profilu użytkownika (wiek, płeć itd.). Na przykład można skonfigurować określoną cechę funkcjonalną, tak aby była dostępna tylko dla pracowników wewnętrznych, 10% użytkowników lub tylko tych użytkowników, którzy są w wieku od 25 do 35 lat. Inne przykłady to Feature API w firmie Etsy oraz biblioteka Archaius w firmie Netflix.

jest jeszcze gotowa, to nadal możemy wdrożyć tę cechę funkcjonalną i ukryć ją za przełącznikiem. Gdy ta usługa ostatecznie stanie się dostępna, możemy włączyć zablokowaną cechę funkcjonalną. Na podobnej zasadzie, gdy wykorzystywana usługa ulegnie awarii, możemy wyłączyć cechę funkcjonalną korzystającą z usługi, aby zapobiec wywołaniom w dole strumienia, a jednocześnie zachować działanie pozostały części aplikacji.

Aby zachować możliwość znajdowania błędów w funkcjonalnościach opakowanych przełącznikami cech funkcjonalnych, należy uruchamiać automatyczne testy akceptacyjne z włączonymi przełącznikami (trzeba także sprawdzać, czy mechanizm włączania-wyłączania funkcjonalności działa poprawnie!).

Przełączniki cech funkcjonalnych pozwalają na oddzielenie wdrażania kodu od publikowania cech funkcjonalnych. W dalszej części książki skorzystamy z przełączników cech funkcjonalnych do zastosowania techniki **HDD** (ang. *hypothesis-driven development* — dosł. „wytwarzanie oprogramowania sterowane hipotezami”) oraz testowania A/B. Techniki te pozwalają jeszcze bardziej zwiększyć naszą zdolność do osiągania oczekiwanych wyników biznesowych.

ŚLEPE URUCHOMIENIA

Przełączniki cech funkcjonalnych pozwalają na wdrażanie cech funkcjonalnych do produkcji bez udostępniania ich użytkownikom za pośrednictwem techniki znanej jako **ślepe uruchomienia** (ang. *dark launching*). W tym trybie najpierw wdrażamy funkcjonalność do produkcji, a następnie przeprowadzamy jej testy w czasie, gdy jest ona nadal niewidoczna dla klientów. W przypadku dużych zmian lub zmian obarczonych ryzykiem często robimy to na wiele tygodni przed uruchomieniem produkcyjnym. To pozwala nam bezpiecznie testować funkcjonalności z przewidywanymi obciążeniami produkcyjnymi.

Załóżmy, że przeprowadzamy ślepe uruchomienie nowej cechy funkcjonalnej, która stwarza znaczne zagrożenie dla wydania — na przykład dodajemy nowe funkcje wyszukiwania, procesy tworzenia kont lub definiujemy nowe zapytania do bazy danych. Po wdrożeniu całego kodu do produkcji możemy przy wyłączonej nowej funkcjonalności zmodyfikować kod obsługi sesji użytkownika, tak aby realizował wywołania do nowych funkcji — zamiast wyświetlać wyniki użytkownikom, zapisujemy je w logu albo po prostu je ignorujemy.

Aby zobaczyć, jak nowa funkcjonalność zachowuje się pod obciążeniem, możemy udostępnić 1% użytkowników online wykonywanie ślepych wywołań nowej cechy funkcjonalnej. Po znalezieniu i rozwiązaniu wszystkich problemów można stopniowo zwiększać symulowane obciążenie poprzez zwiększenie częstotliwości używania nowej funkcjonalności oraz liczby użytkowników, którzy z niej korzystają. W ten sposób możemy bezpiecznie symulować obciążenia podobne do produkcyjnych, co pozwala nam uzyskać pewność, że usługa będzie działała zgodnie z oczekiwaniemi.

Ponadto gdy uruchamiamy funkcjonalność, możemy stopniowo uaktywniać ją dla niewielkich segmentów klientów oraz wstrzymywać wydanie w przypadku znalezienia jakichkolwiek problemów. Dzięki temu możemy zminimalizować liczbę klientów, którzy otrzymują nową cechę funkcjonalną i muszą przestać z niej korzystać, ponieważ znaleziono defekt lub nie można utrzymać wymaganej wydajności.

W 2009 roku, kiedy John Allspaw był wiceprezesem działu operacyjnego w serwisie Flickr, napisał do zespołu zarządzającego Yahoo! o procesie ślepych uruchomień, że „jeśli chodzi o obawy przed problemami związanymi z obciążeniami, zaufanie **wszystkich** osób rośnie prawie do punktu apatii. Nie mam pojęcia, ile wdrożeń kodu do produkcji miało miejsce w jakimś określonym dniu w ciągu ostatnich 5 lat... bo mnie to nie obchodzi. A nie obchodzi mnie przede wszystkim z powodu niewielkiego prawdopodobieństwa spowodowania problemów przez te wdrożenia. Jeśli zmiany spowodują problemy, to wszystkie osoby w firmie Flickr mogą znaleźć na stronie WWW, kiedy wykonano zmianę, kto ją wykonał oraz dokładnie (linijka po linijce), jaka to była zmiana^{*}”.

Później, gdy stworzyliśmy odpowiednie mechanizmy telemetrii produkcji w aplikacji i środowiskach, byliśmy również w stanie zapewnić szybsze cykle sprzężenia zwrotnego, dzięki którym mogliśmy zweryfikować założenia biznesowe i efekty bezpośrednio po wdrożeniu cechy funkcjonalnej do produkcji.

Dzięki temu już nie czekamy do wydania big bang, aby sprawdzić, czy klienci chcą korzystać z funkcjonalności, którą budujemy. Zamiast tego do czasu ogłoszenia i wydania nowej cechy funkcjonalnej mamy już przetestowane hipotezy biznesowe i uruchomione niezliczone eksperymenty, dzięki którym ciągle doskonalimy nasze produkty z prawdziwymi klientami. Możemy potwierdzić, czy nowe funkcjonalności pozwolą klientom osiągnąć pożądane efekty.

Studium przypadku

Ślepe uruchomienia narzędzia Facebook Chat (2008)

Przez prawie dekadę Facebook pod względem liczby przeglądanych stron oraz unikatowych użytkowników jest jednym z najczęściej odwiedzanych serwisów internetowych. W 2008 roku notował ponad 70 milionów aktywnych użytkowników dziennie, co stworzyło wyzwanie dla zespołu pracującego nad nową funkcjonalnością Facebook Chat[†].

* W podobnym tonie wypowiedział się Chuck Rossi, dyrektor inżynierii wydań w firmie Facebook, który napisał: „Cały kod planowany do uruchomienia w ciągu kolejnych sześciu miesięcy już został wdrożony na serwerach produkcyjnych. Pozostało jedynie go uaktywnić”.

† Do roku 2015 Facebook miał ponad miliard aktywnych użytkowników i odnotował 17-procentowy wzrost ich liczby w stosunku do roku poprzedniego.

Eugene Letuchy, inżynier w zespole Chat, pisał, że wielka liczba równoległych użytkowników stanowiła olbrzymie wyzwanie dla inżynierów oprogramowania: „Operacją wymagającą najwięcej zasobów w systemie czatu nie jest wysyłanie wiadomości. Jest nią raczej informowanie użytkowników o statusach online-bezczynność-offline wszystkich ich znajomych, dzięki czemu można rozpocząć rozmowy”.

Implementacja tej intensywnej obliczeniowo cechy funkcjonalnej była jednym z największych przedsięwzięć technicznych, jakie kiedykolwiek były podejmowane na Facebooku, a jej realizacja zajęła prawie rok*. Złożoność projektu częściowo wynika z szerokiego zakresu technologii wymaganych do osiągnięcia pożąданej wydajności, w tym C++, JavaScript i PHP, a także pierwszego użycia języka Erlang w infrastrukturze warstwy backend.

Podczas trwającego niemal rok projektu członkowie zespołu Chat ewidencjonowali kod w repozytorium kontroli wersji, skąd był wdrażany do produkcji co najmniej raz dziennie. Początkowo funkcjonalność czatu była widoczna tylko dla członków zespołu. Później włączono ją dla wszystkich wewnętrznych pracowników, ale była całkowicie ukryta dla zewnętrznych użytkowników Facebooka dzięki mechanizmowi Gatekeeper — usłudze włączania (wyłączania) cech funkcjonalnych w firmie Facebook.

W ramach procesu ślepego uruchamiania w każdej sesji użytkownika Facebooka, gdzie w przeglądarce działał JavaScript, było załadowane środowisko testowe — elementy interfejsu użytkownika czatu były ukryte, ale przeglądarka klienta wysyłała niewidzialne, testowe wiadomości do już wdrożonej do produkcji usługi czatu w warstwie backend. W ten sposób można było symulować obciążenia zbliżone do produkcyjnych w ciągu całego projektu oraz znajdować i rozwiązywać problemy związane z wydajnością na dłucho przed publikacją funkcjonalności dla klientów.

Dzięki temu każdy użytkownik Facebooka stał się częścią ogromnego programu testowania obciążenia, który umożliwił zespołowi zyskać przekonanie, że system jest w stanie obsłużyć realistyczne obciążenia podobne do produkcyjnych. Wydanie i uruchomienie funkcjonalności czatu wymagało zaledwie dwóch kroków: zmodyfikowania ustawienia konfiguracyjnego programu Gatekeeper, aby cecha funkcjonalna czatu stała się widoczna dla pewnej części wewnętrznych użytkowników, oraz wprowadzenia mechanizmu odpowiedzialnego za załadowanie w przeglądarkach użytkowników nowego kodu JavaScript, odpowiedzialnego za wyrenderowanie elementów interfejsu użytkownika czatu oraz wyłączenie niewidzialnego zestawu testów. W przypadku jakichkolwiek problemów te dwa kroki można było wycofać. Kiedy nadszedł dzień premiery funkcjonalności Facebook Chat, okazał się zaskakująco udany i przebiegł bez niespodzianek.

* Ten problem ma najgorszą złożoność obliczeniową — jest ona rzędu $O(n^3)$. Inaczej mówiąc, czas obliczeń zwiększa się wykładniczo jako funkcja liczby użytkowników online, rozmiaru listy ich znajomych oraz częstotliwości zmian statusu online i offline.

Bez żadnego wysiłku liczba użytkowników nowej funkcjonalności z dnia na dzień wzrosła od zera do 70 milionów. Podczas wydania stopniowo włączano funkcjonalność czatu dla coraz większych segmentów populacji klientów — najpierw dla wszystkich wewnętrznych pracowników Facebooka, następnie dla 1% populacji klientów, następnie dla 5% itd. Jak napisał Letuchy: „Sekret pomyślnego wzrostu od zera do 70 milionów z dnia na dzień leży w unikaniu robienia wszystkiego za jednym razem”.

BADANIA NAD STOSOWANIEM TECHNIK CIĄGŁEGO DOSTARCZANIA I CIĄGŁEGO WDRAŻANIA W PRAKTYCE

Termin **ciągłe dostarczanie** (ang. *continuous delivery*) został zdefiniowany przez Jęza Humble'a i Davida Farleya w książce *Continuous Delivery*. Termin **ciągłe wdrażanie** (ang. *continuous deployment*) po raz pierwszy został podany przez Tima Fitza w jego wpisie na blogu zatytułowanym „Continuous Deployment at IMVU: Doing the impossible fifty times a day”. Jednak w 2015 roku, podczas pisania niniejszej książki, Jez Humble skomentował: „W ciągu ostatnich pięciu lat często mylono terminy »ciągłe dostarczanie« i »ciągłe wdrażanie«. Trzeba przyznać, że od czasu napisania książki zmieniło się moje własne myślenie i zmienił się sposób definiowania tych dwóch pojęć. Każda organizacja powinna stworzyć swoją własną odmianę, bazując na własnych potrzebach. Kluczowym elementem, na który powinniśmy zwracać uwagę, są efekty, a nie forma: wdrożenia powinny wiązać się z niskim ryzykiem i sprowadzać do zdarzenia wciśnięcia przycisku, które można wykonać na żądanie”.

Jego zaktualizowane definicje pojęć ciągłego dostarczania i ciągłego wdrażania są następujące:

Gdy wszyscy programiści pracują na małych partiach kodu i rewizji master albo gdy wszyscy pracują na krótkotrwałych gałęziach cech funkcjonalnych, które są regularnie scalane z rewizją master, utrzymywana zawsze w stanie gotowości do wydania, i kiedy możemy przeprowadzić wydanie na żądanie przez naciśnięcie przycisku w normalnych godzinach pracy, to realizujemy ciągłe dostarczanie. Gdy programista wprowadzi jakkolwiek zmianę powodującą regresję (mogą to być defekt, problemy wydajności, problemy zabezpieczeń, użyteczności itp.), szybko otrzymuje informację zwrotną. Znalezione problemy są natychmiast rozwiązywane, tak aby rewizja master była ciągle w stanie gotowości do wdrożenia.

Oprócz wyżej wymienionych własności, kiedy regularnie i samoobsługowo (przez pracowników zespołu Dev lub Ops) są wdrażane do produkcji dobre komplikacje — co zazwyczaj oznacza, że wdrażanie do produkcji następuje co najmniej raz dziennie na programistę lub być może nawet automatycznie po zaewidencjonowaniu każdej zmiany w repozytorium kontroli wersji — mamy do czynienia z ciągłym wdrażaniem.

Jeśli przyjmiemy taką postać definicji, to trzeba uznać ciągłe dostarczanie za warunek wstępny dla ciągłego wdrażania — podobnie jak ciągła integracja jest warunkiem wstępny dla ciągłego dostarczania. Ciągłe wdrażanie jest zwykle stosowane w kontekście usług sieciowych dostarczanych przez internet. Jednak ciągłe dostarczanie może być stosowane w niemal każdym kontekście, gdzie dążymy do wdrożeń i wydań charakteryzujących się wysoką jakością, szybkimi czasami realizacji oraz przynoszącymi wysoce przewidywalne, związane z niskim ryzykiem efekty. Może to dotyczyć zarówno systemów wbudowanych, produktów COTS, jak i aplikacji mobilnych.

W firmach Amazon i Google praktyki ciągłego dostarczania wykorzystuje większość zespołów, choć praktyki ciągłego wdrażania stosują tylko niektóre z nich. W związku z tym istnieje pomiędzy nimi znaczące zróżnicowanie częstotliwości wdrażania kodu i sposobów jego realizacji. Zespoły są uprawnione do wyboru sposobu wdrażania na podstawie specyficznych zagrożeń. Na przykład zespół Google App Engine często przeprowadza wdrażanie raz dziennie, natomiast zespół właściwości Google Search robi to kilka razy w tygodniu.

Większość studiów przypadków zaprezentowanych w tej książce dotyczy ciągłego dostarczania. Przykładem może być wbudowane oprogramowanie działające na drukarkach HP LaserJet, usługa drukowania rozliczeń CSG uruchomiona na 20 platformach technicznych, włącznie z aplikacją w COBOL-u na komputery mainframe, a także firmy Facebook i Etsy. Te same wzorce mogą być stosowane dla oprogramowania działającego na telefonach komórkowych, naziemnych stacjach kontroli zarządzających satelitami itd.

PODSUMOWANIE

Jak pokazują przykłady firm Facebook, Etsy i CSG, wydania i wdrożenia nie muszą być obarczonymi wysokim ryzykiem, dramatycznymi przedsięwzięciami, wymagającymi pracy dziesiątek lub setek inżynierów przez całą dobę. Mogą być one wykonywane całkowicie rutynowo, w ramach codziennej pracy.

Dzięki temu można skrócić czasy realizacji wdrożeń z miesięcy do minut, umożliwiając organizacjom szybkie dostarczanie wartości do klienta bez powodowania chaosu i zakłóceń. Ponadto dzięki wspólnej pracy zespołów Dev i Ops praca działu operacyjnego wreszcie może mieć humanitarną formę.

13

Architektura dla wydań niskiego ryzyka

W prawie każdym znanym przypadku firm dokonujących transformacji DevOps można mówić o „doświadczenia bliskich śmierci” z powodu problemów architektonicznych. Tego rodzaju sytuacje przytaczano w historiach dotyczących takich firm, jak LinkedIn, Google, eBay, Amazon i Etsy. W każdym przypadku udało się z powodzeniem przeprowadzić migrację do odpowiednieszej architektury, która rozwiązywała bieżące problemy i potrzeby organizacyjne.

To jest zasada ewolucji architektury — Jez Humble zauważał, że architektura „każdego udanego produktu lub organizacji z pewnością ewoluje podczas swojego cyklu życia”. Randy Shoup przed przejściem do Google pełnił w latach 2004 – 2011 funkcję głównego inżyniera i architekta w firmie eBay. Zaobserwował, że „zarówno eBay, jak i Google po raz piąty przepisały całą swoją architekturę — od góry do dołu”.

Dalej mówi: „Patrząc dzisiaj, niektóre technologie [i architektoniczne wybory] wyglądają na perspektywiczne, podczas gdy inne zdradzają krótkowzroczność. Każda decyzja najprawdopodobniej możliwie najlepiej służyła celom organizacji w czasie, gdy ją podjęto. Gdybyśmy w 1995 roku próbowali wdrożyć mechanizm równoważny mikrousługom, to prawdopodobnie ponieślibyśmy porażkę, upadlibyśmy pod własnym ciężarem i pociągnęlibyśmy za sobą całą firmę”.

* Architektura w serwisie eBay przeszła przez następujące fazy: Perl i pliki (v1, 1995), C++ i Oracle (v2, 1997), XSL i Java (v3, 2002), Java dla pełnego stosu (v4, 2007), mikrousługi Polyglot (2013+).

Prawdziwym wyzwaniem jest migracja z architektury, którą mamy, do architektury, której potrzebujemy. W przypadku serwisu eBay po pojawiению się potrzeby zmiany architektury najpierw przeprowadzono niewielki projekt pilotażowy, którego zadaniem było udowodnienie sobie, że problem jest wystarczająco dobrze rozumiany, by warto było podjąć wysiłek. Na przykład gdy zespół Shoupa zaplanował w 2006 roku migrację określonych części witryny do pełnego stosu Javy, poszukiwano obszaru, którego migracja przyniesie największe zyski. W tym celu posortowano strony witryny według generowanych przychodów. Wybrano obszary przynoszące najwyższe dochody i wyeliminowano te, dla których zwrot z inwestycji nie był wystarczający dla uzasadnienia wysiłku.

To, co zespół Shoupa zrobił w serwisie eBay, jest podręcznikowym przykładem projektu ewolucyjnego z wykorzystaniem techniki znanej pod nazwą **strangler application pattern** (dosł. „wzorzec dusiciela”), gdzie zamiast „wycofywać i wymieniać” stare usługi o architekturze niezapewniającej wsparcia celów organizacyjnych, ukrywa się istniejące funkcjonalności za interfejsem API i unika się wprowadzania do nich dalszych zmian. Następnie wszystkie nowe funkcjonalności są implementowane w postaci nowych usług z wykorzystaniem wywołań do starego systemu, gdy jest to konieczne.

Wzorzec *strangler application* przydaje się szczególnie do wspomagania migracji fragmentów monolitycznych aplikacji lub usług ściśle sprzężonych do postaci usług o luźnych sprzężeniach. Bardzo często jesteśmy zmuszeni do pracy w architekturze, na którą składają się zbyt ściśle powiązane i zbyt splecione komponenty często tworzone przez wiele lat (lub nawet dziesięcioleci).

Konsekwencje posługiwania się zbyt ściśle powiązaną architekturą są łatwe do spostrzeżenia: za każdym razem, gdy próbujemy ewidencjonować kod w rewizji master albo zrealizować wdrożenie do produkcji, ryzykujemy globalną awarię (np. przestają przechodzić czyjeś testy lub funkcjonalność, za którą jest odpowiedzialny ktoś inny, albo następuje awaria całej witryny). Aby tego uniknąć, każda, nawet niewielka zmiana wymaga ogromnej ilości komunikacji i koordynacji trwających wiele dni lub tygodni, a także uzgodnień z wieloma zespołami, których mogą dotyczyć wprowadzane zmiany. Wdrożenia również stają się problematyczne — liczba zmian, które są połączone w paczkę w ramach pojedynczego wdrożenia, rośnie, co jeszcze bardziej komplikuje wysiłki związane z integracją i testami oraz zwiększa już i tak wysokie prawdopodobieństwo awarii.

Wdrażanie nawet niewielkich zmian może wymagać koordynacji z setkami (a nawet tysiącami) innych programistów. Każda taka zmiana może spowodować katastrofalny błąd, którego znalezienie i rozwiązanie potencjalnie wymaga wielu tygodni (skutkuje to kolejnym symptomem: „Programiści poświęcają tylko 15% czasu na kodowanie — resztę czasu zajmują im spotkania”).

To wszystko przyczynia się do bardzo niebezpiecznego systemu pracy, gdzie niewielkie zmiany mogą mieć nieprzewidywalne i katastrofalne konsekwencje. Często

przyczynia się również do powstawania obaw przed integracją i wdrażaniem kodu oraz samonapędzającej się spirali spadku częstotliwości wdrożeń.

Z perspektywy architektury korporacyjnej ta spirala spadku jest konsekwencją drugiego prawa architektonicznej termodynamiki — zwłaszcza w dużych, złożonych organizacjach. Charles Betz, autor książki *Architecture and Patterns for IT Service Management, Resource Planning, and Governance: Making Shoes for the Cobbler's Children*, zauważa: „[Właścicieli projektów IT] nie obarcza się odpowiedzialnością za swój wkład do ogólnej entropii systemu”. Innymi słowy, zmniejszenie ogólnej złożoności i zwiększanie wydajności pracy wszystkich zespołów programistycznych rzadko jest celem indywidualnego projektu.

W tym rozdziale opiszemy kroki, które można podjąć w celu odwrócenia spirali spadku, dokonamy przeglądu głównych archetypów architektonicznych, zbadamy atrybuty architektury wpływające na wydajność pracy programistów, testowalność, możliwości wdrażania i bezpieczeństwo, a także dokonamy oceny strategii, które pozwalają przeprowadzić bezpieczną migrację od bieżącej architektury do takiej, która zapewnia łatwiejsze osiągnięcie wyznaczonych celów organizacyjnych.

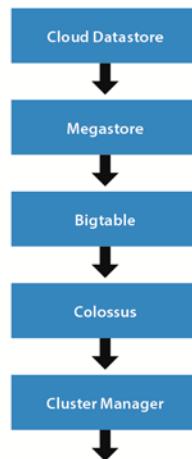
ARCHITEKTURA, KTÓRA UŁATWIA WYDAJNOŚĆ, TESTOWALNOŚĆ I BEZPIECZEŃSTWO

W przeciwieństwie do architektury o ścisłych sprzężeniach, która może ujemnie wpływać na wydajność i zdolność do bezpiecznego wprowadzania zmian, architektura o luźnych sprzężeniach, z dobrze zdefiniowanymi interfejsami, które wymuszają sposób połączenia modułów ze sobą, promuje wydajność i bezpieczeństwo. Umożliwia współpracę małych, wydajnych zespołów „na dwie pizze”, zdolnych do wprowadzania niewielkich zmian, które mogą być wdrażane bezpiecznie i niezależnie od siebie. A ponieważ każda usługa ma również dobrze zdefiniowane API, łatwiejsze staje się również testowanie usług oraz tworzenie kontraktów i umów SLA pomiędzy zespołami.

Jak opisuje Randy Shoup: „Tego typu architektura służyła firmie Google bardzo dobrze — w przypadku takiej usługi jak Gmail pod spodem jest pięć lub sześć innych warstw. Każda z nich ma bardzo ściśle określona, konkretną funkcję. Każda usługa jest obsługiwana przez mały zespół, który buduje usługę i uruchamia jej funkcjonalność. Każdy zespół może wybrać własną technologię. Innym przykładem jest usługa Google Cloud Datastore, która jest jedną z największych usług NoSQL na świecie (rysunek 22). Pomimo rozmiarów jest obsługiwana przez zespół składający się z zaledwie około ośmiu osób, głównie dlatego, że bazuje na warstwach, które z kolei korzystają z innych warstw niezależnych usług”.

Google Cloud Datastore

- **Cloud Datastore:** usługa NoSQL
 - Wysoka skalowalność i elastyczność
 - Silna spójność transakcyjna
 - Bogate możliwości tworzenia zapytań, podobne do magazynów SQL
- **Megastore:** strukturalna baza danych o skali globalnej
 - Transakcje wielowierszowe
 - Synchroniczna replikacja pomiędzy wieloma centrami danych
- **Bigtable:** strukturalny magazyn danych na poziomie klastra
 - (wiersz, kolumna, znacznik czasu) -> zawartość komórki
- **Colossus:** klastrowy system plików nowej generacji
 - Dystrybucja i replikacja bloków
- **Infrastruktura zarządzania klastrami**
 - Planowanie zadań, przydzielanie maszyn



Rysunek 22. Google Cloud Datastore (źródło: Shoup, „From the Monolith to Micro-services”)

Tego rodzaju architektura zorientowana na usługi umożliwia niewielkim zespołom pracę nad mniejszymi i prostszymi jednostkami programistycznymi, które każdy z zespołów może wdrażać niezależnie, szybko i bezpiecznie. Shoup zauważył: „Organizacje korzystające z tego typu architektur, takie jak Google i Amazon, udowodniły, jak bardzo mogą one wpływać na struktury organizacyjne oraz elastyczność i skalowalność. W obu tych organizacjach pracują dziesiątki tysięcy programistów, a pomimo to małe zespoły mogą być bardzo wydajne”.

ARCHETYPY ARCHITEKTONICZNE. MONOLITY A MIKROUSŁUGI

W pewnym momencie historii większość organizacji DevOps była ograniczona przez ściśle powiązane, monolityczne architektury, które — choć okazały się niezwykle skuteczne w ustanowieniu pozycji produktu na rynku — stwarzały zagrożenie niepowodzenia w przypadku zwiększenia skali (np. monolityczna aplikacja w C++ serwisu eBay w 2001 r., monolityczna aplikacja OBIDOS firmy Amazon w 2001, monolityczny frontend Twittera Rails w 2009 r. oraz monolityczna aplikacja serwisu LinkedIn Leo w 2011 roku). W każdym z tych przypadków udało się zmodyfikować architekturę systemów i ustawić scenę nie tylko po to, by przetrwać, ale również, by rozwijać się i poprawiać pozycję na rynku.

Monolityczne architektury nie są z natury złe — w rzeczywistości często są one najlepszym wyborem dla organizacji, której produkt jest we wczesnej fazie swojego cyklu życia. Jak zauważył Randy Shoup: „Nie istnieje jedna doskonała architektura dla wszystkich produktów i we wszystkich skalach. Każda architektura pozwala osiągnąć

określony zbiór celów lub spełnić zbiór wymagań i ograniczeń — na przykład czas do pojawienia się na rynku, łatwość programowania funkcjonalności, skalowanie itp. Funkcjonalność jakiegokolwiek produktu lub usługi prawie na pewno będzie z czasem ewoluować — nie powinno dziwić, że potrzeby architektoniczne również mogą się zmieniać. To, co działa w skali 1×, rzadko działa w skali 10× lub 100×”.

Główne archetypy architektoniczne zestawiono w tabeli 3. Każdy wiersz wskazuje na inną ewolucyjną potrzebę organizacji, a w każdej kolumnie wymieniono zalety i wady każdego z archetypów. Jak wynika z tabeli, architektura monolityczna, która obsługuje początkową fazę projektu (np. szybkie prototypy nowych funkcji i potencjalne zwroty lub duże zmiany w strategii), bardzo różni się od architektury, która wymaga setek zespołów programistów, z których każdy musi mieć możliwość samodzielnego dostarczenia wartości do klientów. Wspieranie architektur ewolucyjnych pozwala zagwarantować, że architektura zawsze będzie służyła bieżącym potrzebom organizacji.

Tabela 3. Archetypy architektoniczne

	Zalety	Wady
Monolityczna v1 (cała funkcjonalność w jednej aplikacji)	<ul style="list-style-type: none"> Początkowo wydaje się prosta Niewielkie opóźnienia pomiędzy procesami Jedna baza kodu, jedna jednostka wdrażania Wydajne gospodarowanie zasobami przy niewielkiej skali 	<ul style="list-style-type: none"> Wraz ze wzrostem rozmiarów zespołu zwiększa się obciążenia związane z koordynacją Słabe wymuszenie modularności Słabe skalowanie Wdrażanie zgodnie z zasadą „wszystko albo nic” (przestoje, awarie) Długie czasy budowania
Monolityczna v2 (zbiór monolitycznych warstw: „fronton prezentacyjny”, „serwer aplikacji”, „warstwa bazy danych”)	<ul style="list-style-type: none"> Początkowo wydaje się prosta Tworzenie kwerend ze złączaniami jest łatwe Wdrożenie z pojedynczym schematem Wydajne gospodarowanie zasobami przy niewielkiej skali 	<ul style="list-style-type: none"> Z biegiem czasu tendencja do coraz ścisłej sprężyny Słabe skalowanie i redundancja („wszystko albo nic”, skalowanie tylko w pionie) Trudności z właściwym dostrajaniem Zarządzanie schematem zgodnie z zasadą „wszystko albo nic”
Mikrosługi (modularne, niezależne relacje grafowe zamiast warstw, utrwalanie w izolacji)	<ul style="list-style-type: none"> Każda jednostka jest prosta Niezależne skalowanie i wydajność Niezależne testowanie i wdrażanie Możliwość optymalnego dostrajania (buforowanie, replikacja itp.) 	<ul style="list-style-type: none"> Wiele współpracujących ze sobą jednostek Wiele niewielkich repozytoriów Wymaga więcej zaawansowanych narzędzi i zarządzania zależnościami Opóźnienia sieciowe

(Źródło: Shoup, „From the Monolith to Micro-services”)

Studium przypadku

Architektura ewolucyjna w firmie Amazon (2002)

Jedną z najczęściej badanych transformacji architektury przeprowadzono w firmie Amazon. W wywiadzie z laureatem Nagrody Turinga ACM i posiadaczem tytułu Microsoft Technical Fellow, Jimem Grayem, dyrektorem technicznym firmy Amazon, Wernerem Vogels, wyjaśnił, że serwis Amazon.com rozpoczął działalność w 1996 roku, bazując na „monolitycznej aplikacji, działającej na serwerze WWW i komunikującej się z serwerem bazy danych na zapleczu. Ta aplikacja, znana pod nazwą Obidos, rozwinęła się do stanu, w którym zawierała całą logikę biznesową, całą logikę wyświetlania oraz wszystkie funkcjonalności, z których ostatecznie serwis Amazon stał się znany: podobieństwa, rekommendacje, Listmania, opinie itp.”.

Z biegiem czasu aplikacja Obidos stała się zbyt popłatana, zawierała złożone relacje współdzielenia, co oznaczało, że nie można było skalować pojedynczych elementów według potrzeb. Vogels powiedział Grayowi, że „wiele cech, które byłyby pożądane w dobrym środowisku oprogramowania, w tym środowisku nie może już być zrealizowane, ponieważ w jeden system połączono wiele skomplikowanych części oprogramowania. Dalszy rozwój takiego systemu nie był już możliwy”.

Opisując proces myślowy zmierzający do opracowania nowej, pożąданiej architektury, Vogel powiedział Grayowi: „Przeszliśmy przez okres poważnych introspekcji i doszliśmy do wniosku, że architektura zorientowana na usługi dałaby nam poziom izolacji, który pozwoliłby nam szybko i samodzielnie zbudować wiele składników oprogramowania”.

Vogels zauważył: „Duże zmiany architektury, jakie nastąpiły w firmie Amazon w ciągu ostatnich pięciu lat [w latach 2001 – 2005], miały na celu przejście z dwuwarstwowego monolitu do w pełni rozproszonej, zdecentralizowanej platformy usług obsługujących wiele różnych aplikacji. Aby taka architektura stała się faktem, potrzebnych było mnóstwo innowacji, zwłaszcza że byliśmy jednymi z pierwszych, którzy zastosowali takie podejście”. Wnioski z doświadczeń Vogela w firmie Amazon, które są ważne dla naszego zrozumienia zmian w architekturze, są następujące:

- **Wniosek 1.** — stosowane konsekwentnie ścisłe zorientowanie na usługi to doskonała technika do osiągnięcia izolacji. Dzięki temu można osiągnąć poziom własności usługi i kontroli, który wcześniej nie był możliwy.
- **Wniosek 2.** — zakaz bezpośredniego dostępu do bazy danych przez klientów pozwala na wprowadzanie usprawnień skalowania i niezawodności usługi bez wpływu na klientów.
- **Wniosek 3.** — przejście do architektury usług pozwala osiągnąć znaczące korzyści dla procesów rozwoju i operacji. Model usług stał się kluczowym czynnikiem tworzenia zespołów, które mogą wprowadzać szybkie innowacje

z silną orientacją na klienta. Z każdą usługą jest związany zespół. Jest on w całości odpowiedzialny za usługę — od zdefiniowania zakresu funkcjonalności, poprzez tworzenie ich architektury, po ich budowanie i utrzymanie.

Poziom wzrostu wydajności programistów i niezawodności usług wynikający z zastosowania wymienionych wniosków zapiera dech w piersiach. W 2011 r. firma Amazon przeprowadzała około 15 000 wdrożeń dziennie. Do 2015 r. liczba wdrożeń wzrosła do prawie 136 000 wdrożeń dziennie.

WYKORZYSTANIE WZORCA STRANGLER APPLICATION W CELU WYKONYWANIA BEZPIECZNEJ EWOLUCJI ARCHITEKTURY KORPORACYJNEJ

Określenie **strangler application** zostało wymyślone przez Martina Fowlera w 2004 r. Inspiracją były ogromne winorośle strangler, które Fowler zobaczył podczas swojej podróży do Australii. Napisał o nich: „Rozsiewają się w górnej gałęzi drzewa figowego i stopniowo schodzą w dół, aż zakorzenią się w glebie. Przez wiele lat rozwijają się we wspaniałe i piękne kształty, jednocześnie dusząc i zabijając drzewo, które było ich gospodarzem”.

Jeżeli ustaliliśmy, że nasza bieżąca architektura jest zbyt ściśle powiązana, możemy zacząć bezpiecznie oddzielać fragmenty funkcjonalności od istniejącej architektury. W ten sposób pozwalamy zespołom obsługującym rozdzielone funkcjonalności na niezależne rozwijanie, testowanie i wdrażanie swojego kodu do produkcji z zachowaniem autonomii i zasad bezpieczeństwa, a jednocześnie zmniejszamy architektoniczną entropię.

Jak opisano wcześniej, wzorzec application strangler polega na schowaniu istniejących funkcjonalności za interfejsem API, gdzie pozostaje bez zmian, i zimplementowaniu nowej funkcjonalności w pożąданiej architekturze z wywołaniami do starego systemu, gdy jest to konieczne. Gdy implementujemy aplikacje dusicieli, staramy się korzystać ze wszystkich usług za pośrednictwem wersjonowanych interfejsów API, zwanych także **usługami wersjonowanymi** (ang. *versioned services*) lub **usługami niezmienialnymi** (ang. *immutable services*).

Wersjonowane interfejsy API umożliwiają modyfikowanie usług bez wpływu na klientów. Dzięki temu system staje się mniej sprzężony. Jeśli trzeba zmodyfikować argumenty, należy utworzyć nową wersję interfejsu API i dokonać migracji do nowej wersji kodu zespołów, które zależą od naszej usługi. Należy pamiętać, że nie osiągniemy celu zmodyfikowania architektury, gdy pozwolimy nowej aplikacji dusicielowi na ścisłe sprzężenie z innymi usługami (np. bezpośrednie połączenie z bazą danych innej usługi).

Jeśli usługi, które wywołujemy, nie mają czysto zdefiniowanego API, to powinniśmy je zbudować lub przynajmniej ukryć złożoność komunikacji z takimi systemami wewnątrz biblioteki klienckiej, która ma czysto zdefiniowany interfejs API.

Poprzez wielokrotne rozdzielanie funkcjonalności z istniejącego, ścisłe powiązanego systemu przenosimy naszą pracę do bezpiecznego i tępniącego życiem ekosystemu, w którym programiści mogą być bardziej wydajni. W rezultacie zakres funkcjonalności w starszej aplikacji się zmniejsza. Kiedy potrzebne funkcjonalności zostaną całkowicie przeniesione do nowej architektury, stare mogą całkowicie zniknąć.

Tworząc aplikacje dusicieli, unikamy jedynie powielania istniejących funkcjonalności w nowej architekturze lub technologii — ze względu na specyfikę istniejących systemów, które replikujemy, procesy biznesowe często są znacznie bardziej złożone, niż to jest konieczne (w wyniku badań użytkowników często można ponownie zaprojektować proces, dzięki czemu można zaprojektować znacznie prostsze sposoby osiągnięcia celu biznesowego)*.

Martin Fowler wskazuje na zagrożenia związane ze stosowaniem tych technik: „Przez większą część mojej kariery zajmowałem się przepisywaniem kluczowych systemów. Można by pomyśleć, że to łatwe — wystarczy stworzyć nowy system, który robi to samo, co robił stary. Takie przedsięwzięcia zawsze są jednak znacznie bardziej skomplikowane, niż się wydaje, i są przepełnione wieloma zagrożeniami. Zbliżają się wyznaczone terminy i jest duża presja. Chociaż wszystkim podobażą się nowe funkcjonalności (zawsze są jakieś nowe funkcjonalności), to stare muszą pozostać. Często do przepisywanego systemu muszą być dodane nawet stare błędy”.

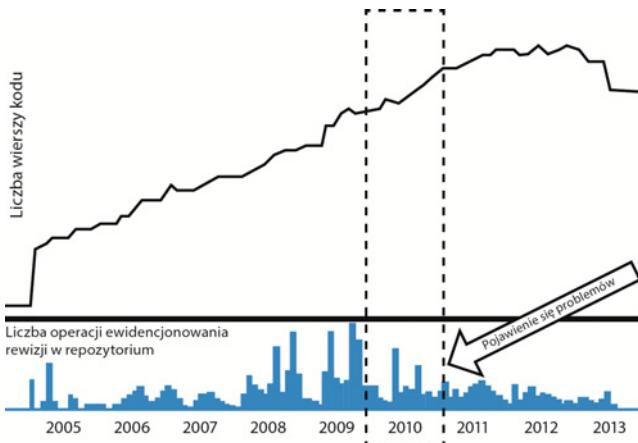
Tak jak w przypadku każdej transformacji, przed przejściem do iteracji dążymy do szybkich zwycięstw i wcześniego dostarczenia przyrostowej wartości. Przeprowadzona „z góry” analiza pomaga zidentyfikować najmniejszą możliwą ilość pracy, która pozwoli skutecznie osiągnąć rezultaty biznesowe przy użyciu nowej architektury.

Studium przypadku

Wzorzec aplikacji dusiciela w firmie Blackboard Learn (2011)

Firma Blackboard, Inc. jest jednym z pionierów w branży dostarczania technologii dla instytucji edukacyjnych. Jej roczny obrót w 2011 roku wynosił około 650 mln dolarów. W tamtym czasie zespół programistów ich flagowego produktu Learn, oprogramowania rozpowszechnianego w pakiecie i instalowanego w siedzibach klientów, zmagał się z codziennymi konsekwencjami starej bazy kodu, stworzonej w oparciu o J2EE, której historia sięgała 1997 roku (rysunek 23).

* Wzorzec aplikacji dusiciela polega na stopniowym zastępowaniu całego systemu (zazwyczaj starszego) systemem zupełnie nowym. Z kolei technika rozgałęziania przez abstrakcję (ang. *branching by abstraction*), opracowana przez Paula Hammanta, polega na stworzeniu warstwy abstrakcji pomiędzy obszarami, które zmieniamy. To pozwala na ewolucyjne projektowanie architektury aplikacji, a jednocześnie wszyscy mogą pracować na gałęzi master i stosować praktyki ciągłej integracji.



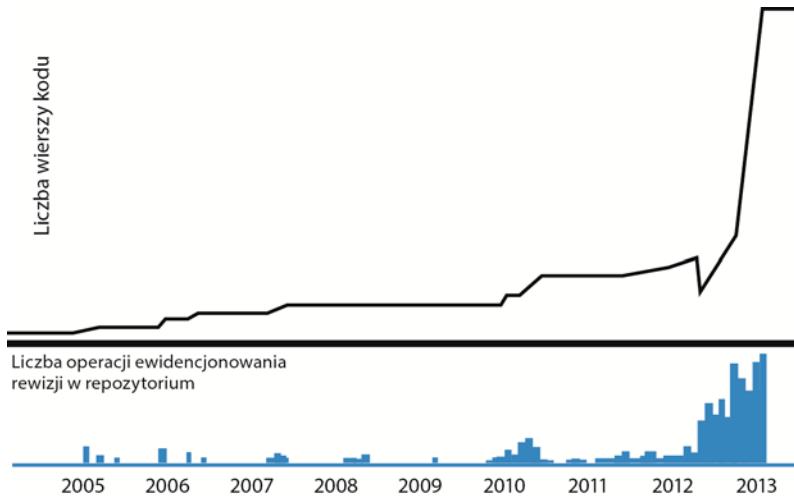
Rysunek 23. Repozytorium kodu źródłowego aplikacji Learn w firmie Blackboard przed wdrożeniem Building Blocks (źródło: „DOES14 — David Ashman — Blackboard Learn — Keep Your Head in the Clouds”, nagranie wideo w serwisie YouTube, 30.43, opublikowane na konferencji DevOps Enterprise Summit 2014, 28 października 2014, <https://www.youtube.com/watch?v=SSmixnMpsI4>)

Jak zaobserwował David Ashman, główny architekt w firmie: „W naszej bazie kodu nadal można znaleźć fragmenty napisane w Perlu”.

W 2010 roku Ashman koncentrował się na złożoności i coraz dłuższych czasach realizacji powiązanych ze starym systemem. Jak zaobserwował: „Nasze procesy budowania, integracji i testowania stają się coraz bardziej złożone i podatne na błędy. I im większe rozmiary miał produkt, tym dłuższe czasy realizacji i gorsze wyniki dla naszych klientów. Uzyskanie informacji zwrotnych z naszego procesu integracji wymaga od 24 do 36 godzin”.

Wpływ złożoności na wydajność programistów można było zaobserwować na wykresach, które Ashman wygenerował na podstawie statystyk repozytorium kodu źródłowego, gromadzonych od 2005 roku.

Na górnym wykresie widocznym na rysunku 24 zaprezentowano liczbę wierszy kodu w repozytorium monolitycznej aplikacji Blackboard Learn. Na dolnym wykresie pokazano liczbę operacji ewidencjonowania kodu w repozytorium. Problem, który stał się oczywisty dla Ashmana, polegał na tym, że liczba operacji ewidencjonowania kodu zaczęła maleć, co obiektywnie wskazywało na coraz większe trudności z wprowadzaniem zmian w kodzie, natomiast liczba wierszy kodu nadal wzrastała. Ashman stwierdził: „Stało się dla mnie oczywiste, że musimy z tym coś zrobić, w przeciwnym razie problem będzie się pogarszał i nie będzie widać jego pozytywnego rozwiązania”.



Rysunek 24. Repozytorium kodu źródłowego aplikacji Learn w firmie Blackboard po wdrożeniu Building Blocks (źródło: „DOES14 — David Ashman — Blackboard Learn — Keep Your Head in the Clouds”, nagranie wideo w serwisie YouTube, 30.43, opublikowane na konferencji DevOps Enterprise Summit 2014, 28 października 2014, <https://www.youtube.com/watch?v=SSmixnMpsI4>)

W rezultacie w 2012 roku Ashman skoncentrował się na realizacji projektu modyfikacji architektury kodu, w którym zastosowano wzorzec aplikacji dusiciela. Zespół dokonał tego poprzez stworzenie mechanizmów, które w firmie nazwano *Building Blocks* (dosł. „bloki konstrukcyjne”). Dzięki nim programiści mogli pracować nad odrębnymi modułami, oddzielonymi od monolitycznej bazy kodu i udostępnionymi za pośrednictwem ustalonych interfejsów API. Umożliwiło to pracę z o wiele większą autonomią i bez konieczności stałego komunikowania się i koordynowania z innymi zespołami programistycznymi.

Po udostępnieniu programistom techniki *Building Blocks* rozmiar monolitycznego repozytorium kodu źródłowego (mierzony liczbą wierszy kodu) zaczął się zmniejszać. Ashman wyjaśnił, że powodem było przenoszenie przez programistów kodu do repozytoriów kodu źródłowego modułów *Building Block*. „W rzeczywistości — opisywał Ashman — każdy programista, jeśli miał wybór, to pracował nad bazą kodu *Building Block*, ponieważ miał tam więcej autonomii, swobody i bezpieczeństwa”.

Wykres zaprezentowany powyżej pokazuje związek pomiędzy wykładowiczym wzrostem liczby wierszy kodu a wykładowiczym wzrostem liczby ewidencjonowania kodu dla repozytoriów kodu modułów *Building Blocks*. Nowa baza kodu modułów *Building Blocks* pozwoliła programistom na poprawienie wydajności, a ich praca stała się bezpieczniejsza, ponieważ błędów powodowały niewielkie, lokalne awarie, a nie — tak jak wcześniej — poważne katastrofy, które miały globalny wpływ na system.

Ashman zakończył: „Umożliwienie programistom pracy w architekturze Building Blocks spowodowało imponujące usprawnienia w modularności kodu. Dzięki temu programiści mogli pracować z większą niezależnością i swobodą. W połączeniu z aktualizacjami procesu budowania programiści mogli uzyskać także szybsze i lepsze informacje zwrotne na temat swojej pracy, a to oznaczało lepszą jakość”.

PODSUMOWANIE

W dużej mierze architektura, w której działają usługi, decyduje o sposobie testowania i wdrażania kodu. Zostało to zatwierdzone w raporcie Puppet Labs *2015 State of DevOps Report*, w którym wykazano, że architektura jest jednym z najważniejszych predyktorów wydajności inżynierów, którzy ją wykorzystują, oraz szybkości i bezpieczeństwa wprowadzanych zmian.

Ponieważ często jesteśmy skazani na architektury, które zoptymalizowano pod kątem innego zbioru celów organizacyjnych albo które pochodzą z dawno minionej ery, musimy mieć możliwość przeprowadzenia bezpiecznej migracji z jednej architektury do drugiej. Studia przypadków zaprezentowane w niniejszym rozdziale, a także pokazane wcześniej studium przypadku firmy Amazon pokazują takie techniki, jak wzorzec aplikacji dusiciela, które mogą nam pomóc w przeprowadzeniu migracji architektur stopniowo, co pozwala nam na dostosowanie się do potrzeb organizacji.

PODSUMOWANIE CZĘŚCI III

W poprzednich rozdziałach części III zaimplementowaliśmy architekturę i praktyki techniczne, które umożliwiają szybki przepływ pracy z Dev do Ops, dzięki czemu można szybko i bezpiecznie dostarczyć wartość do klientów.

W części IV, „Druga droga. Techniczne praktyki sprzężeń zwrotnych”, stworzymy architektury i mechanizmy umożliwiające szybki przepływ informacji zwrotnych od prawej do lewej, co pozwala szybciej znajdować i rozwiązywać problemy, propagować sprzężenie zwrotne i zapewnić lepsze wyniki pracy. Dzięki temu organizacje mogą szybciej dostosowywać się do nowych warunków.

Część IV

*Druga droga
Techniczne praktyki
sprzężeń zwrotnych*

Część IV *Wprowadzenie*

W części III opisaliśmy architekturę i praktyki techniczne wymagane do utworzenia szybkiego przepływu od działu Dev do Ops. Teraz w części IV opowiemy, jak wdrożyć techniczne praktyki **drugiej drogi** — wymagane do utworzenia szybkiego i ciągłego sprzężenia zwrotnego od działu Ops do Dev.

W ten sposób skracamy i wzmacniamy pętle sprzężenia zwrotnego. Dzięki temu możemy zobaczyć problemy natychmiast po ich wystąpieniu oraz promieniujemy te informacje dla wszystkich osób w strumieniu wartości. To pozwala nam szybko znaleźć i rozwiązać problemy we wcześniejszej fazie cyklu życia tworzenia oprogramowania, zanim spowodują katastrofalne błędy.

Ponadto opracujemy system pracy, w którym wiedza nabyta w dole strumienia wartości jest w dziale Ops integrowana z pracą działu Dev oraz kierownictwa produktu. Będziemy mogli zatem szybko wprowadzać ulepszenia i czerpać wiedzę — z problemów produkcji, problemów wdrażania — uzyskiwać wczesne sygnały problemów lub poznawać upodobania klientów.

Dodatkowo stworzymy proces, który pozwala wszystkim uzyskać informacje zwrotne na temat swojej pracy, sprawia, że informacje stają się widoczne — dzięki czemu można czerpać z nich wiedzę — oraz umożliwia szybkie przetestowanie hipotez dotyczących produktu, co pomaga ustalić, czy budowane funkcjonalności pomagają osiągnąć cele organizacji.

Pokażemy również, w jaki sposób można stworzyć telemetrię z procesów budowania, testowania i wdrażania, a także zachowań użytkowników, problemów i awarii w produkcji, problemów inspekcji i naruszeń bezpieczeństwa. Możemy dostrzec wzmacnienie sygnałów dopływających do nas w ramach codziennej pracy i rozwiązać problemy natychmiast po ich wystąpieniu. Przyczyniamy się przez to do powstawania bezpiecznych systemów pracy pozwalających nam śmiało wprowadzać zmiany i uruchamiać eksperymenty dotyczące produktów, wiedząc, że możemy szybko wykryć i skorygować błędy. Wszystko to robimy poprzez wykonywanie następujących czynności:

- tworzenie telemetrii umożliwiających dostrzeganie i rozwiązywanie problemów;
- wykorzystanie telemetrii w celu lepszego przewidywania problemów i osiągania celów;
- integrowanie badań użytkowników i sprzężenia zwrotnego do pracy zespołów produktu;

- wykorzystanie sprzężeń zwrotnych, dzięki czemu działały Dev i Ops mogą bezpieczne przeprowadzać wdrożenia;
- wykorzystanie sprzężeń zwrotnych w celu zwiększenia jakości pracy poprzez wzajemne weryfikowanie kodu i programowanie w parach.

Wzorce zaprezentowane w tym rozdziale pomogą wzmacnić wspólne cele działów zarządzania produktem, rozwoju, walidacji, operacyjnego i bezpieczeństwa informacji i zachęcają pracowników do współdzielenia odpowiedzialności za płynne dostarczenie usług do produkcji oraz współpracę w kierunku usprawnienia systemu jako całości. Tam, gdzie to możliwe, chcemy dążyć do tworzenia powiązań pomiędzy przyczynami a skutkami. Im więcej założeń uda się obalić, tym szybciej można odkrywać i rozwiązywać problemy, ale również tym większa zdolność do nauki i wprowadzania innowacji.

W kolejnych rozdziałach będziemy implementować pętle sprzężenia zwrotnego umożliwiające wszystkim wspólną pracę w kierunku wspólnych celów, dostrzeganie problemów natychmiast po ich wystąpieniu, szybkie wykrywanie awarii i przywracanie stanu działania, a także zadbanie o to, aby funkcjonalności nie tylko działały zgodnie z projektem w produkcji, ale również by pozwalały osiągać cele organizacyjne i wspierały uczenie się w organizacji.

Tworzenie telemetrii umożliwiające dostrzeganie i rozwiązywanie problemów

Awarie są elementem rzeczywistości działań operacyjnych. Niewielkie zmiany mogą skutkować wieloma nieoczekiwanyimi efektami, w tym przestojami i globalnymi awariami, które wywierają wpływ na klientów. To wszystko należy do rzeczywistości złożonych systemów. Żadna pojedyncza osoba nie może zobaczyć całego systemu i zrozumieć wzajemnych współzależności poszczególnych jego części.

Gdy w produkcji wystąpią przestoje albo gdy zdarzą się inne problemy podczas codziennej pracy, często nie posiadamy informacji potrzebnych do rozwiązania problemu. Na przykład podczas przestoju możemy mieć kłopoty z ustaleniem, czy problem jest spowodowany błędym działaniem aplikacji (np. defektem w kodzie), błędem w środowisku (np. problemem z działaniem sieci, złą konfiguracją serwera), czy też czymś dla nas całkowicie zewnętrznym (np. masowy atak typu DoS).

W dziale Ops możemy obsługiwać takie problemy, stosując następującą zasadę „spod dużego palca”: gdy coś pójdzie nie tak w produkcji, po prostu restartujemy serwer. Jeśli to nie zadziała, restartujemy kolejny serwer. Jeśli to także nie zadziała, restartujemy wszystkie serwery. Jeśli to nie zadziała, zwalamy winę na programistów — to oni zawsze są przyczyną awarii.

Natomiast z badania Microsoft Operations Framework (MOF) przeprowadzonego w 2001 roku wynika, że organizacje z najwyższym poziomem usług restartowały swoje serwery 20 razy rzadziej od średniej i pięć razy rzadziej zdarzały się u nich przypadki wystąpienia „niebieskich ekranów śmierci”. Innymi słowy, okazało się, że najwydajniejsze

organizacje znacznie lepiej radziły sobie z diagnozowaniem i usuwaniem zdarzeń serwisowych. Kevin Behr, Gene Kim i George Spafford w książce *The Visible Ops Handbook* określili ten sposób działania „kulturą przyczynowości” (ang. *culture of causality*). Firmy o wysokiej wydajności stosowały zdyscyplinowane podejście do rozwiązywania problemów. Wykorzystywały telemetrię produkcji w celu zrozumienia możliwych czynników powstawania awarii i koncentrowały się na rozwiązywaniu problemów w przeciwieństwie do firm o niższej wydajności, które ślepo restartowały serwery.

Aby skorzystać z tego zdyscyplinowanego podejścia do rozwiązywania problemów, należy zaprojektować systemy w taki sposób, aby stale kreowały **telemetrię**, powszechnie zdefiniowaną jako „zautomatyzowany proces komunikacji, w wyniku którego pomiary oraz inne dane są gromadzone w zdalnych punktach, a następnie są przesyłane do urządzeń odbiorczych w celu monitorowania”. Naszym celem jest stworzenie mechanizmów telemetrii w aplikacjach i środowiskach — zarówno w środowiskach produkcyjnych, jak i przedprodukcyjnych — a także w potoku wdrożeń.

Michael Rembetsy i Patrick McDonnell opisali kluczowe znaczenie monitorowania produkcji podczas transformacji DevOps w firmie Etsy rozpoczętych w 2009 roku. Wynikały one ze standaryzacji i przejęcia całego stosu technologicznego na stos LAMP (Linux, Apache, MySQL i PHP) oraz odrzucenia wielu różnych technologii stosowanych w produkcji, które były coraz bardziej trudne do obsługi.

Na konferencji Velocity Conference 2012 McDonnell opisał zagrożenia związane z przeprowadzanymi transformacjami: „Zmienialiśmy naszą najważniejszą infrastrukturę. Najlepiej by było, aby klienci nigdy tej zmiany nie zauważyci. Jednak z pewnością zauważyliby, gdyby coś nam się nie udało. Potrzebowaliśmy więcej pomiarów, które dałyby nam pewność, że niczego nie zepsuliśmy podczas wprowadzania zmian. Gwarancje te były potrzebne zarówno zespołom inżynierskim, jak i członkom zespołów w obszarach nietechnicznych, takich jak marketing”.

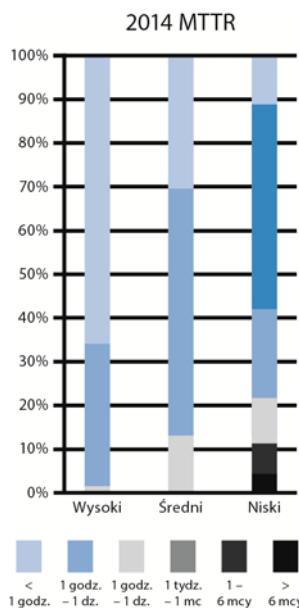
McDonnell wyjaśniał dalej: „Zaczeliśmy zbierać informacje o wszystkich serwerach za pomocą narzędzia o nazwie Ganglia. Zebrane informacje były następnie wyświetlane w programie Graphite — narzędziu open source, w które mocno zainwestowaliśmy. Zaczeliśmy agregowanie wskaźników — począwszy od biznesowych, a skończywszy na wdrożeniowych. Wtedy zmodyfikowaliśmy narzędzie Graphite, wprowadzając technologię określającą jako »niezrównana i nieporównywalna technologia pionowej linii«, którą nakładaliśmy na każdy wykres parametrów podczas realizacji wdrożeń. W ten sposób mogliśmy szybciej zobaczyć wszelkie niepożądane skutki wdrożeń. Zaczeliśmy nawet umieszczać w biurach monitory, dzięki którym wszyscy mogli zaobserwować sposób działania usług”.

Dzięki zezwoleniu programistom dodawania telemetrii do tworzonych przez nich funkcjonalności w ramach ich codziennej pracy powstało wystarczająco dużo mechanizmów telemetrii, aby wdrożenia były bezpieczne. Do roku 2011 firma Etsy monitorowała ponad 200 000 parametrów produkcji w każdej warstwie stosu aplikacji (np. funk-

cjonalności aplikacji, kondycja aplikacji, baza danych, system operacyjny, pamięć masowa, sieci, bezpieczeństwo itp.). Trzydzięci najważniejszych parametrów biznesowych było prezentowanych na „pulpitach nawigacyjnych wdrażania”. Do 2014 r. śledzono już ponad 800 000 parametrów. Dążono do wyposażenia w mechanizmy telemetrii wszystkiego, co się da, i ułatwianie inżynierom tworzenie tego rodzaju mechanizmów.

Jak zażartował Ian Malpass, inżynier w firmie Etsy: „Jeśli inżynierowie w Etsy wyznają jakąś religię, to jest to Kościół Wykresów. Jeśli coś się porusza, to my to śledzimy. Czasami rysujemy wykresy czegoś, co jeszcze nie działa. Tak na wszelki wypadek, gdyby kiedyś zaczęło... Śledzenie wszystkiego jest kluczem do osiągania szybkich postępów, ale jedynym sposobem na szybkie postępy jest zadbanie o to, aby śledzenie było łatwe. Umożliwiamy inżynierom śledzenie wszystkiego, co trzeba śledzić, w łatwy sposób, bez konieczności czasochłonnych zmian konfiguracji lub skomplikowanych procesów”.

Z raportu *2015 State of DevOps Report* wynika, że wysokowydajne firmy potrafią rozwiązywać problemy produkcji 168 razy szybciej niż pozostałe. Mediana współczynnika MTTR dla firm o wysokiej wydajności jest mierzona w minutach, natomiast w przypadku firm o niskiej wydajności mierzy się ją w dniach (rysunek 25). Dwie najważniejsze praktyki techniczne wpływające na wysokie wartości współczynników MTTR to korzystanie z repozytoriów kontroli wersji przez działy Ops oraz zastosowanie telemetrii i proaktywnego monitorowania w środowisku produkcyjnym.



Rysunek 25. Rozwiązywanie problemów przez firmy o wysokim, średnim i niskim poziomie wydajności (źródło: Puppet Labs, 2014 State of DevOps Report)

Celem tego rozdziału (podobnym do tego, który postawiono w firmie Etsy) jest zadbanie o to, aby zawsze mieć wystarczająco dużo wskaźników telemetrycznych do potwierdzenia prawidłowego działania usług w produkcji. A jeśli wystąpią problemy, to trzeba zadbać o możliwość szybkiego ustalenia, co działa niewłaściwie, i podjąć świadome decyzje o sposobie naprawy, najlepiej na dłucho, zanim problemy wywrą ujemny wpływ na klientów. Ponadto mechanizmy telemetryczne umożliwiają zdobycie najlepszego zrozumienia rzeczywistości i wykrycie stanu, w którym zrozumienie rzeczywistości jest nieprawidłowe.

TWORZENIE SCENTRALIZOWANEJ INFRASTRUKTURY TELEMETRII

Monitorowanie i rejestrowanie w celach operacyjnych w żadnym razie nie jest niczym nowym. Wiele pokoleń inżynierów operacyjnych korzysta ze spersonalizowanych framework'ów monitorowania (np. HP OpenView, IBM Tivoli i BMC Patrol/BladeLogic) w celu zagwarantowania dobrej kondycji systemów produkcyjnych. Dane zazwyczaj były zbierane za pośrednictwem agentów, które uruchomiano na serwerach. Stosowano również bezagentowe mechanizmy monitorowania (np. pułapki SNMP lub monitory bazujące na odpytywaniu). Narzędzia tego rodzaju zawsze były wyposażone w warstwę frontend z graficznym interfejsem (GUI) oraz mechanizm raportowania w warstwie backend wspierany przez takie narzędzia, jak Crystal Reports.

Podobnie nie są nowością praktyki tworzenia aplikacji wyposażonych w skuteczne mechanizmy rejestrowania i zarządzania telemetrią — istnieje wiele dojrzałych bibliotek rejestrowania dla prawie wszystkich języków programowania.

Jednak przez dziesięciolecia powstały silosy informacyjne. Działy programistyczne tworzą tylko takie zdarzenia rejestrowania, które są interesujące dla programistów, natomiast inżynierowie operacyjni monitorują jedynie to, czy środowiska działają, czy nie. W rezultacie, gdy nie wystąpi stosowne zdarzenie, to nikt nie jest w stanie stwierdzić, dlaczego cały system nie działa zgodnie z projektem lub jaki konkretny składnik uległ awarii. Brak informacji na ten temat utrudnia przywrócenie systemu do stanu działania.

Aby mieć możliwość wykrycia wszystkich problemów w czasie, gdy występują, trzeba zaprojektować i zaprogramować aplikacje i środowiska w taki sposób, aby generowały wystarczającą ilość telemetryi do tego, by można było zrozumieć sposób zachowania się systemu jako całości. Gdy w mechanizmy monitorowania i rejestrowania wyposażone są wszystkie poziomy stosu aplikacji, możemy skorzystać z innych ważnych możliwości, takich jak tworzenie wykresów i wizualizacja parametrów, wykrywanie anomalii, proaktywne powiadomienia i eskalacja alertów itp.

James Turnbull w książce *The Art of Monitoring* opisał nowoczesną architekturę monitorowania, która została opracowana i jest wykorzystywana przez inżynierów

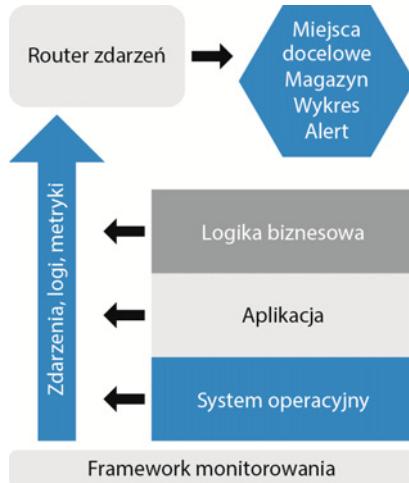
operacyjnych firm internetowych działających w skali globalnej (np. Google, Amazon, Facebook). Na architekturę często składały się narzędzia open source, takie jak Nagios i Zenoss, które zostały spersonalizowane i wdrożone w skali, którą trudno byłoby uzyskać w przypadku stosowania licencjonowanego, komercyjnego oprogramowania. Ta architektura (rysunek 26) ma następujące składniki:

- **Zbieranie danych w warstwach logiki biznesowej, aplikacji i środowiska** — w każdej z tych warstw tworzymy mechanizmy telemetrii w formie zdarzeń, dzienników i parametrów. Dzienniki mogą być przechowywane w plikach specyficznych dla aplikacji na każdym serwerze (np. `/var/log/httpd-error.log`), ale byłoby najlepiej, gdyby wszystkie logi były wysyłane do wspólnej usługi pozwalającej na łatwą centralizację oraz rotację i usuwanie zapisów. Takie mechanizmy dostarcza większość systemów operacyjnych. Przykładem może być syslog w systemie Linux, Log zdarzeń w systemie Windows itp. Ponadto gromadzimy dane we wszystkich warstwach stosu aplikacji. Dzięki temu możemy lepiej zrozumieć, jak zachowuje się nasz system. Na poziomie systemu operacyjnego możemy zbierać takie dane, jak wykorzystanie procesora, pamięci, dysku lub sieci za pomocą takich narzędzi, jak collectd, Ganglia itp. Do innych narzędzi do zbierania informacji dotyczących wydajności można zaliczyć systemy AppDynamics, New Relic i Pingdom.
- **Router zdarzeń odpowiedzialny za przechowywanie zdarzeń i parametrów** — ta funkcja umożliwia potencjalną wizualizację, obserwację trendów, generowanie ostrzeżeń, wykrywanie anomalii itd. Poprzez zbieranie, przechowywanie i agregowanie danych ułatwiamy dalszą analizę i testowanie kondycji usługi. Za pomocą mechanizmów telemetrii możemy także magazynować dane konfiguracji związanych z usługami (oraz pomocniczymi aplikacjami i środowiskami), realizować powiadamianie bazujące na wyznaczonych progach oraz testować kondycję usług^{*}.

Po scentralizowaniu logów można je zliczyć w routerze zdarzeń i przekształcić w parametry — na przykład można policzyć zdarzenia w logu w rodzaju: „Proces pid 14024 sygnał wyjścia błęd segmentacji”, zsumować je i zestawić jako jeden parametr błędów segmentacji na przestrzeni całej infrastruktury produkcyjnej.

Dzięki przekształceniu logów w parametry możemy wykonywać na nich operacje statystyczne — na przykład skorzystać z mechanizmu wykrywania anomalii w celu znalezienia wartości odstających od przeciętnej oraz odchyлеń na wczesnym etapie cyklu życia systemu. Na przykład można skonfigurować mechanizm ostrzeżeń w taki

* Przykładowe narzędzia to Sensu, Nagios, Zabbix, Logsstash, Splunk, Sumo Logic, Datadog i Rieman.



Rysunek 26. Framework monitorowania (źródło: Turnbull, „The Art of Monitoring”, wydanie Kindle, rozdział 2.)

sposób, by powiadomił nas w przypadku przejścia ze stanu „10 błędów segmentacji w zeszłym tygodniu” do „kilka tysięcy błędów segmentacji w ciągu ostatniej godziny”. Takie ostrzeżenie powinno skłonić nas do dalszych badań.

Oprócz telemetrii z usług i środowisk produkcyjnych trzeba również zbierać parametry z potoku wdrożeń, gdy zachodzą w nim ważne zdarzenia — na przykład kiedy przejdą lub zawiodą testy automatyczne lub kiedy są realizowane wdrożenia w dowolnych środowiskach. Warto również gromadzić parametry dotyczące czasu trwania procesu budowania i testowania. W ten sposób można wykryć sytuacje, które mogą wskazywać na problemy — na przykład gdy test wydajności lub komplikacja trwa dwa razy dłużej. Tak możemy znaleźć i naprawić błędy, zanim przedostaną się do produkcji.

Ponadto należy zadbać o to, aby wprowadzanie i pobieranie informacji z infrastruktury telemetrycznej było łatwe. Najlepiej, gdyby wszystkie operacje były wykonywane za pośrednictwem samoobsługowych interfejsów API w przeciwieństwie do konieczności otwierania zleceń roboczych i oczekiwania na otrzymanie raportów.

W idealnej sytuacji powinniśmy stworzyć mechanizmy telemetryczne, które byłyby zdolne do dokładnego poinformowania o tym, kiedy wydarzyło się coś interesującego, a także gdzie to się wydarzyło i w jakich okolicznościach. Mechanizmy telemetryczne powinny być również dostosowane do analizy ręcznej i automatycznej. Powinny także zapewniać możliwość wykonywania analiz zebranych parametrów bez konieczności korzystania z aplikacji, która wygenerowała logi. Jak zauważył Adrian Cockcroft: „Monitorowanie jest tak ważne, że systemy monitorowania powinny być bardziej dostępne i skalowalne niż monitorowane systemy”.

Od tego momentu termin „telemetria” będzie używany zamiennie z terminem „parametry”. W zakres tego terminu wchodzą mechanizmy rejestrowania zdarzeń oraz

parametry tworzone przez usługi na wszystkich poziomach stosu aplikacji i generowane na podstawie danych ze środowisk produkcyjnych i przedprodukcyjnych, a także z potoku wdrożeń.

TWORZENIE TELEMETRII REJESTROWANIA ZDARZEŃ APLIKACJI W CELU WSPOMAGANIA FAZY PRODUKCJI

Teraz, gdy scentralizowana infrastruktura telemetrii jest gotowa, trzeba zadbać o to, aby budowane i wykorzystywane aplikacje generowały wystarczająco dużo danych telemetrycznych. Robimy to przez powierzanie inżynierom Dev i Ops zadań tworzenia telemetrii produkcji w ramach codziennej pracy — zarówno dla nowych, jak i dla istniejących usług.

Scott Prugh, główny architekt i wiceprezes działu rozwoju w firmie CSG, powiedział: „Za każdym razem, gdy NASA wystrzeliwuje rakietę, korzysta z wielu milionów automatycznych sensorów raportujących stan każdego komponentu tego cennego zasobu. Chociaż w przypadku oprogramowania często nie stosujemy takiego poziomu szczegółowości, to doszliśmy do wniosku, że dzięki stworzeniu mechanizmów telemetrii aplikacji i infrastruktury uzyskaliśmy jeden z największych zwrotów z inwestycji. W 2014 roku tworzyliśmy ponad miliard zdarzeń telemetrycznych dziennie za pomocą ponad 100 000 lokalizacji w kodzie”.

W aplikacjach, które tworzymy i obsługujemy, powinniśmy wyposażyć w mechanizmy telemetryczne wszystkie cechy funkcjonalne — jeśli określona własność była wystarczająco ważna, by inżynier ją zaimplementował, to z pewnością zasługuje na stworzenie mechanizmów telemetrii, tak aby można było potwierdzić jej działanie zgodnie z przeznaczeniem oraz osiąganie pożądanych rezultatów*.

Każdy uczestnik strumienia wartości wykorzystuje mechanizmy telemetrii na wiele sposobów. Na przykład programiści mogą czasowo tworzyć więcej mechanizmów telemetrycznych w swoich aplikacjach, tak by lepiej diagnozować problemy na swoich stacjach roboczych, natomiast inżynierowie operacyjni mogą korzystać z telemetrii w celu diagnozowania problemów produkcji. Ponadto inżynierowie Infosec i audytorzy mogą przeglądać parametry w celu potwierdzenia skuteczności wymaganych mechanizmów kontroli, natomiast menedżer produktu może ich używać do śledzenia wyników biznesowych, współczynników użycia funkcji lub współczynników konwersji.

* Istnieją różnorodne biblioteki udostępniające funkcjonalności rejestracji zdarzeń w aplikacjach. Dzięki nim programiści mogą w łatwy sposób tworzyć użyteczne mechanizmy telemetrii. Należy wybrać taką bibliotekę, która pozwala wysyłać wszystkie logi zebrane w aplikacji do scentralizowanej infrastruktury logowania, podobnej do tej, którą stworzyliśmy w poprzednim podrozdziale. Popularnymi przykładami wspomnianych bibliotek są rrd4j i log4j dla Javy oraz log4r i ruby-cabin dla Ruby.

Do obsługi wymienionych różnych modeli użycia stosujemy różne poziomy rejestrowania. Niektóre z nich mogą również generować ostrzeżenia. Na przykład:

- **Poziom DEBUG** — informacje na tym poziomie dotyczą wszystkiego, co dzieje się w programie. Najczęściej takie informacje są wykorzystywane podczas debugowania. Często dzienniki debugowania są wyłączone w fazie produkcji, ale można je czasowo włączyć podczas rozwiązywania problemów.
- **Poziom INFO** — informacje na tym poziomie składają się z działań, które bazują na działaniach użytkownika lub są specyficzne dla systemu (np. „rozpoznanie transakcji kartą kredytową”).
- **Poziom WARN** — informacje na tym poziomie informują nas o warunkach, które mogą przekształcić się w błędy (np. odwołanie do bazy danych trwa dłużej od pewnego predefiniowanego czasu). Najprawdopodobniej zainicjują one alert i akcję rozwiązywania problemów, podczas gdy inne komunikaty logu mogą nam pomóc lepiej zrozumieć, co doprowadziło do tej sytuacji.
- **Poziom ERROR** — informacje na tym poziomie są skoncentrowane na błędach (np. awariach wywołań API, błędach wewnętrznych itp.).
- **Poziom FATAL** — informacje na tym poziomie mówią o sytuacjach, gdy trzeba zakończyć działanie aplikacji (np. demon sieci nie może nawiązać połączenia z gniazdem sieciowym).

Wybór odpowiedniego poziomu rejestrowania jest bardzo ważny. Dan North, były konsultant w firmie ThoughtWorks, który był zaangażowany w kilka projektów, w których nabierały kształtu podstawowe pojęcia ciągłego dostarczania, zauważył: „Przy podejmowaniu decyzji o tym, czy określony komunikat powinien być zarejestrowany jako ERROR, czy jako WARN, wyobraź sobie, że ktoś budzi cię o 4 rano. Niski poziom tonera w drukarce to nie jest komunikat poziomu ERROR”.

Aby mieć pewność, że posiadamy informacje istotne dla niezawodnego i bezpiecznego funkcjonowania usługi, powinniśmy zadbać o to, żeby wszystkie potencjalnie istotne zdarzenia aplikacji generowały wpisy w logu. W tej liczbie powinny się znaleźć wymienione niżej zdarzenia, zebrane przez Antona A. Chuvakina, wiceprezesa ds. badań w zespole GTP Security and Risk Management należącym do firmy Gartner:

- decyzje dotyczące uwierzytelniania i autoryzacji (w tym wylogowywanie);
- dostęp do systemu i danych;
- zmiany w systemie i aplikacjach (zwłaszcza zmiany wymagające dostępu uprzywilejowanego);
- zmiany danych, takie jak dodawanie, edytowanie lub usuwanie;
- nieprawidłowe dane wejściowe (możliwe złośliwe wstrzykiwanie, zagrożenia itp.);

- zasoby (RAM, dysk, CPU, przepustowość sieci lub inne zasoby z ustalonymi twardymi lub miękkimi limitami);
- kondycja i dostępność;
- uruchomienia i zamknięcia systemu;
- awarie i błędy;
- przerwania dopływu zasilania;
- opóźnienia;
- sukcesy (niepowodzenia) tworzenia kopii zapasowych.

W celu ułatwienia interpretacji i nadania sensu wpisom w logu warto utworzyć hierarchiczne kategorie rejestrowania — dla atrybutów niefunkcjonalnych (np. wydajność, bezpieczeństwo) oraz dla atrybutów związanych z cechami funkcjonalnymi (np. wyszukiwanie, ranking).

WYKORZYSTANIE TELEMETRII W ROLI PRZEWODNIKA ROZWIĄZYWANIA PROBLEMÓW

Zgodnie z tym, co napisaliśmy na początku niniejszego rozdziału, firmy charakteryzujące się wysoką wydajnością, stosują zdyscyplinowane podejście do rozwiązywania problemów. Takie podejście różni się od bardziej powszechniej praktyki plotek i pogłosek, które mogą prowadzić do niefortunnego parametru **średniego czasu do uznania za niewinnego** — czyli jak szybko potrafimy przekonać innych, że to nie my jesteśmy odpowiedzialni za przestój.

Gdy w firmie obowiązuje kultura obwiniania za przestoje i problemy, zespoły mogą unikać dokumentowania zmian i wyświetlania telemetrii, które może zobaczyć każdy, po to aby uniknąć obarczenia winą za przestoje.

Do innych negatywnych efektów braku dostępnej publicznie telemetrii można zaliczyć złą, polityczną atmosferę, konieczność formułowania oskarżeń i, co gorsza, niezdolność do budowania wiedzy dotyczącej sposobu powstawania incydentów i czerpania z nich nauki niezbędnej do zapobiegania podobnym incydentom w przyszłości*.

Dla odróżnienia telemetria pozwala na wykorzystanie naukowej metody formułowania hipotez na temat przyczyn określonego problemu oraz działań wymaganych do jego rozwiązywania. Oto przykładowe pytania, na które możemy odpowiedzieć podczas rozwiązywania problemów:

* W 2004 roku Gene Kim, Kevin Behr i George Spafford opisali ten stan jako symptom braku „kultury przyczynowości”. Zauważyl, że z doświadczeń firm charakteryzujących się wysoką wydajnością wynika, że 80% wszystkich przestojów jest spowodowanych przez zmiany, a 80% czasu MTTR zajmują próby ustalenia, co się zmieniło.

- Jakie dowody z systemu monitorowania świadczą o tym, że problem faktycznie występuje?
- Jakie są istotne zdarzenia i zmiany w aplikacjach i środowiskach, które mogły przyczynić się do powstania problemu?
- Jakie możemy sformułować hipotezy potwierdzające związek pomiędzy podawanymi przyczynami a skutkami?
- Jak można udowodnić, które z tych hipotez są prawidłowe, i pomyślnie zastosować poprawki?

Wartość rozwiązywania problemów na podstawie faktów leży nie tylko w znacznie krótszym czasie MTTR (i lepszych wynikach dla klientów), ale również we wzmacnieniu postrzegania obopólnie korzystnej relacji pomiędzy działami Dev i Ops.

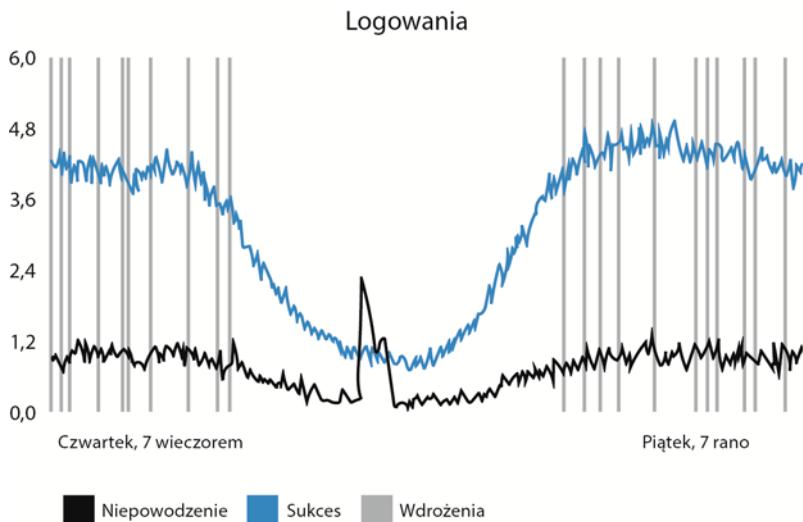
TWORZENIE PARAMETRÓW PRODUKCJI W RAMACH CODZIENNEJ PRACY

Aby umożliwić wszystkim znajdowanie i rozwiązywanie problemów w ich codziennej pracy, musimy umożliwić każdemu uczestnikowi strumienia wartości łatwe tworzenie podczas codziennej pracy parametrów, które można bez trudu wyświetlić i przeanalizować. Aby to zrobić, trzeba stworzyć infrastrukturę i dostarczyć biblioteki niezbędne do tego, aby wszyscy inżynierowie działów Dev i Ops mogli bez trudu tworzyć wskaźniki telemetryczne dla budowanych funkcjonalności. W idealnej sytuacji stworzenie parametrów wyświetlanych na wspólnym pulpicie, dostępnym do oglądania przez wszystkie osoby należące do strumienia wartości, powinno sprowadzać się do napisania linijki kodu.

Taka filozofia stała się przewodnikiem dla rozwoju jednej z najczęściej stosowanych bibliotek obsługi wskaźników telemetrycznych o nazwie StatsD. Biblioteka ta, której nadano status *open source*, powstała w firmie Etsy. Jak opisywał John Allspaw: „Zaprojektowaliśmy bibliotekę StatsD, aby żaden programista nie mógł powiedzieć »wyposażenie mojego kodu w instrumenty do generowania parametrów jest zbyt kłopotliwe«. Teraz można to zrobić za pomocą jednej linijki kodu. Było dla nas ważne, aby zadanie dodania wskaźnika telemetrycznego nie było uważane za równie trudne, jak zmiana schematu bazy danych”.

Biblioteka StatsD umożliwia generowanie timerów i liczników za pomocą jednej linijki kodu (jest dostępna w Ruby, Perlu, Pythonie, Javie i innych językach). Często jest stosowana w połączeniu z narzędziami Graphite lub Grafana, które renderują zdarzenia w postaci wykresów i pulpitów nawigacyjnych.

Na rysunku 27 pokazano przykład stworzenia za pomocą jednej linijki kodu zdarzenia logowania użytkownika (w tym przypadku chodziło o jeden wiersz kodu PHP: `StatsD::increment("login. successes")`). Wynikowy wykres pokazuje liczbę udanych



Rysunek 27. Jedna linijka kodu do wygenerowania wskaźników telemetrycznych przy użyciu biblioteki StatsD i narzędzia Graphite w firmie Etsy (źródło: Ian Malpass, Measure Anything, Measure Everything)

i nieudanych logowań na minutę. Na wykres nałożono pionowe linie reprezentujące wdrożenia w środowisku produkcyjnym.

Podczas generowania wykresów wskaźników telemetrycznych możemy także nakładać na nie zdarzenia zmian produkcyjnych. Wiemy bowiem, że znacząca większość problemów produkcji jest spowodowana przez zmiany w środowisku produkcyjnym. Do tych zmian zaliczają się wdrożenia kodu. Jest to jeden z elementów, które pozwalają na zachowanie wysokiego współczynnika zmian z jednoczesnym zachowaniem bezpiecznego systemu pracy.

Istnieją także biblioteki alternatywne do biblioteki StatsD, które umożliwiają programistom generowanie wskaźników telemetrycznych. Należą do nich biblioteka JMX oraz tzw. parametry codahale. Wśród innych narzędzi do tworzenia parametrów nieocenionych podczas rozwiązywania problemów można wymienić New Relic, AppDynamics i Dynatrace. Podobne funkcjonalności są również dostępne za pośrednictwem takich narzędzi, jak munin i collectd*.

Dzięki generowaniu telemetrycznych wskaźników produkcyjnych w ramach codziennej pracy przyczyniamy się do nabycia coraz lepszej zdolności nie tylko do dostrzegania problemów natychmiast po ich pojawienniu się, ale także do projektowania pracy w taki sposób, aby można było wykrywać problemy w projekcie i działaniu

* Zupełnie inny zestaw narzędzi wspomagający monitorowanie, agregację i zbieranie danych obejmują takie systemy, jak Splunk, Zabbix, Sumo Logic, DataDog, a także Nagios, Cacti, Sensu, RRDTool, Netflix Atlas, Riemann i inne. Analitycy często nazywają tę szeroką kategorię narzędzi „monitorami wydajności aplikacji”.

produkту. Dzięki temu możliwe jest śledzenie coraz większej liczby wskaźników — przekonaliśmy się o tym w studium przypadku Etsy.

TWORZENIE SAMOOPSŁUGOWEGO DOSTĘPU DO TELEMETRII I PROMIENNIKI INFORMACJI

W poprzednich krokach zadbaliśmy o to, aby członkowie zespołów Dev i Ops tworzyli i usprawniali telemetrię produkcji w ramach swojej codziennej pracy. W tym kroku naszym celem jest stworzenie mechanizmów emitowania tych informacji do pozostały części organizacji, co ma dać gwarancję, że każdy, kogo interesują informacje o działających usługach, może je uzyskać bez konieczności dostępu do systemu produkcyjnego lub uprzywilejowanych kont lub konieczności otwierania zlecenia roboczego po to, by ktoś przygotował dla niego wykres.

Dzięki stworzeniu parametrów telemetrycznych, które są szybkie, łatwe do użycia i wystarczająco skonsolidowane, wszyscy uczestnicy strumienia wartości mogą współdzielić wspólne spojrzenie na rzeczywistość. Zazwyczaj oznacza to, że wskaźniki produkcji są publikowane na stronach internetowych generowanych przez skonsolidowany serwer, taki jak Graphite, albo za pośrednictwem innych technologii opisanych w poprzednim podrozdziale.

Chcemy, aby wskaźniki telemetryczne produkcji były dobrze widoczne. To oznacza umieszczenie ich w centralnych obszarach, w których pracują zespoły Dev i Ops. Dzięki temu wszystkie zainteresowane osoby mogą zobaczyć, w jaki sposób działają usługi. Minimum to udostępnienie tych informacji wszystkim osobom w strumieniu wartości — tzn. działom Dev, Ops, kierownictwu produktu oraz działowi Infosec.

Mechanizm udostępniania danych jest często określany jako **promiennik informacji** (ang. *information radiator*). Termin ten został zdefiniowany przez Agile Alliance jako „ogólny termin dla odręcznej, rysowanej, drukowanej lub elektronicznej tablicy umieszczanej przez członków zespołu w widocznym miejscu, tak aby wszyscy członkowie zespołu, a także przechodzący mogli natychmiast zobaczyć najświeższe informacje: liczbę automatycznych testów, czas działania, raporty z incydentów, status ciągłej integracji itp. Ten pomysł pochodzi z systemu Toyota Production System”.

Poprzez umieszczenie radiatorów informacji w dobrze widocznych miejscach promujemy wśród członków zespołu odpowiedzialność, aktywnie prezentując następujące wartości:

- Zespół nie ma nic do ukrycia przed gośćmi (klientami, interesariuszami itp.).
- Zespół nie ma nic do ukrycia przed samym sobą: potwierdza problemy i stawia im czoła.

Teraz, gdy posiadamy infrastrukturę do tworzenia i propagowania wskaźników telemetrycznych na całą organizację, możemy także propagować te informacje do wewnętrznych klientów, a nawet do klientów zewnętrznych. Na przykład możemy to

zrobić poprzez stworzenie dostępnych publicznie stron stanu usługi, tak aby klienci mogli się dowiedzieć, jak działają usługi, od których oni zależą.

Chociaż może istnieć pewien opór przed zapewnieniem przezroczystości na takim poziomie, to jednak warto docenić wartość takiego postępowania. Ernest Mueller opisuje to w następujący sposób:

Jednym z pierwszych działań, które podejmuję w organizacji, jest wdrożenie promienników informacji w celu komunikowania problemów oraz szczegółów wprowadzanych zmian — zazwyczaj jest to bardzo dobrze odbierane przez nasze jednostki biznesowe, które wcześniej często tkwiły w nieswiadomości. Dla zespołów Dev i Ops, które muszą współpracować w celu dostarczenia usługi innym, potrzebujemy stałej komunikacji, informowania oraz opinii zwrotnych.

Tę przezroczystość można rozszerzyć — zamiast dążyć do utrzymania problemów wpływających na klientów w tajemnicy, staramy się propagować te informacje do klientów zewnętrznych. To pokazuje, że cenimy przejrzystość, a tym samym pomagamy w zdobywaniu zaufania klientów (załącznik 10.).^{*}

Studium przypadku

Tworzenie samoobsługowych parametrów w firmie LinkedIn (2011)

Zgodnie z tym, co opisaliśmy w części III, LinkedIn został utworzony w 2003 roku, aby pomóc użytkownikom w nawiązywaniu połączeń „do sieci lepszych miejsc pracy”. W listopadzie 2015 roku LinkedIn miał ponad 350 milionów użytkowników, generujących dziesiątki tysięcy żądań na sekundę, które skutkowały wieloma milionami zapytań na sekundę do serwerów backend serwisu LinkedIn.

Prachi Gupta, dyrektor inżynierii w serwisie LinkedIn, w 2011 roku opisał znaczenie telemetrycznych wskaźników produkcji: „W LinkedIn kładziemy nacisk na zagwarantowanie ciągłego działania witryny oraz tego, by nasi użytkownicy przez cały czas mieli dostęp do kompletnej funkcjonalności serwisu. Realizacja tego zobowiązuje wymaga wykrywania awarii i reagowania na nie oraz wąskie gardła natychmiast po ich pojawienniu się. Dlatego używamy wykresów czasowych do monitorowania witryny, tak by wykrywać incydenty i reagować na nie w ciągu

* Stworzenie prostej tablicy informacyjnej powinno być częścią tworzenia każdego nowego produktu lub usługi — poprawność działania zarówno usługi, jak i tablicy informacyjnej powinny potwierdzać automatyczne testy, które powinny wspomagać klientów i wzmacniać naszą zdolność do bezpiecznego wdrażania kodu.

kilku minut... Ta technika monitorowania okazała się doskonałym narzędziem dla inżynierów. Pozwala nam ona osiągać szybkie postępy oraz daje czas na wykrywanie, selekcję i rozwiązywanie problemów".

Jednak w 2010 r., mimo że generowaliśmy niezwykłe ilości wskaźników telemetrycznych, to inżynierowie mieli bardzo duże trudności, by uzyskać dostęp do tych informacji, nie mówiąc już o możliwości ich analizowania. W ten sposób rozpoczął się w firmie LinkedIn letni, wewnętrzny projekt kierowany przez Erica Wonga. Projekt ten przekształcił się w inicjatywę telemetrii produkcji, która doprowadziła do powstania narzędzia InGraphs.

Wong pisał: „Żeby uzyskać coś tak prostego, jak użycie procesora przez wszystkie hosty, na których była uruchomiona określona usługa, trzeba było wypełnić zlecenie robocze i ktoś musiał poświęcić 30 minut na przygotowanie raportu".

W tamtym czasie do zbierania danych w firmie LinkedIn używano narzędzia Zenoss, ale jak wyjaśnia Wong: „Pobranie danych z systemu Zenoss wymagało »przekopywania się« przez wolny interfejs webowy. Z tego powodu napisałem kilka skryptów w Pythonie, które miały pomóc w usprawnieniu procesu. Chociaż konfiguracja zbierania danych nadal wymagała ręcznej interwencji, to udało mi się skrócić czas, jaki wcześniej musiałem poświęcić na nawigowanie po interfejsie Zenoss".

W ciągu lata Wong dodawał funkcjonalności do systemu InGraphs, dzięki czemu inżynierowie mogli dokładnie zobaczyć to, co chcieli. Wprowadził także możliwość wykonywania obliczeń na wielu zestawach danych, przeglądania trendów tygodni po tygodniu w celu porównywania historycznych danych wydajności, a nawet definiowania niestandardowych pulpitów nawigacyjnych, pozwalających wybrać parametry do wyświetlania na jednej stronie.

Pisząc o efektach dodawania funkcjonalności do systemu InGraphs oraz wartości tych funkcjonalności, Gupta zauważyl: „Skuteczność naszego systemu monitorowania została doceniona natychmiast po tym, gdy nasza funkcjonalność monitorowania w systemie InGraphs powiązana z jednym z głównych dostawców usług pocztowych przez WWW zaczęła pokazywać trendy spadkowe, a dostawca zdał sobie sprawę, że występuje u niego problem dopiero wtedy, gdy się z nim skontaktowaliśmy!".

To, co zaczęło się jako letni projekt stażowy, obecnie jest jedną z najbardziej widocznych części działań operacyjnych w serwisie LinkedIn. InGraphs okazał się tak skuteczny, że generowane w czasie rzeczywistym wykresy są umieszczone w widocznych miejscach w pomieszczeniach inżynierów, w taki sposób, że goście nie mogą ich nie zauważać.

ZNAJDOWANIE I WYPEŁNIANIE WSZELKICH LUK W TELEMETRII

Stworzyliśmy infrastrukturę niezbędną do szybkiego generowania telemetrycznych wskaźników produkcyjnych dla całego stosu aplikacji i propagowania tych informacji w całej organizacji.

W tym kroku postaramy się zidentyfikować wszelkie luki w mechanizmach telemetrii, które przeszkadzają w zdolności szybkiego wykrywania i rozwiązywania problemów — jest to szczególnie istotne, gdy działa Dev i Ops dotychczas stosują telemetrię tylko w niewielkim stopniu (bądź nie stosują jej wcale). Z danych tych skorzystamy później w celu lepszego przewidywania problemów, jak również po to, by umożliwić wszystkim zbieranie informacji potrzebnych do podejmowania lepszych decyzji, a przez to osiąganie celów wyznaczonych przez organizację.

Osiągnięcie tego celu wymaga stworzenia wystarczającej liczby mechanizmów telemetrycznych na wszystkich poziomach stosu aplikacji we wszystkich środowiskach, jak również w potokach wdrożeń, które je obsługują. Potrzebujemy parametrów z następujących poziomów:

- **Poziom biznesowy** — do przykładów należy liczba transakcji sprzedaży, przychody ze sprzedaży, zarejestrowani użytkownicy, wskaźnik migracji, wyniki testów A/B itp.
- **Poziom aplikacji** — przykłady to czas transakcji, czas odpowiedzi użytkownika, błędy aplikacji itp.
- **Poziom infrastruktury (np. baza danych, system operacyjny, sieć, pamięć masowa)** — przykładami są ruch do serwera WWW, obciążenie procesora, użycie dysku itp.
- **Poziom oprogramowania klienckiego (np. JavaScript w przeglądarce klienta, aplikacja mobilna)** — przykładami mogą być błędy i awarie aplikacji, czas transakcji mierzony przez użytkowników itp.
- **Poziom potoku wdrożeń** — przykładami są stan potoku budowania (np. czerwony lub zielony dla różnych zestawów testów automatycznych), czasy realizacji wdrażania zmian, częstotliwość wdrożeń, promocje środowiska testowego oraz status środowiska.

Dzięki pokryciu wskaźnikami telemetrycznymi we wszystkich tych obszarach możemy zaobserwować kondycję wszystkich komponentów, na których bazuje usługa, z wykorzystaniem danych i faktów zamiast plotek, wskazywania palcem, obwiniania itd.

Ponadto dzięki monitorowaniu wszelkich błędów aplikacji i infrastruktury (np. nieprawidłowe zakończenie działania programu, błędy aplikacji i wyjątki oraz błędy serwera i pamięci masowej) umożliwiamy lepsze wykrywanie incydentów zabezpieczeń.

Mechanizmy telemetrii nie tylko dokładniej informują zespoły Dev i Ops o awariach usług, ale błędy te są często wskaźnikami aktywnego wykorzystywania luk w za-bezpieczeniach.

Dzięki szybszemu wykrywaniu i korygowaniu problemów można je rozwiązywać w czasie, gdy ich rozmiary są niewielkie i kiedy są łatwe do rozwiązania oraz wywie-rają mniejszy wpływ na klientów. Ponadto po każdym incydencie w produkcji należy zidentyfikować brakujące wskaźniki telemetryczne, które mogłyby pomóc w szybszym wykryciu awarii i przywróceniu prawidłowego stanu, albo, co byłoby jeszcze lepsze, luki te powinny być zidentyfikowane w procesie przeglądania kodu przez współpracowników.

PARAMETRY POZIOMU APLIKACJI I WSKAŹNIKI BIZNESOWE

Na poziomie aplikacji naszym celem powinno być zadbanie o wygenerowanie wskaźników telemetrycznych nie tylko takich, które dotyczą kondycji aplikacji (np. użycie pamięci, liczba transakcji itd.), ale również takich, które potrafią zmierzyć, w jakim stopniu osiągamy wyznaczone cele organizacyjne (np. liczba nowych użytkowników, zdarzenia logowania użytkowników, czas trwania sesji użytkownika, procent aktywnych użytkowników, częstotliwość używania określonych funkcji itd.).

Na przykład, jeśli mamy usługę wspierającą witrynę e-commerce, to powinniśmy zadbać o generowanie telemetrii dotyczących wszystkich zdarzeń użytkownika prowadzących do udanej transakcji, która wygenerowała przychody. Następnie można wypo-sażyć w mechanizmy generowania wskaźników telemetrycznych wszystkie działania użytkowników, które są wymagane do osiągnięcia pożądanego wpływu na klientów.

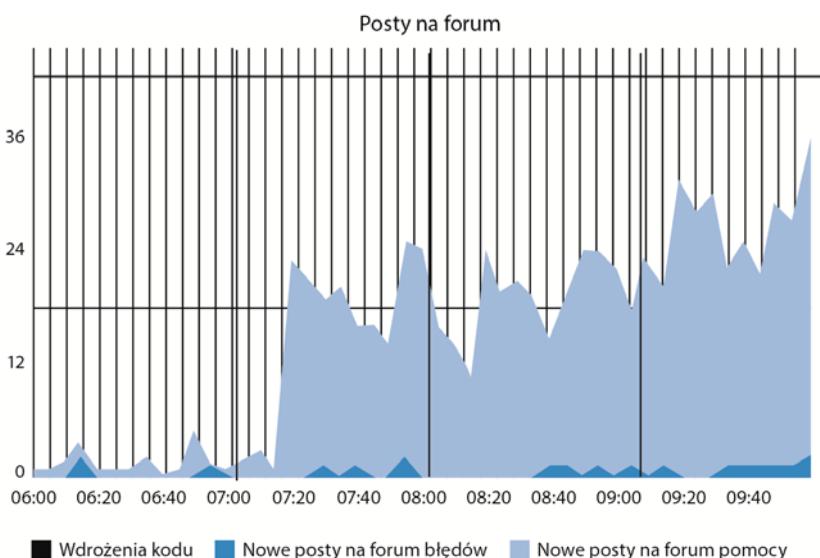
Wskaźniki te mogą być różne w zależności od różnych dziedzin i celów organizacji. Na przykład dla witryn e-commerce może być korzystne zmaksymalizowanie czasu spędzonego na stronie, z kolei w przypadku wyszukiwarek możemy dążyć do skrócenia tego czasu, ponieważ długie sesje mogą wskazywać na to, że użytkownicy mają trudności ze znalezieniem tego, czego szukają.

Ogólnie rzecz biorąc, wskaźniki biznesowe są częścią tzw. lejka nabycia klienta (ang. *customer acquisition funnel*), który określa teoretyczne kroki, które potencjalny klient podejmuje podczas dokonywania zakupu. Na przykład w witrynie e-commerce do mierzalnych zdarzeń pobytu można zaliczyć całkowity czas w witrynie, kliknięcia łączące produktu, dodawanie do koszyka na zakupy oraz wypełnione zamówienia.

Ed Blankenship, starszy menedżer produktu usługi Microsoft Visual Studio Team Services, opisuje: „Często zespół pracujący nad cechą funkcjonalną definiuje własne cele w lejku nabycia, dając do tego, by własność funkcjonalna, nad którą pracuje, była wykorzystywana w codziennej pracy wszystkich klientów. Czasami są oni niefor-malnie opisywani jako »oglądacze«, »aktywni użytkownicy«, »zaangażowani użytkownicy« i »głęboko zaangażowani użytkownicy«; wskaźniki telemetryczne są związane z każdym rodzajem klienta”.

Naszym celem jest to, aby każdy wskaźnik biznesowy był inspiracją do działania — są to najważniejsze wskaźniki, które powinny pomóc w informowaniu o sposobie zmian w produkcie oraz aby skłaniały do eksperymentów i wykonywania testów A/B. Gdy wskaźniki nie skłaniają do działania, to z dużym prawdopodobieństwem są to jedynie próżne liczby dostarczające mało przydatnych informacji — można je gromadzić, ale prawdopodobnie nie ma sensu ich wyświetlanie, nie mówiąc już o inicjowaniu alertów z ich powodu.

W idealnej sytuacji każda osoba przeglądająca radiatory informacji powinna potrafić nadać sens prezentowanym informacjom w kontekście pożądanych rezultatów organizacyjnych, takich jak cele dotyczące przychodów, zdobywania użytkowników, współczynników konwersji itp. (przykład tego rodzaju prezentacji pokazano na rysunku 28). Należy zdefiniować i powiązać każdy parametr ze wskaźnikiem wyniku biznesowego na jak najwcześniejjszym etapie definiowania cechy funkcjonalnej oraz jej rozwoju i zmierzyć wyniki po jej wdrożeniu do produkcji. Ponadto postępowanie w ten sposób pomaga właścicielom produktu opisywać kontekst biznesowy każdej cechy funkcjonalnej dla wszystkich osób należących do strumienia wartości.



Rysunek 28. Miara emocji użytkowników związanych z nowymi funkcjonalnościami dotyczącymi postów na forum po wdrożeniach (źródło: Mike Brittain, „Tracking Every Release”, CodeasCraft.com, 8 grudnia 2010, <https://codeascraft.com/2010/12/08/track-every-release/>)

Można także stworzyć dalszy kontekst biznesowy poprzez uświadomienie sobie i zaprezentowanie w wizualny sposób okresów istotnych dla wysokopoziomowego planowania biznesowego — na przykład długich okresów transakcji związanych ze szczytem sezonowych wyprzedaży, okresów podsumowań finansowych na koniec

kwartału lub zaplanowanych kontroli zgodności. Te informacje mogą być wykorzystywane jako przypomnienie, że nie należy planować ryzykownych zmian wtedy, gdy dostępność ma kluczowe znaczenie, lub że należy unikać pewnych działań podczas trwania kontroli.

Propagowanie informacji dotyczących interakcji klientów z budowanymi produktami w kontekście naszych celów umożliwia generowanie szybkich informacji zwrotnych dla przyszłych klientów, dzięki czemu możemy się przekonać, czy budowane funkcjonalności są faktycznie używane i do jakiego stopnia przyczyniają się do osiągnięcia celów biznesowych. W rezultacie wzmacniamy kulturowe oczekiwania co do tego, że pomiary i analiza sposobu korzystania z produktu przez klientów są również częścią naszej codziennej pracy. Dzięki temu potrafimy lepiej zrozumieć, w jaki sposób nasza praca przyczynia się do osiągania celów organizacji.

PARAMETRY INFRASTRUKTURY

Podobnie jak w przypadku parametrów aplikacji celem dla infrastruktury produkcyjnej i nieprodukcyjnej jest zadbanie o generowanie wystarczającej liczby parametrów do tego, by w sytuacji pojawienia się problemu w dowolnym środowisku można było szybko ustalić, czy to infrastruktura przyczyniła się do pojawienia się problemu. Ponadto musimy mieć możliwość dokładnego wskazania tych elementów infrastruktury, które przyczyniły się do jego powstania (np. baza danych, system operacyjny, pamięć masowa, sieć itp.).

Należy zadbać o to, aby była widoczna jak największa część telemetrii infrastruktury — dla wszystkich interesariuszy technicznych. Najlepiej by było, żeby parametry infrastruktury były zorganizowane według usługi lub według aplikacji. Innymi słowy, gdy jakiś element środowiska zawiedzie, to musimy dokładnie wiedzieć, na jakie aplikacje bądź usługi może to wywrzeć wpływ lub wywiera wpływ*.

W poprzednich dziesięcioleciach tworzenie powiązań pomiędzy usługą a infrastrukturą, od której ona zależała, wymagało działań ręcznych (na przykład korzystania z baz danych ITIL CMDB lub tworzenia definicji konfiguracji wewnętrz narzędzi generowania ostrzeżeń takich jak Nagios). Jednak coraz częściej te powiązania są rejestrowane automatycznie w ramach usług, a następnie dynamicznie wykrywane i wykorzystywane w produkcji za pomocą takich narzędzi, jak Zookeeper, Etcd, Consul itp.

Narzędzia te umożliwiają usługom autonomiczną rejestrację oraz przechowywanie informacji, które mogą być potrzebne innym usługom do interakcji (np. adres IP, numery portów, adresy URI). To eliminuje problem związany z nieautomatycznym charakterem bazy danych ITIL CMDB i jest absolutnie niezbędne w sytuacji, gdy

* Dokładnie według zaleceń bazy danych ITIL Configuration Management Database (CMDB).

usługi składają się z setek (lub tysięcy albo nawet milionów) węzłów, z których każdy ma dynamicznie przypisany adres IP*.

Tworzenie wykresów na podstawie wskaźników biznesowych wraz z parametrami aplikacji i infrastruktury — niezależnie od złożoności usług — pozwala wykryć w nich problematyczne sytuacje. Na przykład możemy zaobserwować, że liczba rejestracji nowych klientów spadła do poziomu 20% dziennej normy, a następnie natychmiast zauważać, że wszystkie zapytania do bazy danych trwają pięć razy dłużej niż zwykle. To pozwala skoncentrować się podczas rozwiązywania problemu na odpowiednim obszarze.

Ponadto wskaźniki biznesowe tworzą kontekst dla parametrów infrastruktury, co umożliwia pracownikom działów Dev i Ops lepiej pracować, aby osiągnąć wspólne cele. Jak zaobserwował Jody Mulkey, dyrektor techniczny firmy Ticketmaster (LiveNation): „Zamiast mierzyć aktywność działu operacyjnego pod kątem czasu przestojów, dużo lepiej jest zmierzyć działania zarówno Dev, jak i Ops pod kątem rzeczywistych konsekwencji biznesowych przestojów: jakie przychody powinniśmy byli osiągnąć, ale ich nie osiągnieliśmy”†.

Należy zauważać, że oprócz monitorowania usług produkcyjnych potrzebne są także wskaźniki telemetryczne dla tych usług w środowiskach przedprodukcyjnych (np. programistycznym, testowym, przedwdrożeniem — ang. *staging*). Dzięki temu możemy znaleźć i rozwiązać problemy, zanim trafią do produkcji. Na przykład możemy wykryć, że z powodu brakującego indeksu tabeli mamy coraz dłuższe czasy wprowadzania rekordów do bazy danych.

NAKŁADANIE INNYCH ISTOTNYCH INFORMACJI NA WSKAŹNIKI

Nawet jeśli stworzyliśmy potok wdrożeń, który pozwala wprowadzać niewielkie i częste zmiany w produkcji, to zmianom zawsze towarzyszą zagrożenia. Niepożądanymi skutkami ubocznymi są nie tylko przestoje, ale także znaczne zakłócenia i odchylenia od zachowań standardowych.

Aby zmiany były widoczne, należy o to zadbać poprzez nałożenie na tworzone wykresy wszystkich działań związanych z wdrożeniami do produkcji. Na przykład dla usługi, która obsługuje dużą liczbę przychodzących transakcji, zmiany w produkcji mogą powodować znacznej długości **okres rozstrzygania**, w którym obniża się wydajność ze względu na chybione wyszukiwania w pamięci podręcznej.

* Na szczególną uwagę zasługuje narzędzie Consul, które tworzy warstwę abstrakcji pozwalającą na łatwe korzystanie z funkcjonalności mapowania usług, monitorowania, blokad, magazynów konfiguracji klucz-wartość, a także mechanizmów host clustering i wykrywania awarii.

† Mogą to być koszty przestojów produkcyjnych lub koszty wynikające z opóźnienia we wdrożeniu cechy funkcjonalnej. Zgodnie z terminologią rozwoju produktu drugi parametr jest określany jako koszty opóźnień i jest kluczem do skutecznych decyzji dotyczących priorytetów.

Aby lepiej zrozumieć i zachować jakość usług, trzeba umieć ustalić, jak szybko wydajność wróci do normy, a w razie potrzeby podjąć kroki zmierzające do poprawy wydajności.

W podobny sposób warto nakładać na wykresy informacje o innych działańach operacyjnych, na przykład kiedy usługa jest poddawana konserwacji lub jest tworzona kopia zapasowa. Należy to robić w tych miejscach, gdzie chcemy wyświetlać alerty lub je pomijać.

PODSUMOWANIE

Usprawnienia uzyskane dzięki zastosowaniu telemetrii produkcji w firmach Etsy i LinkedIn pokazują, jak ważna jest zdolność zauważania problemów w chwili, gdy występują. Dzięki temu możemy znaleźć ich przyczyny i szybko zaradzić powstałej sytuacji. Poprzez wyposażenie wszystkich elementów usługi — niezależnie od tego, czy jest to aplikacja, baza danych, czy środowisko — w mechanizmy emitujące wskaźniki telemetryczne oraz powszechnie udostępnienie tych wskaźników możemy znaleźć i rozwiązać problemy na długo, zanim spowodują problemy, najlepiej na długo przedtem, zanim klient zauważy, że coś dzieje się nie tak. W rezultacie nie tylko mamy bardziej zadowolonych klientów, ale poprzez zmniejszenie liczby akcji „gaszenia pożarów” i kryzysów, gdy coś idzie nie tak, mamy korzystniejsze i bardziej wydajne środowisko pracy, z mniejszą ilością stresu i niższym poziomem wypalenia zawodowego.

Analizowanie telemetrii w celu lepszego przewidywania problemów i realizowania zadań

Jak dowiedzieliśmy się z poprzedniego rozdziału, w aplikacjach i infrastrukturze potrzebna jest odpowiednia liczba mechanizmów telemetrycznych, które pozwalają zauważać problemy oraz je rozwiązywać bezpośrednio po ich wystąpieniu. W tym rozdziale stworzymy narzędzia, które pozwalają odkryć odchylenia oraz coraz słabsze sygnały ukryte w produkcyjnych wskaźnikach telemetrycznych, dzięki czemu możemy uniknąć katastrofalnych awarii. Zaprezentujemy różne techniki statystyczne wraz ze studiami przypadków ich zastosowań.

Doskonały przykład analizy telemetrii w celu proaktywnego znajdowania i rozwiązywania problemów, zanim wywrą one ujemny wpływ na klientów, można zaobserwować w firmie Netflix, która jest globalnym dostawcą przesyłanych strumieniowo filmów i seriali telewizyjnych. W 2015 roku firma Netflix osiągnęła przychody sięgające 6,2 miliarda dolarów, generowane dzięki 75 milionom subskrybentów. Jednym z celów firmy było zapewnienie jak najlepszego komfortu użytkownikom oglądającym nagrania wideo online na całym świecie. Wymagało to rozbudowanej, skalowanej i niezawodnej infrastruktury dostarczania. Roy Rapoport opisał jedno z wyzwań zarządzania usługą Netflix dostarczającą nagrania wideo w chmurze: „Jeśli popatrzymy na stado bydła, gdzie wszystkie sztuki wyglądają i funkcjonują w ten sam sposób, to czy będziemy w stanie znaleźć tę sztukę, która wygląda inaczej niż reszta? Albo konkretniej, jeśli mamy kластer obliczeniowy złożony z tysięcy węzłów, na których jest uruchomione

to samo oprogramowanie i wszystkie w przybliżeniu są poddane temu samemu obciążeniu, wyzwaniem jest znalezienie tych węzłów, które nie wyglądają tak jak pozostałe”.

Jedną z technik statystycznych, zastosowanych przez Netflix w 2012 roku, było tzw. **wykrywanie odstających** (ang. *outlier detection*) — pojęcie zdefiniowane przez Victorię J. Hodge i Jima Austina z University of York jako wykrywanie „nieprawidłowych warunków działania, które mogą spowodować znaczące obniżenie wydajności, na przykład defekt obrotów silnika samolotu lub problem przepływu w rurociągu”.

Rapoport wyjaśnił, że Netflix „zastosował technikę wykrywania odstających w bardzo prosty sposób. Najpierw dla określonej populacji węzłów w klastrze komputerowym obliczono »bieżący stan normalny«. Następnie zidentyfikowano węzły, które nie pasują do tego wzorca, i usunięto je z produkcji”.

Rapoport kontynuował: „Możemy automatycznie oznaczyć niewłaściwie działające węzły bez konieczności rzeczywistego definiowania, co właściwie oznacza »prawidłowe« zachowanie. A ponieważ nasza usługa została zaprojektowana tak, aby mogła działać niezawodnie w chmurze, to nie zlecamy nikomu w dziale operacyjnym, żeby cokolwiek robił — po prostu zabijamy działający nieprawidłowo węzeł komputerowy, a następnie zapisujemy tę informację w logu albo powiadamiamy inżynierów w dowolny sposób”.

Rapoport stwierdził, że „dzięki zaimplementowaniu procesu Server Outlier Detection firma Netflix zdecydowanie zredukowała wysiłek związany z wykrywaniem nieprawidłowo działających serwerów, a co ważniejsze znacznie skróciła czas potrzebny do ich naprawienia, co przyczyniło się do poprawy jakości usługi. Korzyści z zastosowania tych technik w postaci komfortu pracy pracowników, równowagi pomiędzy pracą a życiem prywatnym oraz wzrostu jakości usług są nieocenione”. Prace przeprowadzone w firmie Netflix podkreślają bardzo specyficzny sposób wykorzystania telemetrii do łagodzenia problemów, zanim zdążą one wywrzeć ujemny wpływ na klientów.

W całym niniejszym rozdziale zbadamy wiele technik statystycznych i wizualizacyjnych (włącznie z techniką wykrywania odstających), które można wykorzystać do analizy mechanizmów telemetrycznych w celu lepszego przewidywania problemów. Umożliwia to rozwiązywanie problemów szybciej, taniej i wcześniej niż kiedykolwiek — zanim zdążą wpłynąć na klientów lub kogokolwiek w organizacji. Dodatkowo tworzący jest szerszy kontekst dla danych, co pozwala na podejmowanie lepszych decyzji i osiąganie celów organizacyjnych.

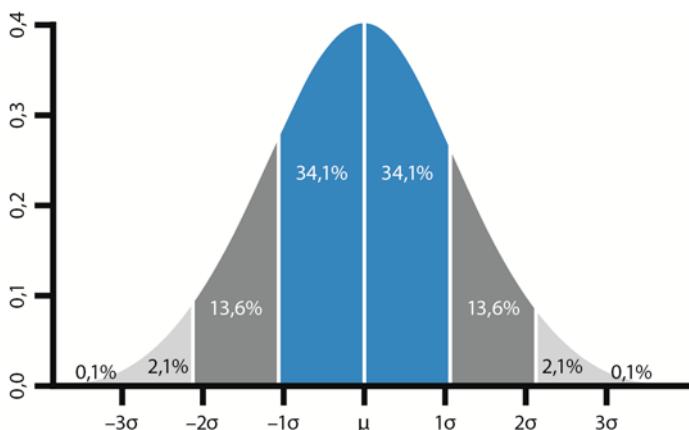
WYKORZYSTANIE WARTOŚCI ŚREDNICH ORAZ ODHYLEŃ STANDARDOWYCH W CELU WYKRYWANIA POTENCJALNYCH PROBLEMÓW

Jedną z najprostszych technik statystycznych, które można wykorzystać do analizy parametrów produkcyjnych, jest obliczanie **średnich** oraz **odchyleń standardowych**. W ten sposób można stworzyć filtr, który wykrywa sytuację, gdy parametr ten różni się

znacząco od normy, a nawet skonfigurować mechanizm ostrzegania, tak aby można było podjąć działania naprawcze (np. powiadomić dyżurujących pracowników o 2 nad ranem, by sprawdzili, co się dzieje, jeśli obsługa zapytań do bazy danych przebiega znacząco wolniej, niż wynosi średnia).

Kiedy problemy występują w kluczowych usługach produkcyjnych, to budzenie ludzi o 2 w nocy może być właściwe. Jednak w przypadku alertów, które nie dają podstaw do działania lub które są fałszywymi alarmami, możemy niepotrzebnie obudzić ludzi w środku nocy. Jak zaobserwował John Vincent, jeden z pierwszych liderów ruchu DevOps: „Zmęczenie alertami to obecnie nasz największy problem... Musimy generować alerty w sposób bardziej inteligentny albo wszyscy oszalejemy”.

Lepsze alerty tworzymy, zwiększając współczynnik sygnału do szumów, koncentrując się na odchyleniach od normy lub znaczących egzemplarzach odstających od reszty. Założymy, że analizujemy liczbę prób nieuprawnionego logowania dziennie. Zebrane dane mają rozkład Gaussa (tzn. rozkład normalny lub krzywa dzwonowa), które odpowiadają wykresowi na rysunku 29. Pionowa linia na środku krzywej dzwonowej oznacza średnią. Pozostałe pionowe linie oznaczają pierwsze, drugie i trzecie odchylenie standardowe. Obejmują one odpowiednio 68%, 95% i 99,7% danych.



Rysunek 29. Odchylenia standardowe (σ) i średnia (μ) z rozkładu Gaussa (źródło: artykuł „Rozkład normalny” w Wikipedii, https://pl.wikipedia.org/wiki/Rozk%C5%82ad_normalny)

Powszechnym zastosowaniem odchyleń standardowych są okresowe testy zbiorów danych pod kątem określonego parametru i generowanie alertu w przypadku, gdy jakaś wartość znacznie odbiega od średniej. Na przykład można ustawić alert, gdy liczba prób nieuprawnionego logowania dziennie przekroczy wartość trzech odchyleń standardowych powyżej średniej. Jeśli ten zbiór danych ma rozkład Gaussa, to możemy oczekiwać, że tylko 0,3% punktów danych spowoduje wyzwolenie alertu.

Nawet ten prosty rodzaj analizy statystycznej jest cenny, ponieważ nie wymaga definiowania statycznych wartości progowych — co jest niemożliwe w przypadku śledzenia tysięcy lub setek tysięcy parametrów produkcyjnych.

W pozostałej części tej książki terminów **telemetria**, **parametry** i **zestawy danych** będziemy używać zamiennie — innymi słowy, **parametr** (np. „czas ładowania strony”) będzie mapowany na **zestaw danych** (np. 2 ms, 8 ms, 11 ms itp.) — tzn. termin używany przez statystyków do określenia macierzy punktów danych, w której każda kolumna reprezentuje zmienną, dla której wykonywane są operacje statystyczne.

INSTRUMENTACJA I ALERTY NA NIEPOŻĄDANE WYNIKI

Tom Limoncelli, współautor książki *The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems* oraz były inżynier niezawodności witryny w firmie Google, zrelacjonował następującą historię dotyczącą monitorowania: „Kiedy ludzie pytają mnie o zalecenia na temat tego, co monitorować, żartuję, że w idealnym świecie należałoby usunąć wszystkie alerty, które obecnie mamy w naszym systemie monitorowania. Następnie po każdej awarii u użytkownika trzeba by było zapytać, jakie wskaźniki mogły przewidzieć awarię, a następnie dodać je do systemu monitorowania wraz z odpowiednio skonfigurowanymi alertami. Powtarzam — teraz mamy skonfigurowane tylko te alerty, które zapobiegają awariom. Nie chcemy być bombardowani alertami już po wystąpieniu awarii”.

W tym kroku postaramy się zreplikować podobne przedsięwzięcie. Jednym z najprostszych sposobów, aby to zrobić, jest analiza najpoważniejszych incydentów z przeszłości (np. 30 dni) i utworzenie listy wskaźników telemetrycznych, które mogły spowodować wcześniejsze wykrycie i szybsze zdiagnozowanie problemu, a także łatwiejsze i szybsze potwierdzenie zastosowania skutecznej naprawy.

Na przykład w przypadku problemu polegającego na zaprzestaniu odpowiadania na żądania przez serwer NGINX przyjrzelibyśmy się wiodącym wskaźnikom, które mogły ostrzec nas wcześniej o odchyleniach względem standardowych działań. Mogłyby to być:

- **Poziom aplikacji** — wydłużenie czasu ładowania strony WWW itp.
- **Poziom systemu operacyjnego** — zaczyna spadać ilość wolnej pamięci na serwerze, wyczerpuje się miejsce na dysku itp.
- **Poziom bazy danych** — transakcje bazodanowe trwają dłużej niż zwykle itp.
- **Poziom sieci** — spada liczba serwerów działających za mechanizmem równoważenia obciążenia itp.

Każdy z tych parametrów jest potencjalnym prekursorem incydentu produkcyjnego. Dla każdego z nich należałoby skonfigurować systemy alarmowania, tak aby w sytuacji

znaczących odchyleń od średniej były generowane powiadomienia. Dzięki temu można by podjąć działania naprawcze.

Dzięki powtarzaniu tego procesu dla coraz słabszych sygnałów awarii można znaleźć problemy na jeszcze wcześniejszym etapie cyklu życia produktu. To powinno skutkować mniejszą liczbą incydentów oraz sytuacji niebezpiecznych ujemnie wpływających na klientów. Innymi słowy, zapobiegamy problemom, a równocześnie pozwalamy na ich szybsze wykrywanie i korygowanie.

PROBLEMY POWSTAJĄCE W PRZYPADKU, GDY DANE TELEMETRYCZNE NIE MAJĄ ROZKŁADU GAUSSA

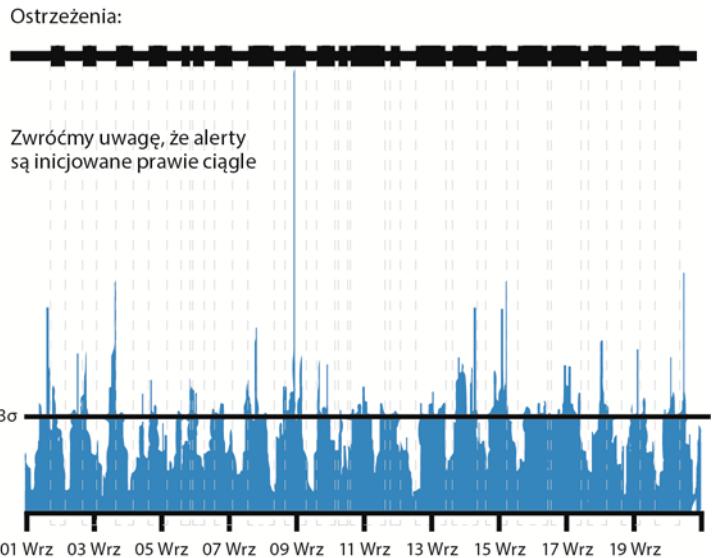
Wykorzystywanie wartości średnich oraz odchyleń standardowych do wykrywania odstępstw od prawidłowego działania może być bardzo przydatne. Jednak stosowanie tych technik dla wielu zestawów danych telemetrycznych, które wykorzystujemy w działaniach operacyjnych, nie przyniesie pożądanych rezultatów. Jak zauważył dr Toufic Boubez: „Będziemy budzeni nie tylko o 2.00 w nocy, ale także o 2.37, 4.13 i 5.17. Tak się będzie działało w przypadku, gdy monitorowane dane nie mają rozkładu Gaussa”.

Innymi słowy, gdy zestaw danych nie ma opisanego wcześniej rozkładu w postaci krzywej dzwonowej Gaussa, to właściwości związane z odchyleniami standardowymi nie mają zastosowania. Rozważmy scenariusz, w którym monitorujemy liczbę operacji pobierania pliku ze strony internetowej na minutę. Chcemy wykryć okresy, w których mamy nienaturalnie dużą liczbę pobrań — na przykład gdy tempo pobierania jest większe niż trzy odchylenia standardowe od średniej. Dzięki temu możemy proaktywnie zwiększać przepustowość.

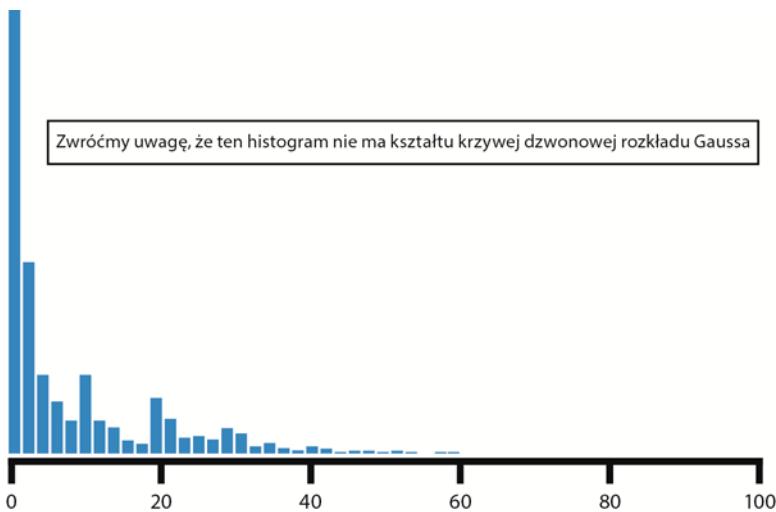
Na rysunku 30 zaprezentowano liczbę równoczesnych operacji pobierania na minutę z poziomym paskiem nałożonym na górną część wykresu. Kiedy pasek jest czarny, liczba pobrań w danym okresie (czasami nazywanym „oknem przesuwnym”) ma wartość różniącą się o co najmniej trzykrotność odchylenia standardowego od średniej. W przeciwnym razie jest szary.

Oczywistym problemem widocznym na wykresie jest generowanie alertów niemal przez cały czas. Powodem jest to, że prawie w każdym momencie zdarzają się przypadki, w których liczba pobrań przekracza próg trzech odchyleń standardowych.

Aby to potwierdzić, wystarczy utworzyć histogram (rysunek 31), który pokazuje częstotliwość pobrań na minutę. Jak można zauważyć, wykres nie ma klasycznego, symetrycznego kształtu krzywej dzwonowej. Wyraźnie widać, że wykres rozkładu jest pochylony ku dołowi, co pokazuje, że przez większość czasu liczba pobrań na minutę jest bardzo niska, ale licznik pobrań często skacze powyżej trzech odchyleń standardowych.



Rysunek 30. Liczba pobrań plików na minutę — nadmierne alarmowanie w przypadku zastosowania reguły „trzech odchylen standardowych” (źródło: dr Toufic Boubez, „Simple math for anomaly detection”)



Rysunek 31. Liczba pobrań plików na minutę — histogram danych nie ma kształtu rozkładu Gaussa (źródło: dr Toufic Boubez, „Simple math for anomaly detection”)

Wiele produkcyjnych zestawów danych nie ma rozkładu Gaussa. Jak wyjaśniła dr Nicole Forsgren: „W działaniach zespołów operacyjnych zbiory danych mają rozkład »chi kwadrat«. Korzystanie z odchyleń standardowych dla takich danych nie tylko skutkuje nadmierną liczbą alertów, ale także może powodować generowanie

wyników pozbawionych sensu". Dalej dr Forsgren mówi: „Gdy obliczamy liczbę równoczesnych pobrania plików jako trzy odchylenia standardowe poniżej średniej, możemy uzyskać liczbę ujemną, co oczywiście nie ma sensu".

Nadmierne ostrzeganie skutkuje budzeniem inżynierów operacyjnych w środku nocy na coraz dłuższe okresy nawet wtedy, gdy nie mogą zbyt wiele zrobić. Problem związany z niedostatecznym ostrzeganiem jest również istotny. Założymy, że monitorujemy liczbę zakończonych transakcji, która spadła o 50% w środku dnia z powodu awarii komponentu oprogramowania. Jeśli spadek ciągle mieści się w limicie trzech odchyleń standardowych od średniej, to nie zostanie wygenerowany żaden alert, co oznacza, że klienci odkryją problem przed nami, a wtedy rozwiążanie problemu może być znacznie bardziej trudne.

Na szczęście istnieją techniki, których możemy użyć do wykrycia anomalii nawet w zestawach danych, które nie mają rozkładu Gaussa. Techniki te zostaną opisane poniżej.

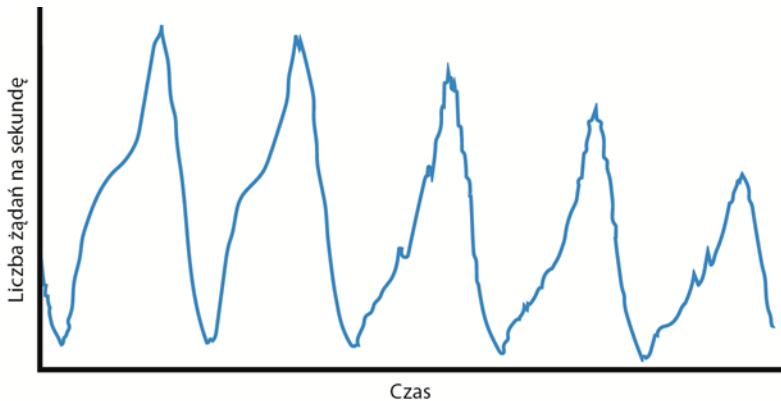
Studium przypadku

Automatyczne skalowanie możliwości obliczeniowych w firmie Netflix (2012)

Inne narzędzie służące do poprawy jakości usług opracowane w firmie Netflix — Scryer — rozwiązuje pewne uchybienia narzędzia Amazon Auto Scaling (AAS), które dynamicznie zwiększa i zmniejsza liczbę serwerów obliczeniowych AWS na podstawie danych dotyczących obciążenia. Zasada działania narzędzia Scryer polega na przewidywaniu żądań klienta na podstawie historycznych wzorców użycia oraz konfigurowaniu potrzebnych możliwości obliczeniowych.

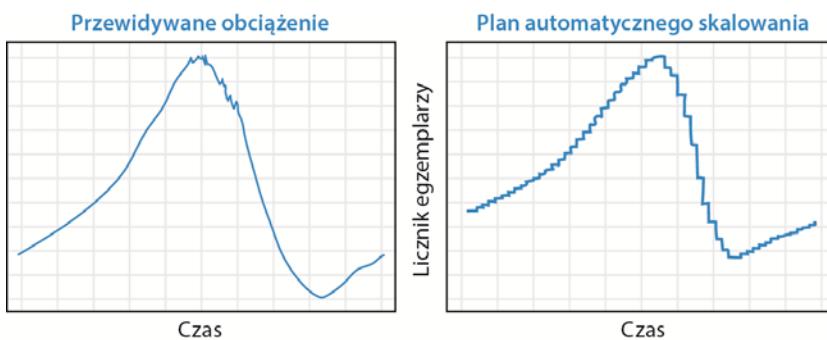
Scryer rozwiązuje trzy problemy mechanizmu AAS. Pierwszy dotyczy obsługi szybkiego wzrostu zapotrzebowania. Ponieważ czasy uruchamiania egzemplarza AWS mogą wynosić od 10 do 45 minut, często dodatkowe możliwości obliczeniowe były dostarczone za późno i nie mogły zaradzić nagłemu wzrostrom zapotrzebowania. Drugi problem polegał na tym, że nagły spadek zapotrzebowania klientów na obliczenia po przestojach doprowadził do tego, że mechanizm AAS zbyt mocno obniżał moc obliczeniową do obsługi przyszłych żądań przychodzących. Trzecim problemem było to, że podczas planowania możliwości obliczeniowych w mechanizmie AAS nie brano pod uwagę znanych wzorców użycia usługi.

W firmie Netflix wykorzystano fakt, że wzorce oglądalności konsumentów były zaskakująco spójne i przewidywalne, mimo że nie miały rozkładu Gaussa. Na rysunku 32 zamieszczono wykres liczby żądań klientów na sekundę przez cały tydzień pracy. Widać na nim regularne i stałe wzorce oglądalności od poniedziałku do piątku.



Rysunek 32. Zapotrzebowanie na moc obliczeniową klientów usługi Netflix w ciągu pięciu dni
 (źródło: Daniel Jacobson, Danny Yuan i Neeraj Joshi, „Scryer: Netflix's Predictive Auto Scaling Engine” — blog techniczny w firmie Netflix, 5 listopada 2013, <http://techblog.netflix.com/2013/11/scryer-netflixs-predictive-auto-scaling.html>)

W narzędziu Scryer wykorzystano kombinację techniki wykrywania odstających w celu odrzucenia fałszywych punktów danych, a następnie użyto takich technik, jak szybkie transformaty Fouriera (ang. *Fast Fourier Transform* — **FFT**) i regresja liniowa w celu wygładzenia danych, a jednocześnie zachowania nagłych wzrostów ruchu powtarzających się wewnętrz danych (rysunek 33). W efekcie w firmie Netflix możliwe jest prognozowanie zapotrzebowania na ruch z zaskakującą dokładnością.



Rysunek 33. Prognozowanie ruchu klientów w firmie Netflix za pomocą narzędzia Scryer i wynik w postaci harmonogramu AWS zasobów obliczeniowych (źródło: Jacobson, Yuan, Joshi, „Scryer: Netflix's Predictive Auto Scaling Engine”)

W zaledwie kilka miesięcy po pierwszym użyciu programu Scryer w firmie Netflix znacznie poprawiono komfort oglądania przez klientów, poprawiono dostępność usług i obniżono koszty Amazon EC2.

ZASTOSOWANIE TECHNIK WYKRYWANIA ANOMALII

Nawet gdy dane nie mają rozkładu Gaussa, stosując różne metody, nadal można znaleźć wymagające uwagi odchylenia od normy. Techniki te są zaliczane do obszernej kategorii **wykrywania anomalii**, która często jest definiowana jako „wyszukiwanie elementów lub zdarzeń, które nie są zgodne z oczekiwany wzorcem”. Niektóre z mechanizmów tego typu można znaleźć wewnątrz narzędzi do monitorowania, podczas gdy inne mogą wymagać pomocy osób z wiedzą z dziedziny statystyki.

Tarun Reddy, wiceprezes ds. rozwoju i operacji w Rally Software, opowiada się za aktywną współpracą pomiędzy działem operacji IT a statystykami. Jak zaobserwował: „Aby poprawić jakość usług, wszystkie parametry produkcyjne są przetwarzane przez Tableau — pakiet oprogramowania do analiz statystycznych. W zespole mamy nawet inżyniera przeszkolonego w dziedzinie statystyki, który pisze kod w R (inny pakiet statystyczny). Ten inżynier ma swój własny zbiór zadań do wykonania, wypełniony prośbami od innych zespołów wewnątrz firmy, które chcą jak najsprawniej i jak najszybciej znaleźć odchylenia od normy — zanim powstaną jeszcze większe odchylenia, które wpłyną ujemnie na klientów”.

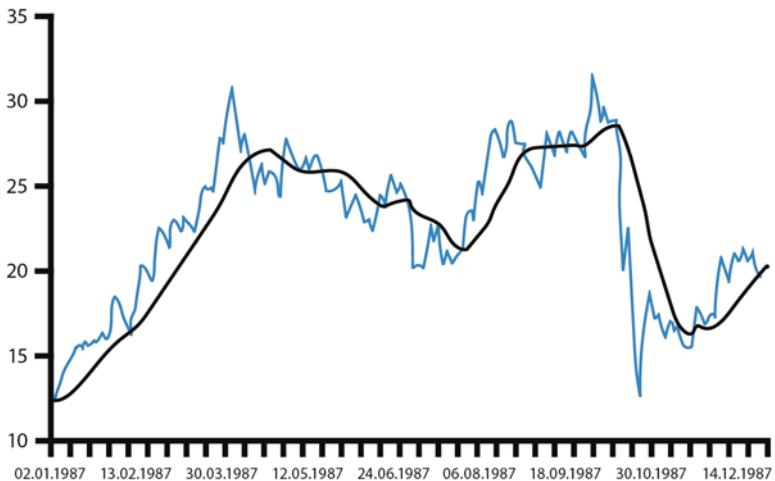
Jedna z technik statystycznych, z których możemy skorzystać, to tzw. **wygładzanie** (ang. *smoothing*). Technika ta jest przydatna zwłaszcza wtedy, gdy dane reprezentują przebiegi czasowe, co oznacza, że z każdym punktem danych jest powiązany stempel czasu (np. zdarzenia pobierania, zdarzenia zakończonych transakcji itp.). Wygładzanie często wiąże się z wykorzystaniem średnich ruchomych (nazywanych też kroczącymi), które przekształcają dane poprzez wyliczenie średniej dla każdego punktu danych z innymi danymi w określonym oknie przesuwnym. Daje to efekt wygładzania krótkoterminowych wahań i podkreśla długoterminowe trendy lub cykle^{*}.

Przykład efektu wygładzania pokazano na rysunku 34. Czarna linia reprezentuje surowe dane, natomiast linia niebieska linia wskazuje 30-dniową średnią ruchomą (tzn. średnią z ostatnich 30 dni)[†].

Istnieją także bardziej egzotyczne techniki filtracji, na przykład szybkie transformaty Fouriera, które szeroko stosowano do przetwarzania obrazów, a także test Kołmogorowa-Smirnowa (wykorzystany w narzędziach Graphite i Grafana), które są często używane do wyszukiwania podobieństw lub różnic w okresowych czy sezonowych danych.

* Wygładzanie, a także inne techniki statystyczne są wykorzystywane również do manipulowania plikami graficznymi i plikami audio. Na przykład wygładzanie obrazu (lub rozmywanie) polega na zastępowaniu każdego piksela przez średnią ze wszystkich jego sąsiadów.

† Do innych przykładów filtrów wygładzania należą ważone średnie ruchome lub wygładzanie wykładnicze (które polegają na stosowaniu odpowiednio liniowo lub wykładniczo większej wagi dla świeższych punktów danych).



Rysunek 34. Ceny akcji firmy Autodesk i filtr 30-dniowej średniej ruchomej
(źródło: Jacobson, Yuan, Joshi, „Scryer: Netflix's Predictive Auto Scaling Engine”)

Można oczekiwać, że duży procent wskaźników telemetrycznych dotyczących danych użytkownika będzie wykazywał okresowe (sezonowe) podobieństwa — ruch WWW, transakcje detaliczne, oglądanie filmów i wiele innych zachowań użytkownika charakteryzuje się bardzo regularnymi i zaskakująco przewidywalnymi wzorami dziennymi, tygodniowymi i rocznymi. Dzięki temu możemy wykrywać sytuacje, które różnią się od norm historycznych — na przykład gdy tempo transakcji zamówień we wtorek po południu spada do 50% wartości tygodniowej normy.

Takie dane mogą być przydatne w prognozowaniu. Warto zatem zwrócić się do osób — na przykład z działów marketingu lub analiz biznesowych — które dysponują wiedzą i umiejętnościami niezbędnymi do ich przeanalizowania. Warto poszukać takich osób i współpracować z nimi w celu identyfikacji wspólnych problemów i wykorzystania poprawionych technik wykrywania anomalii i przewidywania incydentów oraz lepszego ich rozwiązywania*.

* Wśród narzędzi wykorzystywanych do rozwiązywania tego rodzaju problemów należy wymienić Microsoft Excel (jeden z najłatwiejszych i najszybszych sposobów manipulowania danymi dla jednorazowych celów), jak również pakiety statystyczne, takie jak SPSS, SAS i projekt open source R — obecnie jeden z najpowszechniej stosowanych. Utworzono wiele innych narzędzi podobnego typu. Kilka z nich o statusie open source powstało w firmie Etsy: Oculus, który wyszukuje wykresy o podobnych kształtach, które mogą wskazywać na korelację; Opsweekly, który śledzi liczbę i częstotliwość alertów; Skyline, który próbuje zidentyfikować nietypowe zachowania na wykresach aplikacji i systemów.

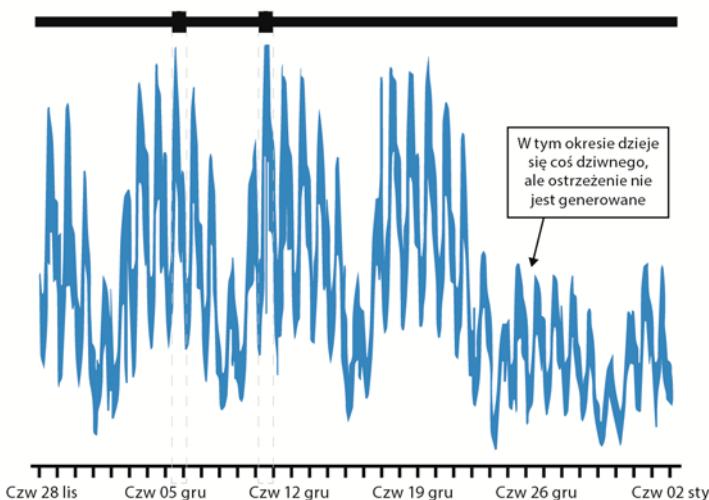
Studium przypadku

Zaawansowane mechanizmy wykrywania anomalii (2014)

W 2014 roku dr Toufic Boubez opisał w serwisie Monitorama możliwości zastosowania technik wykrywania anomalii, w szczególności podkreślając skuteczność testu Kołmogorowa-Smirnowa — techniki często używanej w statystyce do ustalania, czy dwa zestawy danych znacznie różnią się pomiędzy sobą, stosowanej między innymi w popularnych narzędziach Graphite i Grafana. Celem prezentacji tego studium przypadku nie jest przedstawienie samouczka posługiwania się klasą technik statystycznych, ale demonstracja ich użycia w pracy, jak również prawdopodobnego sposobu ich użycia w zupełnie innych zastosowaniach.

Na rysunku 35 pokazano wykres zależności liczby transakcji na minutę w witrynie e-commerce. Zwróćmy uwagę na tygodniową okresowość na wykresie — liczba transakcji zmniejsza się w czasie weekendów. Wystarczy wizualna kontrola, aby zauważać, że coś dziwnego dzieje się czwartego tygodnia, gdy liczba transakcji nie powraca w poniedziałek do normalnego poziomu. Sugeruje to wystąpienie zdarzenia, które należy zbadać.

Ostrzeżenia:

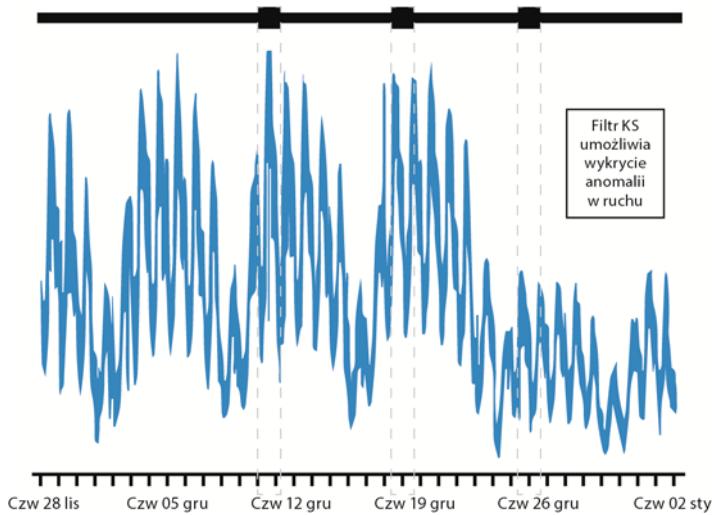


Rysunek 35. Liczba transakcji — braki w alarmowaniu w przypadku zastosowania reguły „trzech odchyleń standardowych” (źródło: dr Toufic Boubez, „Simple math for anomaly detection”)

Zgodnie z regułą trzech odchyleń standardowych ostrzeżenie byłoby generowane tylko dwukrotnie. Spadek liczby transakcji w poniedziałek nie spowodowałby wygenerowania alarmu. Byłoby idealnie, gdybyśmy uzyskali powiadomienie także w przypadku odchylenia od oczekiwanej poniedziałkowej wzorca.

Mówiąc „Kołmogorow-Smirnow» to świetny sposób, aby zaimponować wszystkim», dr Boubez żartuje. „Ale inżynierowie operacyjni powinni przekazać statystkom, że te rodzaje technik **nieparametrycznych** są świetne dla danych operacyjnych, ponieważ nie tworzą założeń dotyczących normalności lub innego rozkładu prawdopodobieństwa, co jest niezwykle istotne dla zrozumienia, co się dzieje w naszych bardzo złożonych systemach. Techniki te porównują dwa rozkłady prawdopodobieństwa, co pozwala na zestawienie okresowych lub sezonowych danych w celu odnalezienia zmian w danych z dnia na dzień lub z tygodnia na tydzień”.

Na rysunku 36 zaprezentowanym na następnej stronie przedstawiono ten sam zestaw danych, ale po zastosowaniu filtra K-S. Na trzecim obszarze wyróżniono nietypowy poniedziałek, w którym liczba transakcji nie powróciła do normalnego poziomu. To zaalarmowałoby nas o występującym w systemie problemie, którego wykrycie za pomocą inspekcji wizualnej albo metody odchyлеń standardowych byłoby praktycznie niemożliwe. W tym scenariuszu takie wczesne wykrywanie mogłoby zapobiec zdarzeniom mającym wpływ na klienta, a także ułatwiłoby osiągnięcie celów organizacyjnych.



Rysunek 36. Liczba transakcji — wykorzystanie testu Kołmogorowa-Smirnowa w celu generowania alertu anomalii (źródło: dr Toufic Boubez, „Simple math for anomaly detection”)

PODSUMOWANIE

W tym rozdziale przeanalizowaliśmy kilka różnych technik statystycznych, które mogą być użyte do analizowania parametrów telemetrycznych. Dzięki temu możemy znaleźć i rozwiązać problemy wcześniej niż kiedykolwiek — często gdy są one jeszcze niewielkie i na długo zanim spowodują katastrofalne skutki. Możemy zatem znajdować coraz słabsze sygnały awarii oraz przystępować do działań naprawczych. To pozwala nam tworzyć coraz bezpieczniejsze systemy pracy, a także zwiększa naszą zdolność do osiągania wyznaczonych celów.

W rozdziale zaprezentowano specyficzne studia przypadków, w tym sposób użycia opisanych technik w firmie Netflix w celu proaktywnego usuwania serwerów obliczeniowych z produkcji oraz automatycznego skalowania ich infrastruktury obliczeniowej. Opowiedzieliśmy również, w jaki sposób można użyć średniej ruchomej oraz filtra Kołmogorowa-Smirnowa — mechanizmów, które są stosowane w popularnych narzędziach do tworzenia wykresów telemetrycznych.

W następnym rozdziale opiszemy sposób integracji produkcyjnych wskaźników telemetrycznych w codziennej pracy działu Dev w celu poprawy bezpieczeństwa wdrożeń oraz usprawnienia systemu jako całości.

Sprzężenia zwrotne poprawiają bezpieczeństwo wdrażania kodu przez zespoły Dev i Ops

W 2006 roku Nick Galbreath pełnił funkcję wiceprezesa ds. inżynierii w firmie Right Media. Był odpowiedzialny za pracę działów Dev i Ops obsługujących platformą ogłoszeniową online, dla której notowano ponad miliard odsłon dziennie.

Galbreath opisywał krajobraz konkurencji, w którym działała firma Right Media:

W naszej branży poziomy rynkowe zmieniały się bardzo dynamicznie, dlatego musieliśmy reagować na nie w ciągu kilku minut. Oznaczało to, że dział Dev musiał mieć możliwość szybkiego wprowadzania zmian w kodzie i jak najszybszego przekazywania ich do produkcji. W przeciwnym razie tracilibyśmy klientów na rzecz szybszych konkurentów. Okazało się, że w przypadku utrzymywania oddzielnych grup do testowania oraz do wdrażania efekty były po prostu zbyt powolne. Trzeba było zintegrować wszystkie te funkcjonalności w jednej grupie oraz wyznaczyć wspólną odpowiedzialność i cele. Można w to wierzyć lub nie, ale naszym największym wyzwaniem było przełamanie u programistów obaw przed wdrażaniem swojego własnego kodu!

Można tu znaleźć interesującą ironię: pracownicy działu Dev często wytykali inżynierom Ops ich strach przed wdrażaniem kodu. Jednak kiedy programiści otrzymali możliwość wdrażania własnego kodu, zaczęli obawiać się wdrażania równie mocno, jak pracownicy zespołów operacyjnych.

Strach przed wdrożeniami kodu odczuwany przez członków zespołów Dev i Ops w firmie Right Media nie jest niczym niezwykłym. Galbreath zaobserwował jednak, że zapewnienie szybszych i częstszych informacji zwrotnych dla inżynierów przeprowadzających wdrożenie (niezależnie od tego, czy byli to inżynierowie Dev, czy Ops), a także zmniejszenie wielkości partii ich pracy poprawiało poczucie bezpieczeństwa i zaufania.

Galbreath, który obserwował tę transformację w wielu zespołach, opisywał uzyskiwane postępy w następujący sposób:

Na początku nikt z pracowników działów Dev i Ops nie jest chętny do wciśnięcia przycisku „Wdrażanie kodu”, inicjującego automatyczny proces wdrażania zbudowanego kodu — z powodu paraliżującego strachu przed byciem pierwszą osobą, która potencjalnie doprowadzi cały system produkcyjny do awarii. Ostatecznie, gdy ktoś zdobędzie się na odwagę i na ochotnika wdroży kod do produkcji, nieuchronnie, ze względu na błędne założenia lub subtelności produkcji, które nie były w pełni uwzględnione, pierwsze wdrożenie nie przechodzi gładko. Ze względu na niewystarczającą liczbę wskaźników telemetrycznych o problemach dowiadujemy się dopiero wtedy, gdy powiedzą nam o nich klienci.

Aby rozwiązać ten problem, zespół pilnie naprawia kod i wdraża go do produkcji, ale tym razem z większą liczbą wskaźników telemetrycznych dodanych do aplikacji i środowiska. W ten sposób możemy potwierdzić, że wprowadzone poprawki przywróciły usługę do prawidłowego działania. Dodatkowo następnym razem będziemy w stanie wykryć tego rodzaju problemy, zanim powie nam o nich klient.

Z biegiem czasu coraz więcej programistów zaczyna wdrażać własny kod do produkcji. Ponieważ pracujemy nad złożonym systemem, to prawdopodobnie znów dojdzie do awarii w produkcji. Tym razem jednak zdołamy szybko zidentyfikować funkcjonalność, która uległa awarii, i błyskawicznie zdecydujemy, czy wycofać zmianę, czy wprowadzić poprawkę, i w ten sposób rozwiążemy problem. Jest to wielkie zwycięstwo dla całego zespołu. Jest powód do świętowania — wreszcie jesteśmy na fali.

Zespół chce jednak poprawiać wyniki wdrożeń, dlatego programiści proaktywnie zbierają opinie dotyczące wprowadzonych zmian w kodzie (dokładniej opisano to w rozdziale 18.). Wszyscy wzajemnie sobie pomagają w pisaniu lepszych testów automatycznych, tak aby można było znaleźć błędy przed wdrożeniem. Ponieważ teraz wszyscy wiedzą, że im mniejsze zmiany produkcyjne, tym mniej problemów, to programiści zaczynają ewidencjonować w potoku wdrożeń coraz mniejsze zmiany. W ten sposób przed przejściem do wprowadzania kolejnej zmiany chcą zyskać pewność, że system będzie pomyślnie działać w produkcji.

Teraz kod jest wdrażany częściej niż kiedykolwiek. Znacznie poprawiła się także stabilność usługi. Ponownie odkryliśmy, że sekret bezproblemowego i nieprzerwanego przepływu leży we wprowadzaniu niewielkich i częstych zmian, które każdy może przejrzeć i bez trudu zrozumieć.

Galbreath zauważał, że wymierne zyski z tego stanu osiągają wszyscy — pracownicy działów Dev, Ops i Infosec. „Jako osoba, która jest również odpowiedzialna za bezpieczeństwo, mogę stwierdzić, że to uspokajające wiedzieć, że poprawki do produkcji można wdrożyć szybko, ponieważ zmiany są wprowadzane do produkcji przez cały dzień. Ponadto zawsze zdumiewa mnie, jak bardzo wzrasta zainteresowanie inżynierów zabezpieczeniami, gdy znajdujemy problemy w kodzie, za który są oni odpowiedzialni, które mogą sami szybko rozwiązać”.

Historia firmy Right Media pokazuje, że nie wystarczy samo zautomatyzowanie procesu wdrożeń — w procesie wdrażania trzeba również zintegrować monitorowanie produkcyjnych wskaźników telemetrycznych, a także ustanowić normy kulturowe, zgodnie z którymi wszyscy są w równym stopniu odpowiedzialni za kondycję całego strumienia wartości.

W tym rozdziale stworzymy mechanizmy sprzężeń zwrotnych, które pozwolą poprawić kondycję strumienia wartości na każdym etapie cyklu życia usługi — od projektu produktu, poprzez rozwój i wdrażanie, aż do eksploatacji i ostatecznie wycofania. W ten sposób gwarantujemy, że nasze usługi są „gotowe do produkcji” nawet w najwcześniejszym stadium projektu, a także integrujemy w przyszłej pracy wiedzę zdobytą w każdym z poprzednich wydań oraz na podstawie problemów, które wystąpiły w produkcji. W rezultacie uzyskujemy lepsze bezpieczeństwo i wydajność dla wszystkich.

WYKORZYSTANIE TELEMETRII DO UZYSKANIA BEZPIECZNIEJSZYCH WDROŻEŃ

W tym kroku pokażemy sposób aktywnego monitorowania telemetrycznych wskaźników produkcyjnych w scenariuszu, w którym wszyscy przeprowadzają wdrożenia produkcyjne, tak jak w historii z firmy Right Media. Dzięki temu wszystkie osoby realizujące wdrożenia — zarówno inżynierowie Dev, jak i Ops — mogą szybko ustalić, czy po wdrożeniu nowego wydania do produkcji funkcjonalności działają zgodnie z projektem. Ostatecznie nigdy nie wolno uznawać wdrożenia kodu lub zmiany produkcyjnej za zrealizowane, dopóki nie zostanie potwierdzone ich prawidłowe działanie w środowisku produkcyjnym.

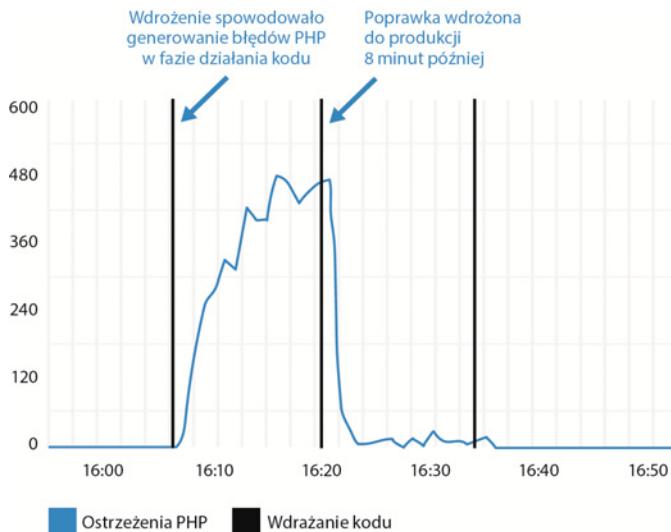
Robimy to przez aktywne monitorowanie podczas wdrożenia parametrów powiązanych z funkcjonalnością. W ten sposób możemy się upewnić, czy przypadkowo nie uszkodziliśmy usługi — lub co gorsza, czy nie uszkodziliśmy innej usługi. Jeśli wprowadzone zmiany powodują awarię lub ograniczenia działania dowolnych innych funkcjonalności, powinniśmy szybko doprowadzić do przywrócenia usługi, korzystając z pomocy wszystkich osób, które mogą przyczynić się do zdiagnozowania i rozwiązania problemu*.

* Postępując w ten sposób oraz tworząc wymaganą architekturę, „optymalizujemy usługę pod kątem wskaźnika MTTR zamiast MTBF”. Jest to popularna maksyma w środowisku DevOps,

Zgodnie z opisem z części III celem powinno być wykrycie błędów w potoku wdrożeń, zanim przedostaną się do produkcji. Nadal jednak będą błędy, które pozostaną niewykryte. Należy polegać na wskaźnikach telemetrycznych w celu szybkiego przywrócenia usługi. Można wyłączyć niedziałające funkcjonalności za pomocą przełączników (co często jest najprostszą i najmniej ryzykowną opcją, ponieważ nie wiąże się z wdrożeniami do produkcji), wprowadzić **poprawkę w przód** (ang. *fix forward*) — tzn. wprowadzić w kodzie zmiany, które eliminują wady, a następnie przenieść je do produkcji za pośrednictwem potoku wdrożeń — lub **wycofać zmiany** (tzn. przełączyć się do poprzedniej wersji przy użyciu przełączników funkcjonalności albo wyeliminować serwery, które uległy awarii, za pomocą rotacji przy użyciu wzorca wdrożenia niebieskie-zielone, wdrożenia kanarkowego itp.).

Chociaż operacja „poprawek w przód” może sprawiać wrażenie niebezpiecznej, to może być ona bardzo bezpieczna, gdy dysponujemy automatycznymi testami i szybkimi procesami wdrożeń oraz odpowiednią liczbą wskaźników telemetrycznych pozwalających na szybkie potwierdzenie, że w produkcji wszystko działa poprawnie.

Na rysunku 37 pokazano wykres wdrażania wprowadzonej w firmie Etsy zmiany w kodzie PHP, która doprowadziła do nagłego wzrostu liczby ostrzeżeń runtime. W tym przypadku programista zauważył problem w ciągu kilku minut, wygenerował poprawkę i wdrożył ją do produkcji. Rozwiążanie problemu zajęło mniej niż dziesięć minut.



Rysunek 37. Wdrożenie w witrynie Etsy.com spowodowało generowanie ostrzeżeń fazy działania kodu PHP. Awaria została szybko naprawiona (źródło: Mike Brittain, Tracking Every Release)

opisującą dążenie do optymalizacji w celu szybkiego przywracania usługi po wystąpieniu awarii w przeciwieństwie do podejmowania prób zapobiegania awariom.

Ze względu na to, że wdrożenia produkcyjne są jedną z głównych przyczyn problemów w produkcji, na wykresy parametrów warto nałożyć wszystkie zdarzenia związane z wdrożeniami i zmianami. W ten sposób można uzyskać pewność, że wszystkie osoby w strumieniu wartości są świadome wykonywanych działań. To poprawia komunikację i koordynację pomiędzy zespołami oraz pozwala na szybsze wykrywanie problemów i przywracanie działania usług po awarii.

INŻYNIEROWIE DEV W DYŻURACH „POD PAGEREM” WSPÓŁNIE Z INŻYNIERAMI OPS

Nawet wtedy, gdy wdrożenia produkcyjne i publikacje kodu działają bezproblemowo, to w przypadku każdej złożonej usługi zdarzają się nieoczekiwane problemy, takie jak incydenty i przestoje o nieodpowiednich porach (codziennie o godzinie 2.00 w nocy). Jeśli pozostawimy te problemy bez rozwiązania, mogą one powodować powracające awarie i kłopoty dla inżynierów Ops pracujących w dole strumienia wartości — zwłaszcza wtedy, gdy kłopoty te nie są widoczne dla inżynierów w górze strumienia, odpowiedzialnych za powstanie problemu.

Nawet jeśli problem spowoduje przypisanie defektu do zespołu pracującego nad cechą funkcjonalną, to może mu być nadany niższy priorytet od dostarczenia nowych funkcjonalności. Problemy mogą powracać przez tygodnie, miesiące lub nawet lata, powodując ciągły chaos i zakłócenia w pracy działu Ops. Jest to przykład tego, gdy lokalna optymalizacja wewnętrz centrów danych w górze strumienia wartości może obniżać wydajność dla całego strumienia wartości.

Aby temu zapobiec, wszystkie osoby pracujące w strumieniu wartości powinny współdzielić odpowiedzialność za obsługę incydentów operacyjnych. Można to zrobić, wyznaczając dyżury programistów, menedżerów zespołów Dev i architektów. W podobny sposób postąpił w 2009 roku Pedro Canahuati, dyrektor inżynierii produkcji Facebooka. Dzięki takim dyżurom zyskujemy gwarancję, że wszystkie osoby w strumieniu wartości otrzymują informacje zwrotne związane z decyzjami na temat architektury i kodu w górze strumienia.

W ten sposób pracownicy działu operacji nie prowadzą w osamotnieniu walki z problemami związanymi z kodem. Zamiast tego wszyscy starają się znaleźć odpowiednią równowagę pomiędzy naprawianiem defektów produkcyjnych a tworzeniem nowych funkcjonalności, bez względu na to, gdzie jest ich miejsce w strumieniu wartości. Jak w 2011 roku zaobserwował Patrick Lightbody, główny wiceprezes ds. zarządzania produktem w firmie New Relic: „Odkryliśmy, że w wyniku zastosowania praktyki budzenia programistów o 2 nad ranem defekty zaczęły być usuwane szybciej niż kiedykolwiek”.

Jednym z efektów ubocznych tej praktyki jest to, że pomaga ona kierownictwu działu Dev uświadomić sobie, że cele biznesowe nie są osiągane po prostu w wyniku

oznaczenia funkcjonalności jako „gotowe”. Funkcjonalność jest gotowa dopiero wtedy, gdy zachowuje się zgodnie z projektem w produkcji, kiedy nie powoduje nadmiernych eskalacji lub niezaplanowanej pracy dla działów Dev lub Ops^{*}.

Opisana praktyka może być skuteczna dla zespołów zorientowanych na cele rynkowe — odpowiedzialnych za rozwijanie funkcjonalności i uruchamianie jej w produkcji — oraz tych, które mają orientację funkcjonalną. Jak zaobserwował podczas prezentacji w 2014 roku Arup Chakrabarti, dyrektor inżynierii operacji w firmie PagerDuty: „W firmach coraz mniej popularne stają się dedykowane zespoły na telefon; zamiast tego oczekuje się, że w przypadku awarii wszyscy będą osiągalni”.

Niezależnie od sposobu organizacji zespołów obowiązują takie same zasady: gdy programiści uzyskują opinie na temat sposobu działania swoich aplikacji w produkcji, włącznie z wprowadzaniem poprawek w przypadku awarii, są bliżej klienta. To tworzy wartość, z której czerpią korzyści wszystkie osoby należące do strumienia wartości.

OBSERWACJA PRZEZ PROGRAMISTÓW PRACY W DOLE STRUMIENIA

Jedną z najbardziej zaawansowanych technik w projektowaniu interakcji i wrażeń użytkowników są zapytania kontekstowe. Polega ona na tym, że zespół produktu obserwuje sposób użytkowania aplikacji przez klienta w jego naturalnym środowisku — często pracując razem z nim przy biurku. Stosowanie tego sposobu często pozwala odkryć zaskakujące sposoby korzystania z aplikacji przez klientów. Na przykład można zauważać, że chcąc wykonać proste zadanie podczas codziennej pracy, klienci muszą wiele razy kliknąć, wycinać i wklejać tekst z wielu ekranów lub robić notatki na papierze. Wszystkie wymienione sytuacje są przykładami zachowań kompensacyjnych i obejść problemów związanych z użytecznością.

Najczęstszą reakcją programistów po sesji obserwacji pracy klienta jest przerażenie. Często pada stwierdzenie: „Straszne było zobaczyć, na jak wiele sposobów potrafiliśmy sprawić kłopoty klientom”. Z sesji obserwacji klientów prawie zawsze płynie nauka oraz powstaje pragnienie poprawy sytuacji dla klienta.

Warto użyć tej samej techniki w celu obserwacji wpływu naszej pracy na klientów wewnętrznych. Programiści powinni zainteresować się pracą w dole strumienia. Dzięki temu mogą zaobserwować sposoby korzystania z ich produktu przez pracowników centrów danych w dole strumienia w celu przygotowania produktu do wdrożenia[†].

* ITIL definiuje gwarancję jako stan, w którym usługa może działać w produkcji niezawodnie i bez interwencji przez ustalony okres (np. dwa tygodnie). Najlepiej, gdyby ta definicja gwarancji została zintegrowana z przyjętą wewnątrz zespołu definicją słowa „gotowe”.

† Dzięki obserwowaniu pracy w dole strumienia można wykryć sposoby usprawnienia przepływu, na przykład poprzez zautomatyzowanie złożonych, ręcznych działań (łączenie przez sześć godzin klastrów serwerów aplikacji w pary); utworzenie pakietu instalacyjnego kodu

Programiści chętnie obserwują pracę w dole strumienia. Dzięki obserwowaniu trudności klientów podejmują lepsze i bardziej świadome decyzje w swojej codziennej pracy.

W ten sposób tworzymy sprzężenia zwrotne na temat niefunkcjonalnych aspektów kodu — wszystkich elementów, które nie są powiązane z funkcjonalnością, z którą styka się klient — oraz identyfikujemy sposoby poprawy możliwości wdrażania, zarządzania, obsługi itd.

Obserwacje wygody użytkowania produktu często wywierają potężny wpływ na obserwatorów. Opisując wrażenia ze swojej pierwszej obserwacji klienta, Gene Kim, założyciel i pełniący przez 13 lat funkcję dyrektora technicznego firmy Tripwire, a także współautor niniejszej książki, powiedział:

„Jeden z najbardziej przełomowych momentów w mojej karierze zawodowej miał miejsce w 2006 roku, kiedy spędziłem ranek, obserwując jednego z klientów podczas korzystania z naszego produktu. Oglądałem go podczas wykonywania operacji, którą zgodnie z oczekiwaniemi klienci powinni wykonywać co tydzień. Ku mojemu ekstremalnemu przerażeniu odkryliśmy, że wymagała ona 63 kliknięć”. Klient wykonujący tę operację ciągle przepraszał, mówiąc rzeczy w stylu: „Proszę mi wybaczyć, jest chyba lepszy sposób, żeby to zrobić”.

Niestety, nie było lepszego sposobu na wykonanie tej operacji. Inny klient opisał początkową konfigurację produktu, która wymagała wykonania 1300 kroków. Nagle zrozumiałem, dlaczego zadanie zarządzania naszym produktem zawsze było przydzielane pracującemu najkrócej inżynierowi w zespole — nikt nie chciał wykonywać zadania polegającego na uruchamianiu naszego produktu. To był jeden z powodów, dla których pomogłem w stworzeniu praktyk UX (od ang. *user experience* — dosł. „doświadczenia użytkowników”) w mojej firmie. Było to odpokutowanie za kłopoty, jakie sprawiliśmy naszym klientom.

Obserwacje UX pozwoliły na stworzenie jakości u źródła i wywołyły znacznie większą empatię dla kolegów pracujących w tym samym strumieniu wartości. W sytuacji idealnej obserwacje UX pomagają w stworzeniu skodyfikowanych wymagań niefunkcjonalnych, które należy dodać do wspólnego rejestru zadań do wykonania. W rezultacie można proaktywnie zintegrować je ze wszystkimi budowanymi usługami — co jest ważnym elementem tworzenia kultury pracy DevOps^{*}.

jednokrotne zamiast wielokrotne na różnych etapach produkcji i walidacji; pracę z testerami w celu automatyzacji zestawów testów wykonywanych ręcznie, a tym samym usunięcie wąskiego gardła przeszkadzającego w częstszych wdrożeniach; stworzenie bardziej użytecznej dokumentacji zamiast rozszyfrowywania uwag programistów aplikacji dotyczących tworzenia pakietów instalacyjnych.

* Później Jeff Sussna podjął próbę dalszego skodyfikowania zadań zmierzających do osiągnięcia celów UX w procesie tzw. „konwersacji cyfrowych”, które mają pomóc organizacjom

POWIERZENIE NA POCZĄTKU EKSPOLOATACJI ZADANIA ZARZĄDZANIA USŁUGĄ PRODUKCYJNĄ PROGRAMISTOM

Pomimo że w codziennej pracy programiści piszą kod i uruchamiają go w środowiskach zbliżonych do produkcyjnych, to wdrożenia do produkcji w dalszym ciągu mogą kończyć się katastrofą, ponieważ często wdrożenie stwarza pierwszą okazję do obserwacji działania kodu w realnych warunkach produkcyjnych. Efekt ten jest spowodowany tym, że informacje o sposobie działania usługi często docierają w zbyt późnej fazie cyklu życia oprogramowania.

Jeśli problemy pozostaną nierozerwiane, efektem jest często tworzenie oprogramowania trudnego w obsłudze. Jak kiedyś powiedział anonimowy inżynier operacyjny: „W naszej grupie większość administratorów systemów zdołała przetrwać zaledwie sześć miesięcy. Oprogramowanie ciągle ulegało awariom w produkcji, godziny pracy były szalone, a wdrożenia aplikacji były niezwykle bolesne. Najgorsze było łączenie klastrów serwerów aplikacji w pary. Operacja ta zajmowała nam sześć godzin. W każdym momencie mieliśmy poczucie, jakby programiści osobiście nas nienawidzili”.

Może to być wynikiem braku wystarczającej liczby inżynierów operacyjnych do obsługi wszystkich zespołów produktów i usług, które już mamy w produkcji, co może się zdarzyć zarówno w zespołach o orientacji funkcjonalnej, jak i rynkowej.

Jednym z potencjalnych środków zaradczych jest postępowanie zgodne z przykładem firmy Google — powierzenie zespołom programistów samodzielnego zarządzania swoimi usługami w środowisku produkcyjnym przed przekazaniem ich scentralizowanemu zespołowi operacyjnemu. Dzięki powierzeniu programistom odpowiedzialności za wdrożenie i wsparcie produkcji przekazanie zadania zarządzania usługą działowi operacyjnemu odbywa się znacznie płynniej*.

Aby zapobiec możliwości przekazania do produkcji problematycznych, samodzielnie zarządzanych usług i stworzeniu zagrożenia dla organizacji, można zdefiniować wymagania uruchomienia, które muszą być spełnione, by usługi mogły być przekazane do wykorzystania przez prawdziwych klientów oraz w warunkach realnych obciążień produkcyjnych. Ponadto w celu udzielenia pomocy zespołom produktów inżynierowie operacyjni powinni spełniać rolę konsultantów wspomagających proces przygotowania usług do produkcji.

w zrozumieniu zachowań klienta jako złożonego systemu, rozszerzając kontekst jakości. Kluczowe pojęcia obejmują projektowanie w kierunku usług, a nie oprogramowania; minimalizacja opóźnień i maksymalizacja siły sprzężeń zwrotnych; projektowanie pod kątem awarii i eksploatacja w celu czerpania nauki; wykorzystanie doświadczeń działu operacji jako danych wejściowych do projektowania oraz poszukiwanie empatii.

* Prawdopodobieństwo szybkiego rozwiązania problemów występujących w produkcji zwiększa się dzięki pozostawieniu zespołów programistów w pierwotnej postaci — bez rozwiązywania ich po zakończeniu projektu.

Dzięki utworzeniu wskazówek uruchamiania pozwalamy zagwarantować wszystkim zespołom produktów czerpanie korzyści ze skumulowanych, zbiorczych doświadczeń całej organizacji, a zwłaszcza działu operacyjnego. Wytyczne i wymagania uruchamiania prawdopodobnie powinny zawierać następujące informacje:

- **Liczba defektów i ich istotność** — czy aplikacja faktycznie działa zgodnie z projektem.
- **Typ (częstotliwość) alertów przez pager** — czy aplikacja generuje niezwykle dużą liczbę alertów w produkcji.
- **Zakres monitorowania** — czy zakres monitorowania jest wystarczający do przywrócenia usługi w przypadku, gdy coś pójdzie nie tak.
- **Architektura systemu** — czy usługa jest wystarczająco luźno powiązana, by mogła obsługiwać wysoki współczynnik zmian i wdrożeń w produkcji.
- **Proces wdrażania** — czy istnieje przewidywalny, deterministyczny i wystarczająco zautomatyzowany proces wdrażania kodu do produkcji.
- **Higiena produkcji** — czy istnieją dowody na istnienie wystarczająco dobrych nawyków produkcyjnych, które pozwoliłyby na zarządzanie wsparciem produkcji przez inne osoby.

Z pozoru wymagania te mogą wydawać się podobne do tych, które znajdują się na tradycyjnej liście kontrolnej fazy produkcji, którą wykorzystywaliśmy w przeszłości. Zasadnicza różnica polega jednak na konieczności istnienia skutecznego mechanizmu monitorowania, niezawodności i determinizmu wdrożeń oraz architektury wspierającej szybkie i częste wdrożenia.

Jeśli podczas przeglądu zostaną znalezione jakiekolwiek braki, to wyznaczony inżynier operacyjny powinien pomóc zespołowi pracującemu nad funkcjonalnością rozwiązać problemy, a nawet — gdy to konieczne — pomóc w modyfikacji projektu usługi, tak aby można było ją łatwo wdrożyć do produkcji, a następnie nią zarządzać.

W tym momencie warto również sprawdzić, czy usługa ma znaczenie w kontekście celów zapewnienia zgodności z przepisami lub czy jest prawdopodobne, że będzie związana z tego rodzaju celami w przyszłości:

- Czy usługa generuje znaczący przychód? (Np. jeśli to jest więcej niż 5% dochodów korporacji będącej spółką na giełdzie, to jest to „znaczące konto”, które musi spełniać sekcję 404 ustawy Sarbanes-Oxley [SOX] z 2002 roku).
- Czy ruch użytkowników w usłudze jest wysoki lub czy koszty przestojów są wysokie? (Np. czy zagrożenia operacyjne generują zagrożenia dostępności lub utraty reputacji?).
- Czy usługa przechowuje informacje dotyczące płatności, takie jak numery kart kredytowych lub dane osobowe, np. numery NIP, PESEL lub zapisy związane

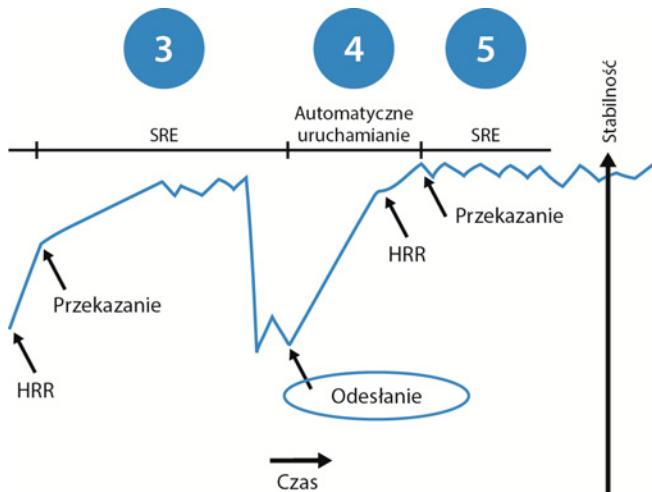
z ochroną zdrowia? Czy istnieją inne kwestie bezpieczeństwa, które mogą stwarzać zagrożenia związane ze zgodnością z przepisami, zobowiązaniemi kontraktowymi, prywatnością lub utratą reputacji?

- Czy usługa podlega innym wymogom prawnym lub kontraktom, takim jak przepisy eksportowe, ustawa PCI-DSS, HIPAA itp.?

Dysponowanie tymi informacjami pozwala skutecznie zarządzać nie tylko zagrożeniami technicznymi związanymi z usługą, ale również wszelkimi potencjalnymi zagrożeniami dla bezpieczeństwa i zgodności z przepisami. Stanowią one również ważne dane wejściowe do projektu środowiska zarządzania produkcją.

Dzięki włączeniu wymagań operatywności na najwcześniejszych etapach procesu rozwoju oraz powierzeniu działowi Dev zadania samodzielnego zarządzania własnymi aplikacjami i usługami proces przekazywania nowych usług do produkcji staje się bardziej płynny, znacznie łatwiejszy i bardziej przewidywalny. Jednak w przypadku usług, które już są w produkcji, potrzebny jest inny mechanizm, który pozwoli ochronić dział Ops przed zetknięciem się w produkcji z nieobsługiwana usługą. Jest to szczególnie istotne dla świadczących usługi operacyjne firm o orientacji funkcjonalnej.

W tym kroku możemy stworzyć **mechanizm odsyłania usługi** (ang. *service handback mechanism*) — innymi słowy, gdy usługa produkcyjna staje się wystarczająco krucha, dział Ops ma możliwość ponownego przekazania odpowiedzialności za obsługę tej usługi działowi Dev (rysunek 38).



Rysunek 38. „Przekazywanie usług” w Google (źródło: SRE@Google: Thousands of DevOps Since 2004, nagranie wideo w serwisie YouTube, 45.57, opublikowane przez USENIX, 12 stycznia 2012, <https://www.youtube.com/watch?v=iluTnhdTzKO>)

Gdy usługa powróci do stanu umożliwiającego zarządzanie przez programistów, rola działu operacji przesuwa się od wsparcia produkcji do konsultacji oraz wspierania zespołu usługi w przygotowaniu usługi do produkcji.

Mechanizm ten odgrywa rolę zaworu bezpieczeństwa, którego funkcją jest niedopuszczenie do postawienia działu Ops w sytuacji, gdy będzie zmuszony do zarządzania kruchą usługą w czasie, gdy rośnie dług techniczny, a lokalny problem stopniowo przekształca się w problem globalny. Ten mechanizm pomaga również działowi Ops zachować możliwości pracy nad usprawnieniami oraz realizacją projektów profilaktycznych.

Mechanizm odsyłania usług ma wieloletnią tradycję w firmie Google i jest prawdopodobnie jednym z najwyraźniejszych dowodów wzajemnego szacunku pomiędzy inżynierami działów Dev i Ops. Dzięki stosowaniu tej praktyki dział Dev może szybko wygenerować nowe usługi, a inżynierowie Ops mogą dołączyć do zespołu, gdy usługi nabiorą dla firmy strategicznego znaczenia. W rzadkich przypadkach, gdy usługa zaczyna sprawiać zbyt wiele problemów, by można je było obsłużyć w produkcji, możliwe jest odesłanie usługi z powrotem do działu Dev^{*}. Poniższe studium przypadku zespołu Site Reliability Engineering w firmie Google opisuje rozwój procesów Hand-off Readiness Review (dosł. „przegląd gotowości do przekazania”) oraz Launch Readiness Review (dosł. „przegląd gotowości do uruchomienia”) oraz wynikające z tego korzyści.

Studium przypadku

Procesy przeglądu gotowości do przekazania i uruchomienia w Google (2010)

Jednym z wielu zaskakujących faktów dotyczących firmy Google jest to, że inżynierowie Ops określani skrótem **SRE** (ang. *Site Reliability Engineers* — dosł. „inżynierowie niezawodności ośrodka”) mają organizację funkcjonalną. Termin SRE został wymyślony przez Bena Treynora Slossa w 2004 roku[†]. Treynor Sloss zaczął pracę w zespole złożonym z siedmiu inżynierów SRE. Liczba ta do roku 2014 wzrosła do ponad 1200. Jak powiedział Treynor Sloss: „Jeśli Google kiedykolwiek upadnie, to będzie to moja wina”. Treynor Sloss opierał się przed stworzeniem

* W organizacjach z finansowaniem bazującym na projektach może nie być działu Dev, do którego można by odesłać usługę, ponieważ zespół mógł już zostać rozwiązany lub być może nie dysponuje już wystarczającym budżetem bądź czasem, by móc взять odpowiedzialność za usługę. Potencjalnymi środkami zaradczymi są: powołanie doraźnego zespołu z zadaniem usprawnienia usługi, tymczasowe finansowanie lub przydział personelu do usługi albo wycofanie usługi z użycia.

† W tej książce używamy terminu „inżynier Ops”, ale terminy „inżynier Ops” oraz „inżynier SRE” mogą być stosowane zamiennie.

jednozdaniowej definicji, czym są inżynierowie SRE. Kiedyś opisał zadania inżynierów SRE jako „pracę wykonywaną w przypadku, gdy programistom zostaną zlecone zadania typowe dla działu operacyjnego”.

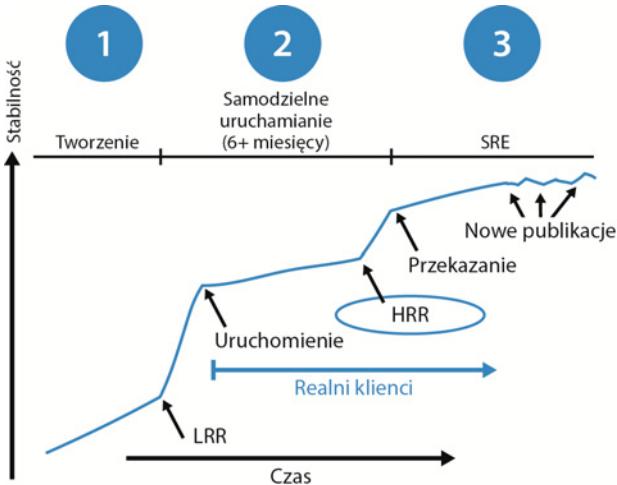
Każdy inżynier SRE składa raporty do organizacji Treynora Slossa, mające na celu zapewnienie spójności jakości personelu i zatrudniania. Inżynierowie SRE są osadzeni w zespołach produktów w firmie Google (które również zapewniają ich finansowanie). Jednak inżynierowie SRE są zbyt nieliczni, aby byli przydzielani do wszystkich zespołów. Przydziela się ich wyłącznie do zespołów tych produktów, które mają najwyższą wagę dla firmy, lub tych, które muszą spełniać wymagania zgodności z przepisami. Ponadto usługi te muszą charakteryzować się niskim obciążeniem operacyjnym. Produkty, które nie spełniają koniecznych kryteriów, są nadal zarządzane przez deweloperów.

Nawet wtedy, gdy nowe produkty stają się na tyle ważne dla firmy, aby zdecydowano się przydzielić do ich zespołów inżynierów SRE, programiści nadal muszą samodzielnie utrzymywać swoje usługi w produkcji przez co najmniej sześć miesięcy, zanim usługi nabędą prawo do przydzielenia do nich inżynierów SRE.

Aby zapewnić zespołom samodzielnie zarządzającym produktami możliwość czerpania korzyści ze zbiorowego doświadczenia SRE organizacji, firma Google stworzyła dwa zestawy kontroli bezpieczeństwa dla dwóch krytycznych etapów publikowania nowych usług znanych jako procesy **LRR** (ang. *Launch Readiness Review*) oraz **HRR** (ang. *Hand-Off Readiness Review*).

Kryteria procesu LRR muszą być spełnione i potwierdzone, zanim jakakolwiek usługa Google stanie się publicznie dostępna dla klientów i zostanie poddana obciążeniom aktywnego ruchu w warunkach produkcyjnych. Z kolei proces HRR jest wykonywany podczas przejęcia usługi do stanu zarządzania przez dział Ops. Zwykle odbywa się to kilka miesięcy po procesie LRR. Listy kontrolne procesów LRR i HRR są podobne, ale HRR ma o wiele bardziej rygorystyczne kryteria i musi przejść przez ostrzejsze standardy akceptacji — dla porównania proces LRR jest przeprowadzany samodzielnie przez zespoły produktu.

Do każdego zespołu realizującego proces LRR lub HRR jest przydzielony inżynier SRE, którego zadaniem jest udzielenie pomocy w zrozumieniu kryteriów oraz ich spełnieniu. Listy kontrolne LRR i HRR ewoluowały w czasie, dlatego wszystkie zespoły mogą czerpać korzyści ze zbiorowych doświadczeń wszystkich poprzednich procesów — zarówno tych, które zakończyły się sukcesem, jak i tych, które zakończyły się niepowodzeniem. Tom Limoncelli podczas swojej prezentacji *SRE@Google: Thousands of DevOps Since 2004*, wygłoszonej w 2012 roku, powiedział: „Za każdym razem, gdy uruchamiamy usługę, uczymy się czegoś nowego. Podczas publikowania i uruchamiania usług zawsze znajdą się ludzie, którzy są mniej doświadczeni niż inni. Listy kontrolne LRR i HRR są sposobem tworzenia tej organizacyjnej pamięci”. Wykorzystanie procesów LRR i HRR w Google zaprezentowano na rysunku 39.



Rysunek 39. Procesy LRR i HRR w firmie Google (źródło: SRE@Google: Thousands of DevOps Since 2004, nagranie wideo w serwisie YouTube, 45.57, opublikowane przez USENIX, 12 stycznia 2012, <https://www.youtube.com/watch?v=iluTnhdTzKO>)

Wymaganie od zespołów produktu samodzielnego zarządzania własnymi usługami w produkcji zmusza inżynierów Dev do tego, by „chodziły w butach” inżynierów Ops, ale z przewodnikiem w postaci procesów LRR i HRR. Dzięki temu przekształcenie usługi nie tylko staje się łatwiejsze i bardziej przewidywalne, ale również pomaga stworzyć empatię pomiędzy centrami pracy w górze i w dole strumienia wartości.

Limoncelli zauważyl: „W najlepszym przypadku zespoły produktu wykorzystywały listę kontrolną LRR jako przewodnika, pracując nad spełnieniem wymienionych w niej kryteriów równolegle z rozwijaniem swojej usługi i korzystając z pomocy inżynierów SRE wtedy, gdy była potrzebna”.

Ponadto, jak zaobserwował Limoncelli: „Zespoły, którym udało się najszybciej uzyskać zatwierdzenie HRR, były tymi, które najwcześniej pracowały z inżynierami SRE — od najwcześniejszych etapów projektu aż do uruchomienia. Wielką rzeczą jest możliwość łatwego uzyskania pomocy od ochronników spośród inżynierów SRE. Każdy inżynier SRE dostrzega wartość w udzielaniu rad zespołom produktów na wczesnych etapach projektu. Z dużym prawdopodobieństwem będzie chętny do udzielania tego rodzaju porad przez kilka godzin lub kilka dni”.

Praktyka udzielania przez inżynierów SRE pomocy zespołom produktów na wczesnych etapach projektu jest ważną normą kulturową, która jest ciągle wzmacniana w firmie Google. Limoncelli wyjaśnił: „Pomoc zespołom produktów to długoterminowa inwestycja, która spłaci się wiele miesięcy później, kiedy nadjdzie czas uruchomienia usługi. Jest to forma »dobrego obywatelstwa« i »usługi społecznościowej«, które są cenione oraz rutynowo uwzględniane podczas ocen inżynierów SRE pod kątem awansów”.

PODSUMOWANIE

W tym rozdziale omówiliśmy mechanizmy sprzężeń zwrotnych, które umożliwiają usprawnianie usług na każdym etapie codziennej pracy. Opisano mechanizmy sprzężeń zwrotnych podczas wdrażania zmian w produkcji, naprawiania kodu, gdy coś pójdzie nie tak, przez inżynierów wzywanych przez pager, zlecanie programistom obserwacji swojej pracy w centrach danych w niższych partiach strumienia wartości, tworzenie wymagań niefunkcjonalnych pomagających zespołowi programistycznemu pisać kod, który jest lepiej przygotowany do przekazania do produkcji, czy nawet odsyłać usługi sprawiające problemy do samodzielnego zarządzania przez dział Dev.

Dzięki stworzeniu tych pętli sprzężenia zwrotnego wdrożenia do produkcji stają się bezpieczniejsze. Ponadto zwiększa się gotowość przekazywania do produkcji kodu tworzonego przez dział Dev, a wzmacnianie poczucia wspólnoty celów, odpowiedzialności i empatii kreuje lepsze relacje pomiędzy działami Dev i Ops.

W kolejnym rozdziale omówimy wykorzystanie wskaźników telemetrycznych w technice wytwarzania oprogramowania sterowanego hipotezami (ang. *hypothesis-driven development*) oraz do przeprowadzania testów A/B. Techniki te pozwalają przeprowadzać eksperymenty, dzięki którym osiągnięcie wyznaczonych celów organizacyjnych i zdobycie przewagi na rynku stają się łatwiejsze.

Integracja technik wytwarzania oprogramowania sterowanego hipotezami i testowania A/B w codziennej pracy

Często się zdarza, że w projektach oprogramowania programiści pracują nad cechami funkcjonalnymi przez wiele miesięcy lub lat. W tym okresie realizują wiele wydań i nawet nie sprawdzają, czy pożądane cele biznesowe zostały osiągnięte — na przykład czy określona cecha funkcjonalna pozwoliła osiągnąć odpowiednie rezultaty lub nawet czy w ogóle jest używana.

Co gorsza nawet w przypadku odkrycia, że określona cecha funkcjonalna nie osiąga pożądanych rezultatów, większy priorytet od dokonania korekt w tej funkcjonalności jest przyznawany tworzeniu nowych funkcjonalności. To sprawia, że cecha funkcjonalna działająca poniżej oczekiwania nigdy nie osiągnie zamierzonego celu biznesowego. Ogólnie rzecz biorąc, jak zauważał Jez Humble: „Najbardziej nieefektywnym sposobem testowania modelu biznesowego lub pomysłu na produkt jest zbudowanie kompletnego produktu tylko po to, żeby się przekonać, czy przewidywane zapotrzebowanie rzeczywiście istnieje”.

Przed rozpoczęciem budowania funkcjonalności należy rygorystycznie zadać sobie pytanie: „Czy powinniśmy budować tę funkcjonalność i dlaczego?”. Następnie należy przeprowadzić możliwie najtańsze i najszybsze eksperymenty w celu zweryfikowania — poprzez badanie użytkowników — czy przewidywana funkcjonalność rzeczywiście pozwoli osiągnąć pożądane rezultaty. W tym celu można zastosować takie techniki, jak wytwarzanie oprogramowania sterowane hipotezami (ang. *hypothesis-driven*

development), lejki popytu (ang. *customer acquisition funnels*) oraz testowanie A/B. Te pojęcia omówimy w niniejszym rozdziale. Interesujący przykład sposobu wykorzystania tych technik do stworzenia uwielbianych przez klientów produktów w celu promowania uczenia się organizacji oraz zdobycia przewagi na rynku udostępniła firma Intuit, Inc.

Firma ta koncentruje się na tworzeniu rozwiązań zarządzania biznesowego i finansowego usprawniających pracę małych firm, konsumentów oraz specjalistów w dziedzinie księgowości. W 2012 r. firma osiągnęła przychód na poziomie 4,5 mld dolarów, zatrudniała 8500 pracowników, jej flagowymi produktami były QuickBooks, TurboTax, Mint, a do niedawna Quicken*.

Scott Cook, założyciel Intuit, od dawna dążył do budowania kultury innowacji, zachęcał zespoły do eksperymentowania w procesie rozwijania produktu oraz przekonywał kierownictwo do wsparcia tych dążeń. Jak powiedział: „Zamiast skupiać się na głosie szefa... należy położyć nacisk na to, by prawdziwi ludzie naprawdę wykonywali prawdziwe eksperymenty, i na tej podstawie podejmować decyzje”. Jest to typowy przykład naukowego podejścia do rozwoju produktu.

Cook wyjaśniał, że potrzebny jest „system, w którym każdy pracownik może prowadzić szybkie, dynamiczne eksperymenty... Dan Maurer przewodzi działowi obsługi konsumentów... [który] prowadzi witrynę internetową TurboTax. Kiedy objął swoje stanowisko, przeprowadzaliśmy około siedmiu eksperymentów rocznie”.

Kontynuował: „Dzięki zastosowaniu szalonej kultury innowacji [w 2010 roku] teraz przeprowadzamy 165 eksperymentów w ciągu trzech miesięcy sezonu podatkowego [USA]. Efekt biznesowy? Współczynnik konwersji witryny internetowej wzrósł o 50 procent... Ludzie [członkowie zespołu] wprost to uwielbiają, bo teraz ich pomysły mogą trafić na rynek”.

Oprócz wpływu na współczynnik konwersji witryny internetowej jednym z najbardziej zaskakujących elementów tej historii jest to, że firma TurboTax wykonuje eksperymenty produkcyjne w szczytce swojego sezonu. Przez dziesięciolecia, zwłaszcza w handlu detalicznym, zagrożenia związane z przestojami wpływającymi na dochody podczas sezonu wakacyjnego były tak wysokie, że często zamrażano wszelkie zmiany w okresie od połowy października do połowy stycznia.

Jednak dzięki temu, że wdrożenia oprogramowania i publikacje stały się szybkie i bezpieczne, zespół TurboTax przeprowadził eksperymenty z użytkownikami online i zaczął wprowadzać zmiany do produkcji podczas okresów największego ruchu oraz sezonów największych dochodów.

To podkreśla opinię, zgodnie z którą eksperymenty mają największą wartość podczas sezonów ruchu szczytowego. Gdyby z wdrożeniem zmian zespół TurboTax czekał do 16 kwietnia — daty przypadającej na jeden dzień po terminie wypełniania

* W 2016 r. firma Intuit sprzedała produkt Quicken prywatnej firmie H.I.G. Capital.

zeznań podatkowych w USA — firma mogłaby stracić wielu potencjalnych klientów, a nawet niektórych z istniejących klientów na rzecz konkurencji.

Im szybciej eksperymentujemy, iterujemy i integrujemy opinie z produktem lub usługą, tym szybciej jesteśmy w stanie pokonać konkurencję. A to, jak szybko możemy zintegrować opinie, zależy od zdolności do wdrażania i publikowania oprogramowania.

Przykład firmy Intuit pokazuje, że zespół Intuit TurboTax był w stanie wykorzystać tę sytuację na swoją korzyść i w rezultacie osiągnąć przewagę na rynku.

KRÓTKA HISTORIA TESTÓW A/B

Jak pokazuje historia produktu TurboTax firmy Intuit, potężną techniką badania użytkowników jest zdefiniowanie lejka popytu klienta i przeprowadzenie testów A/B. Pionierską dziedziną technik testowania A/B był **marketing bezpośredni** — jedna z dwóch głównych kategorii strategii marketingowych. Druga nazywa się **marketingiem masowym** lub **marketingiem marki** i często bazuje na maksymalizacji liczby wyświetleń reklam w celu wywarcia wpływu na decyzje zakupowe.

W poprzednich epokach — przed czasem wiadomości e-mail i mediów społeczeństwowych — marketing bezpośredni oznaczał wysyłanie tysięcy pocztówek lub ulotek za pośrednictwem poczty z prośbą do odbiorców o przyjęcie oferty przez wybranie numeru telefonu, odesłanie kartki pocztowej lub złożenie zamówienia.

W trakcie tych kampanii przeprowadzano eksperymenty w celu znalezienia oferty o najwyższym współczynniku konwersji. Eksperymentowano z modyfikowaniem i przystosowywaniem oferty, zmianą jej brzmienia, różnymi stylami, projektami i typografią, opakowaniami itd. w celu znalezienia metody, która byłaby najskuteczniejsza w wygenerowaniu pożądanego działania (np. wybrania numeru lub złożenia zamówienia).

Każdy eksperyment często wymagał zmiany projektu i wydrukowania nowej partii materiałów, wysłania tysięcy ofert i oczekiwania przez wiele tygodni na odpowiedzi. Każdy eksperyment zazwyczaj kosztował dziesiątki tysięcy dolarów i zajmował wiele tygodni lub miesięcy. Jednak pomimo wydatków iteracyjne testowanie szybko się zwrażało, jeśli udało się znacznie zwiększyć współczynniki konwersji (np. procent respondentów, którzy zdecydowali się na zamówienie produktu, wzrósł z 3 do 12%).

Dobrze udokumentowane przypadki testów A/B obejmują kampanie zbierania funduszy, marketing internetowy oraz metodologię Lean Startup. Co ciekawe, technikę tę zastosował również rząd brytyjski do przygotowania listów, które okazały się najbardziej skuteczne w ściąganiu zaległych podatków od obywateli*.

* Istnieje wiele innych sposobów prowadzenia badań użytkownika przed przystąpieniem do projektu rozwojowego. Do najtańszych metod można zaliczyć ankiety, tworzenie prototypów (mogą to być makietki wykonane za pomocą takich narzędzi jak Balsamiq albo wersje interaktywne napisane w kodzie) oraz wykonywanie testów użyteczności. Alberto Savoia, dyrektor

INTEGRACJA TESTÓW A/B DO PROCESU TESTOWANIA CECHY FUNKCJONALNEJ

Najczęściej stosowaną techniką testowania A/B wśród nowoczesnych praktyk UX jest stworzenie witryny internetowej, gdzie odwiedzającym jest losowo wyświetlna jedna z dwóch wersji witryny „A” lub „B”. Na podstawie analizy statystycznej późniejszego zachowania tych dwóch grup użytkowników można pokazać, czy istnieje znacząca różnica wyników pomiędzy tymi dwoma grupami. W ten sposób można ustawić związek przyczynowo-skutkowy pomiędzy zastosowanym sposobem (np. zmianą cechy funkcjonalnej, wyglądu elementu, koloru tła), a uzyskanym wynikiem (np. współczynnik konwersji, średnia wielkość zamówienia).

Stosując tę metodę, można na przykład przeprowadzić eksperyment, który ma ustalić, czy modyfikacja tekstu bądź koloru przycisku „kup” spowoduje zwiększenie dochodów lub czy spowolnienie czasu odpowiedzi witryny internetowej (poprzez wprowadzenie sztucznego opóźnienia) zmniejsza dochody. Testy A/B tego typu pozwalają ustalić kwotę wynikającą z poprawy wydajności.

Testy A/B czasami są również nazywane kontrolowanymi eksperymentami online lub testami podziału (ang. *split tests*). Możliwe jest również przeprowadzanie eksperymentów z więcej niż jedną zmienną. W ten sposób można zaobserwować sposób interakcji zmiennych. Technikę tę określa się jako testowanie wielu zmiennych.

Wyniki testów A/B często są zaskakujące. Ronny Kohavi, posiadacz tytułu Distinguished Engineer oraz dyrektor generalny grupy analiz i eksperymentów w firmie Microsoft, zauważał, że po „przeprowadzeniu oceny dobrze zaprojektowanych i wykonanych eksperymentów cech funkcjonalnych zaprojektowanych w celu poprawy kluczowych parametrów, tylko około jedna trzecia przyniosła sukces w postaci poprawy tych parametrów!”. Mówiąc inaczej, dwie trzecie eksperymentów albo wywarło znikomy wpływ na wskazane parametry lub nawet pogorszyło sytuację. Kohavi zauważał, że wszystkie te cechy funkcjonalne pierwotnie były uważane za rozsądne, dobre pomysły. To jeszcze bardziej podkreśla przewagę badań użytkowników nad intuicją i opiniemi ekspertów.

Implikacje wynikające z danych Kohaviego są zdumiewające. Jeśli nie przeprowadzimy badań użytkowników, to może się okazać, że dwie trzecie spośród budowanych cech funkcjonalnych dostarczy zerowy przychód lub przyniesie **straty** organizacji, choćby wynikające ze zwiększonej złożoności bazy kodu, co przyczynia się do zwięk-

inżynierii w firmie Google, wymyślił termin „prototyping”, opisujący praktykę stosowania prototypów w celu weryfikacji, czy budujemy prawidłowy produkt. Badania użytkowników to na tyle tani i łatwy sposób w porównaniu z wysiłkiem i kosztami budowy bezużytecznej cechy funkcjonalnej w kodzie, że prawie w każdym przypadku nie należy tworzyć cechy funkcjonalnej bez jakiejś formy sprawdzenia jej poprawności.

szonych kosztów utrzymania i trudniejszego modyfikowania oprogramowania. Co więcej, wysiłek potrzebny do budowania tych cech funkcjonalnych często jest ponoszony kosztem tych cech funkcjonalnych, które dostarczyłyby wartość (tzw. koszt alternatywny). Jak zażartował Jez Humble: „Gdyby zastosować podejście ekstremalne, to można by dojść do wniosku, że wysłanie wszystkich pracowników firmy na wakacje przyniosłyby firmie i jej klientom więcej korzyści niż budowanie jednej z tych niewartych cech funkcjonalnych”.

Środkiem zaradczym jest zintegrowanie testowania A/B ze sposobem projektowania, implementowania, testowania i wdrażania dostarczanych cech funkcjonalnych. Przeprowadzanie sensownych badań użytkowników i eksperymentów daje pewność, że podejmowane wysiłki pomagają w osiągnięciu celów klientów i organizacji oraz pomogą w osiągnięciu przewagi na rynku.

INTEGRACJA TESTOWANIA A/B Z PROCESEM PUBLIKOWANIA

Szybkie, iteracyjne testy A/B są możliwe dzięki zdolności do szybkiej i łatwej realizacji wdrożeń produkcyjnych na żądanie przy użyciu przełączników cech funkcjonalnych oraz możliwości jednoczesnego dostarczenia wielu wersji kodu do różnych segmentów klientów. Korzystanie z tej techniki wymaga dostępności użytecznych wskaźników telemetrycznych na wszystkich poziomach stosu aplikacji.

Dzięki podpięciu do przełączników cech funkcjonalnych możemy zarządzać tym, jaki procent użytkowników uzyska dostęp do eksperymentalnej wersji produktu. Na przykład grupa poddawana badaniu może stanowić połowę naszych klientów, a drugiej poowie jest wyświetlany następujący komunikat: „Podobne towary prowadzą do pozycji niedostępnych w koszyku”. W ramach eksperymentu porównujemy zachowania w grupie kontrolnej (gdzie nie złożono oferty) z zachowaniami grupy badanej (gdzie ofertę złożono) i na przykład dokonujemy pomiaru liczby zakupów dokonanych w tej sesji.

W firmie Etsy opracowano framework do przeprowadzania eksperymentów o statusie open source Feature API (dawniej znany jako Etsy A/B API), który obsługuje nie tylko testy A/B, ale również eksperymenty online oraz funkcje „dławienia” dostępu do eksperymentów. Do innych produktów oferujących funkcjonalność testów A/B należą Optimizely, Google Analytics itp.

W 2014 r. w wywiadzie z Kendrickiem Wangiem z Apptimize Lacy Rhoades z Etsy opisywał swoje doświadczenia: „Źródłem inspiracji do eksperymentów w firmie Etsy jest dążenie do podejmowania świadomych decyzji i zyskanie gwarancji, że jeśli uruchamiamy cechy funkcjonalne dla milionów użytkowników, to one działają. Zbyt często zdarzało się nam pracować nad pewnymi cechami funkcjonalnymi przez długi czas i musieliśmy je utrzymywać, nie mając jakiegokolwiek dowodu na to, że odniosły

sukces lub zyskały popularność wśród użytkowników. Testy A/B pozwalają nam potwierdzić, że... cecha funkcjonalna była warta poświęcania jej czasu niemal natychmiast po rozpoczęciu pracy nad nią”.

WŁĄCZENIE TESTÓW A/B DO PROCESU PLANOWANIA CECH FUNKCJONALNYCH

Gdy już mamy infrastrukturę do obsługi publikowania i testowania A/B, powinniśmy zadbać o to, aby właściciele produktu myśleli o każdej własności funkcjonalnej jak o hipotezie, a wydania produkcyjne wykorzystywali jako eksperymenty z prawdziwymi użytkownikami w celu udowodnienia swoich hipotez. Konstruowanie eksperymentów powinno być projektowane w kontekście ogólnego lejka popytu klientów. Barry O'Reilly, współautor książki *Lean Enterprise: How High Performance Organizations Innovate at Scale*, zaproponował następujący sposób formułowania hipotez w procesie wytwarzania oprogramowania:

Wierzymy, że zwiększenie rozmiaru zdjęć hoteli na stronie rezerwacji

Spowoduje większe zaangażowanie klientów i wyższą konwersję

Będziemy mieć pewność, że warto kontynuować, gdyauważymy 5-procentowy wzrost liczby klientów przeglądających zdjęcia hoteli, którzy następnie w ciągu 48 godzin zdecydują się na rezerwację.

Zastosowanie eksperimentalnego podejścia do rozwoju produktu wymaga od nas nie tylko podzielenia pracy na małe jednostki (historyjki lub wymagania), ale również zweryfikowania, czy każda jednostka pracy przynosi oczekiwane rezultaty. Jeśli tak nie jest, możemy zmodyfikować naszą mapę drogową za pomocą alternatywnych ścieżek, które spowodują faktyczne osiągnięcie spodziewanych wyników.

Studium przypadku

Podwojenie wzrostu przychodów dzięki eksperimentom z szybkimi cyklami wydań w Yahoo! Answers (2010)

Im szersze iteracje oraz wcześniejsza integracja sprzężeń zwrotnych dla produktów i usług dostarczanych klientom, tym szybciej uczymy się i tym większy wpływ na uzyskiwane wyniki. Olbrzymi wpływ szybszych cykli na uzyskiwane wyniki można zaobserwować na przykładzie Yahoo! Answers po przejściu z publikowania jednego wydania co sześć tygodni do wielu wydań co tydzień.

W 2009 roku Jim Stoneham pełnił funkcję dyrektora generalnego grupy Yahoo! Communities, która obejmowała grupy Flickr i Answers. Wcześniej był przede

wszystkim odpowiedzialny za Yahoo! Answers i konkurował z innymi firmami Q&A, takimi jak Quora, Aardvark i Stack Exchange.

W tamtym czasie serwis Answers miał około 140 milionów odwiedzających miesięcznie, ponad 20 milionów aktywnych użytkowników udzielających odpowiedzi na pytania w ponad 20 różnych językach. Jednak wzrost liczby użytkowników i dochody wyhamowały, tendencję spadkową zaczęły wykazywać także współczynniki zaangażowania użytkowników.

Stoneham zauważał, że „Yahoo! Answers była i nadal jest jedną z największych gier społecznościowych w internecie; dziesiątki milionów ludzi aktywnie próbują »przejść na wyższy poziom« poprzez udzielanie wartościowych odpowiedzi na pytania szybciej od innych członków społeczności. Było wiele okazji do dostrojenia mechaniki gry, »pętli zarażania« (ang. *viral loops*) oraz innych interakcji społecznościowych. Gdy ma się do czynienia z zachowaniami prawdziwych ludzi, trzeba wykonywać szybkie iteracje i testy, tak by się przekonać, co się sprawdza”.

Dalej pisał: „Te eksperymenty doskonale przeprowadzono dla serwisów Twitter, Facebook i Zynga. Organizacje te przeprowadzały eksperymenty co najmniej dwa razy w tygodniu — wykonywane zmiany przed wdrożeniem przeglądano tak, aby upewnić się, że kierunek jest właściwy. A więc jestem w tym miejscu. Kieruję największym serwisem typu Q&A na rynku i dążę do iteracyjnego testowania cech funkcjonalnych, ale nie jestem w stanie przeprowadzać wydań szybciej niż raz na cztery tygodnie. Dla odróżnienia inne firmy na rynku mogą korzystać z pętli sprzężenia zwrotnego 10 razy szybciej od nas”.

Stoneham zauważał, że pomimo iż właściciele produktu i programiści mówią o stosowaniu podejścia bazującego na parametrach, to jeśli eksperymenty nie są wykonywane często (codziennie lub co tydzień), to działania koncentrują się jedynie na funkcjonalności, nad którą trwają prace, a nie na wynikach dla klienta.

Gdy zespołu Yahoo! Answers udało się przejść od cotygodniowych wdrożeń do wielu wdrożeń na tydzień, ich zdolność do eksperymentowania z nowymi funkcjonalnościami dramatycznie wzrosła. Dzięki zdumiewającym wynikom i wiedzy uzyskanej w ciągu kolejnych 12 miesięcy eksperymentów liczba odwiedzin miesięcznie wzrosła o 72%, zainteresowanie użytkowników wzrosło trzykrotnie, a zespół podwoił uzyskiwane przychody. W celu kontynuowania dobrej passy zespół skoncentrował się na optymalizacji następujących najważniejszych parametrów:

- Czas do pierwszej odpowiedzi — jak szybko została udzielona odpowiedź na pytanie zadane przez użytkownika.
- Czas do najlepszej odpowiedzi — jak szybko społeczność użytkowników nagrodziła najlepszą odpowiedź.
- Głosy popierające odpowiedź — ile razy odpowiedź była oznaczona głosem użytkownika społeczności.
- Odpowiedzi na tydzień na osobę — ile odpowiedzi tworzyli użytkownicy.

- Współczynnik drugiego wyszukiwania — jak często odwiedzający musieliby przeprowadzić wyszukiwanie po raz drugi, aby uzyskać odpowiedź (im niższa wartość, tym lepiej).

Stoneham zakończył: „Właśnie takiej wiedzy potrzebowaliśmy, aby osiągnąć przewagę na rynku. Dzięki niej zmieniło się więcej niż tylko dynamika publikowania funkcjonalności. Przeprowadziliśmy transformację z zespołu pracowników na zespół właścicieli. Gdy poruszasz się w takim tempie i codziennie obserwujesz liczby i wyniki, poziom inwestycji radykalnie się zmienia”.

PODSUMOWANIE

Sukces wymaga nie tylko szybkiego wdrażania i publikowania oprogramowania, ale także eksperymentowania w celu osiągnięcia przewagi nad konkurencją. Stosowanie takich technik, jak wytwarzanie oprogramowania sterowane hipotezami, definiowanie i pomiary lejka popytu klientów oraz testy A/B, umożliwia przeprowadzanie eksperymentów z użytkownikami bezpiecznie i łatwo, co pozwala nam wyzwolić kreatywność i innowacyjność oraz utworzyć system uczenia się w organizacji. I chociaż sukces jest ważny, to nauka dla organizacji, która płynie z eksperymentowania, daje pracownikom poczucie tożsamości z celami biznesowymi firmy oraz zadowolenie klientom. W następnym rozdziale przeanalizujemy procesy przeglądania i koordynowania jako sposób na podniesienie jakości wykonywanej pracy.

Tworzenie procesów przeglądu i koordynacji w celu poprawy jakości bieżącej pracy

W poprzednich rozdziałach pokazaliśmy sposób tworzenia wskaźników telemetrycznych niezbędnych do obserwowania i rozwiązywania problemów w produkcji oraz na wszystkich etapach potoku wdrożeń, a także sposób tworzenia szybkich pętli sprzężeniu zwrotnych od klientów w celu usprawnienia procesu uczenia się w organizacji — zdobywania wiedzy, która zachęca do poczucia własności i odpowiedzialności za satysfakcję klienta i wydajność funkcjonalności, a dzięki temu pomaga osiągnąć sukces.

Celem w tym rozdziale jest umożliwienie działom Dev i Ops zmniejszenia ryzyka wprowadzenia zmian w produkcji, zanim zostaną wprowadzone. Zgodnie z tradycyjnym podejściem przy przeglądaniu zmian na potrzeby wdrażania zwykle polegamy na opiniach, przeglądach i zatwierdzeniach tuż przed wdrożeniem. Często te zatwierdzenia są dokonywane przez zewnętrzne zespoły, które są zbyt daleko od pracy, aby mogły podejmować świadome decyzje na temat tego, czy zmiana jest ryzykowna, czy nie, a czas wymagany do uzyskania niezbędnych zgód dodatkowo wydłuża czas realizacji zmiany.

Proces przeglądu kodu w firmie GitHub to doskonały przykład na skuteczność stosowania przeglądów w celu poprawy jakości, zwiększenia bezpieczeństwa wdrożeń oraz integracji wdrożeń w przepływ codziennej pracy. Firma GitHub była pionierem procesu o nazwie *pull request* (dosł. „żądanie ściągnięcia”) — jednej z najpopularniejszych form wzajemnego przeglądania kodu, która uwzględnia członków zespołów Dev i Ops.

Scott Chacon, szef działu informatyki i współzałożyciel firmy GitHub, napisał w swojej witrynie internetowej, że żądania ściągnięcia są mechanizmem, który pozwala inżynierom opowiedzieć innym o zmianach, które zostały zaewidencjonowane w repozytorium w GitHub. Po wysłaniu żądania ściągnięcia zainteresowane strony mają możliwość przeglądania zbioru zmian, omówienia potencjalnych modyfikacji, a nawet zaewidencjonowania kolejnych zmian, jeśli to konieczne. Inżynierowie przesyłający żądania ściągnięcia często żądają „+ 1”, „+ 2” itd., w zależności od tego, ile przeglądów potrzebują albo od ilu inżynierów „@mention” chcą uzyskać opinie.

W firmie GitHub żądania ściągnięcia są również mechanizmem używanym do wdrażania kodu do produkcji za pośrednictwem zbioru praktyk o nazwie „GitHub Flow” — opisuje on sposób, w jaki inżynierowie żądają przeglądu kodu, jak zbierają opinie, jak z nich korzystają oraz w jaki sposób zapowiadają wdrożenie kodu do produkcji (czyli do gałęzi „master”) (rysunek 40).

The screenshot shows a GitHub pull request interface. At the top, there's a code snippet with line numbers 35-38 and a line 27. The code is:

```
35   27
36   28
37   29
38   30
      +   opts[:options] [:stripnl] || = false
          timeout opts.delete(:timeout) || DEFAULT_TIMEOUT do
              begin
                  Pygments.highlight(text, opts)

```

Below the code, there's a comment from user brianmario:

brianmario repo collab
Zatem jakie tutaj są ustawienia domyślne, skoro nie jest przekazywane kodowanie ani nie jest przekazywany obiekt lexer?
Poza tym jest co najmniej jedno dodatkowe miejsce, gdzie API powinno pobierać klucz kodowania (niezagnieździone w :options key/hash) –
<https://github.com/github/github/blob/master/app/models/gist.rb#L114>
Jedynym powodem, dla którego zrobilem to tutaj w ten sposób, była chęć wyabstrahowania faktu, że obecnie stosujemy pigmenty do kolorowania (a nie że planujemy to wkrótce zmienić...)

Underneath, another comment from user Josh:

Josh repo collab
W porządku, przekażę to dalej w celu kolorowania.
Dodaj komentarz

Rysunek 40. Komentarze i sugestie dla żądań ściągnięcia w GitHub (źródło: Scott Chacon, „GitHub Flow” — ScottChacon.com, <http://scottchacon.com>, 31 sierpnia 2011 roku, <http://scottchacon.com/2011/08/31/github-flow.html>)

Proces GitHub Flow składa się z pięciu kroków:

1. W celu rozpoczęcia pracy na czymś nowym inżynier tworzy gałąź z pnia i nadaje jej nazwę opisową (np. „new-oauth2-scopes”).
2. Inżynier ewidencjonuje zmiany na tej gałęzi lokalnie. Co jakiś czas przesyła pracę do gałęzi o takiej samej nazwie na serwerze.
3. Gdy są potrzebne opinie lub pomoc albo gdy inżynier uznaje gałąź za gotową do scalenia, otwiera żądanie ściągnięcia.

4. Po uzyskaniu żądanego opinii oraz wszelkich niezbędnych uzgodnień dla funkcjonalności inżynier może ją scalić z gałęzią master.
5. Gdy zmiany w kodzie zostaną scalone i zaewidencjonowane na gałęzi master, inżynier wdraża zmiany do produkcji.

Pokazane praktyki, które łączą zadania przeglądu i koordynacji z codzienną pracą, pozwoliły firmie GitHub szybko i niezawodnie dostarczyć funkcjonalności na rynek z zachowaniem wysokiej jakości i bezpieczeństwa. Na przykład w 2012 roku firma GitHub przeprowadziła niesamowitą liczbę 12 602 wdrożeń. W szczególności 23 sierpnia, po szczytce firmowej, na którym omówiono wiele ciekawych pomysłów, firma odnotowała najbardziej intensywny dzień wdrożeń w roku. Wykonano 563 komplikacje i 175 udanych wdrożeń do produkcji. Wszystko to było możliwe dzięki procesowi żądania ściągania.

W tym rozdziale pokażemy sposób zintegrowania praktyk podobnych do tych, które stosowano w firmie GitHub w celu wyeliminowania okresowych kontroli i uzgodnień oraz przejścia do ciągłego procesu zatwierdzania wykonywanego w ramach codziennej pracy. Celem powinno być zapewnienie stałej współpracy pomiędzy inżynierami Dev, Ops i Infosec, tak aby zmiany wprowadzane w systemie działały niezawodnie, bezpiecznie, pewnie i zgodnie z projektem.

NIEBEZPIECZEŃSTWA ZWIĄZANE Z PROCESAMI ZATWIERDZANIA ZMIAN

Awaria w firmie Knight Capital to jeden z najbardziej znanych błędów wdrażania oprogramowania w ostatnich latach. Błąd wdrażania i brak możliwości zablokowania usług produkcyjnych przez 15 minut doprowadził do 440 milionów dolarów strat. Straty finansowe spowodowały zagrożenie dla działania firmy, która musiała być sprzedana w ciągu weekendu, tak aby dłużej nie narażać całego systemu finansowego.

John Allspaw zauważył, że po zajściu takich sytuacji, jak ta związana z wdrożeniem w firmie Knight Capital, zazwyczaj krążą dwa alternatywne poglądy na temat powodów wystąpienia incydentu^{*}.

Pierwszy mówi o tym, że incydent spowodował błędne zarządzanie zmianami. Wydaje się to prawidłowe, ponieważ można sobie wyobrazić sytuację, w której stosowanie praktyk lepszego zarządzania zmianami mogło pozwolić na wcześniejsze wykrycie zagrożenia i zapobiec wdrożeniu zmiany do produkcji. A jeśli nie można było

* Myślenie alternatywne (ang. *counterfactual thinking*) to termin używany w psychologii, który uwzględnia ludzką skłonność do tworzenia alternatyw zdarzeń, które już się wydarzyły. W inżynierii niezawodności często obejmuje ono opisy „systemu zgodnego z wyobrażeniami” w przeciwieństwie do „systemu w rzeczywistości”.

zapobiec zmianom, to być może można było podjąć kroki zmierzające do szybszego wykrywania problemów i przywrócenia stanu prawidłowego.

Drugi pogląd mówi, że do incydentu doszło z powodu błędu testowania. To również wydaje się prawdziwe, ponieważ stosując lepsze praktyki testowania, można było zidentyfikować zagrożenia wcześniej i wyeliminować ryzykowne wdrożenie albo przynajmniej podjąć kroki umożliwiające szybsze wykrywanie i odzyskiwanie.

Zaskakująca rzeczywistość jest taka, że w środowiskach, w których panuje kultura niskiego zaufania, kultura dowodzenia i kontroli, wyniki stosowania wymienionych mechanizmów zaradczych w postaci zarządzania zmianami i testowania często powodują zwiększone prawdopodobieństwo ponownego wystąpienia problemów — potencjalnie z jeszcze gorszymi efektami.

Gene Kim (współautor niniejszej książki) opisuje uświadomienie sobie faktu, że mechanizmy zarządzania zmianami i testowania mogą przynieść skutek odwrotny do zamierzonego, jako „jeden z najważniejszych momentów swojej kariery zawodowej. Ten moment »aha« był wynikiem rozmowy przeprowadzonej w 2013 roku z Johnem Allspawem i Jezem Humble’em, dotyczącej incydentu w firmie Knight Capital. W efekcie zacząłem kwestionować jedno z moich podstawowych przekonań, które uformowało się we mnie w ciągu ostatnich 10 lat, zwłaszcza podczas szkoleń z prowadzenia audytów”.

Jak kontynuuje Gene Kim: „Choć był to moment przykry, to był on również niezwykle twórczy. Moi rozmówcy nie tylko zdołali mnie przekonać, że mają rację, ale również poglądy te zostały sprawdzone i zestawione w *2014 State of DevOps Report*, co doprowadziło do zdumiewających wniosków wzmacniających twierdzenie, że budowanie kultury wysokiego zaufania jest prawdopodobnie największym wyzwaniem zarządzania bieżącej dekady”.

POTENCJALNE ZAGROŻENIA ZWIĄZANE ZE „ZBYTNIM KONTROLOWANIEM ZMIAN”

Tradycyjne mechanizmy kontroli zmian mogą prowadzić do niepożądanych efektów, takich jak długie czasy realizacji, a także mogą zmniejszyć siłę i natychmiastowość sprzężenia zwrotnego z procesu wdrażania. Aby zrozumieć, jak to się dzieje, spróbujmy zbadać mechanizmy, które są często stosowane, gdy system kontroli zmian zawiedzie:

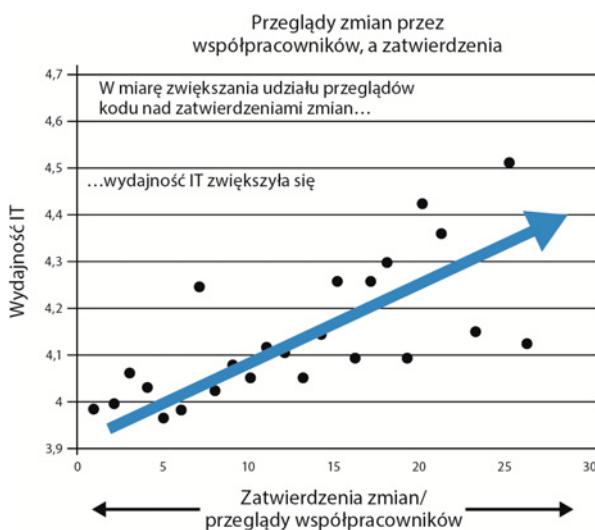
- Wprowadzenie dodatkowych pytań, na które trzeba odpowiedzieć na formularzu żądania zmiany.
- Wymaganie więcej zatwierdzeń — na przykład dodatkowy poziom akceptacji kierownictwa (np. zatwierdza nie tylko wiceprezes działu operacji, ale również dyrektor ds. informatyki) albo większa liczba interesariuszy (np. inżynierowie sieci, komisja przeglądu architektury itp.).

- Wymaganie dłuższego czasu realizacji dla zatwierdzeń zmian, tak aby żądania zmian mogły być właściwie ocenione.

Te mechanizmy kontroli często wprowadzają dodatkowe tarcia do procesu wdrażania. Zwiększą liczbę realizowanych kroków i zatwierdzeń oraz wielkości partii i wydłużają czas realizacji wdrażania, co jak wiadomo, zmniejsza prawdopodobieństwo skutecznych efektów produkcji zarówno dla działów Dev, jak i Ops. Te mechanizmy kontroli wydłużają również czas uzyskania opinii na temat pracy.

Zgodnie z jednym z podstawowych przekonań w Toyota Production System, „ludzie znajdujący się najbliżej problemu, zazwyczaj wiedzą o nim najwięcej”. Staje się to bardziej widoczne wraz ze wzrostem złożoności i dynamiki wykonywanej pracy oraz systemu, w którym prace są wykonywane — co jest typowe w strumieniach wartości DevOps. W takich przypadkach tworzenie etapów polegających na zatwierdzaniu przez osoby, które są coraz dalej od wykonywanej pracy, może w efekcie zmniejszyć prawdopodobieństwo sukcesu. Jak wielokrotnie sprawdzono, im większy dystans pomiędzy osobą wykonującą pracę (tzn. realizatorem zmiany) a osobą podejmującą decyzję o wykonaniu pracy (tzn. zatwierdzającym zmiany), tym gorsze wyniki.

W raporcie Puppet Labs *State of DevOps Report* z 2014 r. jednym z najważniejszych ustaleń było stwierdzenie, że wysokowydajne organizacje bazują w większym stopniu na przeglądach zmian dokonywanych przez współpracowników niż na zewnętrznych procesach zatwierdzania zmian. Na rysunku 41 pokazano, że im bardziej organizacja polega na zatwierdzaniu zmian, tym gorsza jest jej wydajność IT pod względem zarówno stabilności (średni czas do przywrócenia usługi i zmiany współczynnika awarii), jak i przepustowości (czasy realizacji wdrażania, częstotliwość wdrażania).



Rysunek 41. Organizacje, które bazują na przeglądach współpracowników, mają wyższą wydajność od tych, które stosują zatwierdzenia zmian (źródło: Puppet Labs, DevOps Survey Of Practice 2014)

W wielu organizacjach ważną rolę w zarządzaniu procesami dostarczania i koordynowaniu ich spełniają doradcze komisje zarządzania zmianami. Ich zadania nie powinny jednak polegać na ręcznej ocenie każdej zmiany. Kodeks ITIL nie zaleca stosowania tego rodzaju praktyk.

Aby zrozumieć, dlaczego tak się dzieje, wyobraźmy sobie, że jesteśmy członkiem takiej doradczej komisji i przeglądamy skomplikowaną zmianę obejmującą kilkaset tysięcy wierszy kodu, utworzoną przez setki inżynierów.

Z jednej strony nie jesteśmy w stanie wiarygodnie przewidzieć, czy zmiana przyniesie sukces — ani przez czytanie obszernego opisu zmiany, ani przez weryfikację spełnienia warunków wymienionych na liście kontrolnej. Z drugiej strony jest mało prawdopodobne, aby wnikliwe i żmudne sprawdzanie wielu tysięcy wierszy kodu spowodowało ujawnienie jakichkolwiek nowych informacji. Po części wynika to z charakteru wprowadzania zmian w złożonym systemie. Nawet inżynierowie, którzy na co dzień pracują z bazą kodu, często są zaskoczeni skutkami ubocznymi zmian, które nie powinny stwarzać zbyt wielkiego ryzyka.

Ze względu na wszystkie wymienione powody należy stworzyć skuteczne praktyki kontroli, które bardziej przypominają przeglądy współpracowników, i zredukować wpływ korzystania z organów zewnętrznych do autoryzowania zmian. Trzeba również skutecznie koordynować i planować zmiany. Oba te mechanizmy zostaną omówione w kolejnych dwóch podrozdziałach.

KOORDYNACJA I PLANOWANIE ZMIAN

Zawsze wtedy, gdy nad systemami pracuje wiele grup, które są od siebie wzajemnie, trzeba zadbać o koordynację zmian (np. szeregowanie, tworzenie pakietów i ustawianie kolejek), tak aby mieć pewność, że poszczególne grupy wzajemnie sobie nie przeszkadzią. Ogólnie rzecz biorąc, im luźniejsze sprzężenia w architekturze, tym mniejsza potrzeba komunikacji i koordynacji z zespołami innych komponentów — gdy architektura jest rzeczywiście zorientowana na usługi, to zespoły mogą wprowadzać zmiany z dużym stopniem autonomii i istnieje niewielkie prawdopodobieństwo, aby lokalne zmiany spowodowały globalne awarie.

Jednak nawet w przypadku luźnych sprzężeń, kiedy wiele zespołów realizuje codziennie kilkaset niezależnych wdrożeń, może istnieć zagrożenie, że zmiany będą wzajemnie ze sobą kolidować (np. jednocześnie testy A/B). W celu złagodzenia tych zagrożeń można korzystać z czatów do ogłaszenia zmiany i proaktywnie wyszukiwać istniejące kolizje.

W przypadku bardziej złożonych organizacji oraz organizacji stosujących ścisłe sprzężone architektury czasami trzeba świadomie zaplanować zmiany. Wtedy przedstawiciele poszczególnych zespołów spotykają się nie po to, żeby zatwierdzać zmiany, ale po to, żeby je zaplanować i uszeregować, tak aby ryzyko wystąpienia incydentów było jak najmniejsze.

Istnieją jednak obszary — na przykład zmiany w globalnej infrastrukturze (wymiana głównego przełącznika sieciowego itp.) — które zawsze będą stwarzały większe zagrożenia. Te zmiany zawsze wymagają stosowania technicznych środków zaradczych, takich jak redundancja, mechanizmy pracy awaryjnej (ang. *failover*), kompleksowe testy i (najlepiej) symulacje.

PRZEGŁĄDY ZMIAN PRZEZ WSPÓŁPRACOWNIKÓW

Zamiast wymagać zatwierdzeń przez zewnętrzne organy przed wdrażaniem, możemy wymagać od inżynierów wzajemnego przeglądu dokonywanych przez nich zmian. W działach Dev praktykę tę określa się terminem **przeglądów kodu** (ang. *code review*), ale równie dobrze może być ona zastosowana w odniesieniu do dowolnych zmian wprowadzanych w aplikacji lub środowiskach — w tym serwerach, urządzeniach sieciowych i bazach danych*. Celem tej czynności jest znalezienie błędów poprzez powierzenie współpracownikom zadania przeanalizowania wprowadzonych zmian. Przeglądy kodu poprawiają jakość wprowadzanych zmian. Dodatkowe korzyści wynikające z tego procesu to przekrojowe szkolenia, wzajemne uczenie się i doskonalenie umiejętności.

Logicznym momentem do zażądania przeglądu jest zaewidencjonowanie kodu w gałęzi master w repozytorium kontroli wersji, ponieważ w tym momencie zmiany mogą wywierać wpływ na pracę zespołu bądź innych zespołów. Minimalnym wymaganiem jest dokonanie przeglądu kodu przez współpracownika, ale w obszarach podwyższonego ryzyka, takich jak zmiany w bazie danych lub krytycznych komponentach słabo pokrytych automatycznymi testami, możemy wymagać kolejnego przeglądu przez eksperta w określonej dziedzinie (np. inżyniera bezpieczeństwa informacji, inżyniera bazy danych) lub wielu opinii (np. „+ 2” zamiast tylko „+ 1”).

Do przeglądów kodu ma również zastosowanie zasada małych partii. Im większy rozmiar zmiany, która musi być poddana przeglądowi, tym dłużej zajmuje jej zrozumienie i tym większe obciążenie inżyniera dokonującego przeglądu. Jak zauważył Randy Shoup: „Istnieje nieliniowa zależność pomiędzy wielkością zmiany i potencjalnymi zagrożeniami związanymi z jej integracją — przy przejściu ze zmiany w kodzie obejmującej 10 wierszy do 100 wierszy ryzyko, że coś pójdzie nie tak, jest ponad 10 razy wyższe i tak dalej”. Dlatego właśnie jest tak ważne, aby programiści wprowadzali niewielkie, przyrostowe zmiany, a nie utrzymywali długowieczne gałęzie rozwoju funkcjonalności.

Ponadto zdolność do konstruktywnej krytyki kodu maleje wraz ze wzrostem rozmiarów zmiany. Giray Özil napisał na Twitterze: „Poproś programistę, żeby dokó-

* W tej książce terminy „przeglądy kodu” i „przeglądy zmian” będą używane zamiennie.

nał przeglądu 10 wierszy kodu, to znajdzie w nim 10 problemów. Poproś go, aby zrobił to dla 500 wierszy, to powie, że kod wygląda dobrze”.

Oto kilka wytycznych dotyczących przeglądów kodu:

- Każdy musi mieć kogoś, kto dokonuje u niego przeglądu wprowadzanych zmian (np. w kodzie, środowisku itp.) przed zaewidencjonowaniem zmian w gałęzi master.
- Każdy powinien monitorować strumień ewidencjonowania zmian przez współpracowników z zespołu, tak aby można było zidentyfikować i przejrzeć potencjalne konflikty.
- Należy zdefiniować zmiany, które kwalifikują się jako zmiany wysokiego ryzyka i mogą wymagać przeglądu przez wyznaczonego eksperta z określonej dziedziny (np. zmiany w bazie danych, zmiany w modułach wrażliwych z punktu widzenia zabezpieczeń — na przykład uwierzytelniania itp.)^{*}.
- Jeśli ktoś zaewidencjonuje zmianę, która jest zbyt duża, aby można było ją łatwo ocenić — innymi słowy, nie można zrozumieć, jaki wywrze ona wpływ, po kilkakrotnym przeczytaniu albo trzeba zadawać pytania autorowi o wyjaśnienia — należy ją podzielić na wiele mniejszych zmian, które można zrozumieć bez kłopotów.

Aby mieć pewność, że przeglądy nie są jedynie ślepym „przyklepywaniem”, można sprawdzić statystyki przeglądów kodu w celu określenia liczby zatwierdzonych propozycji zmian w zestawieniu z niezatwierdzonymi albo przeanalizować przykłady określonych przeglądów kodu.

Przeglądy kodu mogą mieć różne formy:

- **Programowanie w parach** — programiści pracują w parach (podrozdział poniżej).
- **„Zapuszczanie żurawia”** — jeden programista patrzy „przez ramię” programisty wprowadzającego zmiany w kodzie.
- **Wiadomości e-mail z powiadomieniami o zmianach** — system zarządzania kodem źródłowym po zaewidencjonowaniu zmiany automatycznie wysyła wiadomość e-mail do recenzentów.
- **Przegląd kodu z wykorzystaniem narzędzi** — autorzy i recenzenci używają specjalistycznych narzędzi przeznaczonych do przeglądu kodu (np. Gerrit, żądania ściągnięcia GitHub itp.) lub mechanizmów dostarczanych przez repo-

^{*} Najprawdopodobniej lista obszarów kodu i środowisk podwyższonego ryzyka została stworzona wcześniej przez wyznaczony zespół zarządzania zmianami (ang. *change advisory board* — CAB).

zytoria kodu źródłowego (np. GitHub, Mercurial, Subversion, jak również inne platformy, np. Gerrit, Atlassian Stash i Atlassian Crucible).

Ścisła analiza zmian w wielu postaciach jest skutecznym sposobem na wyszukanie błędów, które wcześniej zostały przeoczone. Przeglądy kodu mogą ułatwić obsługę większej liczby operacji ewidencjonowania oraz wdrożeń produkcyjnych. Wspomagają także wdrożenia bazujące na gałęzi master oraz mechanizmy ciągłego dostarczania na dużą skalę, co opisano w poniższym studium przypadku.

Studium przypadku

Przeglądy kodu w Google (2010)

Google jest doskonałym przykładem firmy, której pracownicy stosują mechanizmy wdrażania na bazie gałęzi master oraz ciągłe dostarczanie na dużą skalę. Jak wspomniano wcześniej w tej książce, Eran Messeri pisał o procesach przeprowadzonych w Google w 2013 r., które pozwoliły ponad 13 000 programistów korzystać z jednego drzewa kodu źródłowego i wykonywać ponad 5500 operacji ewidencjonowania kodu w repozytorium tygodniowo, czego efektem były setki wdrożeń produkcyjnych tygodniowo. W 2010 roku było ponad 20 zmian ewidencjonowanych w gałęzi master co minutę. W rezultacie każdego miesiąca zmianom podlegało 50% bazy kodu.

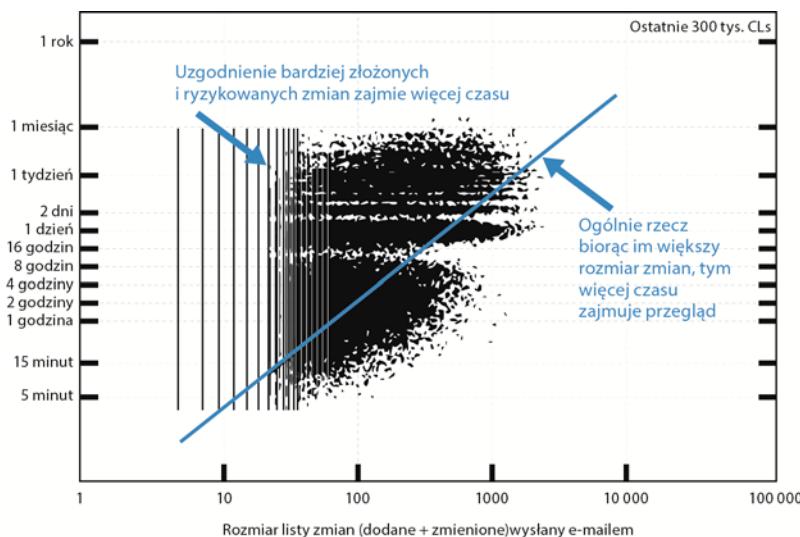
Wymagało to znaczcej dyscypliny od członków zespołu Google i obowiązkowych przeglądów kodu, które obejmowały następujące obszary:

- czytelność kodu (wymuszona przez przewodnik stylu);
- przydzielanie odpowiedzialności za poddrzewa kodu w celu utrzymania spójności i poprawności;
- przejrzystość kodu i wpływ kodu na inne zespoły.

Na rysunku 42 pokazano wpływ rozmiaru zmian na czasy realizacji przeglądów kodu. Na osi X zaznaczono wielkość zmiany, natomiast na osi Y czas wymagany do przeprowadzenia procesu przeglądu kodu. Ogólnie rzecz biorąc, im większe zmiany przekazane do przeglądu, tym dłuższy czas wymagany do zebrania potrzebnych opinii. Punkty danych w lewym górnym rogu reprezentują zmiany bardziej złożone i potencjalnie wprowadzające większe zagrożenia, a tym samym wymagające więcej dyskusji.

Randy Shoup w czasie swojej pracy na stanowisku dyrektora inżynierii w Google rozpoczął własny projekt, mający na celu rozwiązywanie problemów technicznych, z jakimi borykała się jego organizacja. Jak powiedział: „Pracowałem nad tym projektem przez wiele tygodni i wreszcie dotarłem do momentu, w którym poprosiłem dziedzinowego eksperta o przegląd mojego kodu. To było prawie 3000 linii kodu i analiza zajęła recenzentowi wiele dni pracy. Powiedział do mnie: »Proszę, nie rób mi tego nigdy więcej!«. Byłem wdzięczny temu inżynierowi

za to, że poświęcił mi swój czas. Wtedy także nauczyłem się, że przeglądy kodu powinny być częścią codziennej pracy”.



Rysunek 42. Rozmiar zmian a czas realizacji przeglądu w firmie Google
(źródło: Ashish Kumar, Development at the Speed and Scale of Google,
prezentacja na konferencji QCon, San Francisco, CA, 2010)

POTENCJALNE ZAGROŻENIA WYNIKAJĄCE Z WIĘKSZEGO UDZIAŁU TESTÓW RĘCZNYCH I ZAMRAŻANIA ZMIAN

Kiedy już stworzyliśmy mechanizm wzajemnych przeglądów zmian, który pozwala zmniejszyć zagrożenia wynikające z wprowadzania zmian oraz skrócić czas realizacji procesów ich zatwierdzania i umożliwić stosowanie ciągłego dostarczania na dużą skalę (tak jak opisano w studium przypadku firmy Google), to spróbujmy pokazać sytuacje, w których środki zaradcze dla procesu testowania mogą czasami trafić w nas rykoszetem. W przypadku wykrycia awarii podczas testowania typową reakcją jest zwykle przeprowadzenie większej liczby testów. Jeśli jednak wykonujemy więcej testów tylko na koniec projektu, to możemy pogorszyć osiągane wyniki.

Dotyczy to zwłaszcza tych sytuacji, kiedy wykonujemy testy ręczne. Testowanie ręczne jest naturalnie wolniejsze i bardziej uciążliwe od testowania automatycznego, dlatego wykonanie „dodatkowych testów” często skutkuje znacznie dłuższym czasem testowania. To oznacza rzadsze wdrażanie, a tym samym zwiększenie rozmiarów wdrażanej paczki. Wiemy zarówno z teorii, jak i z praktyki, że zwiększenie rozmiaru paczki wdrożenia powoduje obniżenie współczynnika sukcesu zmiany i zwiększenie

współczynników liczby incydentów i MTTR (średniego czasu naprawy) — jest to efekt przeciwny do zamierzonego.

Zamiast przeprowadzać testy na dużych partiach zmian, zaplanowanych w okresach zamrożenia modyfikacji, powinniśmy w pełni zintegrować testowanie do codziennej pracy — wkomponować je w proces płynnego i ciągłego przepływu do produkcji oraz zwiększyć częstotliwość wdrażania. W ten sposób budujemy jakość, co pozwala nam testować, wdrażać i wydawać coraz mniejsze partie zmian.

PROGRAMOWANIE W PARACH W CELU POPRAWY JAKOŚCI WPROWADZANYCH ZMIAN

Programowanie w parach to proces, w którym dwóch inżynierów współpracuje ze sobą na tej samej stacji roboczej. Metoda ta została spopularyzowana dzięki nurtowi programowania ekstremalnego wraz z technikami Agile na początku XXI w. Podobnie jak w przypadku przeglądów kodu, praktykę tę zaczęli stosować programiści, ale w równym stopniu może ona znaleźć zastosowanie w pracy każdego inżyniera w naszym strumieniu wartości. W tej książce będziemy używać terminów **praca parami** (ang. *pairing*) i **programowanie parami** (ang. *pair programming*) zamiennie, aby pokazać, że praktyka ta nie dotyczy wyłącznie programistów.

W jednym z popularnych wzorców pracy w parach jeden inżynier odgrywa rolę **kierowcy**, tzn. jest osobą, która faktycznie pisze kod, podczas gdy drugi jest **nawigatorem** lub **obserwatorem**, czyli osobą, która obserwuje pracę w czasie jej wykonywania. Podczas przeglądania obserwator może również analizować strategiczny kierunek pracy, zgłaszać pomysły usprawnień oraz proponować obszary, którymi należałoby się zająć. Dzięki temu kierowca może skupić całą swoją uwagę na taktycznych aspektach realizacji zadania, korzystając z pomocy obserwatora w roli zabezpieczenia oraz jako przewodnika. Gdy dwaj inżynierowie mają różne specjalności, automatycznym efektem ubocznym jest transfer umiejętności, który odbywa się za pośrednictwem szkolenia ad hoc lub w wyniku wymiany poglądów na temat stosowanych technik i sposobów rozwiązywania problemów.

Inny wzorzec programowania parami wzmacnia ideę programowania sterowanego testami (ang. *test-driven development* — **TDD**), gdy jeden inżynier pisze zautomatyzowane testy, natomiast drugi implementuje kod. Jeff Atwood, jeden z założycieli firmy Stack Exchange, napisał: „Nie mogę przestać zastanawiać się nad tym, czy programowanie w parach nie jest niczym innym jak wzmocnioną formą przeglądów kodu. Zaletą programowania w parach jest natychmiastowy efekt: nie można zignorować recenzenta, gdy siedzi obok ciebie”.

Dalej pisał: „Większość ludzi biernie rezygnuje z przeglądów kodu, jeśli ma taką możliwość. W przypadku programowania w parach jest to niemożliwe. Każdy członek pary musi rozumieć kod — natychmiast, gdy jest pisany. Praca parami może być

inwazyjna, ale również wymusza komunikację na poziomie, który w innym wypadku byłby niemożliwy do osiągnięcia”.

Dr Laurie Williams w 2001 roku przeprowadziła badania, z których wynika, że „programiści pracujący w parach są o 15% wolniejsi od dwóch niezależnych programistów, ale współczynnik błędnego kodu tworzonego przez nich wzrasta z 70% do 85%. Ponieważ testowanie i debugowanie często są wielokrotnie droższe niż początkowe programowanie, to trzeba uznać ten wynik za imponujący. Pary zazwyczaj rozwijały więcej możliwości projektowych niż programiści pracujący samodzielnie. W rezultacie powstają projekty prostsze i łatwiejsze w utrzymaniu. Ponadto podczas pracy parami wcześniej są wykrywane błędy”. Dr Williams poinformowała również, że 96% jej respondentów twierdziło, że odczuwa większą satysfakcję z pracy w przypadku, gdy programuje parami, w porównaniu z sytuacją, gdy programują osobno*.

Programowanie w parach przynosi dodatkowe korzyści w postaci rozpowszechniania wiedzy w organizacji oraz zwiększenia przepływu informacji wewnątrz zespołu. Powierzenie bardziej doświadczonym inżynierom zadania przeglądania kodu, a mniej doświadczonym kodowania to również skuteczny sposób nauczania i bycia uczonym.

Studium przypadku

Programowanie w parach zamiast niewłaściwych przeglądów kodu w firmie Pivotal Labs (2011)

Elisabeth Hendrickson, wiceprezes firmy Engineering Pivotal Software, Inc. oraz autorka książki *Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing*, głosiła pogląd popierający powierzenie zespołom odpowiedzialności za jakość własnego kodu w odróżnieniu od pozostawiania tej odpowiedzialności innym działom. Twierdziła, że w ten sposób nie tylko poprawia się jakość, ale również znacznie zwiększa się przepływ pracy do produkcji.

W swojej prezentacji na konferencji 2015 DevOps Enterprise Summit opisywała stan z 2011 roku, kiedy to w firmie Pivotal akceptowano dwie metody przeglądów kodu: programowanie w parach (w wyniku czego zyskiwano pewność przeanalizowania każdego wiersza kodu przez dwie osoby) lub przeglądanie kodu zarządzane za pomocą systemu Gerrit (w efekcie dla każdej operacji

* Niektóre organizacje wymagają programowania parami, podczas gdy w innych inżynierowie decydują się na ten rodzaj programowania podczas pracy w obszarach wymagających więcej analizy (na przykład przed zaewidencjonowaniem kodu) lub podczas realizacji trudnych zadań. Inną powszechną praktyką jest wyznaczenie kilku godzin w ciągu dnia dla programowania parami — na przykład cztery godziny od przerwy śniadaniowej do popołudnia.

ewidencjonowania kodu były wyznaczone dwie osoby — przegląd „+ 1” — zanim zmiany mogły zostać zarejestrowane w gałęzi master).

W przypadku przeglądów kodu Hendrickson zaobserwowała problem: otóż często zdarzało się, że na przegląd kodu programiści byli zmuszeni czekać nawet tydzień. Co gorsza, utalentowani programiści doznawali „frustrującego doświadczenia, polegającego na braku możliwości wprowadzenia prostych zmian do bazy kodu ze względu na stworzone sztucznie wąskie gardła”.

Hendrickson ubolewała nad tym, że „jedynymi osobami zdolnymi do wprowadzania zmian »+ 1« byli starsi inżynierowie, którzy mieli wiele innych obowiązków i często niezbyt przejmowali się poprawkami wprowadzanymi przez młodszych programistów czy też ich produktywnością. Doprowadzało to do bardzo niekorzystnej sytuacji — podczas oczekiwania na przegląd zaproponowanych zmian inni programiści ewidencjonowali nowe zmiany w repozytorium kodu źródłowego. W związku z tym przez tydzień oczekiwania trzeba było scalać zmiany na swoim laptopie, ponownie uruchomić wszystkie testy, aby mieć pewność, że wszystko nadal działa, a (czasami) ponownie przesyłać kod do przeglądu”.

Aby rozwiązać ten problem i wyeliminować niepotrzebne opóźnienia, skończyło się na wyeliminowaniu procesu przeglądów kodu za pomocą narzędzia Gerrit. Zamiast tego wprowadzono obowiązek programowania w parach w celu wprowadzania zmian w kodzie. W ten sposób skrócono czas potrzebny do zrealizowania przeglądu kodu z tygodni do godzin.

Hendrickson zauważała, że proces przeglądów kodu dobrze się sprawdza w wielu organizacjach, ale wymaga kultury traktowania procesu przeglądania kodu jak jego pisania. Jeśli nie ma takiej kultury, to programowanie w parach może odgrywać rolę cennej praktyki tymczasowej.

OCENA EFEKTYWNOŚCI PROCESÓW ŻĄDAŃ ŚCIĄGNIĘCIA

Ponieważ proces przeglądu kodu przez współpracowników jest ważną częścią środowiska zarządzania, potrzebny jest mechanizm, który pozwoli zweryfikować, czy jest on skuteczny, czy nie. Jedna z metod polega na obserwacji przestojów produkcyjnych i analizowaniu procesu przeglądu dla wszelkich istotnych zmian.

Inną metodę zaproponował Ryan Tomayko, dyrektor ds. informatyki i współzałożyciel firmy GitHub oraz jeden ze współtwórców procesu żądania ściągnięcia. Poproszony o podanie różnicy pomiędzy złym żądaniem ściągnięcia a dobrym żądaniem ściągnięcia odpowiedział, że ma to niewiele wspólnego z wynikami produkcji. Złe żądanie to takie, które nie ma wystarczającego kontekstu dla czytelnika, ma niewiele

lub nie ma żadnej dokumentacji bądź opisu celu zmiany. Na przykład żądanie ściągnięcia, do którego dołączono jedynie następujący tekst: „Usuwa problem #3616 i #3841”*.

W ten sposób opisano rzeczywiste, wewnętrzne żądanie ściągnięcia w firmie GitHub, które Tomayko skrytykował: „Prawdopodobnie ten opis stworzył jakiś nowy inżynier. Przede wszystkim nie wymieniono żadnego konkretnego inżyniera. Minimalne wymaganie to podanie mentora lub eksperta w obszarze, w którym wprowadzamy zmianę, by zyskać pewność, że odpowiednia osoba dokona przeglądu wprowadzanych zmian. Co gorsza, nie ma żadnego wyjaśnienia, czego dotyczyły wprowadzane zmiany, dlaczego są ważne ani też jaki był sposób myślenia osoby, która zmiany wprowadziła”.

Z drugiej strony, poproszony o opisanie dobrych żądań ściągnięcia wskazujących na skuteczny proces przeglądu, Tomayko szybko wymienił niezbędne elementy: muszą zawierać wystarczająco dużo szczegółów na temat powodów, dla których wprowadzono zmianę, a także wszelkie zidentyfikowane zagrożenia oraz zastosowane środki zaradcze.

Tomayko podkreśla również zalety dobrej dyskusji o zmianie możliwej dzięki kontekstowi dostarczonemu przez żądanie ściągnięcia — często wskazywane są dodatkowe zagrożenia, pomysły na lepszy sposób implementacji żądanej zmiany, pomysły na sposoby złagodzenia zagrożeń itd. A jeśli po wdrożeniu zdarzy się coś złego lub nieoczekiwanej, to do żądania ściągnięcia dodawana jest odpowiednia notatka wraz z łączem do podanego problemu. Dyskusje są prowadzone bez podawania innych. Zamiast tego jest szczera rozmowa na temat zapobiegania podobnym problemom w przyszłości.

Jako przykład Tomayko podał inne wewnętrzne żądanie ściągnięcia w GitHub dla migracji bazy danych. Miał wiele stron, zawierało obszerne dyskusje na temat potencjalnych zagrożeń, prowadzące do następującej instrukcji wprowadzonej przez autora żądania: „Przesyłam zmianę. Kompilacje dla tej gałęzi wykazują błąd ze względu na brakującą kolumnę w serwerach CI (łącze do raportu Post-Mortem: przestój MySQL)”.

Następnie autor zmiany przeprosił za przestój, opisując, jakie warunki i błędne założenia doprowadziły do incydentu, a także wymienił listę proponowanych środków zaradczych zapobiegających powtarzaniu się sytuacji w przyszłości. Dalej było wiele stron dyskusji. Czytając to żądanie ściągnięcia, Tomayko uśmiechnął się i powiedział: „**To** jest świetne żądanie ściągnięcia”.

Zgodnie z powyższym opisem możemy ocenić skuteczność procesu przeglądu kodu poprzez analizę próbek żądań ściągnięcia z całej populacji albo tylko tych, które są istotne dla wskazanego incydentu w produkcji.

* Gene Kim dziękuje Shawnowi Davenportowi, Jamesowi Frymanowi, Willowi Farroowi i Ryanowi Tomaykwi z firmy GitHub za omówienie różnic pomiędzy dobrymi a złymi żądaniami ściągnięcia.

ELIMINOWANIE PROCESÓW BIUROKRATYCZNYCH

Do tej pory omawialiśmy procesy przeglądów i programowania w parach, które pozwalają nam podnieść jakość wykonywanej pracy bez korzystania z zatwierdzania zmian przez podmioty zewnętrzne. Jednak w wielu firmach nadal obowiązują długotrwałe procesy zatwierdzania. Przebrnięcie przez nie często wymaga wielu miesięcy. Takie procesy zatwierdzania mogą znacznie zwiększyć czasy realizacji i nie tylko przeszkodzić w szybkim dostarczeniu wartości dla klientów, ale również zwiększyć zagrożenia dla wyznaczonych celów organizacyjnych. Gdy tak się stanie, należy zmodyfikować projekt procesów w taki sposób, by osiąganie celów mogło być szybsze i bezpieczniejsze.

Jak zaobserwował Adrian Cockcroft: „Doskonałym parametrem publikacji jest liczba spotkań i zleceń roboczych wymaganych do realizacji publikacji — należy dążyć do ciągłego zmniejszania wysiłku potrzebnego do wykonania pracy i dostarczenia jej do klienta”.

W podobnym tonie wypowiedział się dr Tapabrata Pal, posiadacz tytułu technical fellow w firmie Capital One, opisując program prowadzony w firmie Capital One o nazwie Got Goo?, który obejmował powołanie dedykowanego zespołu do usuwania przeskódek — w tym narzędzi, procesów i zatwierdzeń — które utrudniały pracę. Jason Cox, główny dyrektor inżynierii systemów w firmie Disney, w swojej prezentacji na szczytce DevOps Enterprise Summit w 2015 roku opisał program o nazwie Join The Rebellion (dosł. „dołącz do buntu”), który miał na celu usuwanie przeskódek i niepotrzebnego wysiłku z codziennej pracy.

W firmie Target w 2012 połączenie procesów TEAP (Technology Enterprise Adoption Process) oraz LARB (Lead Architecture Review Board) doprowadziło do powstania skomplikowanego, długotrwałego procesu zatwierdzania dla wszystkich, którzy dążyli do wprowadzenia nowej technologii. Każdy, kto chciał, aby była zastosowana nowa technologia (np. nowa baza danych lub technologia monitorowania), musiał wypełnić formularz TEAP. Złożone wnioski były poddawane ocenie, a te, które zostały uznane za odpowiednie, były wprowadzane na agendę comiesięcznego spotkania LARB.

Heather Mickman i Ross Clanton, odpowiednio dyrektor ds. rozwoju i dyrektor operacyjny w firmie Target, Inc., pomagali w przewodzeniu zmianom DevOps w firmie Target. W czasie realizacji projektu DevOps Mickman znalazła technologię niezbędną do zastosowania inicjatywy (w tym przypadku chodziło o systemy Tomcat i Cassandra). Na spotkaniu LARB podjęto decyzję, że dział operacyjny nie może udzielić wsparcia we wdrożeniu tych technologii. Ponieważ jednak Mickman była przekonana co do potrzeby zastosowania wymienionych technologii, zaproponowała, że zamiastdziału operacyjnego za wsparcie serwisowe, a także za integrację, dostępność i bezpieczeństwo będzie odpowiedzialny dział rozwoju, którym kierowała.

„W czasie realizacji procesu chciałam lepiej zrozumieć, dlaczego proces TEAP-LARB trwał tak długo. W tym celu zastosowałam technikę »pięciu pytań dlaczego«... co ostatecznie doprowadziło do zadania pytania o sens istnienia procesu TEAP-LARB. Zaskakujące było to, że nikt dokładnie nie wiedział, że potrzebowaliśmy jakiegoś rodzaju procesu zarządzania. Wiele osób wiedziało, że kilka lat wcześniej była w firmie jakaś katastrofa, która nigdy nie miała się powtórzyć, ale nikt nie pamiętał dokładnie, czego ta katastrofa dotyczyła” — zaobserwowała Mickman.

Mickman doszła do wniosku, że ten proces nie będzie potrzebny jej grupie w przypadku powierzenia obowiązków operacyjnych dotyczących wprowadzanych technologii. Dodała: „Chciałabym wszystkich poinformować, że w przyszłości wszystkie technologie, które będą wdrażane, nie będą musiały przechodzić przez proces TEAP-LARB”.

W efekcie system Cassandra został pomyślnie wdrożony w firmie Target i ostatecznie powszechnie przyjęty. Ponadto proces TEAP-LARB wycofano. W uznaniu dla zasług w usuwaniu barier dla technologii w firmie Target zespół przyznał Mickman nagrodę Lifetime Achievement Award.

PODSUMOWANIE

W tym rozdziale omówiliśmy sposób integracji z codzienną pracą praktyk, które zwiększają jakość wprowadzanych zmian i zmniejszają zagrożenia związane ze złymi wdrożeniami oraz eliminują zbytnią zależność od procesów zatwierdzania. Studia przypadków z firm GitHub i Target pokazują, że stosowanie tych praktyk nie tylko poprawia wyniki, ale również znacznie skraca czasy realizacji i zwiększa wydajność programistów. Wykonywanie tego rodzaju pracy wymaga kultury wysokiego zaufania.

Weźmy pod uwagę historię, którą John Allspaw opowiadał o nowo zatrudnionym młodszym inżynierze: inżynier zapytał, czy może wdrożyć niewielkie zmiany w HTML, a Allspaw odpowiedział: „Nie wiem, czy to są niewielkie zmiany”. Następnie zapytał: „Czy poprosiłeś kogoś o przejrzenie zmian? Czy wiesz, kto jest najlepszą osobą, do której należy się zwrócić w przypadku zmian tego typu? Czy zrobiłeś absolutnie wszystko, co można zrobić, aby zyskać pewność, że ta zmiana działa w produkcji zgodnie z projektem? Jeśli tak zrobiłeś, to mnie nie pytaj — po prostu wprowadź tę zmianę!”.

Odpowiadając w ten sposób, Allspaw przypomniał inżynierowi, że to on ponosi wyjątkową odpowiedzialność za jakość zmian — jeśli zrobił wszystko, co w swoim odczuciu powinien zrobić, aby uzyskać pewność, że zmiana będzie działać, to nie ma potrzeby, aby prosił kogokolwiek o zgodę. Powinien po prostu wprowadzić zmianę.

Tworzenie warunków umożliwiających osobom wprowadzającym zmiany przyjęcie pełnej odpowiedzialności za jakość tych zmian jest kluczową częścią generatywnej kultury wysokiego zaufania, którą staramy się budować. Ponadto warunki te pozwalają nam tworzyć coraz bezpieczniejszy system pracy, gdzie wszyscy wzajemnie sobie

pomagają osiągnąć wyznaczone cele, niezależnie od granic, które trzeba pokonać, żeby te cele osiągnąć.

WNIOSKI DO CZĘŚCI IV

W części IV pokazaliśmy, że zastosowanie pętli sprzężeń zwrotnych umożliwia wszystkim wspólną pracę w kierunku wspólnych celów. Dzięki temu można zauważać problemy natychmiast po ich wystąpieniu oraz szybko je rozwiązać. W ten sposób nie tylko gwarantujemy, że funkcjonalności działają w produkcji zgodnie z projektem, ale także możemy osiągać cele organizacji i wspierać zdobywanie wiedzy. Pokazaliśmy również, w jaki sposób wyznaczyć wspólne cele, obejmujące działy Dev i Ops, tak aby poprawić kondycję całego strumienia wartości.

Teraz jesteśmy gotowi, by przejść do części V, „Trzecia droga. Techniczne praktyki ciągłego uczenia się i eksperymentowania”. Pokażemy w niej sposób tworzenia możliwości uczenia się, które występują wcześniej, szybciej i taniej. Dzięki nim zdołamy uwolnić kulturę innowacji i eksperymentowania, pozwalającą wszystkim wykonywać sensowną pracę oraz przyczynić się do osiągnięcia sukcesu przez organizację.

Część V

Trzecia droga

*Techniczne praktyki ciągłego
uczenia się i eksperymentowania*

Część V *Wprowadzenie*

W części III, „Pierwsza droga. Techniczne praktyki przepływu”, omówiliśmy sposób wdrożenia praktyk potrzebnych do umożliwienia szybkiego przepływu w naszym strumieniu wartości. W części IV, „Druga droga. Techniczne praktyki sprzężeń zwrotnych”, naszym celem było stworzenie jak najwięcej sprzężeń zwrotnych z jak największej liczby obszarów w systemie — wcześniej, szybciej i taniej.

W części V, „Trzecia droga. Techniczne praktyki ciągłego uczenia się i eksperymentowania”, prezentujemy praktyki stwarzające możliwości uczenia się — tak szybko, wcześniej, często i tanio, jak to możliwe. Obejmuje to praktyki uczenia się na podstawie wypadków i awarii, które są nieuniknione, gdy pracujemy w złożonych systemach, jak również organizowania i projektowania systemów pracy w taki sposób, aby stale eksperymentować i uczyć się oraz ciągle dążyć do tego, by system stawał się coraz bezpieczniejszy. W efekcie zyskujemy większą odporność i ciągle rosnącą zbiorową wiedzę na temat rzeczywistego sposobu pracy, dzięki czemu możemy łatwiej osiągać wyznaczone cele.

W kolejnych rozdziałach zaprezentujemy rytuały, które zwiększą bezpieczeństwo, ciągłe doskonalenie i naukę za pośrednictwem następujących mechanizmów:

- ustanowienie kultury uczciwości w celu stworzenia warunków dla bezpiecznej pracy;
- wstrzykiwanie awarii produkcyjnych w celu uzyskania odporności;
- konwersja lokalnych odkryć na globalne usprawnienia;
- zarezerwowanie czasu na stworzenie organizacyjnych mechanizmów usprawnień i uczenia się.

Ponadto stworzymy mechanizmy, dzięki którym nowa wiedza zdobyta w jednym obszarze organizacji będzie mogła być błyskawicznie wykorzystana w całej organizacji, co przyczyni się do przekształcenia lokalnych usprawnień w globalne innowacje. W ten sposób nie tylko będziemy uczyć się szybciej niż konkurencja, co pomoże nam osiągnąć przewagę konkurencyjną na rynku, ale także stworzymy kulturę bezpieczniejszej, bardziej elastycznej pracy, z której wykonywania pracownicy będą dumni, co pozwoli im osiągnąć największy potencjał.

*Stworzenie warunków do uczenia się
podczas codziennej pracy*

Pracując w złożonym systemie, nie jesteśmy w stanie przewidzieć wszystkich efektów działań, które podejmujemy. Z tego powodu powstają nieoczekiwane sytuacje, których skutki czasami są katastrofalne. Występują one pomimo stosowania statycznych środków ostrożności takich, jak listy kontrolne i instrukcje działania kodyfikujące nasze obecne rozumienie systemu.

Aby umożliwić nam bezpieczną pracę w złożonych systemach, organizacje muszą poprawić swoje możliwości w zakresie samodiagnostyki i samodoskonalenia i muszą posiadać umiejętności wykrywania problemów, rozwiązywania ich oraz wzmacniania uzyskiwanych efektów dzięki udostępnieniu rozwiązań w całej organizacji. W ten sposób tworzy się dynamiczny system uczenia się, który pozwala zrozumieć popełniane błędy i przekształcić to zrozumienie na działania, które zapobiegają powtórzeniu podobnych błędów w przyszłości.

W efekcie powstaje coś, co dr Steven Spear opisuje jako elastyczne organizacje, które są „zdolne do wykrywania problemów, rozwiązywania ich oraz wzmacnienia efektów poprzez udostępnienie rozwiązań w całej organizacji”. Takie organizacje mogą się same leczyć. „Dla takich organizacji reagowanie na sytuacje kryzysowe nie jest pracą idiosynkratyczną. Jest to działanie, które jest wykonywane przez cały czas. To jest ten rodzaj responsywności, która jest źródłem niezawodności”.

Uderzający przykład niesamowitej odporności, którą można uzyskać w wyniku stosowania tych zasad i praktyk, można było zaobserwować 21 kwietnia 2011 roku,

gdy awarii uległa cała strefa dostępności Amazon AWS US-EAST, uniemożliwiając dostęp do usług wszystkim klientom, którzy na niej polegali, włącznie z serwisami Reddit i Quora*. Jednak usługa Netflix była zaskakującym wyjątkiem. Masowa awaria usługi AWS wydawała się nie wywierać na nią wpływu.

Po tym wydarzeniu pojawiło się wiele spekulacji na temat tego, w jaki sposób udało się firmie Netflix zachować ciągłość działania swoich usług. Popularna teoria głosiła, że ze względu na to, że Netflix był jednym z największych klientów Amazon Web Services, zastosowano dla nich jakieś specjalne środki, co pozwoliło im zachować ciągłość działania. Jednak w poście na blogu *Netflix Engineering* wyjaśniono, że tę wyjątkową odporność Netflix zawdzięcza decyzjom projektowym w zakresie architektury podjętym w 2009 roku.

W 2008 roku usługa dostarczania wideo Netflix działała w oparciu o monolityczną aplikację J2EE, hostowaną w jednym z centrów danych. Jednak od 2009 roku zaczęto modyfikować projekt architektury tego systemu w kierunku tzw. **natywnej chmury** — zaprojektowano ją do działania całkowicie w chmurze publicznej Amazon oraz tak, aby była wystarczająco elastyczna do tego, by przetrwać znaczące błędy.

Jednym z konkretnych celów projektowych było zagwarantowanie działania usług Netflix nawet wtedy, gdy przestała działać cała strefa dostępności AWS, tak jak stało się w przypadku strefy US-EAST. Aby cel mógł być osiągnięty, konieczne były luźne sprzężenia systemu. Dla każdego komponentu określono agresywne timeoutry, tak by awarie komponentów nie spowodowały awarii całego systemu. Każda funkcjonalność i komponent zostały zaprojektowane tak, by zapewniały degradację „z wdziękiem”. Na przykład podczas okresów największego ruchu, kiedy obserwowało gwałtowny wzrost użycia procesora zamiast listy filmów spersonalizowanych dla użytkownika prezentowano statyczną zawartość. Były to wyniki zbuforowane albo pozbawione personalizacji, dzięki czemu wymagały mniej obliczeń.

Dalej w tym samym poście wyjaśniano, że oprócz zastosowania tych wzorców architektonicznych dodatkowo zbudowano i uruchomiono zaskakującą i zuchwałą usługę o nazwie *Chaos Monkey*, która symulowała awarie AWS poprzez stałe i losowe zabijanie serwerów produkcyjnych. Zrobiono to ze względu na to, aby „zespoły inżynierskie przyzwyczaiły się do stałego poziomu awarii w chmurze”, tak aby usługi mogły być „automatycznie przywracane bez ręcznej interwencji”.

Innymi słowy, zespół Netflix uruchomił usługę Chaos Monkey w celu osiągnięcia operacyjnej odporności — poprzez stałe wstrzykiwanie błędów do swoich środowisk przedprodukcyjnych i produkcyjnych.

* W styczniu 2013 r. na konferencji re:Invent James Hamilton, wiceprezes i posiadacz tytułu Distinguished Engineer usługi Amazon Web Services, powiedział, że w samym regionie US East było ponad 10 centrów danych, oraz dodał, że typowe centrum danych zawiera od 50 do 80 tysięcy serwerów. Z tych obliczeń wynika, że awaria EC2 z 2011 roku miała wpływ na klientów korzystających z ponad pół miliona serwerów.

Jak można oczekwać, gdy po raz pierwszy wprowadzono usługę Chaos Monkey do środowisk produkcyjnych, usługi uległy awariom w sytuacjach, których nikt nie mógł przewidzieć lub sobie wyobrazić. Dzięki stałemu poszukiwaniu i naprawianiu tego rodzaju problemów podczas normalnych godzin pracy inżynierowie Netflix szybko i iteracyjnie stworzyli bardziej odporną usługę, a jednocześnie uzyskali nową wiedzę organizacyjną (podczas normalnych godzin pracy!), która umożliwiała im rozwiązywanie systemów i znaczne wyprzedzenie konkurencji.

Chaos Monkey to tylko jeden przykład integracji procesu uczenia się w codziennej pracy. Historia ta pokazuje również, w jaki sposób organizacje uczące się myślą o awariach, incydentach i błędach — powinny być one postrzegane jako okazja do zdobycia nowej wiedzy, a nie powód do tego, by kogoś ukarać. W tym rozdziale pokazano, jak stworzyć system uczenia się w organizacji oraz jak ustanowić **kulturę sprawiedliwości** (ang. *just culture*), a także jak przeprowadzać rutynowe ćwiczenia i celowo prowokować błędy, aby przyspieszyć zdobywanie wiedzy.

USTANOWIENIE KULTURY SPRAWIEDLIWOŚCI I UCZENIA SIĘ

Jednym z warunków wstępnych dla kultury uczenia się jest zadbanie o to, aby w przypadku incydentów (które niewątpliwie się zdarzą) reakcja na nie była po prostu „sprawiedliwa”. Dr Sidney Dekker, który pomógł w kodyfikacji niektórych kluczowych elementów kultury bezpieczeństwa i wymyślił termin „kultura sprawiedliwości”, pisał: „Gdy reakcje na incydenty i wypadki są postrzegane jako niesprawiedliwe, to prowadzenie dochodzeń dotyczących bezpieczeństwa staje się trudniejsze. Wśród ludzi wykonujących prace krytyczne dla bezpieczeństwa zaczyna narastać strach przed odpowiedzialnością zamiast ostrożności. W ten sposób organizacje stają się bardziej burokratyczne, a nie ostrożniejsze. Profesjonalisci dbają o utrzymywanie własnych tajemnic zawodowych, uchylają się od odpowiedzialności oraz starają się zapewnić ochronę dla samych siebie”.

Pojęcie kary jest obecne — subtelnie lub jawnie — poprzez sposób, w jaki wielu menedżerów działało w ubiegłym stuleciu. Powszechnie stosowano następujący sposób myślenia: aby osiągnąć cele organizacji, liderzy muszą zarządzać, nadzorować, ustanawiać procedury w celu wyeliminowania błędów i egzekwować zgodność z tymi procedurami.

Dr Dekker praktykę eliminowania błędów poprzez eliminowanie osób, które spowodowały błędy, określa jako teorię **zgniłego jabłka**. Twierdzi, że jest ona niewłaściwa, ponieważ „to nie ludzkie błędy są przyczyną kłopotów; są one raczej konsekwencją projektu narzędzi, które im daliśmy”.

Skoro incydenty nie są spowodowane przez „zgniłe jabłka”, ale są raczej konsekwencją nieuniknionych problemów projektowych w złożonych systemach, które tworzymy, to zamiast stosowania praktyki „nazwij, obwiń i zawstydz” w stosunku

do osoby, która spowodowała błąd, należy raczej dążyć do zmaksymalizowania możliwości nauki w organizacji oraz stale wzmacniać przekonanie, że wartościowe są działania, w wyniku których ujawniamy wiedzę o problemach i dzielimy się nią w codziennej pracy. Takie postępowanie pozwala poprawić jakość i bezpieczeństwo systemu, w którym działamy, oraz wzmacnić relacje pomiędzy osobami, które działają w jego ramach.

Dzięki przekształceniu informacji w wiedzę i wbudowaniu wyników uczenia się do naszych systemów zaczynamy osiągać cele kultury sprawiedliwości — równoważyć potrzeby bezpieczeństwa i odpowiedzialności. Jak powiedział John Allspaw, dyrektor ds. technicznych w firmie Etsy: „Naszym celem w Etsy jest postrzeganie pomyłek, błędów, niedociągnięć, braków itp. z perspektywy możliwej nauki”.

Gdy inżynierowie popełniają błędy i czują się bezpiecznie, przekazując szczegółowe dotyczące tych błędów, to nie tylko wyrażają zgodę na przyjęcie odpowiedzialności, ale także są nastawieni entuzjastycznie do udzielenia pomocy pozostałym osobom w firmie, by mogły uniknąć popełnienia tych samych błędów w przyszłości. Takie działanie tworzy system uczenia się organizacji. Z drugiej strony, jeśli ukarzemy inżyniera, który popełni błąd, to wszyscy będą niechętnie udzielali informacji o szczegółach niezbędnych do uzyskania zrozumienia mechanizmu błędu, jego patologii i sposobu działania. W takiej sytuacji z całą pewnością błąd powtórzy się w przyszłości.

Dwie skuteczne praktyki, które pomagają tworzyć kulturę sprawiedliwości i uczenia się, to „analizy post-mortem” bez podawania winnych oraz kontrolowane wprowadzanie awarii do produkcji w celu stworzenia możliwości do ćwiczenia zachowań w sytuacjach awaryjnych — nieuniknionych w złożonych systemach. Najpierw przyjrzymy się „analizom post-mortem”, a następnie wyjaśnimy, dlaczego awarie mogą być dobre.

PLANOWANIE ANALIZ POST-MORTEM PO WYSTĄPIENIU AWARII

Aby ułatwić stworzenie kultury sprawiedliwości po wystąpieniu incydentów (np. nieudane wdrożenie, problem w produkcji, który wywarł wpływ na klientów), po naprawieniu awarii należy przeprowadzić **analizę post-mortem bez poszukiwania winnych**^{*}. Analizy post-mortem bez szukania winnych — termin wymyślony przez Johna Allspawa — mają pomóc przeanalizować „błędy w sposób, który koncentruje się na sytuacyjnym aspekcie mechanizmu awarii i procesu decyzyjnego osób w czasie bliskim awarii”.

Aby to zrobić, można zaplanować analizę post-mortem możliwie najszybciej po wystąpieniu awarii — zanim osłabią się wspomnienia i powiązania przyczynowo-

* Praktyka ta jest określana również jako przeglądy powypadkowe (ang. *post-incident review*) lub retrospektyny pozdarzeniowe (ang. *post-event retrospectives*). Warto zauważyć podobieństwo do rutynowych retrospektyw, które są częścią wielu metodologii iteracyjnych oraz technik zinnego wytwarzania oprogramowania.

-skutkowe lub zmienią się okoliczności (oczywiście należy poczekać do rozwiązania problemu, tak aby nie rozpraszać uwagi osób, które nadal aktywnie działają w tej sprawie).

W spotkaniu dotyczącym analizy post-mortem bez określania winnych wykonuje się następujące czynności:

- skonstruowanie osi czasu i zebranie danych dotyczących awarii z wielu perspektyw z dbałością o to, aby nie karać osób, które popełniły pomyłki;
- zobowiązanie wszystkich inżynierów do poprawienia bezpieczeństwa poprzez przekazanie szczegółowych sprawozdań dotyczących ich udziału w awarii;
- zachęcenie osób, które popełniły błędy, do zostania ekspertami, którzy będą edukować pozostałe osoby w organizacji w zakresie niedopuszczenia do popełnienia podobnych pomyłek w przyszłości;
- zaakceptowanie faktu, że wszędzie tam, gdzie ludzie decydują o tym, czy podjąć działania, czy nie, zawsze jest miejsce na interpretację, a ocena zasadności podjętej decyzji jest możliwa tylko z perspektywy czasu;
- zaproponowanie środków zaradczych w celu zapobieżenia podobnym incydentom w przyszłości i zadbanie o to, aby — w celu obserwacji — środki zaradcze zostały zarejestrowane wraz z datą wdrożenia i osobą odpowiedzialną.

Aby możliwe było zyskanie właściwego zrozumienia problemu, na spotkaniu powinni być obecni następujący interesariusze:

- osoby biorące udział w podejmowaniu decyzji, które mogły przyczynić się do powstania problemu;
- osoby, które zidentyfikowały problem;
- osoby, które zareagowały na problem;
- osoby, które zdiagnozowały problem;
- osoby, które zostały dotknięte skutkami problemu;
- wszystkie inne osoby zainteresowane uczestnictwem w spotkaniu.

Pierwszym zadaniem podczas spotkania poświęconego analizie post-mortem jest zarejestrowanie najlepszego zrozumienia osi czasu istotnych wydarzeń w miarę ich występowania. Dotyczy to rejestracji wszystkich podjętych działań wraz z czasem, kiedy zostały podjęte (najlepiej wraz z logami rozmów — np. przez IRC lub Slack), obserwowanymi skutkami (najlepiej w formie konkretnych wskaźników telemetrycznych w przeciwieństwie do samej subiektywnej narracji), wszystkimi ścieżkami podjętych badań oraz przyjętymi rozwiązaniami.

Aby umożliwić korzystanie z tych wyników, należy rygorystycznie podchodzić do wszystkich szczegółów rejestracji i wzmacniać kulturę możliwości współdzielenia

informacji bez obaw o kary lub odwet. Z tego powodu, zwłaszcza w przypadku kilku pierwszych analiz post-mortem, pomocne może okazać się prowadzenie spotkania przez wyszkolonego moderatora, który nie był zaangażowany w incydent.

Podczas spotkania i późniejszego stosowania środków zaradczych należy jawnie zabronić stosowania zwrotów „trzeba było” czy też „można było”, ponieważ są to hipotezy przeciwnie, będące wynikiem ludzkiej skłonności do tworzenia możliwych alternatyw zdarzeń, które już nastąpiły.

Hipotezy przeciwnie w rodzaju „mógłbym...” lub „gdybym to wiedział, to...” umiejscawiają problem w kategoriach **systemu zgodnego z wyobrażeniami** zamiast **systemu, który faktycznie istnieje**. Należy ograniczyć się do rozpatrywania sytuacji w kontekście wyłącznie tego drugiego (dodatek 8.).

Jednym z potencjalnie zaskakujących rezultatów spotkań post-mortem jest to, że ludzie często obwiniają siebie za rzeczy poza ich kontrolą lub kwestionują własne umiejętności. Ian Malpass, inżynier w firmie Etsy, zauważał: „W chwili, gdy robimy coś, co powoduje, że cała witryna przestaje działać, odczuwamy efekt »lodowatej wody spływającej wzdułż kręgosłupa« i prawdopodobnie pierwszą myślą, jaką przechodzi przez głowę, jest: »Zawaliłem i nie mam pojęcia, co robić«. Trzeba temu przeciwdziałać, ponieważ jest to droga do szaleństwa, rozpaczki i poczucia bycia oszustem. Do tego nie można dopuścić, jeśli chce się być dobrym inżynierem. Lepszym pytaniem, na którym warto się skupić, jest: »Dlaczego podjęcie tego działania miało dla mnie sens?«”.

Na spotkaniu należy zarezerwować sobie wystarczająco dużo czasu na burzę mózgów i podjęcie decyzji co do tego, jakie środki zaradcze należy wdrożyć. Po zidentyfikowaniu środków zaradczych, muszą one być traktowane priorytetowo. Powinien być do nich przypisany właściciel oraz oś czasu na implementację. Postępowanie w taki sposób pokazuje, że większą wartość przypisujemy usprawnieniom niż codziennej pracy samej w sobie.

Dan Milstein, jeden z głównych inżynierów w firmie Hubspot, napisał, że wszystkie spotkania post-mortem zaczyna od zdania: „Staramy się przygotować na przyszłość, w której będziemy tak samo głupi, jak jesteśmy dziś”. Inaczej mówiąc, niedopuszczalne jest stosowanie środka zaradczego w postaci „bądźmy bardziej uważni” lub „nie bądźmy tacy nierożważni” — zamiast tego trzeba zaprojektować rzeczywiste środki zaradcze, które uchronią nas przed powtórzeniem podobnych błędów w przyszłości.

Przykładem takich środków zaradczych są nowe automatyczne testy do wykrywania niebezpiecznych sytuacji występujących w potoku wdrożeń, wprowadzanie nowych produkcyjnych wskaźników telemetrycznych, zidentyfikowanie kategorii zmian wymagających dodatkowych przeglądów kodu oraz regularne wykonywanie ćwiczeń podobnej kategorii błędów.

OPUBLIKOWANIE WNIOSKÓW Z ANALIZ POST-MORTEM DLA JAK NAJSZERSZEGO GRONA ODBIORCÓW

Po przeprowadzeniu spotkania poświęconego analizie post-mortem, należy opublikować dla jak naajserszego grona odbiorców notatki ze spotkania oraz powiązane z nimi artefakty (np. osie czasu, logi czatów IRC, logi z komunikacji zewnętrznej). Informacje te powinny być (najlepiej) umieszczone w centralnej lokalizacji, do której mogą uzyskać dostęp osoby z całej organizacji, tak aby mogły czerpać naukę z incydentu. Prowadzenie analiz post-mortem jest na tyle ważne, że czasami warto wstrzymać decyzję o zamknięciu incydentu produkcyjnego do czasu zakończenia spotkania post-mortem.

Postępowanie w taki sposób pomaga nam nadać lokalnej wiedzy i usprawnieniom charakter globalny. Randy Shoup, były dyrektor inżynierii grupy Google App Engine, opisywał ogromną wartość dokumentacji spotkań post-mortem dla innych osób w organizacji: „Jak można sobie wyobrazić, w Google wszystko może być przeszukiwane. Wszystkie dokumenty post-mortem są rozmieszczone w lokalizacjach, gdzie mogą je zobaczyć inni pracownicy serwisu Google. Proszę mi wierzyć, że gdy w jakiejkolwiek grupie zdarzy się incydent, który wygląda podobnie do czegoś, co wydarzyło się wcześniej, w pierwszej kolejności są czytane i studiowane dokumenty post-mortem”*.

Publikowanie notatek ze spotkań post-mortem dla szerokiego grona odbiorców oraz zachęcanie innych osób w organizacji do ich czytania poprawia wiedzę organizacji. Coraz powszechniejszą praktyką firm prowadzących serwisy online jest publikowanie notatek ze spotkań post-mortem dotyczących awarii mających wpływ na klientów. Takie działanie często znacznie zwiększa przejrzystość działań firmy w stosunku do klientów wewnętrznych i zewnętrznych, co z kolei zwiększa ich zaufanie do nas.

Dążenie do prowadzenia jak największej liczby spotkań post-mortem w firmie Etsyc doprowadziło do pewnych problemów — w ciągu czterech lat zgromadzono dużą liczbę notatek ze spotkań post-mortem w formie stron wiki. Stały się one bardzo trudne do przeszukiwania, zapisywania i czerpania z nich informacji.

Aby rozwiązać ten problem, opracowano narzędzie o nazwie Morgue, którego zadaniem jest łatwe rejestrowanie wszystkich aspektów każdego incydentu (np. współczynnik MTTR dla awarii, istotność, strefy czasowe, które stały się istotne ze względu na powiększającą się grupę zdalnych pracowników firmy Etsyc) oraz uwzględnienie

* Można także rozszerzyć filozofię Transparent Uptime do raportów post-mortem i oprócz pulpitu nawigacyjnego usługi możemy również udostępniać publicznie (być może nieco okrojone) raporty ze spotkań post-mortem. Do powszechnie podziwianych publicznych raportów post-mortem należą raporty opublikowane przez zespół Google App Engine po poważnej awarii w 2010 roku, jak również analizy post-mortem po awarii Amazon DynamoDB w 2015 roku. Co ciekawe, firma Chef publikuje notatki ze swoich spotkań post-mortem na blogu. Są na nim również publikowane sprawozdania wideo z tych spotkań.

również innych danych takich, jak tekst sformatowany w formacie Markdown, osadzone zdjecia, tagi i historia.

Narzędzie Morgue zaprojektowano w celu ułatwienia zespołom rejestrowania następujących faktów:

- Czy przyczyną problemu było zdarzenie zaplanowane, czy niezaplanowane?
- Gospodarz spotkania post-mortem.
- Istotne zapisy czatów IRC (szczególnie ważne dla problemów zgłaszanych o 3 nad ranem, gdy mogą występować problemy ze sporządzaniem dokładnych notatek).
- Istotne zlecenia robocze JIRA dla działań naprawczych oraz terminy ich wykonania (te informacje są szczególnie ważne dla kierownictwa).
- Linki do postów na forach klientów (tam gdzie klienci narzekają na występujące problemy).

Po opracowaniu i zastosowaniu narzędzia Morgue liczba zarejestrowanych spotkań post-mortem w firmie Etsy znacznie wzrosła w porównaniu z sytuacją, gdy używano stron wiki. Zmiana była szczególnie wyraźna dla incydentów P2, P3 i P4 (czyli tych o niższym poziomie istotności). Ten efekt wzmacnił hipotezę, że w przypadku ułatwienia dokumentowania spotkań post-mortem za pomocą takich narzędzi jak Morgue więcej osób będzie rejestrować szczegółowe wyniki spotkań post-mortem, a dzięki temu poprawi się efekt uczenia się w organizacji.

Dr Amy C. Edmondson, profesor w dziedzinie kierowania i zarządzania w Harvard Business School oraz współautor książki *Building the Future: Big Teaming for Audacious Innovation*, napisał:

Remedium — które niekoniecznie pociąga za sobą wiele czasu i kosztów — polega na zmniejszeniu ryzyka niepowodzenia. Eli Lilly postępuje w taki sposób od początku lat 90. poprzez organizowanie »święta porażek« w celu uczczenia intelligentnych, wysokiej jakości doświadczeń naukowych, w których nie udało się osiągnąć pożądanych rezultatów. Przyjęcia nie kosztują dużo, a wcześniejsza zmiana przydziału cennych zasobów — szczególnie naukowców — do nowych projektów pozwala zaoszczędzić setki tysięcy dolarów, nie wspominając już o możliwości dokonania potencjalnych nowych odkryć.

ZMNIEJSZENIE PROGU TOLERANCJI W CELU WYSZUKIWANIA CORAZ SŁABSZYCH SYGNAŁÓW AWARII

Nieuchronną konsekwencją zdobywania doświadczenia przez organizacje w efektywnym dostrzeganiu i rozwiązywaniu problemów jest potrzeba zmniejszenia progu sygnału oznaczającego problem. Tylko wtedy można bowiem kontynuować zdobywanie wiedzy. W tym celu dążymy do wzmacniania słabych sygnałów błędów. Na

przykład zgodnie z opisem w rozdziale 4., kiedy w firmie Alcoa udało się zmniejszyć częstotliwość występowania wypadków w miejscu pracy, tak że przestały one być codziennością, Paul O'Neill, dyrektor ds. inżynierii w firmie Alcoa, zaczął być powiadomiany nie tylko o faktycznych wypadkach w miejscu pracy, ale także o sytuacjach, w których o mało nie doszło do wypadku.

Dr Spear, podsumowując dokonania O'Neilla w Alcoa, pisze: „Choć wszystko zaczęło się od koncentrowania się na problemach związanych z bezpieczeństwem w miejscu pracy, wkrótce okazało się, że problemy bezpieczeństwa odzwierciedlają ignorancję procesu i że ta ignorancja może również objawiać się innymi problemami, takimi jak jakość, punktualność i wydajność”.

Kiedy pracujemy w ramach złożonych systemów, to potrzeba wzmacniania słabych sygnałów jest kluczem do uniknięcia katastrofalnych awarii. Obrazowym przykładem może być sposób, w jaki NASA obsługiwała sygnały awarii podczas ery wahadłowców. W 2003 roku, w 16 dni po rozpoczęciu misji promu kosmicznego *Columbia*, nastąpił wybuch po ponownym wejściu wahadłowca w atmosferę ziemską. Teraz wiemy, że podczas startu kawałek pianki izolacyjnej uszkodził zewnętrzny zbiornik paliwa.

Jednak przed ponownym wejściem w atmosferę promu *Columbia* grupa inżynierów NASA średniego szczebla zgłaszała ten incydent, ale ich głosy zostały zignorowane. Zaobserwowali oni uderzenie pianki na monitorach video podczas sesji analizy po startie i natychmiast powiadomili menedżerów NASA, ale powiedziano im, że problem pianki nie był niczym nowym. Przesunięcia pianki uszkadzały wahadłowce w poprzednich startach, ale nigdy nie spowodowały wypadku. Uznano to za problem konserwacyjny i nie podjęto żadnych działań do czasu, aż było za późno.

Michael Roberto, Richard M.J. Bohmer i Amy C. Edmondson w 2006 roku opublikowali artykuł w *Harvard Business Review* opisujący sposób, w jaki do powstania problemu przyczyniły się zwyczaje panujące w NASA. Opisywali oni, że organizacje zazwyczaj mają dwa modele struktury: **model standardowy**, w którym wszystkim rządzą procedury i systemy (dotyczy to również ścisłego przestrzegania terminów i budżetów), i **model eksperymentalny**, w którym codziennie są analizowane i dyskutowane wszystkie nowe doświadczenia i informacje zgodnie ze zwyczajami podobnymi do tych, które obowiązują w laboratoriach badawczo-rozwojowych.

Zaobserwowali oni: „Firmy wpadają kłopoty w przypadku, gdy zastosują dla organizacji nieodpowiedni sposób myślenia (który decyduje o sposobie reagowania na lub, zgodnie z terminologią tej książki,)... Od 1970 roku w NASA stworzono kulturę sztywnej standaryzacji, a wahadłowce były reklamowane Kongresowi jako tani statek kosmiczny wielokrotnego użytku”. NASA preferowała ścisłą zgodność z przepisami zamiast modelu eksperymentalnego, gdzie wszystkie informacje muszą być poddane ocenie zaraz po ich wystąpieniu. Brak ciągłego uczenia się i eksperymentowania przyniósł tragiczne konsekwencje. Autorzy wyciągnęli wniosek, zgodnie z którym liczą się zwyczaje i sposób

myślenia, a nie tylko „zachowanie ostrożności” — jak pisali: „Sama czujność nie zapobiegnie przekształceniu się zagrożeń niejednoznacznych (awarii o słabych sygnałach) w awarie kosztowne (a czasami mające tragiczne konsekwencje)”.

Prace w strumieniu wartości technologii — takim jak wyprawy kosmiczne — powinny być uważane za eksperymentalne i w taki też sposób powinny być zarządzane. Wszelkie prace, które wykonujemy, są potencjalnie ważnymi hipotezami i źródłami danych, a nie rutynowym stosowaniem i weryfikowaniem praktyk wykorzystywanych wcześniej. Zamiast traktować pracę w strumieniu technologii jako całkowicie ustandaryzowaną, w sytuacji, w której dążymy do zachowania zgodności z procesami, powinniśmy stale dążyć do wyszukiwania coraz słabszych sygnałów awarii, tak aby lepiej zarządzać systemem, w którym działamy, i lepiej go rozumieć.

ZMIANA DEFINICJI NIEPOWODZENIA I ZACHĘCANIE DO PODEJMOWANIA SKALKULOWANEGO RYZYKA

Liderzy organizacji — celowo lub przypadkowo — poprzez swoje działania wzmacniają zwyczaje i wartości obowiązujące w organizacji. Audytorzy, eksperci rachunkowości i etyki od dawna zaobserwowali, że „sygnał z góry” pozwala przewidzieć prawdopodobieństwo oszustw oraz innych nieetycznych praktyk. W celu wzmacnienia kultury uczenia się i podejmowania skalkulowanego ryzyka należy nakłonić liderów do tego, aby nieustannie wzmacniali przekonanie, że wszyscy powinni mieć możliwość uczenia się na błędach i być odpowiedzialni za naukę wynikającą z błędów.

Roy Rapoport z Netflix zauważał: „Raport 2014 State of DevOps Report przekonał mnie, że wysokowydajne organizacje stosujące praktyki DevOps częściej popełniają błędy. Nie tylko to jest w porządku, ale to jest właśnie to, co jest organizacjom potrzebne! Można to wyraźnie zaobserwować w danych: jeśli wysokowydajne organizacje publikują 30 razy częściej, a współczynnik awarii jest o połowę niższy niż w firmach o niższej wydajności, to wyraźnie widać, że w tych pierwszych awarii jest więcej”.

Dalej pisał: „Rozmawiałem ze współpracownikiem o ogromnej skali awarii, którą właśnie mieliśmy w Netflix — szczerze mówiąc, była ona spowodowana głupim błędem. Wywołał ją inżynier, który w ciągu ostatnich 18 miesięcy dwukrotnie doprowadził do awarii na podobną skalę. Ale co oczywiste, jest to osoba, której nigdy nie zwolnimy. W ciągu tych samych 18 miesięcy dzięki temu inżynierowi uzyskaliśmy postęp w działaniach i automatyzacji nie o mile, ale o lata świetlne. Dzięki jego pracy zdolaliśmy bezpiecznie przeprowadzać wdrożenia z dnia na dzień. Osobiście inżynier ten zrealizował ogromną liczbę wdrożeń produkcyjnych”.

Na koniec stwierdził: „DevOps musi pozwalać na tego rodzaju innowacje i wynikające z nich ryzyko popełniania błędów. To prawda, że takie postępowanie spowoduje błędy w produkcji. Ale są one dobre i nie należy za nie karać”.

WSTRZYKIWANIE AWARII PRODUKCYJNYCH W CELU POPRAWY ODPORNOŚCI I UCZENIA SIĘ

Jak dowiedzieliśmy się z wprowadzenia do rozdziału, wstrzykiwanie awarii w środowisku produkcyjnym (za pomocą takich mechanizmów, jak Chaos Monkey) jest jednym ze sposobów na zwiększenie odporności. W tym podrozdziale opiszemy procesy związane z realizacją prób i wstrzykiwaniem błędów do systemu w celu potwierdzenia jego prawidłowego projektu i architektury, tak aby awarie zdarzały się w sposób specyficzny i kontrolowany. Można to robić, wykonując regularnie (lub nawet stale) testy, które mają potwierdzić, że nasze systemy ulegają awariom „z wdziękiem”.

Jak skomentował Michael Nygard, autor książki *Release It! Design and Deploy Production-Ready Software*: „Tak jak w samochodach projektuje się strefy zgnotu, których celem jest absorbowanie skutków zderzeń i zapewnienie bezpieczeństwa pasażerów, tak w przypadku systemów można wskazać funkcjonalności, które są niezbędne, i wbudować je w tryby awaryjne, dzięki którym funkcjonalności te zachowają działanie pomimo awarii. Jeśli tryby awaryjne nie zostaną zaprojektowane — pojawią się nieprzewidziane — i zazwyczaj niebezpieczne efekty”.

Odporność wymaga, aby najpierw zdefiniować tryby awarii, a następnie przeprowadzić testy, których zadaniem jest potwierdzenie działania tych trybów zgodnego z przeznaczeniem. Jednym ze sposobów realizacji tego przedsięwzięcia jest wstrzykiwanie błędów do środowiska produkcyjnego i ćwiczenie awarii na dużą skalę, tak aby uzyskać pewność przywrócenia stanu po awarii natychmiast po wystąpieniu błędów — w idealnej sytuacji w taki sposób, aby to nie wywarło wpływu na klientów.

Zaprezentowana we wprowadzeniu do niniejszego rozdziału historia firmy Netflix i przestaju usług Amazon AWS-EAST z 2012 roku to jeden z przykładów tego rodzaju sytuacji. Jeszcze bardziej interesujący przykład odporności firmy Netflix można było zaobserwować podczas awarii znanej jako „Great Amazon Reboot of 2014”, gdy prawie 10% całej floty serwerów Amazon EC2 musiało zostać uruchomionych ponownie, w celu zastosowania nadzwyczajnej poprawki zabezpieczeń Xen. Jak wspomina Christos Kalantzis z zespołu Netflix Cloud Database Engineering: „Gdy otrzymaliśmy informacje o nadzwyczajnych restartach EC2, opadły nam szczećki. Kiedy dostaliśmy listę węzłów Cassandra, na które będą miały wpływ wspomniane restarty, poczułem się chory. Ale — kontynuował Kalantzis — potem przypomniałem sobie wszystkie ćwiczenia z narzędziem Chaos Monkey, które przeprowadziliśmy. Moją reakcją było: »Trzeba przywrócić je do działania!«”.

Jak łatwo można sobie wyobrazić, wyniki były zadziwiające. Spośród ponad 2700 węzłów Cassandra wykorzystywanych w produkcji 218 zostało uruchomionych ponownie, a w przypadku 22 operacja restartu zakończyła się niepomyślnie. Jak napisali Kalantzis i Bruce Wong z zespołu Netflix Chaos Engineering: „Podczas tego weekendu Netflix miał zero przestojów. Wielokrotne i regularne ćwiczenia awarii — nawet

w warstwie utrwalania (bazy danych) — powinny być częścią planu pracy nad odpornością w każdej firmie. Gdyby dla węzłów Cassandra wcześniej nie przeprowadzano ćwiczeń z narzędziem Chaos Monkey, ta historia mogłaby zakończyć się zupełnie inaczej”.

Jeszcze bardziej zaskakujące było to, że ze względu na awarię węzłów Cassandra nie tylko nie było zespołu, który aktywnie pracowałby nad incydentem, ale nawet nie było nikogo w biurze — wszyscy byli w Hollywood na przyjęciu z okazji osiągnięcia kolejnego kamienia milowego w projekcie. Jest to kolejny przykład, który udowadnia, że proaktywne skoncentrowanie się na odporności często oznacza dla firm możliwość obsłużenia w zwykły, rutynowy sposób zdarzeń, które dla większości organizacji mogą być przyczyną kryzysów* (dodatek 9.).

WYZNACZENIE DNI GIER W CELU TRENOWANIA PRÓBNYCH AWARII

W tym podrozdziale opiszemy konkretne próby przywracania po awarii, określane jako tzw. **dni gier** (ang. *game days*). Termin ten został spopularyzowany przez Jesse'ego Robbinsa, jednego z założycieli społeczności Velocity Conference oraz współzałożyciela firmy Chef. W ten sposób nazwał przedsięwzięcia prowadzone w firmie Amazon, gdzie był odpowiedzialny za zapewnienie dostępności witryny i był powszechnie znany w firmie jako tzw. „Mistrz Katastrofy” (ang. *Master of Disaster*). Koncepcja dni gier wywodzi się z dziedziny **inżynierii odporności** (ang. *resilience engineering*). Robbins zdefiniował inżynierię odporności jako „ćwiczenia mające na celu zwiększenie odporności poprzez iniekcję awarii na dużą skalę w kluczowych systemach”.

Robbins zauważał, że „ilekroć stajemy przed wyzwaniem zaprojektowania systemu na dużą skalę, to najlepszym, na co możemy liczyć, jest zbudowanie niezawodnej platformy oprogramowania na bazie komponentów, które są całkowicie niewiarygodne.

* Specyficzne wzorce architektoniczne, które wdrożono, uwzględniały tzw. szybkie upadki — ang. *fail fasts* — (ustawienie agresywnych timeoutów w taki sposób, że komponenty ulegające awariom nie powodowały znacznego spowolnienia lub zatrzymania całego systemu), komponenty zastępcze — ang. *fallbacks* (projektowanie każdej funkcji w taki sposób, aby w przypadku awarii degradowała się do reprezentacji o niższej jakości) — oraz usuwanie własności (usuwanie niekrytycznych funkcjonalności w przypadku ich powolnego działania, aby nie dopuścić do pogorszenia komfortu pracy użytkowników). Kolejnym zadziwiającym przykładem odporności, uzyskanej przez firmę Netflix, poza zachowaniem ciągłości działania podczas przestoju usługi AWS był fakt, że usługa Netflix przetrwała ponad sześć godzin przestoju AWS, zanim zadeklarowano incydent o pierwszym stopniu istotności. Zakładano, że usługi AWS ostatecznie zostaną przywrócone („przecież zwykle tak bywa”). Dopiero po sześciu godzinach trwania awarii uruchomiono procedury związane z zachowaniem ciągłości biznesu.

Z tego powodu mamy do czynienia ze środowiskiem, w którym złożone awarie są zarówno nieuniknione, jak i nieprzewidywalne”.

W związku z tym trzeba zapewnić ciągłość działania usług w warunkach awarii dotyczących różnych części systemu — idealnie bez powodowania kryzysu lub nawet ręcznej interwencji. Jak żartuje Robbins: „Usługa nie jest przetestowana naprawdę, jeśli nie nastąpi jej awaria w produkcji”.

Celem dnia gry jest pomoc zespołom w symulowaniu i ćwiczeniu incydentów, tak by mogły zyskać doświadczenie w ich rozwiązywaniu. Najpierw należy zaplanować zdarzenie o katastrofalnych skutkach — na przykład symulowaną destrukcję całego centrum danych — w pewnym momencie w przeszłości. Następnie należy dać zespołom czas na przygotowanie się do tego zdarzenia — wyeliminowanie wszystkich pojedynczych punktów awarii i stworzenie niezbędnych procedur monitorowania, pracy w warunkach awarii itp.

Zespół organizujący dzień gry definiuje i wykonuje działania — na przykład sprawdza funkcjonowanie bazy danych w warunkach awarii (czyli symuluje uszkodzenie bazy danych i sprawdza, czy pomocnicza baza danych działa) lub wyłącza ważne połączenie sieciowe w celu ujawnienia problemów w zdefiniowanych procesach. W przypadku wystąpienia wszelkiego rodzaju problemów i trudności są one identyfikowane, tworzone są dla nich środki zaradcze, a następnie są testowane ponownie.

Później, w zaplanowanym czasie, jest symulowana awaria. Jak opisuje Robbins, w Amazon „dosłownie wyłączają usługi — bez powiadomienia — a następnie pozwalają systemom w naturalny sposób ulec awarii, a pracownikom realizować odpowiednie procedury”.

W ten sposób eksponujemy w systemie **ukryte wady** — tzn. problemy, które pojawiają się tylko ze względu na to, że do systemu zostały wstrzyknięte awarie. Robbins wyjaśnia: „Możemy odkryć, że określone systemy monitorowania lub zarządzania, kluczowe dla procesu przywracania, zostały wyłączone w wyniku sprowokowanej awarii. [Albo] możemy wykryć pojedyncze punkty awarii, o których dotychczas nic nie wiedzieliśmy”. Podobne ćwiczenia są następnie prowadzone w coraz bardziej intensywny i złożony sposób. Celem jest to, aby stały się one częścią przeciętnego dnia.

Dzięki prowadzeniu dni gry stopniowo tworzymy coraz bardziej elastyczne usługi i zyskujemy wyższy stopień przekonania, że zdolamy przywrócić działanie usługi, gdy wystąpią niepożądane zdarzenia, a także uda się nam zdobyć więcej wiedzy i zbudować bardziej odporną organizację.

Doskonałym przykładem symulowanej katastrofy jest prowadzony w Google program Disaster Recovery Program (DiRT). Kripa Krishnan pełni funkcję dyrektora programu technicznego w Google, a w czasie pisania tej książki od siedmiu lat kierowała wspomnianym programem. W tym czasie symulowano trzęsienie ziemi w Dolinie Krzemowej, które spowodowało wyłączenie usług Google w całym kampusie Mountain

View. Ćwiczono całkowity zanik zasilania w najważniejszych centrach danych, a nawet atak kosmitów na miasta, w których pracują inżynierowie.

Jak napisał Krishnan: „Obszarem, który często jest pomijany w testowaniu, jest proces biznesowy i komunikacja. Systemy i procesy w dużym stopniu przeplatają się pomiędzy sobą, a oddzielenie testowania systemów od testowania procesów biznesowych nie wydaje się realistyczne: awaria systemu biznesowego wpływa na proces biznesowy, a działający system nie przyda się zbytnio bez odpowiedniego personelu”.

Podczas ćwiczeń zdobyto między innymi następującą wiedzę:

- Gdy została utracona łączność, mechanizmy failover do stacji roboczych inżynierów nie działały.
- Inżynierowie nie wiedzieli, w jaki sposób uzyskać dostęp do mostu połączenia konferencyjnego, albo most pozwalał jedynie na podłączenie 50 osób lub był potrzebny nowy dostawca połączeń konferencyjnych, który pozwoliłby na połączenie wszystkich inżynierów.
- Gdy w centrach danych zabrakło oleju napędowego dla generatorów, nikt nie znał procedur dokonywania awaryjnych zakupów za pośrednictwem dostawcy. W efekcie ktoś musiał użyć osobistej karty kredytowej do zakupu oleju napędowego o wartości 50 000 dolarów.

Tworząc awarie w kontrolowanych warunkach, możemy ćwiczyć i tworzyć potrzebne instrukcje postępowania. Do innych rezultatów dni gry można zaliczyć uświadomienie pracownikom, do kogo powinni zadzwonić lub z kim rozmawiać podczas awarii. Dzięki ćwiczeniom tego rodzaju są rozwijane relacje z pracownikami z innych działów, co pozwala na współpracę podczas incydentu. Ponadto świadome działania przekształcają się w działania instynktowne, a ostatecznie rutynowe.

PODSUMOWANIE

Aby stworzyć kulturę wspierającą uczenie się w organizacji, trzeba zmodyfikować kontekst tzw. awarii. Jeśli odpowiednio postępujemy z błędami, możemy stworzyć dynamiczne środowisko nauki, gdzie wszyscy interesariusze czują się wystarczająco bezpiecznie, sprawdzane są pomysły i obserwacje oraz gdzie łatwiejsze jest tworzenie nowych zespołów z projektów, które nie działają zgodnie z oczekiwaniemi.

Spotkania post-mortem bez szukania winnych oraz wstrzykiwanie błędów w środowisku produkcyjnym wzmacnia poczucie komfortu oraz odpowiedzialność za ujawnianie awarii i czerpanie z nich nauki. W przypadku odpowiedniego zmniejszenia liczby incydentów należy zmniejszyć poziom tolerancji błędów, tak aby móc kontynuować uczenie się. Jak mawia Peter Senge: „Jedyną trwałą przewagą konkurencyjną jest zdolność organizacji do uczenia się szybciej niż konkurencja”.

20

Konwersja lokalnych odkryć w globalne usprawnienia

W poprzednim rozdziale omawialiśmy rozwijanie kultury uczenia się poprzez zachęcanie do dyskusji na temat błędów i incydentów podczas spotkań post-mortem bez szukania winnych. Omówiliśmy także zagadnienie znajdowania coraz słabszych sygnałów awarii, a także wzmacniania i nagradzania eksperymentowania i podejmowania ryzyka. Ponadto poprawiliśmy odporność systemu pracy, wykorzystując proaktywne planowanie i testowanie scenariuszy awarii. Dzięki wykrywaniu i naprawianiu ukrytych wad systemy stały się bezpieczniejsze.

W tym rozdziale stworzymy mechanizmy, dzięki którym nowa wiedza i usprawnienia lokalne będą mogły być przechwycone i udostępniane globalnie w całej organizacji, wzmacniając efekt globalnej wiedzy i udoskonaleń. W ten sposób przyczyniamy się do poprawy stanu wiedzy w całej organizacji. Dzięki temu wszyscy mogą czerpać korzyści ze skumulowanych doświadczeń całej organizacji.

WYKORZYSTANIE CHAT ROOMÓW I CHAT BOTÓW DO AUTOMATYZACJI I PRZECHWYTYWANIA WIEDZY ORGANIZACJI

W wielu organizacjach stworzono chat roomy w celu umożliwienia szybkiej komunikacji pomiędzy zespołami. Chat roomy są jednak również wykorzystywane do inicjowania automatyzacji.

Pionierem tej techniki był projekt ChatOps prowadzony w firmie GitHub. Jego celem było zastosowanie narzędzi automatyzacji wewnątrz konwersacji prowadzonej w chat roomach. Miało to poprawić przezroczystość oraz ułatwić dokumentowanie pracy. Jak opisuje Jesse Newland, inżynier systemów w GitHub: „Nawet gdy jesteś nowy w zespole, możesz zatrzymać w logi czatu, by zobaczyć, jakie są stosowane praktyki. To tak, jakby praktykować programowanie w parach ze wszystkimi i przez cały czas”.

Najpierw stworzono program *Hubot* — aplikację, która komunikowała się z zespołem Ops w ich chat roomach. Aby zainicjować działanie, wystarczyło przesłać polecenie za pomocą chat roomu (np. „@hubot deploy owl to production”). Wyniki również były przesyłane do chat roomu.

Wykonywanie tych działań automatycznie za pośrednictwem chat roomu (w przeciwieństwie do uruchamiania automatycznych skryptów z linii poleceń) przynosiło liczne korzyści, w tym:

- Wszyscy dokładnie widzieli, co się dzieło.
- Inżynierowie już pierwszego dnia swojej pracy mogli zobaczyć, jak wyglądała ich praca oraz w jaki sposób była wykonywana.
- Gdy pracownicy zobaczyli, że inni wzajemnie sobie pomagają, byli bardziej skłonni do udzielania pomocy.
- Schemat wspierał szybkie uczenie się organizacji i kumulowanie wiedzy.

Ponadto poza powyższymi korzyściami wynikającymi z testowania stosowanie chat roomów w naturalny sposób umożliwiło rejestrowanie i publiczne udostępnianie całej komunikacji. Dla odróżnienia wiadomości e-mail są z natury prywatne, a informacje, które są w nich zgromadzone, nie mogą być łatwo odkryte lub propagowane wewnątrz organizacji.

Zintegrowanie automatyzacji z chat roomami pomaga dokumentować i współdzieć obserwacje i rozwiązywanie problemów jako nieodłączny element wykonywania naszej pracy. To wzmacnia kulturę przejrzystości i współpracy we wszystkim, co robimy.

Jest to również niezwykle skuteczny sposób konwersji lokalnego uczenia się w globalną wiedzę. W GitHub cały personel działu operacyjnego pracuje zdalnie — w rzeczywistości nie ma dwóch inżynierów, którzy pracowaliby w tym samym mieście. Jak przypomina sobie Mark Imbriaco, były wiceprezes działu operacyjnego w GitHub: „W GitHub nie było fizycznej schładzarki do wody. Tę rolę odgrywał chat room”.

W narzędziu Hubot uwzględniono technologie automatyzacji, w tym Puppet, Capistrano, Jenkins, resque (biblioteka bazująca na Redis do tworzenia zadań w tle) oraz graphme (do generowania wykresów z programu Graphite).

Do działań wykonywanych za pośrednictwem Hubot należało sprawdzanie kondycji usług, wdrażanie kodu do produkcji oraz wyciszczenie alertów w przypadku przełączenia usług do trybu konserwacji. Działania, które były wykonywane wiele razy,

takie jak ściąganie logów testów dymnych w przypadku awarii wdrożeń, rotacje serwerów produkcyjnych, przywracanie do mastera dla produkcyjnych usług warstwy front-end lub nawet przepraszanie inżynierów dyżurujących pod telefonami — to także były działania wykonywane przez program Hubot^{*}.

Podobnie zadania ewidencjonowania w repozytorium kodu źródłowego oraz polecenia inicjujące wdrożenia do produkcji również emitowały komunikaty w chat roomie. Ponadto w chat roomie był publikowany status zmian przechodzących przez potok wdrożeń.

Oto typowa, przykładowa wymiana komunikatów w chat roomie:

```
„@sr: @jnewland, w jaki sposób pobiera się listę wielkich repo?  
↳disk_hogs albo jakoś tak?”  
“@jnewland: /disk-hogs”
```

Newland zauważył, że niektóre pytania, które były zadawane we wcześniejszej fazie projektu, teraz są zadawane rzadko. Na przykład inżynierowie mogą pytać siebie nawzajem: „Jak idzie wdrażanie?” lub „Czy ty to wdrażasz, czy raczej ja powinniem?” albo „Jak wygląda obciążenie?”.

Wśród wszystkich zalet, które opisuje Newland — włącznie z szybszym wdrażaniem się nowych inżynierów oraz poprawą produktywności wszystkich inżynierów — najważniejszym efektem było według niego to, że dzięki możliwości szybkiego wykrywania problemów i udzielania sobie wzajemnej pomocy praca inżynierów operacyjnych stała się bardziej humanitarna.

Firma GitHub stworzyła środowisko sprzyjające współpracy i lokalnemu uczeniu się, które łatwo można było przekształcić w globalny system współdzielenia wiedzy w całej organizacji. W dalszej części niniejszego rozdziału omówimy sposoby tworzenia i rozpowszechniania nowej wiedzy organizacyjnej.

AUTOMATYZACJA STANDARDOWYCH PROCESÓW W OPROGRAMOWANIU W CELU ICH WIELOKROTNEGO UŻYCIA

Często zdarza się, że standard i procesy dla architektury, testowania, wdrażania i infrastruktury są „pisane prozą” i zapisywane w dokumentach Worda składowanych gdzieś na serwerach. Problem w tym przypadku polega na tym, że inżynierowie, którzy budują nowe aplikacje lub środowiska, często nie wiedzą, że te dokumenty istnieją, albo nie mają czasu na stosowanie się do udokumentowanych standardów. W rezultacie tworzą własne narzędzia i procesy, ze wszystkimi złymi efektami, jakich można się

* Hubot często wykonywał zadania poprzez wywoływanie skryptów powłoki, które następnie mogły być uruchamiane z dowolnego miejsca chat roomu — w tym z telefonu inżyniera.

w takim przypadku spodziewać: kruchymi, niezabezpieczonymi i trudnymi w utrzymaniu aplikacjami i środowiskami, które są trudne do uruchomienia, utrzymania i rozwijania.

Zamiast umieszczać doświadczenia w dokumentach Worda, należy przekształcić udokumentowane standardy i procesy obejmujące sumę zdobytej w organizacji wiedzy na formę wykonywalną, która jest łatwiejsza do wielokrotnego użycia. Jednym z najlepszych sposobów, dzięki którym ta wiedza może być wykorzystywana wielokrotnie, jest umieszczenie jej w centralnym repozytorium kodu źródłowego i udostępnienie narzędzia wszystkim do jej przeszukiwania i wykorzystywania.

Justin Arbuckle, były naczelnym architekt w GE Capital, w 2013 roku powiedział: „Musielibyśmy stworzyć mechanizm, który umożliwiłby zespołom łatwe przestrzeganie zasad — przepisów krajowych, regionalnych i branżowych — rozproszonych na dziesiątki framework'ów obejmujących tysiące aplikacji uruchomionych na dziesiątkach tysięcy serwerów w kilkunastu centrach danych”.

Mechanizm, który stworzono, otrzymał nazwę ArchOps. System ten „pozwolił naszym inżynierom być budowniczymi, a nie murarzami. Dzięki wprowadzeniu standardów projektowania do zautomatyzowanych planów, które mogły być łatwo wykorzystywane przez wszystkich, osiągnęliśmy spójność jako produkt uboczny”.

Zaimplementowaniem procesów ręcznych w kodzie, który może być automatycznie uruchomiony, umożliwiamy powszechnie przyjęcie procedury. To może dostarczać wartość dla wszystkich, którzy ją stosują. Arbuckle zakończył: „Rzeczywista zgodność organizacji z przepisami jest wprost proporcjonalna do stopnia, w jakim przepisy są wyrażone w kodzie”.

Maksymalne ułatwienie tego procesu jest kluczem do osiągnięcia celu. Pozwalamy na powszechnie przyjęcie praktyk — możemy nawet zastańowić się nad przekształceniem ich w usługi współdzielone, wspierane przez organizację.

STWORZENIE JEDNEGO, WSPÓLNEGO REPOZYTORIUM KODU ŹRÓDŁOWEGO DLA CAŁEJ ORGANIZACJI

Współdzielone repozytorium kodu źródłowego obejmujące całą firmę to jeden z najpotężniejszych mechanizmów pozwalających na zintegrowanie lokalnych odkryć z wiedzą całej organizacji. Kiedy w repozytorium kodu źródłowego dokonujemy aktualizacji czegokolwiek (np. biblioteki współdzielonej), to zmiana ta szybko i automatycznie rozprzestrzenia się na wszystkie inne usługi, które używają tej biblioteki, i jest zintegrowana za pośrednictwem potoku wdrożeń każdego zespołu.

Jednym z najlepszych przykładów firmy wykorzystującej wspólne repozytorium kodu źródłowego, obejmujące całą organizację, jest Google. W 2015 r. firma Google posługiwała się jednym, wspólnym repozytorium kodu źródłowego, obejmującym ponad miliard plików i ponad dwa miliardy wierszy kodu. To repozytorium jest

wykorzystywane przez każdego spośród 25 000 inżynierów pracujących w Google, w tym Google Search, Google Maps, Google Docs, Google+, Google Calendar, Gmail i YouTube*.

Jednym z cennych wyników tego stanu rzeczy jest możliwość wykorzystywania przez inżynierów różnorodnych doświadczeń wszystkich osób w organizacji. Rachel Potvin, menedżer w Google nadzorująca pracę grupy Developer Infrastructure powiedziała Wiredowi, że każdy inżynier w Google ma dostęp do „bogactwa bibliotek”, ponieważ „prawie wszystko już zostało zrobione”.

Ponadto Eran Messeri, inżynier w grupie Google Developer Infrastructure, wyjaśnił, że jedną z zalet używania jednego repozytorium jest to, że pozwala ono użytkownikom na łatwy dostęp do całego kodu w najbardziej aktualnej postaci, bez potrzeby koordynacji.

Umieszczamy w naszym repozytorium kodu źródłowego nie tylko kod źródłowy, ale także inne artefakty, które kodują wiedzę i naukę, w tym:

- standardy konfiguracji dla bibliotek, infrastruktury i środowiska (receptury Chef, manifesty Puppet itp.);
- narzędzia do instalacji;
- standardy i narzędzia testowania, w tym zabezpieczenia;
- narzędzia obsługi potoku wdrożeń;
- narzędzia monitorowania i analizy;
- samouczki i standardy.

Kodowanie wiedzy i współdzielenie jej za pośrednictwem repozytorium jest jednym z najpotężniejszych mechanizmów do propagowania wiedzy, jakimi dysponujemy. Jak opisuje Randy Shoup: „Najpotężniejszym mechanizmem zapobiegania awariom w Google jest pojedyncze repozytorium kodu źródłowego. Zawsze, gdy ktoś zaewidencjonuje jakąkolwiek zmianę w repozytorium, inicjowana jest nowa komplikacja, w której zawsze są wykorzystywane najnowsze wersje wszystkiego. Wszystko jest budowane ze źródeł zamiast dynamicznego łączenia w czasie wykonywania — zawsze istnieje jedna wersja biblioteki, która jest aktualna i jest w użyciu, i ona jest statycznie łączona podczas procesu budowania”.

Tom Limoncelli jest współautorem książki *The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems*. W przeszłości pracował na stanowisku Site Reliability w firmie Google. W swojej książce stwierdził, że wartość posiadania pojedynczego repozytorium dla całej organizacji, jest tak wielka, że nawet trudno to wyjaśnić.

* Projekty Chrome i Android są przechowywane w oddzielnym repozytorium kodu, a niektóre algorytmy — na przykład PageRank są tajne — i są dostępne tylko dla niektórych zespołów.

Narzędzie napisane raz może być wykorzystywane we wszystkich projektach. Masz stuprocentową wiedzę na temat zależności biblioteki. Dzięki temu możesz ją zrefaktoryzować i mieć stuprocentową pewność, na kogo wpłyną wprowadzone zmiany oraz kto będzie musiał przeprowadzić testy regresji. Mógłbym prawdopodobnie przytoczyć jeszcze ze sto przykładów. Nie można wyrazić słowami, jak wielką przewagę konkurencyjną zyskała dzięki temu firma Google.

W Google każda biblioteka (np. libc, OpenSSL, jak również biblioteki opracowane wewnętrznie, takie jak biblioteki obsługi wątków w Javie) ma właściciela, który jest odpowiedzialny nie tylko za to, że biblioteka się skompiluje, ale również za to, że pomyślnie przejdzie ona testy dla wszystkich projektów, które od niej zależą. Właściciel ten jest również odpowiedzialny za przeprowadzanie migracji każdego projektu z jednej wersji do następnej.

Rozważmy przykład rzeczywistej organizacji, w której działa w produkcji 81 różnych wersji biblioteki framework Java Struts. Wszystkie z wyjątkiem jednej mają krytyczne luki w zabezpieczeniach, a utrzymywanie wszystkich tych wersji, z których każda ma własne dziwactwa i zawiłości, stwarza znaczne obciążenie operacyjne i wywołuje dodatkowy stres. Ponadto ze względu na różnorodność aktualizowanie wersji staje się ryzykowne i niebezpieczne, a to z kolei zniechęca programistów do dokonywania aktualizacji. Cykl trwa.

Pojedyncze repozytorium kodu źródłowego rozwiązuje większość spośród tych problemów. Pomaga również posiadanie automatycznych testów, dzięki którym zespoły mogą bezpiecznie i pewnie migrować do nowych wersji.

Jeśli nie jesteśmy w stanie zbudować całego oprogramowania z jednego drzewa kodu źródłowego, to musimy znaleźć inny sposób utrzymywania dobrych wersji bibliotek i ich zależności. Możemy stworzyć repozytorium kodu źródłowego dla całej organizacji — na przykład Nexus, Artifactory albo RPM lub Debian. Gdy zostaną odkryte luki w zabezpieczeniach, to należy zaktualizować kod zarówno w tych repozytoriach, jak i w systemach produkcji.

ROZPOWSZECHNIANIE WIEDZY ZA POMOCĄ ZAUTOMATYZOWANYCH TESTÓW W ROLI DOKUMENTACJI ORAZ SPOŁECZNOŚCI PRAKTYK

Gdy w organizacji używamy współdzielonych bibliotek, to powinniśmy zapewnić szybkie rozprzestrzenianie się wiedzy i usprawnień. Dzięki uzupełnieniu każdej z tych bibliotek w odpowiednią liczbę testów automatycznych każda z nich jest dokumentacją samą dla siebie i pokazuje innym inżynierom, w jaki sposób można z niej korzystać.

W przypadku stosowania technik TDD, gdy testy automatyczne są pisane przed kodem, korzyć tę osiągamy niemal automatycznie. Stosowanie tej dyscypliny powoduje przekształcenie zestawów testowych w żyjącą, aktualną specyfikację systemu.

Każdy inżynier, chcąc zrozumieć, jak należy korzystać z systemu, może przeanalizować zestaw testów, gdzie może znaleźć działające przykłady użycia API systemu.

W idealnej sytuacji każda biblioteka powinna mieć jednego właściciela lub jeden zespół wspierający. Dzięki temu można bez trudu wskazać, gdzie są wiedza i doświadczenie związane z biblioteką. Ponadto należy (w idealnej sytuacji) pozwolić na istnienie tylko jednej wersji w produkcji. Dzięki temu zyskujemy pewność, że cokolwiek jest w produkcji — odzwierciedla najlepszą, zbiorową wiedzę organizacji.

W tym modelu właściciel biblioteki jest jednocześnie odpowiedzialny za bezpieczną migrację za pośrednictwem repozytorium, z jednej wersji do następnej. To z kolei wymaga szybkiego wykrywania błędów regresji. Jest to możliwe dzięki kompleksowym testom automatycznym i stosowaniu zasad ciągłej integracji we wszystkich systemach, które korzystają z biblioteki.

W celu szybszego propagowania wiedzy można również stworzyć grupy dyskusyjne lub czaty dla każdej biblioteki lub usługi. Dzięki temu każdy, kto ma pytania dotyczące biblioteki, może uzyskać odpowiedź od innych użytkowników, którzy często odpowiadają szybciej niż programiści.

W wyniku zastosowania tego typu narzędzia komunikacji zamiast z odizolowanej wiedzy rozproszonej po całej organizacji korzystamy z łatwiejszej wymiany umiejętności i doświadczeń. W ten sposób możemy zagwarantować pracownikom możliwość udzielania sobie wzajemnej pomocy w zakresie rozwiązywania problemów i stosowania nowych wzorców.

PROJEKTOWANIE OPERACJI ZA POŚREDNICTWEM UJEDNOLICONYCH WYMAGAŃ NIEFUNKCJONALNYCH

Jeśli pracownicy działu Dev obserwują pracę w dole strumienia wartości i uczestniczą w rozwiązywaniu problemów w produkcji, to aplikacja staje się coraz lepiej przystosowana do wykorzystania przez dział operacji. Ponadto kiedy zaczniemy celowo projektować kod i aplikacje pod kątem wykorzystania szybkiego przepływu i łatwości wdrażania, to z dużym prawdopodobieństwem zdołamy także zidentyfikować zbiór wymagań niefunkcjonalnych, które warto zintegrować ze wszystkimi usługami produkcyjnymi.

Zaimplementowanie tych wymagań niefunkcjonalnych ułatwi również wdrażanie usług oraz ich działanie w środowisku produkcyjnym, gdzie będziemy mogli szybko wykryć i rozwiązać problemy oraz zagwarantować degradację „z wdziękiem” w przypadku awarii komponentów. Do przykładów wymagań niefunkcjonalnych można zaliczyć zadbanie o:

- odpowiednią liczbę produkcyjnych wskaźników telemetrycznych w aplikacji i środowiskach;
- możliwość dokładnego śledzenia zależności;

- elastyczność usług oraz degradację „z wdziękiem”;
- zgodność w przód i wstecz pomiędzy wersjami;
- możliwość archiwizacji danych w celu zarządzania rozmiarem zestawu produkcyjnych danych;
- możliwość łatwego wyszukiwania i zrozumienia zapisów w logach usług;
- możliwość śledzenia żądań od użytkowników za pośrednictwem wielu usług;
- proste, skoncentrowane środowisko działania z wykorzystaniem flag funkcji i podobnych mechanizmów.

Dzięki skodyfikowaniu tego rodzaju wymagań niefunkcjonalnych możemy ułatwić wykorzystanie zbiorowej wiedzy i doświadczenia w organizacji dla wszystkich nowych i istniejących usług. Wszystko to są obowiązki zespołu budującego usługę.

UWZGLĘDNIANIE W PROCESIE ROZWOJU OPERACYJNYCH HISTORYJEK UŻYTKOWNIKA WIELOKROTNEGO UŻYTKU

Gdy mamy zadania operacyjne, których nie można w pełni zautomatyzować lub przekształcić w usługi samoobsługowe, naszym celem powinno być doprowadzenie do tego, aby te cykliczne zadania były w maksymalnym stopniu powtarzalne i deterministyczne. Aby to osiągnąć, należy ustandaryzować potrzebną pracę, w maksymalnym stopniu ją zautomatyzować oraz udokumentować, tak aby zespoły produktu mogły lepiej planować i wspomagać zasobami te działania.

Zamiast ręcznie budować serwery, a następnie wprowadzać je do produkcji zgodnie z tworzonymi ręcznie listami kontrolnymi, należy w jak największym stopniu zautomatyzować tę pracę. O ile niektórych czynności nie da się zautomatyzować (np. montaż serwera na półce oraz jego okablowanie), o tyle — w celu maksymalnego skrócenia czasu realizacji i możliwości popełnianych błędów — należy dążyć do zbiorowego i jak najbardziej czytelnego zdefiniowania przełączeń pracy. To pozwala również na lepsze planowanie i tworzenie harmonogramu podobnych czynności w przyszłości. Na przykład możemy użyć narzędzi, takich jak Rundeck, do zautomatyzowania i uruchomienia przepływów pracy albo systemów zarządzania zleceniami roboczymi, takich jak JIRA lub ServiceNow.

W idealnej sytuacji dla wszystkich powtarzających się zadań operacyjnych będziemy znać następujące fakty: jakie prace są wymagane, kto jest potrzebny do ich wykonania, jakie kroki są wymagane do zakończenia prac itp. Na przykład: „Wiemy, że wdrażanie mechanizmów wysokiej dostępności obejmuje 14 kroków, które wymagają pracy czterech różnych zespołów, a ostatnie pięć razy wykonanie tego zadania zajęło średnio trzy dni”.

Podobnie jak w dziale Dev tworzymy historyjki użytkownika, które następnie umieszczamy w rejestrze backlog, a następnie je z niego pobieramy w celu realizacji, możemy stworzyć dobrze zdefiniowane „operacyjne historyjki użytkownika”, reprezentujące prace, które mogą być wielokrotnie wykonane w różnych projektach (np. wdrożenia, zwielokrotnianie, zabezpieczenia itd.). Tworząc te dobrze zdefiniowane historyjki użytkownika Ops, eksponujemy powtarzalne prace działu operacji IT obok zadań działu Dev, co pozwala na lepsze planowanie i bardziej powtarzalne wyniki.

ZADBAJ O TO, ABY WYBRANE TECHNOLOGIE POMAGAŁY OSIĄGNĄĆ CELE ORGANIZACJI

Gdy jednym z celów jest zmaksymalizowanie wydajności programistów oraz gdy mamy architekturę zorientowaną na usługi, to małe zespoły usług mogą budować i uruchamiać usługę, korzystając z dowolnego języka programowania oraz frameworka, które najlepiej służą spełnianiu ich specyficznych potrzeb. W niektórych przypadkach taka architektura i struktura zespołów pozwalają najłatwiej osiągnąć wyznaczone cele organizacyjne.

Są jednak scenariusze, gdy jest odwrotnie — na przykład gdy wiedza dotycząca kluczowej usługi jest skumulowana tylko w jednym zespole i tylko ten jeden zespół może wprowadzić zmiany lub naprawić problemy. W takiej sytuacji tworzy się wąskie gardło. Innymi słowy, możemy zoptymalizować organizację pod kątem wydajności zespołu, ale niechcący utrudnić osiągnięcie celów organizacyjnych.

Często tak się dzieje w przypadku istnienia grup operacyjnych zorientowanych na funkcjonalności, odpowiedzialnych za wszystkie aspekty wsparcia usługi. W takich scenariuszach, aby zapewnić fachową obsługę specyficznych technologii, trzeba zagwarantować działowi operacji możliwość wywierania wpływu na to, które komponenty będą wykorzystywane w produkcji, albo zwolnić go z odpowiedzialności za działanie nieobsługiwanych platform.

Jeśli nie mamy listy technologii obsługiwanych przez dział operacji, kolektywnie generowanej przez działy Dev i Ops, to powinniśmy systematycznie analizować infrastrukturę produkcji i usług, a także wszystkie ich obecnie obsługiwane zależności. W ten sposób można znaleźć te, które generują nieproporcjonalnie dużą liczbę żądań naprawy oraz nieplanowane prace. Naszym celem powinno być zidentyfikowanie takich technologii, które:

- utrudniają lub spowalniają przepływ pracy;
- tworzą nieproporcjonalnie dużą liczbę niezaplanowanych zadań;
- tworzą nieproporcjonalnie dużą liczbę żądań obsługi;
- są najbardziej niespójne z pożdanymi wynikami dotyczącymi architektury (np. przepustowość, stabilność, bezpieczeństwo, niezawodność, ciągłość biznesowa).

Usunięcie tej problematycznej infrastruktury i platform z listy technologii obsługiwanych przez dział Ops umożliwia im skupienie się na tej infrastrukturze, która najlepiej pomaga w osiągnięciu globalnych celów organizacji.

Jak opisuje Tom Limoncelli: „Kiedy pracowałem w Google, mieliśmy jeden oficjalny język kompilowany, jeden język skryptowy i jeden oficjalny język interfejsu użytkownika. Oczywiście inne języki były obsługiwane w ten czy inny sposób, ale trzymanie się zasad »wielkiej trójki« ułatwiało obsługę bibliotek, narzędzi oraz zapewniało łatwiejszy sposób wyszukiwania współpracowników”*. Stosowane standardy były dodatkowo wzmacniane przez proces przeglądu kodu oraz przez języki używane wewnętrznych platformach.

Olivier Jacques i Rafael Garcia w prezentacji na konferencji 2015 DevOps Enterprise Summit stwierdzili:

Wewnętrznie opisaliśmy nasz cel jako tworzenie „boi, ale nie granic”. Zamiast rysowania twardych granic, w których wszyscy muszą pozostać, możemy ustawić boje, które wskazują obszary kanału, w których jesteśmy bezpieczni i mamy wsparcie. Możemy przekraczać boje pod warunkiem stosowania zasad obowiązujących w organizacji. W końcu jak moglibyśmy kiedykolwiek wprowadzić jakąkolwiek innowację pozwalającą nam zyskać przewagę konkurencyjną, gdybyśmy nie odkrywali i nie testowali na granicach? Liderzy powinni być nawigatorami podróży — wytyczać drogę, ale jednocześnie pozwalać pracownikom odkrywać to, co znajduje się poza nią.

Studium przypadku

Standaryzacja nowego stosu technologii w Etsy (2010)

W wielu organizacjach przyjmujących DevOps programiści opowiadają popularną historiękę: „Dział Ops nie potrafił zapewnić nam tego, czego potrzebowaliśmy, więc po prostu włączyliśmy jego zadania do własnych i sami się obsłużyliśmy”. Jednak w początkowej fazie transformacji Etsy kierownictwo techniczne zastosowało odwrotne podejście — zdecydowano o znacznym zmniejszeniu liczby technologii obsługiwanych w produkcji.

* W Google używano języka C++ jako oficjalnego języka kompilowanego, Pythona (a później Go) jako oficjalnego języka skryptowego oraz Javy, JavaScriptu i frameworka Google Web Toolkit jako oficjalnych języków tworzenia interfejsu użytkownika.

W 2010 r., po prawie katastrofalnym szczycie sezonu wakacyjnego, zespół Etsy zdecydował się znacznie zmniejszyć liczbę technologii wykorzystywanych w produkcji, wybierając kilka, które mogły mieć wsparcie w całej organizacji. Pozostałe wyeliminowano*.

Celem było ustandaryzowanie i rozmyślne ograniczenie obsługiwanej infrastruktury i konfiguracji. Jedną z pierwszych decyzji była migracja całej platformy Etsy do PHP i MySQL. Była to przede wszystkim decyzja filozoficzna, a nie technologiczna — menedżerowie z Etsy chcieli, aby działy Dev i Ops rozumiały pełny stos technologii, tak aby każdy mógł tworzyć kod dla wybranej platformy, ale również by miał możliwość czytania, przepisywania i naprawiania kodu innych. W ciągu najbliższych kilku lat — jak przypomina Michael Rembetsy, który był wtedy dyrektorem operacyjnym Etsy: „Odesłaliśmy na emeryturę kilka wielkich technologii, wycofując je całkowicie z produkcji”. Zrezygnowano między innymi z takich technologii, jak lighttpd, Postgres, MongoDB, Scala, CoffeeScript czy Python.

Dan McKinley, programista zespołu funkcjonalności, który w 2010 roku wprowadził do Etsy technologię MongoDB, napisał na swoim blogu, że wszystkie korzyści z posiadania bazy danych bez schematu zostały zanegowane ze względu na problemy operacyjne, które zespół był zmuszony rozwijać. Obejmowało to problemy związane z rejestrówaniem, wykresami, monitorowaniem, telemetrią produkcji i kopiami zapasowymi oraz przywracaniem, a także wiele innych problemów, którymi programiści zazwyczaj nie muszą się martwić. W rezultacie porzucono MongoDB. Nową usługę przeniesiono do już wcześniej wspieranej infrastruktury MySQL.

PODSUMOWANIE

Techniki opisane w tym rozdziale umożliwiają włączenie nowo zdobytej nauki do wspólnej wiedzy organizacji, co wzmacnia efekt uczenia się. Można to osiągnąć poprzez aktywne i powszechnie komunikowanie nowej wiedzy — na przykład za pośrednictwem chat roomów oraz dzięki wykorzystaniu takich technologii, jak architektura jako kod, współdzielone repozytoria kodu źródłowego, standaryzacja technologii itp. W ten sposób praktyki będą stosowane nie tylko przez pracowników działów Dev i Ops, ale również w całej organizacji, dzięki temu wszyscy, którzy wykonują swoją pracę, będą mogli korzystać ze skumulowanych doświadczeń całej organizacji.

* W tym czasie w Etsy korzystano z takich platform i języków programowania, jak: PHP, lighttpd, Postgres, MongoDB, Scala, CoffeeScript, Python, i z wielu innych.

Zarezerwuj czas na stworzenie organizacyjnego systemu uczenia się i doskonalenia

Jedna z praktyk stanowiących część Toyota Production System nazywa się *improvement blitz* (lub czasami *kaizen blitz*). Definiuje się ją jako dedykowany i skoncentrowany okres do rozwiązywania określonego problemu — często kilka dni. Dr Spear wyjaśnia: „*Blitz* często przyjmuje następującą formę: grupa zbiera się po to, by skoncentrować się uważnie na procesie, w którym występują problemy... Blitz trwa kilka dni, a jego celem jest doskonalenie procesów. Stosowane są wtedy środki polegające na skoncentrowanym wykorzystaniu osób spoza procesu w celu przekazania wiedzy tym, którzy standardowo są wewnętrz procezu”.

Spear zaobserwował, że często efektem pracy zespołu *improvement blitz* jest opracowanie nowego podejścia do rozwiązywania problemu — na przykład zastosowanie nowego układu urządzeń, nowych środków transportu materiałów i informacji, lepiej zorganizowane stanowisko pracy lub praca bardziej ustandardyzowana. Zespół tworzy też czasami listę zmian do wprowadzenia.

Przykładem pracy zespołu *improvement blitz* jest program *Monthly Challenge* w ośrodku DevOps Dojo firmy Target. Ross Clanton, dyrektor operacyjny firmy Target, jest odpowiedzialny za przyspieszenie przyjęcia technik. Jednym z głównych mechanizmów służących temu celowi jest Technology Innovation Center, bardziej znane pod nazwą DevOps Dojo.

DevOps Dojo zajmuje około 18 000 metrów kwadratowych otwartej przestrzeni biurowej, gdzie trenerzy DevOps pomagają zespołom z organizacji technicznych firmy

Target poprawić stan stosowanych praktyk. Najbardziej intensywny format zajęć to tzw. „30-dniowe wyzwania”, podczas których wewnętrzne zespoły programistów pracują przez miesiąc pod obserwacją dedykowanych trenerów i inżynierów Dojo. Zespoły pracują nad swoimi zadaniami. Celem jest rozwiązywanie wewnętrznego problemu, z którym się zmagali, oraz uzyskanie przełomu w ciągu 30 dni.

W ciągu 30 dni członkowie zespołu pracują intensywnie z trenerami Dojo nad rozwiązyaniem problemu — planując i realizując zadania oraz demonstrując ich realizację w dwudniowych sprintach. Po zakończeniu 30-dniowego wyzwania zespoły wracają do swojej pracy nie tylko z rozwiązyaniem poważnego problemu, ale również z nową wiedzą dla swoich współpracowników.

Clanton opisuje: „Mamy obecnie możliwości przeprowadzania 30-dniowych wyzwań dla ośmiu zespołów jednocześnie, dlatego skupiamy się na najbardziej strategiccznych projektach dla organizacji. Dotychczas przez ośrodek Dojo przeszli członkowie zespołów wykonujących najbardziej kluczowe zadania — w tym zespoły punktów sprzedaży (ang. *Point Of Sale — POS*), zarządzania zapasami, cenami i promocjami”.

Dzięki posiadaniu pełnoetatowego personelu Dojo oraz skoncentrowaniu się na tylko jednym celu zespoły realizujące 30-dniowe wyzwania czynią niezwykłe postępy.

Ravi Pandey, menedżer zespołu programistów w firmie Target, który uczestniczył w programie, wyjaśnił: „Kiedyś na stworzenie środowiska testowego musieliszyśmy czekać sześć tygodni. Dzisiaj możemy je uzyskać w ciągu kilku minut. Pracujemy ramię w ramię z inżynierami Ops, którzy pomagają nam zwiększyć naszą wydajność, budując oprzyrządowanie ułatwiające osiągnięcie wyznaczonych celów”. Clanton dodaje: „Nie należy do rzadkości sytuacja, w której zespoły osiągają w ciągu kilku dni cele, które zwykle zajmowały im od trzech do sześciu miesięcy. Do tej pory przez proces w ośrodku Dojo przeszło 200 uczniów, którzy zrealizowali 14 programów 30-dniowych wyzwań”.

Ośrodek Dojo obsługuje również mniej intensywne modele zaangażowania, w tym program Flash Builds, w którym zespoły spotykają się na dwu- lub trzydniowe szkolenia. Celem tych przedsięwzięć jest dostarczenie minimalnego produktu (ang. *minimal viable product — MVP*) lub funkcjonalności. Co dwa tygodnie są również prowadzone dni otwarte Open Labs. Podczas ich trwania każdy może odwiedzić ośrodek Dojo, aby porozmawiać z trenerami Dojo, uczestniczyć w demonstracjach lub szkoleniach.

W tym rozdziale opiszemy różne sposoby rezerwowania czasu dla organizacyjnego uczenia się i doskonalenia oraz dalszej instytucjonalizacji praktyk poświęcania czasu na usprawnianie codziennej pracy.

INSTYTUCJONALIZACJA RYTUAŁÓW W CELU SPŁACANIA DŁUGU TECHNICZNEGO

W tym podrozdziale pokażemy, jak zaplanować rytuały, które pomagają wyegzekwować praktyki rezerwowania czasu zespołów Dev i Ops na prace związane z usprawnieniami.

Jednym z najprostszych sposobów, aby to zrobić, jest zaplanowanie i przeprowadzenie trwających dzień lub tydzień przedsięwzięć *improvement blitz*, podczas których wszystkie osoby w zespole (lub w całej organizacji) samodzielnie organizują się w celu rozwiązywania problemów, które ich interesują — nie jest dozwolona praca nad funkcjonalnościami. Zespół może się wtedy zajmować problematycznymi obszarami kodu, środowiskiem, architekturą, narzędziami itp. Zespoły obejmują osoby z całego strumienia wartości — często w ich skład wchodzą inżynierowie działów Dev, Ops i Infosec. Zespoły, które zwykle nie pracują ze sobą, łączą swoje umiejętności i wysiłki w celu poprawy wybranego obszaru, a następnie demonstrują usprawnienia pozostałej części firmy.

Oprócz terminów pochodzących z Lean, *kaizen blitz* i *improvement blitz*, do nazywania technik specjalnych rytałów dotyczących prac, które są związane z usprawnieniami, wykorzystuje się również określeń *spring* lub *fall cleanings* (dosł. „wiosenne lub jesienne sprzątanie”) oraz *ticket queue inversion weeks* (dosł. „tygodnie odwróconej kolejki zleceń”). Spotyka się również inne terminy — *hack days*, *hackathon* oraz 20% *innovation time*. Niestety, te specyficzne rytały czasami koncentrują się na wprowadzaniu innowacji do produktów oraz tworzeniu prototypów nowych koncepcji rynkowych zamiast na usprawnianiu pracy. Co gorsza, często ograniczają się do inżynierów Dev, co sprawia, że cele tych przedsięwzięć znacznie się różnią od celów *improvement blitz**.

Celem przedsięwzięć blitz nie jest jedynie eksperymentowanie i wprowadzanie innowacji dla samego testowania nowych technologii, ale usprawnienie codziennej pracy — na przykład usunięcie codziennych obejść. Chociaż eksperymenty mogą również prowadzić do usprawnień, przedsięwzięcia *improvement blitz* są skoncentrowane na rozwiązywaniu konkretnych problemów, jakie napotykamy w codziennej pracy.

Możemy zaplanować trwające tydzień przedsięwzięcia *improvement blitz*, podczas których prace Dev i Ops mają wyższy priorytet od celów uzyskania usprawnień. Zarządzanie przedsięwzięciami *improvement blitz* jest proste: wybierany jest tydzień, podczas którego wszyscy w organizacji technicznej jednocześnie pracują nad usprawnieniami. Na koniec okresu każdy zespół robi prezentację dla swoich współpracowników, w której omawia problem, jaki był rozwiązywany, oraz uzyskane efekty. Stosowanie tej praktyki wzmacnia kulturę rozwiązywania problemów dzięki pracy inżynierów z całego strumienia wartości. Ponadto wzmacnia praktykę rozwiązywania problemów w ramach codziennej pracy i pokazuje, że spłacanie dlużu technicznego jest opłacalne.

Przedsięwzięcia *improvement blitz* przynoszą efekty, ponieważ pozwalają osobom będącym najbliżej pracy w ciągły sposób identyfikować i rozwiązywać własne problemy.

* W dalszej części książki terminy „hack week” i „hackathon” będą używane zamiennie z „improvement blitz”, a nie w kontekście założenia „możesz pracować, nad czym tylko chcesz”.

Przypuśćmy, że nasz skomplikowany system jest jak pajęczyna, z przeplatającymi się nitkami, które stale się osłabiają i przerywają. Jeśli zostanie przerwana odpowiednia kombinacja nitek, cała sieć ulega zniszczeniu. Nie ma dobrego poziomu zarządzania i kontroli, które mogłyby poinstruować pracowników, jak po kolei naprawiać poszczególne nitki. Zamiast tego trzeba stworzyć kulturę organizacyjną i normy, dzięki którym wszyscy będą mogli stale wyszukiwać i naprawiać przerwane nitki w ramach codziennej pracy. Jak zauważył dr Spear: „Nie ma niczego dziwnego w tym, że pajęki naprawiają nitki w pajęczynie natychmiast, kiedy powstanie uszkodzenie. Nie czekaj, aż niedoskonałości się skumulują”.

Doskonały przykład sukcesu koncepcji *improvement blitz* opisał Mark Zuckerberg — CEO Facebooka. W rozmowie z Jessicą Stillman z Inc.com powiedział: „Co kilka miesięcy mamy hackathon, gdzie wszyscy budują prototypy nowych pomysłów. Na koniec cały zespół spotyka się i analizuje wszystkie produkty, które zostały zbudowane. Efektem hackatonów było wiele naszych najbardziej udanych produktów, w tym Timeline, czat, video, framework tworzenia aplikacji mobilnych oraz niektóre z najważniejszych elementów infrastruktury, takie jak kompilator HipHop”.

Na szczególną uwagę zasługuje kompilator PHP HipHop. W 2008 r. Facebook zmagał się ze znaczącymi problemami wydajności. Obsługiwał ponad 100 milionów aktywnych użytkowników i dynamicznie się rozwijał, co stwarzało ogromne problemy dla całego zespołu inżynierskiego. W czasie trwania dnia *hack day* Haiping Zhao, Senior Server Engineer w firmie Facebook, zaczął eksperymentować z konwersją kodu PHP na C++, z nadzieją na znaczną poprawę wydajności istniejącej infrastruktury. W ciągu kolejnych dwóch lat sformowano mały zespół, który zbudował produkt znany dziś pod nazwą kompilator HipHop, który konwertował wszystkie usługi produkcyjne Facebooka z interpretowanego kodu PHP na kompilowane binaria C++. Zastosowanie kompilatora HipHop pozwoliło platformie Facebook obsłużyć sześć razy większe obciążenia produkcyjne w porównaniu z natywnym kodem PHP.

W wywiadzie dla Cade'a Metza z magazynu *Wired* Drew Paroski, jeden z inżynierów, którzy pracowali nad projektem, zauważył: „Był taki moment, w którym uważałem, że gdyby nie było HipHopa, to bylibyśmy w prawdziwych tarapatach. Prawdopodobnie potrzebowalibyśmy do obsługi więcej maszyn, niż zdołalibyśmy zdobyć w tamtym czasie. HipHop był naszą modlitwą »Zdrowaś Mario«, która pomogła”.

Później Paroski i współpracujący z nim inżynierowie Keith Adams i Jason Evans uznali, że zdołają poprawić wydajność kompilatora HipHop i wyeliminować niektóre z jego ograniczeń obniżających wydajność programistów. W efekcie powstał projekt maszyny wirtualnej HipHop („HHVM”), w którym zastosowano podejście kompilacji just-in-time. Do 2012 roku HHVM całkowicie zastąpił kompilator HipHop w produkcji. W projekcie uczestniczyło prawie 20 inżynierów.

Realizacja zaplanowanych regularnie przedsięwzięć *improvement blitz* i *hack week* sprawiła, że wszyscy uczestniczący w strumieniu wartości byli dumni i czuli się odpo-

wiedzialni za wprowadzane innowacje. Stopniowo integrujemy usprawnienia do systemu, co wzmacnia ducha bezpieczeństwa, niezawodności i nauki.

UMOŻLIWIENIE WSZYSTKIM NAUCZANIA I UCZENIA SIĘ

Dynamiczna kultura nauki stwarza warunki nie tylko do tego, aby wszyscy się uczyli, ale także by nauczali innych — czy to tradycyjnymi metodami dydaktycznymi (np. organizowanie kursów i szkoleń), czy w sposób bardziej empiryczny i otwarty (np. konferencje, warsztaty, doradztwo). Jednym ze sposobów wspierania nauczania i uczenia się jest poświęcenie na to czasu organizacji.

Steve Farley, wiceprezes ds. informatyki w Nationwide Insurance powiedział: „Mamy 5000 profesjonalistów technicznych, których nazywamy »partnerami«. Od 2011 r. dążymy do tworzenia kultury uczenia się — część działań podejmowanych w tym kierunku realizujemy za pośrednictwem programu, który określamy jako *Teaching Thursday*. W ramach tego programu co tydzień wygospodarowujemy naszym partnerom czas na naukę. Przez dwie godziny każdy partner może nauczać lub uczyć się. Tematy są wybierane spośród propozycji przekazywanych przez partnerów — niektóre z nich dotyczą technologii, procesu wytwarzania nowego oprogramowania, technik usprawniania procesu lub nawet sposobów lepszego zarządzania karierą. Najcenniejszą rzeczą jest to, że każdy partner może być mentorem lub może się uczyć od innych partnerów”.

Z lektury niniejszej książki jasno wynika, że niektóre umiejętności są coraz bardziej potrzebne wszystkim inżynierom — nie tylko programistom. Na przykład coraz większego znaczenia dla wszystkich inżynierów operacyjnych i testowych nabiera zapoznanie się z technikami, rytuałami i umiejętnościami związanymi z wytwarzaniem oprogramowania — takimi jak kontrola wersji, testowanie automatyczne, potoki wdrożeń, zarządzanie konfiguracją i automatyzacja. Znajomość technik wytwarzania oprogramowania pomaga inżynierom operacyjnym w pracy w sytuacji, w której coraz więcej strumieni wartości technicznych przyjmuje zasady i wzorce DevOps.

Mimo że perspektywa uczenia się czegoś nowego może być zastraszająca lub może powodować u niektórych osób poczucie zażenowania czy wstydu, to nie powinno tak być. Karthik Gaekwad, który uczestniczył w transformacji DevOps w firmie National Instruments, powiedział: „Próba nauczenia się automatyzacji przez inżynierów operacyjnych nie jest czymś, czego należy się obawiać — wystarczy zapytać kolegę programistę, programiści uwielbiają pomagać”.

Pomóc w doskonaleniu umiejętności nauczania w codziennej pracy można za pomocą przeglądów kodu, w których uczestniczą obie strony. Dzięki temu uczymy się przez działanie, jak również dzięki wspólnej pracy inżynierów Dev i Ops przy rozwiązywaniu niewielkich problemów. Na przykład możemy poprosić inżyniera z działu Dev, aby wytłumaczył kolegom z działu Ops, w jaki sposób przeprowadzić uwierzytelnianie

w aplikacji i zalogować się w celu uruchomienia automatycznych testów różnych części aplikacji, aby sprawdzić, czy kluczowe komponenty działają prawidłowo (np. kluczowe funkcjonalności aplikacji, transakcje do bazy danych bądź kolejki komunikatów). Następnie należy zintegrować nowe automatyczne testy z potokiem wdrożeń i okresowo je uruchamiać, wysyłając wyniki do systemów monitorowania i ostrzegania. Dzięki temu można wcześniej wykryć sytuacje, w których krytyczne elementy zawodzą.

Jak zażartował Glenn O'Donnell z Forrester Research podczas swojej prezentacji DevOps Enterprise Summit 2014: „Dla wszystkich specjalistów technicznych, którzy kochają innowacje i kochają zmiany, istnieje znakomita i dynamiczna przyszłość”.

DZIELENIE SIĘ DOŚWIADCZENIAMI Z KONFERENCJI DEVOPS

W wielu organizacjach, w których dużą uwagę poświęca się obniżeniu kosztów, inżynierowie często są zniechęcani do uczestnictwa w konferencjach i uczenia się od współpracowników. Aby pomóc w budowaniu organizacji uczącej się, należy zachętać inżynierów (zarówno z działów Dev, jak i Ops) do udziału w konferencjach, wygłaszań na nich wykładów oraz gdy jest to konieczne, do samodzielnego tworzenia i organizowania wewnętrznych lub zewnętrznych konferencji.

Jedną z najbardziej dynamicznych seryjnych wydarzeń zorganizowanych przez społeczność jest konferencja DevOpsDays. Podczas ich trwania współdzielono i ogłoszono wiele praktyk DevOps. Konferencja jest darmowa lub prawie darmowa. Jest wspierana przez energiczną społeczność osób praktykujących DevOps oraz dostawców technologii.

Konferencję DevOps Enterprise Summit stworzono w 2014 r. dla liderów technicznych w celu wymiany doświadczeń w przyjęciu zasad i praktyk DevOps. Program jest zorganizowany przede wszystkim na bazie raportów z doświadczeń liderów technicznych podczas transformacji DevOps, jak również prezentacji ekspertów na tematy wybrane przez członków społeczności.

Studium przypadku

Wewnętrzne konferencje techniczne w firmach Nationwide Insurance, Capital One i Target (2014)

Oprócz uczestnictwa w zewnętrznych konferencjach wiele firm, w tym także te, które zostały opisane w tym studium przypadku, prowadzi wewnętrzne konferencje dla swoich pracowników technicznych.

Nationwide Insurance jest wiodącym dostawcą usług finansowych i ubezpieczeniowych. Działa w branżach ściśle regulowanych przepisami. Do szerokiej oferty firmy należą ubezpieczenia pojazdów i właścicieli nieruchomości. Firma należy do grupy najważniejszych dostawców planów emerytalnych sektora publicznego oraz ubezpieczeń zwierząt domowych. W roku 2014 r. firma miała 195 mld dolarów aktywów oraz przychód w wysokości 24 miliardów dolarów. Od 2005 roku firma Nationwide stosuje zasady Agile i Lean w celu poprawienia jakości praktyk stosowanych przez 5000 profesjonalistów technicznych oraz umożliwienia wprowadzania oddolnych innowacji.

Steve Farley, wiceprezes ds. technologii informatycznych, przypomina: „W tamtym czasie zaczęły pojawiać się ekscytujące konferencje techniczne, takie jak krajowa konferencja Agile. W 2011 roku kierownictwo technologiczne w firmie Nationwide uzgodniło, że należy utworzyć konferencję techniczną o nazwie TechCon. Przez utrzymywanie tego wydarzenia chcemy stworzyć lepszy sposób wzajemnego nauczania, a także zagwarantować, aby wszyscy uczestnicy znali kontekst firmy Nationwide — w przeciwieństwie do wysyłania wszystkich na konferencje zewnętrzne”.

Capital One, jeden z największych banków w USA z ponad 298 miliardami dolarów aktywów i 24 miliardami dolarów przychodów, w 2015 roku przeprowadził pierwszą konferencję poświęconą inżynierii oprogramowania, częściowo w ramach dążenia do utworzenia organizacji technicznej światowej klasy. Misją konferencji było promowanie kultury współdzielenia i współpracy, zbudowanie relacji pomiędzy profesjonalistami technicznymi i stworzenie warunków do uczenia się. Konferencja obejmowała 13 ścieżek nauki oraz 52 sesje. Uczestniczyło w niej 1200 wewnętrznych pracowników.

Dr Tapabrata Pal, posiadacz tytułu *technical fellow* w Capital One i jeden z organizatorów imprezy, opisuje: „Mieliśmy nawet salę expo, gdzie było 28 stanowisk, na których zespoły Capital One prezentowały wszystkie doskonałe funkcjonalności, nad którymi pracowały. Bardzo świadomie podjęliśmy decyzję, że nie będzie tam dostawców, ponieważ chcemy skupić się na celach firmy Capital One”.

Firma Target jest szóstym co do wielkości sprzedawcą detalicznym w USA, z 72 miliardami dolarów przychodów w 2014 r., 1799 sklepami detalicznymi i 347 000 pracowników na całym świecie. Heather Mickman, dyrektor działu rozwoju, i Ross Clanton od 2014 r. przeprowadzili sześć wewnętrznych wydarzeń DevOpsDays. Zyskali ponad 975 zwolenników wśród członków ich wewnętrznej społeczności uformowanej na wzór wydarzenia DevOpsDays przeprowadzonego w firmie ING w Amsterdamie w 2013 roku*.

* Nawiąsem mówiąc, pierwsze wewnętrzne wydarzenie DevOpsDays zostało zamodelowane po pierwszym wydarzeniu DevOpsDays w ING, którego organizatorami byli Ingrid Algra, Jan-Joost Bouwman, Evelijn Van Leeuwen i Kris Buytaert w 2013 r., po tym jak niektórzy członkowie zespołu ING wzięli udział w wydarzeniu DevOpsDays w Paryżu w 2013 roku.

Mickman i Clanton po powrocie z konferencji DevOps Enterprise Summit w 2014 zaczęli organizować własną konferencję. Zapraszają na nią wielu prelegentów z firm zewnętrznych, którzy mają okazję opisać swoje doświadczenia z przewodnictwa transformacjom. Clanton opisuje: „2015 był rokiem, w którym kierownictwo zwróciło na nas uwagę i kiedy nadaliśmy naszym działaniom rozped. Po tym wydarzeniu wiele osób przychodziło do nas z pytaniami o to, w jaki sposób mogłyby się zaangażować i jak mogłyby to nam pomóc”.

KORZYSTANIE Z WEWNĘTRZNYCH KONSULTACJI I TRENERÓW W CELU ROZPOWSZECHNIANIA PRAKTYK

Tworzenie organizacji z wewnętrznym systemem szkoleń i doradztwa to powszechnie stosowana metoda rozpowszechniania wiedzy w całej organizacji. Może to przyjąć wiele różnych form. W Capital One są wyznaczeni eksperci dziedzinowi, którzy w ustalonych godzinach udzielają porad, odpowiadają na pytania i pomagają wszystkim, którzy się do nich zwrócą.

We wcześniejszej części tej książki rozpoczęliśmy historię zespołu Testing Grouplet, który począwszy od 2005 roku, opracował w Google doskonałe praktyki testowania automatycznego. W tym rozdziale będziemy kontynuowali tę historię. Zespół ten dążył do poprawy stanu automatyzacji testów w całej firmie Google, korzystając z dedykowanych wydarzeń *improvement blitz*, wewnętrznych trenerów, a nawet własnego programu certyfikacji.

Bland powiedział, że w tamtym czasie wprowadzono w Google strategię 20% czasu na innowacje. Dzięki niej programiści mogli poświęcać średnio jeden dzień w tygodniu na projekty związane z Google poza ich podstawowym zakresem odpowiedzialności. Niektórzy inżynierowie zdecydowali się na stworzenie **groupletów** (ang. *grouplets*), czyli zespołów inżynierów o podobnych poglądach tworzonych ad hoc, którzy chcieli wykorzystać swoje 20% czasu na realizację ukierunkowanych przedsięwzięć *improvement blitz*.

Bharat Mediratta i Nick Lesiecki stworzyli grouplet zajmujący się testowaniem, a jego misją stało się kierowanie przyjęciem w firmie Google zwyczajów testowania automatycznego. Mimo że grouplet nie miał własnego budżetu ani formalnego kierownictwa, to jak opisywał Mike Bland: „Nikt nie nakładał na nas wyraźnych ograniczeń. I skorzystaliśmy z tego”.

Wykorzystano kilka mechanizmów, które miały poprawić akceptację praktyki testowania. Jedną z najbardziej znanych był tygodnik *Testing on the Toilet* (lub TotT) poświęcony testowaniu. Co tydzień w prawie każdej toalecie i w prawie wszystkich biurach firmy Google na całym świecie był publikowany biuletyn. Bland powiedział: „Naszym celem było podniesienie stopnia zaawansowania wiedzy o testowaniu w całej firmie. Wątpię, aby bazowanie wyłącznie na publikacji online zaangażowało ludzi w takim samym stopniu”.

Bland kontynuował: „Jeden z najważniejszych epizodów TotT nosił tytuł »Test Certified: Lousy Name, Great Results«^{*}, bo przedstawiono w nim dwie inicjatywy, które pozwoliły odnieść znaczący sukces w promowaniu stosowania testowania automatycznego”.

Test Certified (TC) dostarczył mapy drogowej, która umożliwiła poprawę stanu testowania automatycznego. Zgodnie z tym, co opisuje Bland: „Celem było zmodyfikowanie priorytetów skoncentrowanej na pomiarach kultury Google... i pokonanie pierwszych, przerażających przeskód braku wiedzy o tym, gdzie lub od czego zacząć. Poziom 1. polegał na szybkim ustaleniu podstawowych parametrów, poziom 2. to ustanowienie zasad i osiągnięcie celu: pokrycie kodu automatycznymi testami, a poziom 3. to dążenie do utrzymania celu długoterminowo”.

Druga inicjatywa polegała na dostarczeniu mentorów podejścia *Test Certified* do wszystkich zespołów, które oczekiwali porad lub pomocy, oraz tzw. *Test Mercenaries* — dosł. „najemnicy testowania” (tzn. pełnoetatowy zespół wewnętrznych konsultantów i trenerów) — do pracy z zespołami w celu poprawy stosowanych praktyk testowych i jakości kodu. „Najemnicy” robili to, stosując wiedzę, narzędzia i techniki groupletu »Testing Grouplet« do kodu zespołu z wykorzystaniem TC zarówno w roli przewodnika, jak i celu. Bland ostatecznie został liderem zespołu *Testing Grouplet* w latach 2006 – 2007 i członkiem zespołu *Test Mercenaries* w latach 2007 – 2009.

Bland kontynuował: „Naszym celem było, aby wszystkie zespoły osiągnęły 3. poziom programu TC, niezależnie od tego, czy zostali objęci naszym programem, czy nie. Ponadto ściśle współpracowaliśmy z wewnętrznymi zespołami pracującymi nad narzędziami testowymi, dostarczając opinii na temat pracy pełnej wyzwań związanych z testowaniem. Byliśmy żołnierzami na pierwszej linii, stosowaliśmy narzędzia, które budowaliśmy, i ostatecznie udało się nam wyeliminować popularne usprawiedliwienie »nie mam czasu na testowanie«”.

Dalej Bland mówił: „Poziomy TC wykorzystywały obowiązującą w Google kulturę bazowania na parametrach — trzy poziomy testowania były czymś, o czym ludzie mogli rozmawiać i czym mogli się chwalić podczas spotkań poświęconych przeglądowi wydajności. Zespół *Testing Grouplet* ostatecznie otrzymał budżet na *Test Mercenaries* — etatowy zespół wewnętrznych konsultantów. To był ważny krok, ponieważ teraz zarządzanie było »w pełni na pokładzie« — jego praca nie była usankcjonowana edyktami, ale rzeczywistym budżetem”.

Inną ważną konstrukcją było wykorzystywanie w całej firmie wydarzeń *improvement blitz* znanych pod nazwą „Fixit”. Bland opisuje wydarzenia Fixit w następujący sposób: „Sesje, podczas których zwykli inżynierowie z pomysłem i poczuciem misji wykorzystują całą wiedzę inżynierską w Google w celu przeprowadzenia jednodniowych,

* „Poświadczone testami: kiepska nazwa, świetne rezultaty” — przyp. tłum.

intensywnych sprintów reformowania kodu i korzystania z narzędzi”. Bland zorganizował cztery wydarzenia Fixit obejmujące całą firmę, z których dwa były poświęcone wyłącznie testowaniu, natomiast dwa pozostałe w większym stopniu były skoncentrowane na narzędziach. W ostatnim wzięło udział ponad 100 ochronników z 20 biur w 13 krajach. Od 2007 do 2008 roku Bland przewodniczył również zespołowi Fixit Grouplet.

Z wydarzeń Fixit, zgodnie z tym, co opisywał Bland, wynika, że należy dostarczyć skoncentrowanych misji w krytycznych punktach czasu w celu wygenerowania entuzjazmu i energii, które pomagają osiągnąć „postęp w sztuce”. To pomaga dzięki dużym, widocznym wysiłkom osiągnąć nowy stan plateau podczas długoterminowej misji zmian w kulturze.

Wyniki stosowania kultury testowania są oczywiste — wystarczy wspomnieć chociażby o niesamowitych osiągnięciach firmy Google, co wielokrotnie było przedstawiane w tej książce.

PODSUMOWANIE

W tym rozdziale opisano sposób wykorzystania rytuałów, które pomagają wzmacnić świadomość, że ludzie uczą się przez całe życie oraz że cenniejsze są usprawnienia codziennej pracy niż codzienna praca sama w sobie. Robimy to poprzez zarezerwowanie czasu na spłacanie dłużu technicznego, tworzenie forów, które pozwalają wszystkim uczyć się i nauczać innych zarówno wewnętrz organizacji, jak i poza nią. Oprócz tego udostępniamy ekspertów pomagających wewnętrznym zespołom poprzez coaching bądź konsultacje lub nawet poprzez wyznaczenie godzin na udzielanie odpowiedzi na pytania.

Dzięki udzieleniu wzajemnej pomocy w uczeniu się na co dzień zdobywamy większą wiedzę i zwyciężamy z konkurencją, co w efekcie pozwala nam uzyskać przewagę na rynku. Ale także pomagamy sobie nawzajem — jako ludzie — w osiągnięciu pełnego potencjału.

PODSUMOWANIE CZĘŚĆ V

W części V omówiliśmy stosowanie praktyk, które pomagają stworzyć w organizacji kulturę uczenia się i eksperymentowania. Uczenie się na podstawie incydentów, tworzenie współdzielonych repozytoriów i dzielenie się wiedzą jest niezbędne podczas pracy w złożonych systemach. Pomaga przekształcić kulturę w bardziej sprawiedliwą, a systemy w bezpieczniejsze i bardziej elastyczne.

W części VI omówimy sposoby rozszerzania przepływu, mechanizmów odbierania informacji zwrotnych, uczenia się i eksperymentowania w celu osiągnięcia celów związanych z bezpieczeństwem informacji.

Część VI

*Zarządzanie zmianami
i zapewnienie zgodności
z przepisami*

Część VI *Wprowadzenie*

W poprzednich rozdziałach omawialiśmy sposoby zapewnienia szybkiego przepływu pracy — od pobrania kodu z repozytorium do wydania — a także tworzenie szybkiego przepływu informacji zwrotnych. Omówiliśmy rytuały kulturowe wzmacniające tempo uczenia się w organizacji oraz wykrywanie słabych sygnałów awarii, co pomaga tworzyć coraz bezpieczniejszy system pracy.

W części VI rozszerzymy te działania w taki sposób, aby nie tylko osiągać cele działań Dev i Ops, ale także cele InfoSec. To pomoże nam uzyskać wysoki stopień pewności co do poufności, integralności i dostępności usług i danych.

Zamiast sprawdzać zabezpieczenia w produkcie na końcu procesu, stworzymy zintegrowane mechanizmy kontroli bezpieczeństwa w codziennej pracy działów Dev i Ops. Dzięki temu praca nad zabezpieczeniami stanie się częścią codziennej pracy każdego z nas. W idealnej sytuacji ta praca powinna być zautomatyzowana i znaleźć swoje miejsce w potoku wdrożeń. Ponadto rozszerzymy stosowane ręczne praktyki: procesy akceptacji i zatwierdzania o zautomatyzowane mechanizmy — bazując mniej na takich dziedzinach, jak podział obowiązków — i procesy zatwierdzania zmian.

Dzięki zautomatyzowaniu tych działań możemy na żądanie wygenerować dowody potwierdzające skuteczność mechanizmów takim osobom, jak audytorzy, asesorzy lub dowolne inne osoby należące do strumienia wartości.

Ostatecznie nie tylko poprawiamy bezpieczeństwo, ale także tworzymy procesy, które są łatwiejsze do kontroli i które świadczą o skuteczności mechanizmów wspierających przestrzeganie przepisów i umów. Robimy to poprzez:

- stworzenie z bezpieczeństwa obowiązku wszystkich osób;
- zintegrowanie mechanizmów prewencyjnych ze wspólnym repozytorium kodu źródłowego;
- zintegrowanie zabezpieczeń z potokiem wdrożeń;
- zintegrowanie zabezpieczeń z mechanizmami telemetrycznymi w celu łatwiejszego wykrywania luk w systemie bezpieczeństwa i wykonywania działań naprawczych;
- ochronę potoku wdrożeń;
- zintegrowanie działań związanych z wdrażaniem z procesami zatwierdzania zmian;
- poleganie w mniejszym stopniu na podziale obowiązków.

Integrując zadania związane z bezpieczeństwem w codzienną pracę wszystkich osób i przekazując odpowiedzialność za bezpieczeństwo wszystkim pracownikom, pomagamy organizacji w osiągnięciu większego stopnia bezpieczeństwa. Lepsze bezpieczeństwo oznacza, że jesteśmy w stanie bronić się przed zagrożeniami dla naszych danych oraz zachowujemy czujność. Zapewniamy niezawodność i ciągłość działania biznesu oraz większą dostępność i łatwiejsze przywracanie usługi po wystąpieniu problemów. Ponadto jesteśmy zdolni do przewyciężenia niektórych problemów zabezpieczeń, zanim spowodują katastrofalne skutki, oraz potrafimy zwiększyć przewidywalność obsługiwanych systemów. I, co być może jest najważniejsze, potrafimy zabezpieczyć systemy i dane lepiej niż kiedykolwiek.

22

Bezpieczeństwo informacji jako codzienne zadanie każdego z nas

Jedno z najczęstszych zastrzeżeń do wdrażania zasad i wzorców DevOps brzmi: „Nie pozwalają nam na to zasady bezpieczeństwa informacji i zgodności z przepisami”. Tymczasem zasady DevOps mogą być jednym z lepszych sposobów integracji zasad bezpieczeństwa informacji z codzienną pracą wszystkich osób należących do technologicznego strumienia wartości.

Realizacja zadań Infosec w formie silosu poza działami Dev i Ops może być przyczyną wielu problemów. James Wickett, jeden z twórców narzędzia zabezpieczeń Gauntlet i organizator konferencji DevOpsDays Austin, a także konferencji Lonestar Application Security, zauważył:

W jednej z interpretacji DevOps mówi się o tym, że nurt wywodzi się z potrzeby wyzwolenia produktywności programistów, ponieważ po wzroście liczby programistów nie było wystarczającej liczby inżynierów operacyjnych do wdrożenia wszystkich wyników prac. Te braki są jeszcze większe w dziedzinie Infosec — współczynnik liczby inżynierów w działach Dev, Ops i Infosec w typowej organizacji technicznej zwykle wynosi 100:10:1. Ze względu na te niewielkie zasoby personalne bez automatyzacji i integracji zadań bezpieczeństwa w codzienną pracę działów Dev i Ops działania inżynierów Infosec ograniczają się tylko do sprawdzania zgodności z przepisami, co jest przeciwieństwem inżynierii bezpieczeństwa — a poza tym sprawia, że nikt nas nie lubi.

James Wickett i Josh Corman, byli CTO firmy Sonatype i szanowany naukowiec w dziedzinie bezpieczeństwa informacji, napisali o konieczności uwzględnienia celów bezpieczeństwa informacji w praktykach DevOps. Ten zbiór zasad i praktyk jest określany jako Rugged DevOps (dosł. „surowy DevOps”). Podobne pomysły zostały przedstawione przez dr. Tapabrata Pala, dyrektora i posiadacza tytułu Platform Engineering Technical Fellow w Capital One, oraz zespół w Capital One, którzy opisują swoje procesy jako *DevOpsSec*, a zadania Infosec są zintegrowane ze wszystkimi fazami SDLC. Historia związana z Rugged DevOps została opisana w książce Gene'a Kima, Paula Love'a i George'a Spafforda *Visible Ops Security*.

W poprzednich rozdziałach tej książki badaliśmy sposób pełnej integracji celów inżynierów QA i operacyjnych w całym technologicznym strumieniu wartości. W tym rozdziale opiszemy, jak w podobny sposób zintegrować z codziennymi zadaniami cele Infosec. Dzięki temu można zwiększyć produktywność programistów i inżynierów operacyjnych, a także podnosić poziom zaufania i bezpieczeństwa.

PREZENTOWANIE REALIZACJI ZADAŃ BEZPIECZEŃSTWA PODCZAS DEMONSTRACJI PRODUKTU NA KONIEC ITERACJI

Jednym z naszych celów jest zaangażowanie zespołów funkcjonalności w zadania Infosec na jak najwcześniejszym etapie cyklu życia oprogramowania w przeciwieństwie do angażowania się w te zadania głównie na końcu projektu. Zapraszamy inżynierów Infosec do demonstracji produktu na koniec każdego okresu rozwojowego. Dzięki temu można lepiej zrozumieć cele zespołu w kontekście celów organizacyjnych, obserwować implementacje na bieżąco oraz zapewnić wskazówki i informacje zwrotne w jak najwcześniejjszej fazie projektu — gdy jest najwięcej czasu i najwięcej swobody dokonywania korekt.

Justin Arbuckle, były główny architekt w GE Capital, powiedział: „Zauważliśmy, że w przypadku zadań bezpieczeństwa informacji i zgodności z przepisami blokady na koniec projektu były znacznie droższe niż na początku — a skutki blokad w dziedzinie Infosec należały do najbardziej dotkliwych. »Zgodność z przepisami przez demonstrację« stała się jednym z rytuałów, które stosujemy w celu przeniesienia złożonych zadań Infosec na wcześniejszą fazę procesu”.

Kontynuując tę myśl, stwierdził: „Dzięki uczestnictwu inżynierów Infosec w tworzeniu każdej nowej funkcjonalności potrafiliśmy znacznie zmniejszyć stopień wykorzystania statycznych list kontrolnych i w większym stopniu polegać na wiedzy inżynierów Infosec w całym procesie rozwoju oprogramowania”.

To pomogło organizacji osiągnąć jej cele. Snehal Antani, była CIO Enterprise Architecture w GE Capital Americas pisała, że trzema kluczowymi miarami biznesowymi były: „szybkość tworzenia oprogramowania (tzn. szybkość dostarczania funkcjonalności na rynek), niepowodzenia interakcji z klientami (tzn. awarie i błędy) oraz czas reakcji na zgodność z przepisami (tzn. czas, jaki upływa od żądania audytu do dostarczenia wszystkich informacji ilościowych i jakościowych wymaganych do spełnienia żądania)”.

Kiedy inżynierowie Infosec są częścią zespołu — nawet gdy są tylko na bieżąco informowani i obserwują proces — to uzyskują kontekst biznesowy potrzebny do podejmowania lepszych decyzji bazujących na analizie ryzyka. Ponadto inżynierowie Infosec są w stanie pomóc zespołom funkcji dowiedzieć się, co jest potrzebne do spełnienia celów związanych z bezpieczeństwem i zgodnością z przepisami.

UWZGLĘDNIENIE BEZPIECZEŃSTWA W ŚLEDZENIU DEFEKTÓW I SPOTKANIACH POST-MORTEM

Jeśli to możliwe, to wszystkie problemy zabezpieczeń należy rejestrować w tym samym systemie śledzenia pracy, który jest stosowany przez działy Dev i Ops. Dzięki temu można zagwarantować widoczność pracy oraz ustalanie priorytetów razem z pozostałymi zadaniami. Sposób ten bardzo różni się od tradycyjnego sposobu działania inżynierów Infosec polegającego na rejestrowaniu wszystkich luk w zabezpieczeniach w systemie **GRC** (ang. *governance, risk, and compliance* — dosł. „nadzór, ryzyko i zgodność”), do którego mają dostęp wyłącznie inżynierowie Infosec. Zamiast tego wszystkie wymagane zadania są rejestrowane w systemach wykorzystywanych przez działy Dev i Ops.

Podczas prezentacji na konferencji 2012 Austin DevOpsDays Nick Galbreath przez wiele lat stojący na czele działu bezpieczeństwa informacji w firmie Etsy opisywał podejście do kwestii zabezpieczeń w następujący sposób: „Wszystkie zadania związane z bezpieczeństwem zarejestrowaliśmy w systemie JIRA używanym przez wszystkich inżynierów podczas codziennej pracy i oznaczyliśmy je jako »P1« albo »P2«, co oznacza, że muszą być naprawione natychmiast lub do końca tygodnia, nawet wtedy, gdy problem dotyczy tylko aplikacji wykorzystywanej wewnętrznie”.

Ponadto stwierdził: „Zawsze, kiedy mieliśmy problem z zabezpieczeniami, prowadziliśmy analizę post-mortem, ponieważ mogła ona doprowadzić do lepszej edukacji inżynierów w zakresie przeciwdziałania zajściu podobnego zdarzenia ponownie w przyszłości, a także jest to fantastyczny mechanizm transferu wiedzy do naszych zespołów inżynierskich”.

INTEGRACJA ZAPOBIEGAWCZYCH MECHANIZMÓW KONTROLI ZABEZPIECZEŃ ZE WSPÓLNYMI REPOZYTORIAMI KODU ŹRÓDŁOWEGO ORAZ USŁUGAMI WSPÓŁDZIELONYMI

W rozdziale 20. stworzyliśmy współdzielone repozytorium kodu źródłowego, dzięki któremu każdy może łatwo odkrywać i wykorzystywać wspólną wiedzę w organizacji — nie tylko w odniesieniu do kodu, ale również zestawu narzędzi, potoku wdrożeń, standardów itp. W ten sposób każdy może skorzystać ze zbiorowego doświadczenia wszystkich osób w organizacji.

W tym rozdziale pokażemy, jak dodać do współdzielonego repozytorium kodu źródłowego różnego rodzaju mechanizmy lub narzędzia ułatwiające zapewnienie bezpieczeństwa aplikacjom i środowiskom. Dodamy biblioteki, które są wstępnie zatwierdzone przez dział Infosec jako te, które spełniają specyficzne cele Infosec — na przykład biblioteki i usługi związane z uwierzytelnianiem i szyfrowaniem. Ponieważ wszystkie osoby należące do strumienia wartości DevOps korzystają z systemu kontroli wersji do wszystkiego, co budują lub wspierają, to umieszczenie tam artefaktów związanych z bezpieczeństwem informacji znacznie ułatwia wpływanie na codzienną pracę działów Dev i Ops. Dzieje się tak dlatego, że wszystko, co tworzymy, staje się dostępne, możliwe do wyszukania i zdatne do wielokrotnego użycia. System kontroli wersji służy również jako mechanizm wielokierunkowej komunikacji, dzięki któremu wszyscy mają świadomość wprowadzanych zmian.

Jeśli mamy organizację scentralizowaną wokół współdzielonych usług, możemy również współpracować z nimi w celu utworzenia i obsługi współdzielonych platform dotyczących zabezpieczeń, takich jak uwierzytelnianie, autoryzacja, rejestrowanie, a także innych usług zabezpieczeń lub inspekcji potrzebnych działom Dev i Ops. Gdy inżynierowie używają jednej z tych wstępnie zdefiniowanych bibliotek lub usług, to nie muszą planować osobnych przeglądów zabezpieczeń projektu dla tego modułu. Mogą korzystać z udzielonych wskazówek dotyczących hartowania konfiguracji, ustawień zabezpieczeń bazy danych, długości kluczy itd.

Aby zwiększyć prawdopodobieństwo prawidłowego wykorzystywania dostarczonych usług i bibliotek, możemy przeprowadzić szkolenia z zabezpieczeń dla inżynierów działów Dev i Ops, jak również dokonać przeglądu tego, co już zostało zrobione. W ten sposób można zagwarantować prawidłową implementację celów zabezpieczeń — zwłaszcza w przypadku zespołów używających tego rodzaju narzędzi po raz pierwszy.

Ostatecznym celem jest dostarczenie bibliotek lub usług zabezpieczeń wymaganych przez wszystkie nowoczesne aplikacje lub środowiska — na przykład uwierzytelnianie użytkowników, zarządzanie hasłami, szyfrowanie danych itd. Ponadto możemy dostarczyć członkom zespołów Dev i Ops skutecznych ustawień konfiguracji,

specyficznych dla bezpieczeństwa poszczególnych komponentów wykorzystywanych w stosach aplikacji — na przykład do rejestracji, uwierzytelniania lub szyfrowania. Warto uwzględnić następujące elementy:

- Biblioteki kodu oraz zalecane konfiguracje (np. 2FA (biblioteki uwierzytelniania dwuskładnikowego), obsługa skrótów haseł bcrypt, rejestracja).
- Zarządzanie informacjami poufnymi (np. ustawienia połączenia, klucze szyfrowania) za pomocą takich narzędzi, jak Vault, sneaker, Keywhiz, credstash, Troussseau, Red October itp.
- Pakiety systemu operacyjnego i kompilacje (np. NTP do synchronizacji czasu, bezpieczne wersje OpenSSL z prawidłowymi konfiguracjami, OSSEC lub Tripwire do monitorowania integralności plików, konfiguracje syslog do zapewnienia logowania kluczowych informacji zabezpieczeń w skoncentrowanym stosie ELK — Elasticsearch, Logstash, Kibana).

Umieszczenie wszystkich tych elementów we wspólnym repozytorium kodu źródłowego pozwala wszystkim inżynierom na prawidłowe tworzenie i wykorzystywanie w aplikacjach i środowiskach standardów rejestracji i szyfrowania bez konieczności udziału inżynierów Infosec.

Inżynierowie Infosec powinni również współpracować z zespołami inżynierów Ops podczas tworzenia bazowych receptorów lub budowania obrazu systemu operacyjnego, baz danych i innych elementów infrastruktury (np. NGINX, Apache, Tomcat). Powinni przy tym pokazywać, że komponenty te są w znany, bezpiecznym stanie i nie wprowadzają zagrożeń. Wspólne repozytorium nie tylko staje się miejscem, skąd można pobrać najnowsze wersje kodu, ale również miejscem, gdzie możemy współpracować z innymi inżynierami w zakresie monitorowania i zgłoszenia zmian wprowadzanych w modułach wrażliwych z punktu widzenia zabezpieczeń.

INTEGRACJA ZABEZPIECZEŃ Z POTOKIEM WDROŻEŃ

W poprzednich epokach prace nad hartowaniem i zabezpieczaniem aplikacji rozpoczęły się przeglądami zabezpieczeń po zakończeniu prac rozwojowych nad aplikacją. Wynikiem tego przeglądu był zazwyczaj liczący kilkaset stron dokument PDF z listą słabych punktów, który przekazywano działom Dev i Ops. Problemy wskazywane w tym dokumencie często pozostawały bez żadnej reakcji ze względu na presję terminów albo ze względu na to, że usterki zostały znalezione w cyklu życia oprogramowania zbyt późno na to, aby można je było łatwo skorygować.

W tym kroku zautomatyzujemy jak największą liczbę testów bezpieczeństwa informacji, tak aby były one uruchamiane razem ze wszystkimi innymi zautomatyzowanymi testami w potoku wdrożeń. Najlepiej by było, aby automatyczne testy były automatycznie inicjowane po każdym zaewidencjonowaniu zmian w repozytorium

kodu przez inżynierów Dev lub Ops i nawet w najwcześniejszych fazach projektu oprogramowania.

Celem jest zapewnienie działom Dev i Ops szybkich informacji zwrotnych na temat ich pracy, tak aby były powiadamiane po wprowadzeniu zmian, które są potencjalnie niebezpieczne. W ten sposób włączamy inżynierów Dev i Ops do procesu szybkiego wykrywania i rozwiązywania problemów zabezpieczeń w ramach codziennej pracy, co sprzyja uczeniu się i zapobiega popełnianiu podobnych błędów w przyszłości.

W idealnej sytuacji te zautomatyzowane testy zabezpieczeń powinny być uruchamiane w potoku wdrożeń razem z innymi narzędziami do statycznej analizy kodu.

Narzędzia takie jak Gauntlet zostały zaprojektowane z myślą o możliwości integracji z potokami wdrożeń. Pozwalają one na uruchamianie automatycznych testów zabezpieczeń dotyczących aplikacji, zależności, środowiska itp. Zadziwiające jest to, że Gauntlet pozwala na pisanie testów zabezpieczeń w skryptach testów zapisanych z wykorzystaniem składni Gherkin — powszechnie używanej przez programistów do pisania testów jednostkowych i funkcjonalnych. W ten sposób testy zabezpieczeń są pisane za pomocą frameworka znajomego dla programistów. Dzięki temu mogą być uruchamiane w potoku wdrożeń po zaewidencjonowaniu każdej zmiany równie łatwo, jak programy do statycznej analizy kodu, programy sprawdzające wrażliwe zależności lub testy dynamiczne. Podobne możliwości zapewnia Jenkins (rysunek 43).

Jenkins					
S	W	Name	Last Success	Last Failure	Last Duration
		Static analysis scan	7 days 1 hr - #2	N/A	6.3 sec
		Check known vulnerabilities in dependencies	N/A	7 days 1 hr - #2	1.6 sec
		Download and unit test	7 days 1 hr - #2	N/A	32 sec
		Scan with OWASP ZAP	7 days 1 hr - #2	N/A	4 min 43 sec
		Start	7 days 1 hr - #2	N/A	5 min 46 sec
		Virus scanning	7 days 1 hr - #2	N/A	4.7 sec

Rysunek 43. Jenkins z uruchomionymi automatycznymi testami zabezpieczeń

(źródło: James Wicket i Gareth Rushgrove, Battle-tested code without the battle — prezentacja na konferencji Velocity 2014 wysłana do Speakerdeck.com, 24 czerwca 2014, <https://speakerdeck.com/garethr/battle-tested-code-without-the-battle>)

W ten sposób zapewniamy wszystkim w strumieniu wartości jak najszybsze opinie na temat zabezpieczeń tworzonych elementów. Dzięki temu inżynierowie Dev i Ops mogą szybko znaleźć i naprawić występujące problemy.

ZAPEWNIENIE BEZPIECZEŃSTWA APLIKACJI

Testowanie w dziale Dev bardzo często koncentruje się na poprawności funkcjonalności oraz przepływu logiki. Ten typ testowania często jest określany jako tzw. **ścieżka szczęśliwa**, która sprawdza poprawne działania użytkowników (a czasami ścieżki alternatywne), gdzie wszystko idzie zgodnie z oczekiwaniami, bez wyjątków i błędów.

Z drugiej strony osoby praktykujące zadania QA, Infosec i Fraud często skupiają się na **ścieżkach smutnych**, które zdarzają się w przypadku, gdy coś idzie nie tak — zwłaszcza w odniesieniu do błędów związanych z bezpieczeństwem (tego rodzaju błędy związane z bezpieczeństwem często są żartobliwie określane jako **ścieżki złe**).

Załóżmy, że mamy witrynę e-commerce z formularzem wejściowym do wprowadzania danych o kartach kredytowych w ramach generowania zamówień klienta. Chcemy zdefiniować wszystkie ścieżki smutne i złe wymagane do zagwarantowania odrzucenia wszystkich oszustw i eksploitów bezpieczeństwa, takich jak wstrzykiwanie SQL, przepełnienia bufora, oraz zapobiec innym niepożądanym efektom.

Zamiast przeprowadzać te testy ręcznie, w idealnej sytuacji należałoby wygenerować je w ramach automatycznych testów jednostkowych lub funkcjonalnych, tak aby mogły być stale uruchamiane w potoku wdrożeń. W ramach testowania należy uwzględnić następujące przedsięwzięcia:

- **Analiza statyczna** — testy, które są wykonywane w środowisku innym niż runtime — najlepiej w potoku wdrożeń. Zazwyczaj narzędzie analizy statycznej sprawdza kod programu pod kątem wszystkich możliwych zachowań w fazie działania programu i wyszukuje braki w kodzie, tzw. tylne wejścia (ang. *backdoors*) i potencjalnie złośliwy kod (taki sposób testowania czasami jest nazywany „testowaniem od wewnętrz na zewnętrz”). Przykłady narzędzi wykorzystywanych do tego celu to Brakeman, Code Climate oraz wyszukiwanie niedozwolonych funkcji w kodzie (np. instrukcji `exec()`).
- **Analiza dynamiczna** — w przeciwieństwie do testowania statycznego analiza dynamiczna obejmuje testy uruchamiane w czasie pracy programu. Testy dynamiczne monitorują takie elementy, jak pamięć systemowa, zachowania funkcjonalne, czas reakcji, oraz ogólną wydajność systemu. Ta metoda (czasami określana jako „testowanie z zewnątrz do wewnątrz”) przypomina to, jak złośliwy, zewnętrzny kod może współdziałać z aplikacją. Do przykładów narzędzi należą Arachni oraz OWASP ZAP (ang. *Zed Attack Proxy*)*. Niektóre rodzaje testów penetracyjnych również można przeprowadzić w sposób automatyczny.

* Projekt OWASP (ang. *Open Web Application Security Project*) to aplikacja non profit skoncentrowana na poprawie zabezpieczeń oprogramowania.

Takie testy należy uwzględnić w ramach analizy dynamicznej. Do ich wykonywania można korzystać z takich narzędzi jak Nmap i Metasploit. Automatyczne testy dynamiczne najlepiej przeprowadzać w fazie automatycznego testowania funkcjonalnego w potoku wdrożeń albo nawet poddawać im usługi podczas pracy w warunkach produkcyjnych. Aby zapewnić odpowiednią obsługę zabezpieczeń, narzędzia takie jak OWASP ZAP można skonfigurować tak, by atakowały usługi przez webowy serwer proxy oraz aby w ramach środowiska testowego kontrolowały ruch sieciowy.

- **Skanowanie zależności** — innym typem testów statycznych, które zazwyczaj wykonuje się w fazie komplikacji, wewnątrz potoku wdrożeń, jest inwentaryzacja wszystkich zależności od binariów i plików wykonywalnych oraz zadbanie o to, aby te zależności — nad którymi często nie mamy kontroli — były wolne od słabych punktów lub złośliwego kodu. Wśród dostępnych narzędzi można wymienić Gemnasium, wbudowany program audytu dla Ruby, Maven dla Javy oraz program do sprawdzania zależności OWASP.
- **Integralność kodu źródłowego i podpisywanie kodu** — wszyscy programiści powinni mieć własny kod PGP, najlepiej tworzony i zarządzany w takim systemie jak *keybase.io*. Wszystkie operacje ewidencjonowania w repozytorium kontroli wersji powinny być podpisane — można to łatwo skonfigurować za pomocą narzędzi open source, takich jak gpg i git. Ponadto wszystkie pakiety instalacyjne tworzone w procesie CI powinny być podpisane; podpisany powinien być również ich skrót w centralnej usłudze rejestracji do celów audytu.

Ponadto należy zdefiniować wzorce projektowe, które pomagają projektantom pisać kod, który przeciwdziała nadużyciom, na przykład wprowadzenie limitów szybkości dla usług i dezaktywacja przycisków zatwierdzania formularzy po ich kliknięciu. OWASP publikuje wiele przydatnych wskazówek, na przykład serię Cheat Sheet (dosł. „ściągawka”). Można tam znaleźć między innymi następujące informacje:

- jak przechowywać hasła;
- jak postępować w przypadku zapomnianych haseł;
- jak obsługiwać logowanie;
- jak zapobiec wrażliwości na ataki za pomocą skryptów krzyżowych (XSS).

Studium przypadku

Testowanie statycznych zabezpieczeń w serwisie Twitter (2009)

Prezentacja *10 Deploys per Day: Dev and Ops Cooperation at Flickr* autorstwa Johna Allspawa i Paula Hammonda stynie z wpływem, jaki wywarła na społeczność Dev i Ops w 2009 roku. Odpowiednikiem takiej prezentacji dla społeczności Infosec może być wykład Justina Collinsa, Alexa Smolenia i Neila Matatalla dotyczący transformacji związanej z obsługą zabezpieczeń w serwisie Twitter na konferencji AppSecUSA w 2012 roku.

Z powodu dynamicznego rozwoju serwis Twitter zmagał się z wieloma wyzwaniami. Przez wiele lat, gdy Twitter nie był w stanie sprostać zapotrzebowaniu użytkowników, wyświetlała się słynna strona z komunikatem o błędzie „Fail Whale”, przedstawiająca grafikę wieloryba podnoszonego przez osiem ptaków. Skala wzrostu liczby użytkowników zapierała dech w piersiach — od stycznia do marca 2009 roku liczba aktywnych użytkowników Twittera wzrosła z 2,5 do 10 milionów.

W tym okresie Twitter miał również problemy z zabezpieczeniami. Na początku 2009 roku miały miejsce dwa poważne incydenty związane z bezpieczeństwem. Najpierw w styczniu na Twitterze zhakowano konto @BarackObama. Następnie w kwietniu w wyniku przeprowadzenia siłowego (ang. *brute-force*) ataku słowniowego naruszono Twitterowe konta administracyjne. W wyniku tych wydarzeń komisja **FTC** (ang. *Federal Trade Commission*) orzekła, że Twitter wprowadzał w błąd swoich użytkowników, twierdząc, że ich konta były bezpieczne, i nakazała Twitterowi, aby podjął działania, które pozwoliłyby zapewnić zgodność serwisu z regułami FTC.

Zgodnie z nakazem w ciągu 60 dni Twitter miał ustanowić zbiór procedur, które miały obowiązywać przez następne 20 lat. Obejmowały one następujące przedsięwzięcia:

- Wyznaczenie pracownika lub pracowników odpowiedzialnych za plan ochrony informacji Twittera.
- Identyfikacja przewidywalnych zagrożeń, zarówno wewnętrznych, jak i zewnętrznych, które mogłyby prowadzić do incydentów włamań, oraz stworzenie i zaimplementowanie planu przeciwdziałania tym zagrożeniom*.

* Strategie zarządzania tymi zagrożeniami obejmują zapewnienie pracownikom szkolenia i odpowiedniego kierowania, przemyślenie projektu systemów informacyjnych włącznie z sieciami i oprogramowaniem oraz ustanowienie procesów do przeciwdziałania, wykrywania i reagowania na ataki.

- Utrzymanie prywatności informacji użytkowników — nie tylko z poziomu źródeł zewnętrznych, ale również wewnętrznych. Wykonanie zarysu możliwych źródeł weryfikacji i testów bezpieczeństwa oraz prawidłowości tych implementacji.

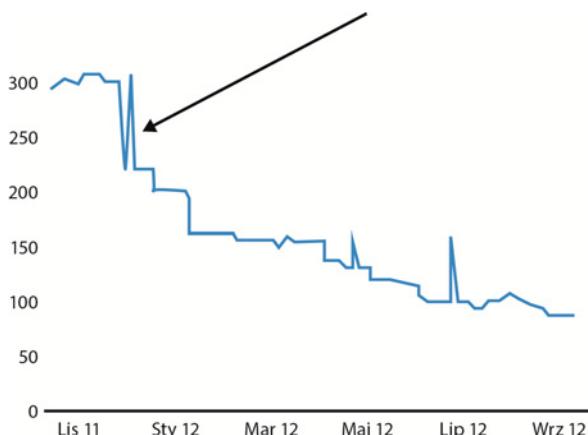
Grupa inżynierów przypisana do rozwiązymania tego problemu miała za zadanie włączenie zabezpieczeń do codziennej pracy działów Dev i Ops oraz zamknięcie luk w zabezpieczeniach, które pozwoliły na wystąpienie naruszeń.

We wcześniej wspomnianej prezentacji Collins Smolen i Matatall zidentyfikowali kilka problemów do rozwiązania:

- **Uniemożliwienie powtarzania błędów dotyczących bezpieczeństwa** — okazało się, że stale naprawiano te same defekty i słabe punkty. Trzeba było zmodyfikować system pracy i narzędzia automatyzacji, aby zapobiec powtarzaniu się tych samych problemów.
- **Integracja celów zabezpieczeń w istniejących narzędziach dla programistów** — autorzy prezentacji szybko doszli do wniosku, że głównym źródłem luk w zabezpieczeniach były problemy z kodem. Nie było sensu uruchamiać narzędzi, które wygenerowałoby olbrzymi raport PDF, i następnie przesyłać go do kogoś z działów Dev lub Ops. Zamiast tego programista, który napisał kod ze słabym punktem, powinien otrzymać dokładne informacje potrzebne do rozwiązania problemu.
- **Utrzymanie zaufania do działu Dev** — należało zdobyć i utrzymać zaufanie do działu Dev. Oznaczało to, że była potrzebna świadomość przesyłania do działu Dev fałszywych alarmów, tak aby można było naprawić błąd będący przyczyną fałszywego alarmu bez niepotrzebnego marnotrawienia czasu przez dział Dev.
- **Utrzymanie szybkiego przepływu przez Infosec dzięki automatyzacji** — nawet wtedy, gdy skanowanie luk w zabezpieczeniach w kodzie byłoby zautomatyzowane, inżynierowie Infosec nadal byli zmuszeni do wykonywania mnóstwa pracy ręcznej i długiego czekania. Musieli czekać na zakończenie skanowania, przeglądać ogromne stosy raportów, interpretować je, a następnie znaleźć osoby odpowiedzialne za naprawienie błędów. A po wprowadzeniu zmian wszystko musiało być wykonane od początku. Dzięki automatyzacji ręcznej pracy wykonywali mniej głupich zadań w rodzaju „naciśnij klawisz”, co pozwalało na więcej kreatywności i rzeczowej oceny.
- **Przekształcenie jak największej liczby zadań związanych z zabezpieczeniami w czynności samoobsługowe** — inżynierowie Infosec powinni ufać, że większość ludzi chce postępować uczciwie, dlatego konieczne jest dostarczenie całego kontekstu i wszystkich informacji niezbędnych do rozwiązyania powstałych problemów.
- **Holistyczne podejście do osiągnięcia celów Infosec** — celem było prowadzenie analizy z uwzględnieniem wszystkich perspektyw: kodu źródłowego, środowiska produkcyjnego, a nawet tego, co widzą klienci.

Pierwszy wielki przełom w zespole Infosec nastąpił podczas obejmującego całą firmę tygodnia hakowania, gdy zajmowano się integracją statycznej analizy kodu w proces komplikacji Twittera. Zespół skorzystał z narzędzia Brakeman, które skanuje aplikacje Ruby on Rails pod kątem występowania luk w zabezpieczeniach. Celem było zintegrowanie skanowania zabezpieczeń na jak najwcześniejszych etapach procesu rozwoju, a nie tylko w momencie ewidencjonowania kodu do repozytorium kodu źródłowego.

Wyniki integracji testowania zabezpieczeń z procesem rozwoju zapierały dech w piersiach. Z biegiem lat dzięki stworzeniu szybkiej pętli zwrotnej dla programistów, gdy pisali kod ze słabymi punktami, i pokazywaniu sposobu naprawiania luk, co było możliwe dzięki narzędziu Brakeman, liczba znajdowanych słabych punktów zmniejszyła się o 60%, co widać na rysunku 44 (gwałtowne skoki zwykle są związane z nowymi wydaniami programu Brakeman).



Rysunek 44. Liczba luk w zabezpieczeniach wykrytych przez program Brakeman

Te studia przypadków pokazują potrzebę integracji narzędzi dotyczących zabezpieczeń i praktyk DevOps w codzienną pracę i udowadniają, jak mogą być skuteczne. To zmniejsza zagrożenia związane z zabezpieczeniami, zmniejsza prawdopodobieństwo występowania luk w systemie i pomaga uczyć programistów pisania bezpieczniejszego kodu.

ZAPEWNIENIE BEZPIECZEŃSTWA ŁAŃCUCHA DOSTAW OPROGRAMOWANIA

Josh Corman zauważał: „My, programiści, już nie piszemy spersonalizowanego oprogramowania — zamiast tego montujemy to, czego potrzebujemy, z open source’owych części, które stały się łańcuchem dostaw oprogramowania, na który jesteśmy zdani”. Innymi słowy, kiedy używamy komponentów lub bibliotek — komercyjnych lub

open source — w oprogramowaniu, to nie tylko dziedziczymy ich funkcjonalność, ale również wszelkie zawarte w nich luki w zabezpieczeniach.

Przy wyborze oprogramowania potrafimy wykryć, kiedy projekty bazują na komponentach lub bibliotekach ze znanyymi lukami, i możemy pomóc programistom wybrać komponenty świadomie i z należytą starannością, wskazując te składniki (np. projekty open source), które charakteryzują się historią potwierdzającą szybkie eliminowanie luk w oprogramowaniu. Szukamy również wielu wersji tej samej biblioteki, używanej w środowisku produkcyjnym, w szczególności pod kątem występowania starszych wersji bibliotek, zawierających znane luki w zabezpieczeniach.

Badanie naruszeń prywatności danych posiadaczy kart pokazuje, jak ważne może być bezpieczeństwo komponentów typu open source, które wybieramy. Od 2008 r. coroczny raport **DBIR** (ang. *Data Breach Investigation Report*), przygotowywany przez Verizon PCI, jest jednym z najbardziej autorytatywnych głosów dotyczących naruszeń zabezpieczeń w przypadkach, w których doszło do utraty lub kradzieży danych posiadaczy kart. W raporcie za rok 2014 zbadano ponad 85 000 naruszeń, aby lepiej zrozumieć, skąd pochodziły ataki, w jaki sposób dochodziło do kradzieży i jakie czynniki się do tego przyczyniły.

Z raportu DBIR wynika, że w 2014 roku 10 luk (tzw. CVE) było użytych w prawie 97% eksplotach występujących w zbadanych naruszeniach danych posiadaczy kart.

W raporcie *2015 Sonatype State of the Software Supply Chain Report* dokonano dokładnej analizy danych słabych punktów z centralnego repozytorium Nexus. W 2015 roku to repozytorium dostarczało artefaktów komplikacji dla ponad 605 000 projektów open source, obsługiwało ponad 17 miliardów żądań pobrania artefaktów i zależności przede wszystkim dla platformy Javy, pochodzących ze 106 000 organizacji.

Raport zawierał zaskakujące wyniki:

- Typowa organizacja bazowała na ponad 7601 artefaktach komplikacji (np. dostawcach oprogramowania lub komponentów) i używała 18 614 różnych wersji (tzn. komponentów oprogramowania).
- Spośród używanych komponentów 7,5% miało znane luki w zabezpieczeniach, a ponad 66% z tych luk miało ponad dwa lata i pozostawało bez rozwiązania.

Ostatnią statystykę potwierdzają inne badania bezpieczeństwa informacji, przeprowadzone przez dr. Dana Geera i Joshua Cormana, które wykazały, że spośród projektów typu open source ze znanyimi lukami w zabezpieczeniach, zarejestrowanych w bazie danych National Vulnerability Database, tylko 41% luk kiedykolwiek zostało naprawionych, a średni czas do opublikowania poprawki wynosił 390 dni. W przypadku luk w zabezpieczeniach oznaczonych najwyższym poziomem ważności (tzn. luk poziomu 10. według CVSS) opublikowanie poprawek wymagało 224 dni*.

* Wśród narzędzi, które mogą pomóc w zapewnieniu integralności zależności oprogramowania, można wymienić OWASP Dependency Check i Sonatype Nexus Lifecycle.

ZAPEWNIENIE BEZPIECZEŃSTWA ŚRODOWISKA

W tym kroku należy zrobić wszystko, co jest potrzebne do zagwarantowania środowiskom stanu wzmacnienia i redukcji zagrożeń. Chociaż znane, dobre konfiguracje być może zostały stworzone już wcześniej, to należy wprowadzić mechanizmy monitorowania, tak aby wszystkie egzemplarze produkcyjne odpowiadały tym znanym, dobrym stanom.

Aby to zrobić, należy wygenerować testy automatyczne, które sprawdzą, czy wszystkie właściwe ustawienia zostały poprawnie zastosowane w celu zahartowania konfiguracji, ustawień zabezpieczeń bazy danych, długości kluczy itd. Ponadto należy skorzystać z testów do skanowania środowiska pod kątem występowania znanych luk w zabezpieczeniach*.

Inną kategorią weryfikacji zabezpieczeń jest zrozumienie rzeczywistych środowisk (tzn. „jakie są faktycznie”). Przykładem narzędzi służących do tego celu są: Nmap, który pozwala zagwarantować otwarcie wyłącznie oczekiwanych portów, i Metasploit, który pozwala sprawdzić, czy odpowiednio wzmacniliśmy środowiska przed znanymi zagrożeniami (np. skanowanie połączone z atakami wstrzykiwania SQL). Dane wyjściowe z tych narzędzi należy wprowadzić do repozytorium artefaktów i porównać z poprzednimi wersjami w ramach procesu testowania funkcjonalności. Takie działanie pomoże nam wykryć wszelkie niepożądane zmiany jak najszybciej po ich wystąpieniu.

Studium przypadku

Automatyczne zapewnienie zgodności z przepisami przez zespół 18F dla rządu federalnego z wykorzystaniem narzędzia Compliance Masonry

Agencje rządu federalnego USA w 2016 roku wydały na zadania związane z IT prawie 80 mld dolarów, wspierając misję wszystkich agencji wykonawczych. Niezależnie od agencji przeniesienie dowolnego systemu ze stanu „zakończony rozwój” do stanu „aktywne w produkcji” wymaga uzyskania zgody na użytkowanie (ang. *Authority to Operate — ATO*) od wyznaczonego organu zatwarzającego (ang. *Designated Approving Authority — DAA*). Zasady, które rządzą zgodnością z przepisami w agencjach rządowych, składają się z dziesiątek dokumentów, które razem liczą ponad 4000 stron zapełnionych takimi akronimami, jak FISMA, FedRAMP czy FITARA. Nawet w przypadku systemów, które wymagają niskiego poziomu poufności, integralności i dostępności, trzeba zaimplementować,

* Narzędzia, które mogą pomóc podczas testowania poprawności zabezpieczeń (tzn. „jak być powinno”), obejmują zautomatyzowane systemy zarządzania konfiguracjami (np. Puppet, Chef, Ansible, Salt), jak również takie narzędzia, jak ServerSpec oraz programy należące do Netflix Simian Army (np. Conformity Monkey, Security Monkey itd.).

udokumentować i przetestować ponad 100 mechanizmów. Uzyskanie ATO zwykle zajmuje od 8 do 14 miesięcy od momentu osiągnięcia stanu „prace rozwojowe zakończone”.

Zespół 18F, należący do działu General Services Administration w rządzie federalnym, podjął wielokierunkowe podejście do rozwiązymania tego problemu. Mike Bland wyjaśnia: „Zespół 18F został stworzony w General Services Administration w celu wykorzystania impetu powstałego podczas przywracania serwisu *Healthcare.gov* do zreformowania sposobu budowania i kupowania oprogramowania przez agencje rządowe”.

Jednym z efektów wysiłków 18F jest system rodzaju Paas (platforma jako usługa) o nazwie *Cloud.gov*, utworzony z komponentów typu open source. *Cloud.gov* działa obecnie na bazie chmury AWS GovCloud. Platforma nie tylko obsługuje wiele zadań operacyjnych, które bez niej musiałyby być zrealizowane przez zespoły dostaw, takich jak rejestrowanie, monitorowanie, alarmowanie oraz zarządzanie cyklem życia usług. Obsługiwała także większość zadań związanych z zapewnieniem zgodności z przepisami. Dzięki uruchomieniu tej platformy zdecydowana większość mechanizmów, które muszą być zaimplementowane w systemach rządowych, może być załatwionych na poziomie infrastruktury i platformy. Pozostają do sprawdzenia i udokumentowania pozostałe mechanizmy — te, które należą do warstwy aplikacji. W ten sposób znacznie zmniejszą się obciążenia związane ze zgodnością z przepisami oraz czas potrzebny do otrzymania ATO.

Chmura AWS GovCloud już została zatwierdzona do użytku przez systemy rządu federalnego wszystkich typów, w tym tych, które wymagają wysokiego poziomu poufności, integralności i dostępności. Oczekuje się, że do czasu, kiedy ta książka trafi w ręce Czytelników, platforma *Cloud.gov* zostanie zatwierdzona dla wszystkich systemów wymagających umiarkowanego poziomu poufności, integralności i dostępności*.

Ponadto zespół *Cloud.gov* buduje framework automatyzacji tworzenia planów ochrony systemu (SSP), które stanowią „kompleksowy opis architektury systemu, zaimplementowanych mechanizmów oraz ogólnej postawy w zakresie bezpieczeństwa, co często jest niezwykle złożone i ma objętość kilkuset stron”. Wspomniany zespół stworzył prototyp narzędzia o nazwie *compliance masonry*, dzięki któremu dane SSP są przechowywane w czytelnym dla maszyn formacie YAML, a następnie automatycznie konwertowane na dokumenty GitBook i pliki PDF.

Zespół 18F jest dedykowany do pracy w środowiskach otwartych. Publikuje swoje prace jako open source w domenie publicznej. Narzędzie *compliance masonry* oraz komponenty wchodzące w skład platformy *Cloud.gov* można znaleźć w repozytoriach GitHub zespołu 18F. Można nawet stworzyć własny egzemplarz platformy *Cloud.gov*. Prace nad otwartą dokumentacją SSP są realizowane w ścisłej współpracy ze społecznością OpenControl.

* Wspomniane zatwierdzenia są znane jako FedRAMP JAB P-ATO.

INTEGRACJA BEZPIECZEŃSTWA INFORMACJI Z TELEMETRIĄ PRODUKCJI

Marcus Sachs, jeden z naukowców Verizon Data Breach, zaobserwował w 2010 roku: „Rok za rokiem w znacznej większości naruszeń danych posiadaczy kart kredytowych organizacje wykrywały naruszenia kilka miesięcy lub kwartałów po ich wystąpieniu. Co gorsza, naruszenie zazwyczaj wykrywane było nie za pomocą wewnętrznego monitoringu, ale przez kogoś spoza organizacji — zwykle partnera biznesowego lub klienta, który zauważył fałszywe transakcje. Jednym z głównych powodów tego stanu rzeczy było to, że nikt w organizacji nie przeglądał regularnie plików dzienników”.

Innymi słowy, wewnętrzne mechanizmy bezpieczeństwa często są nieskuteczne w wykrywaniu naruszeń w sposób terminowy — albo z powodu „martwego pola” w systemie monitorowania, albo dlatego, że nikt z organizacji nie analizuje właściwych wskaźników telemetrycznych w codziennej pracy.

W rozdziale 14. omawialiśmy tworzenie w działach Dev i Ops kultury, zgodnie z którą wszystkie osoby należące do strumienia wartości tworzą telemetrię produkcji i parametry, udostępniają je innym w widocznych, publicznych miejscach, tak aby wszystkie osoby w organizacji mogły zobaczyć, w jaki sposób usługi działają w produkcji. Ponadto zbadaliśmy konieczność nieustannego wyszukiwania coraz słabszych sygnałów awarii, tak aby można było znaleźć i rozwiązać problemy, zanim spowodują katastrofalny błąd.

W tym podrozdziale pokażemy, jak zastosować w aplikacji i środowiskach mechanizmy monitorowania, rejestrowania i alarmowania wymagane do realizacji celów bezpieczeństwa informacji, a także jak zagwarantować ich właściwą centralizację w celu ułatwienia prostych i sensownych mechanizmów analizy i reagowania.

Można to zrobić poprzez zintegrowanie mechanizmów telemetrycznych do tych samych narzędzi, których używają inżynierowie działów Dev, QA i Ops, dając wszystkim osobom należącym do strumienia wartości wgląd w sposób działania aplikacji i środowisk w warunkach wrogiego otoczenia, w którym napastnicy stale próbują wykorzystywać luki w zabezpieczeniach, uzyskać nieautoryzowany dostęp, zainstalować tylne wejścia, popełniać oszustwa, wykonywać ataki DoS itd.

Propagując informacje na temat przewidywanych sposobów ataków na usługi w środowisku produkcyjnym, wzmacniamy poczucie obowiązku wszystkich osób zwracania uwagi na zagrożeniach dla bezpieczeństwa oraz projektowania środków zaradczych podczas codziennej pracy.

TWORZENIE W APLIKACJI TELEMETRII ZABEZPIECZEŃ

Aby wykrywać problematyczne działania użytkownika mogące świadczyć o oszustwach i nieautoryzowanym dostępie albo o powstawaniu sytuacji sprzyjających tego rodzaju naruszeniom, trzeba stworzyć w aplikacjach odpowiednie mechanizmy telemetryczne.

Oto przykłady:

- Udane i nieudane logowania.
- Resetowanie hasła użytkownika.
- Resetowanie adresu e-mail użytkownika.
- Zmiany danych karty kredytowej użytkownika.

Na przykład jako wczesny wskaźnik ataków siłowych zmierzających do uzyskania nieuprawnionego dostępu można wyświetlić wartość stosunku prób nieudanego logowania do logowania pomyślnego. Oczywiście należy również stworzyć mechanizmy ostrzegania o istotnych zdarzeniach, tak aby można było szybko wykryć i skorygować problemy.

TWORZENIE W ŚRODOWISKU TELEMETRII ZABEZPIECZEŃ

Oprócz instrumentacji aplikacji należy również utworzyć odpowiednią liczbę mechanizmów telemetrycznych w środowisku. Dzięki temu można wykrywać wczesne sygnały nieautoryzowanego dostępu — zwłaszcza w tych komponentach, które są uruchomione na bazie infrastruktury poza naszą kontrolą (np. środowiska hostingowe bądź w chmurze).

Potrzebne są mechanizmy monitorowania i potencjalnego alarmowania dotyczące następujących sytuacji:

- zmian w systemie operacyjnym (np. w produkcji lub w środowisku komplikacji);
- zmian w grupach zabezpieczeń;
- modyfikacji konfiguracji (np. OSSEC, Puppet, Chef, Tripwire);
- zmian w infrastrukturze chmury (np. VPC, grupach zabezpieczeń, użytkownikach i uprawnieniach);
- prób ataków XSS (czyli ataków z wykorzystaniem „skryptów krzyżowych”);
- prób SQLi (czyli ataków wstrzykiwania SQL);
- błędów serwera WWW (np. błędów 4XX oraz 5XX).

Trzeba również potwierdzić prawidłową konfigurację rejestrowania, tak aby wszystkie mechanizmy telemetryczne były wysyłane w odpowiednie miejsce. Po wykryciu ataku oprócz zarejestrowania, że taki atak miał miejsce, należy również zablokować dostęp i zapisać informacje dotyczące jego źródła, tak aby ułatwić wybór najlepszego działania zaradczego.

Studium przypadku

Instrumentacja środowiska w firmie Etsy (2010)

W 2010 roku Nick Galbreath był dyrektorem inżynierii w Etsy, odpowiedzialnym za bezpieczeństwo informacji, zapobieganie oszustwom i zapewnienie prywatności. Galbreath zdefiniował oszustwo jako sytuację, w której „system działa niepoprawnie, pozwala na wprowadzanie nieprawidłowych lub niekontrolowanych danych wejściowych, powodując straty finansowe, kradzież danych, przestoje, vandalizm lub atak na inny system”.

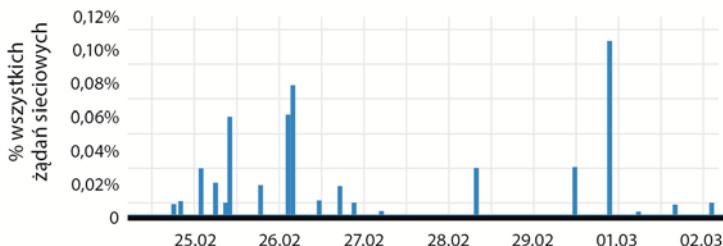
Aby osiągnąć postawione sobie cele, Galbreath nie stworzył osobnego działu kontroli oszustw lub bezpieczeństwa informacji. Zamiast tego rozmieścił te obowiązki wewnętrz strumienia wartości DevOps.

Galbreath stworzył mechanizmy telemetryczne związane z zabezpieczeniami, które były wyświetlane obok innych parametrów Dev i Ops i z którymi rutynowo miał styczność każdy inżynier w Etsy:

- **Nieprawidłowe zakończenia programów produkcyjnych (np. błędy segmentacji, błędy powodujące zrzut pamięci — ang. core dump itp.):** „Szczególnie interesowaliśmy się, dlaczego niektóre procesy powodują zrzuty pamięci w całym środowisku produkcji lub ciągle są inicjowane przez ruch pochodzący z jednego adresu IP. Podobne zainteresowanie wzbudzały w nas błędy HTTP typu »500 Internal Server Error«. Są to sygnały wykorzystywania luki w celu uzyskania nieautoryzowanego dostępu do systemów oraz pilnej konieczności zainstalowania łaty”.
- **Błąd składniowy bazy danych:** „Zawsze szukaliśmy błędów składni bazy danych wewnętrz kodu — takie błędy albo umożliwiałły przeprowadzanie ataków wstrzykiwania SQL, albo oznaczały właśnie trwające ataki. Z tego powodu stosowaliśmy zasadę »zero tolerancji dla błędów składniowych bazy danych w kodzie«, ponieważ były one jednymi z głównych kierunków ataków używanych do złamania naszych systemów”.
- **Oznaki ataków wstrzykiwania SQL:** „To był śmiesznie prosty test — inicjowaliśmy alert w sytuacji, gdy w polach wprowadzanych przez użytkownika pojawił się ciąg »UNION ALL«. To prawie zawsze wskazywało na atak wstrzykiwania SQL. Dodaliśmy również testy jednostkowe, których zadaniem było sprawdzenie, czy tego rodzaju niekontrolowane dane wejściowe użytkownika nigdy nie zostaną dopuszczone do zapytań do bazy danych”.

Wykres zamieszczony na rysunku 45 to przykład grafiki prezentowanej wszystkim programistom. Pokazano na nim liczbę ataków iniekcji SQL, które próbowano przeprowadzić w środowisku produkcyjnym. Jak zaobserwował Galbreath: „Nic tak nie pomaga zrozumieć programistom, jak wrogie jest środowisko operacyjne, jak obserwowanie swojego kodu, który jest atakowany w czasie rzeczywistym”.

Potencjalne iniekcje SQL



Rysunek 45. Programiści w Etsy mogą obserwować próby ataków iniekcji SQL w systemie Graphite (źródło: DevOpsSec: Applying DevOps Principles to Security, DevOpsDays Austin 2012, SlideShare.net, opublikowane przez Nicka Galbreatha, 12 kwietnia 2012, <http://www.slideshare.net/nickgsuperstar/devopssec-apply-devops-principles-to-security>)

Galbreath zauważył: „Jednym z wyników pokazywania tego wykresu było uświadomienie programistom, że byli atakowani przez cały czas! To było niesamowite, ponieważ doprowadziło do zmiany sposobu myślenia programistów o bezpieczeństwie ich kodu już podczas jego pisania”.

OCHRONA POTOKU WDROŻEŃ

Infrastruktura, która wspiera procesy ciągłej integracji i ciągłego wdrażania, prezentuje również nowy obszar narażony na ataki. Na przykład jeśli ktoś włamie się do serwerów, na których działa potok wdrożeń i gdzie są poświadczenia do systemu kontroli wersji, to może umożliwić komuś kradzież kodu źródłowego. Co gorsza, jeśli potok wdrożeń ma uprawnienia zapisu, to napastnik może również wstrzykiwać złośliwe zmiany do repozytorium kontroli wersji, a w związku z tym wstrzykiwać złośliwe zmiany do naszych aplikacji i usług.

Jak zaobserwował Jonathan Claudius, były starszy tester zabezpieczeń w TrustWave SpiderLabs: „Serwery ciągłej komplikacji i testowania są niesamowite i sam ich używam. Zacząłem jednak myśleć o wykorzystywaniu mechanizmów CI/CD do wstrzykiwania złośliwego kodu. Doprowadziło mnie to do pytania o to, gdzie jest dobre miejsce do ukrycia złośliwego kodu? Odpowiedź była oczywista: w testach jednostkowych. Nikt specjalnie nie ogląda testów jednostkowych, a są one uruchamiane za każdym razem, gdy ktoś ewidencjonuje kod w repozytorium”.

To pokazuje, że w celu właściwej ochrony integralności aplikacji i środowisk trzeba również zablokować kierunki ataku na potoku wdrożeń. Zagrożenie stwarzają programiści wprowadzający kod, który umożliwia nieautoryzowany dostęp (neutralizujemy je za pomocą takich mechanizmów, jak testowanie kodu, przeglądy kodu i testy penetracyjne), a także nieautoryzowani użytkownicy uzyskujący dostęp do kodu lub środowiska (to zagrożenie neutralizujemy dbaniem o to, aby konfiguracje były znymi, prawidłowymi stanami, oraz skutecznym instalowaniem łat).

Jednak w celu ochrony potoku wdrożeń obejmujących komplikację, integrację lub wdrożenia do stosowanych strategii neutralizowania zagrożeń możemy zaliczyć:

- Hartowanie serwerów ciągłego budowania i integracji oraz dbanie o możliwość odtworzenia ich w sposób zautomatyzowany, podobnie jak czynimy to w odniesieniu do infrastruktury obsługującej usługi produkcyjne, z którymi ma styczność klient. To pomaga zapobiec włamaniom do serwerów ciągłego budowania i integracji.
- Przeglądanie wszystkich zmian ewidencjonowanych w systemie kontroli wersji — albo poprzez programowanie w parach w momencie zatwierdzania, albo przez proces przeglądania kodu pomiędzy zaewidencjonowaniem a zintegrowaniem z gałęzią master — aby uniemożliwić uruchamianie niekontrolowanego kodu (np. testy jednostkowe mogą zawierać złośliwy kod, który umożliwia nieautoryzowany dostęp).
- Wyposażenie repozytorium w mechanizmy wykrywania prób ewidencjonowania w repozytorium kodu zawierającego podejrzane wywołania API (np. testy jednostkowe uzyskujące dostęp do systemu plików lub sieci). Takie pliki można na przykład poddać kwarantannie lub można zainicjować natychmiastowy przegląd kodu.
- Zadbanie o to, aby każdy proces CI działał we własnym odizolowanym kontenerze lub na maszynie wirtualnej.
- Zadbanie o to, aby poświadczenia kontroli wersji wykorzystywane przez system CI były tylko do odczytu.

PODSUMOWANIE

W tym rozdziale opisaliśmy sposoby integracji celów bezpieczeństwa informacji na wszystkich etapach codziennej pracy. Robimy to poprzez zintegrowanie mechanizmów kontroli bezpieczeństwa z instrumentami, które stworzyliśmy wcześniej. Dbanie o to, aby wszystkie środowiska tworzone na żądanie również znajdowały się w zahartowanym, zmniejszającym zagrożenia stanie — poprzez zintegrowanie testowania zabezpieczeń z potokiem wdrożeń oraz zapewnienie tworzenia wskaźników telemetrycznych zabezpieczeń w środowiskach przedprodukcyjnym i produkcyjnym. W ten sposób umożliwiamy podniesienie wydajności programistów i zadań operacyjnych, a jednocześnie poprawiamy ogólny poziom bezpieczeństwa. Następnym krokiem jest ochrona potoku wdrożeń.

23

Ochrona potoku wdrożeń

W tym rozdziale opowiemy o ochronie potoku wdrożeń, a także o tym, jak osiągnąć w środowisku cele związane z zabezpieczeniami i zgodnością z przepisami, włącznie z zarządzaniem zmianami i podziałem obowiązków.

INTEGRACJA MECHANIZMÓW ZABEZPIECZEŃ I ZGODNOŚCI Z PRZEPISAMI Z PROCESAMI ZATWIERDZANIA ZMIAN

W prawie każdej organizacji IT o znaczących rozmiarach istnieją procesy zarządzania zmianami, które są podstawowymi mechanizmami ograniczenia zagrożeń dla operacji i bezpieczeństwa. Menedżer zgodności z przepisami i menedżerowie zabezpieczeń w celu sprawdzania wymagań zgodności polegają na procesach zarządzania zmianami i zwykle wymagają dowodów na to, że wszystkie zmiany zostały odpowiednio autoryzowane.

Jeśli poprawnie zbudowaliśmy potok wdrożeń, tak aby wdrożenia wiązały się z niskim ryzykiem, to większość wprowadzanych zmian nie będzie powodować konieczności stosowania ręcznego procesu zatwierdzania zmian ze względu na wykorzystanie takich mechanizmów, jak testy automatyczne oraz proaktywne monitorowanie produkcji.

W tym kroku zrobimy wszystko to, co jest potrzebne do zagwarantowania skutecznej integracji mechanizmów bezpieczeństwa i zgodności z przepisami z dowolnym istniejącym procesem zarządzania zmianami. Skuteczne zasady zarządzania zmianami

uwzględniają fakt istnienia różnych zagrożeń powiązanych z różnego rodzaju zmianami oraz to, że zmiany te wymagają różnego traktowania. Procesy te są zdefiniowane w zasadach ITIL, które dzielą zmiany na trzy kategorie:

- **Zmiany standardowe** — zmiany obarczone niskim ryzykiem, zgodne z ustalonym i zatwierdzonym procesem. Mogą być również wstępnie zatwierdzone. Obejmują comiesięczne aktualizacje tabel podatkowych lub kodów krajów, zmiany treści i stylu witryny oraz niektóre rodzaje aplikacji lub latek systemów operacyjnych o zrozumiałym wpływie na inne komponenty. Wnioskodawca zmiany nie wymaga zatwierdzenia przed wdrożeniem zmiany. Wdrożenie zmiany można całkowicie zautomatyzować. Zmiany powinny być rejestrowane w celu zachowania zdolności śledzenia.
- **Zmiany normalne** — są to zmiany o wyższym ryzyku, które wymagają przeglądu lub zatwierdzenia przez ustalone organy zatwierdzające. W wielu organizacjach odpowiedzialność za tego rodzaju zatwierdzanie jest nieprawidłowo powierzona doradczej komisji zmian (ang. *change advisory board* — **CAB**) lub awaryjnej doradczej komisji zmian (ang. *emergency change advisory board* — **ECAB**). Członkom takiej komisji może brakować wiedzy potrzebnej do pełnego zrozumienia wpływu zmiany na system, co często prowadzi do niedopuszczalnie długich terminów realizacji. Problem ten jest szczególnie istotny w przypadku wdrożeń obszernych fragmentów kodu, które mogą zawierać setki tysięcy (lub nawet miliony) linii nowego kodu, ewidencjonowanego przez setki programistów w ciągu kilku miesięcy. W celu uzyskania autoryzacji dla normalnych zmian komisja CAB musi korzystać z dobrze zdefiniowanego formularza zmiany (ang. *request for change* — **RFC**), który określa rodzaj informacji potrzebnych do uzyskania pozytywnej bądź negatywnej decyzji dotyczącej wdrożenia zmiany. Formularz RFC zwykle zawiera pożądane wyniki biznesowe, planowane narzędzia i gwarancje*, biznesplan z zagrożeniami i alternatywami oraz proponowany harmonogram†.

* ITIL definiuje narzędzie (ang. *utility*) jako odpowiedź na pytanie „co usługa robi”, podczas gdy gwarancja (ang. *warranty*) jest definiowana jako odpowiedź na pytanie „jak usługa jest dostarczana oraz jak można ją wykorzystać w celu stwierdzenia, czy jest »zdatna do użytku«”.

† Aby dokładniej zarządzać ryzykiem związanym ze zmianami, możemy również zdefiniować reguły — na przykład wymaganie, aby określone zmiany mogły być wprowadzane tylko przez określoną grupę osób lub osobę (np. zmiany w schemacie baz danych mogą wprowadzać tylko administratorzy baz danych — **DBA**). Tradycyjnie spotkania komisji CAB odbywają się co tydzień. W czasie tych spotkań są zatwierdzane żądania zmiany oraz planowane ich wdrożenia. Począwszy od ITIL w wersji 3., zmiany mogą być akceptowane tylko w formie elektronicznej, na zasadzie *just-in-time* oraz za pośrednictwem narzędzia zarządzania zmianami. Zasada ITIL zaleca również: „W celu promowania wydajności standardowe zmiany

- **Zmiany pilne** — są to zmiany wprowadzane w warunkach awarii, a w konsekwencji są potencjalnie związane z wysokim ryzykiem. Muszą zostać wdrożone do produkcji natychmiast (np. pilne poprawki zabezpieczeń, usługa przywracania). Zmiany tego rodzaju często wymagają zgody kadry kierowniczej wyższego szczebla, ale dokumentacja może być wykonana po fakcie. Głównym celem stosowania praktyk DevOps jest usprawnienie procesu zwykłych zmian, tak aby mógł być stosowany również w odniesieniu do zmian pilnych.

ZMIANA KATEGORII WIĘKSZOŚCI ZMIAN NISKIEGO RYZYKA NA STANDARDOWE

W idealnej sytuacji samo posiadanie niezawodnego potoku wdrożeń powinno być równoznaczne z reputacją szybkich, niezawodnych i niedramatycznych wdrożeń. W tym momencie należy dążyć do uzyskania zgody z działu operacyjnego oraz odpowiednich władz autoryzujących zmiany, że proponowane zmiany wiążą się z ryzykiem na tyle niskim, że można je zdefiniować jako zmiany standardowe, wstępnie zatwierdzone przez komisję CAB. To pozwala wdrożyć je do produkcji bez konieczności dalszego zatwierdzania, choć nadal powinny być poprawnie zarejestrowane.

Jednym ze sposobów na poparcie twierdzenia, że zmiany wiążą się z niskim ryzykiem, jest zaprezentowanie historii zmian w znaczącym okresie (np. na przestrzeni kilku miesięcy lub kwartałów) i podanie pełnej listy problemów produkcyjnych w tym samym okresie. Jeśli możemy zaprezentować wysokie wartości współczynnika sukcesu zmian i niskie wartości współczynnika MTTR, to możemy przyjąć, że mamy środowisko kontroli zmian, które skutecznie zapobiega błędom wdrażania, jak również udowadniajemy zdolność do skutecznego i szybkiego wykrywania i korygowania powstających problemów.

Nawet gdy zmiany są sklasyfikowane jako standardowe, nadal muszą być widoczne i zarejestrowane w systemach zarządzania zmianami (np. Remedy albo ServiceNow). W idealnej sytuacji wdrożenia powinny być przeprowadzone automatycznie przez system zarządzania konfiguracją oraz narzędzia potoku wdrożeń (np. Puppet, Chef, Jenkins), a wyniki powinny być automatycznie zarejestrowane. Dzięki temu wszystkie osoby w organizacji (z DevOps oraz pozostałe) będą miały wgląd w realizowane zmiany, tak samo jak we wszystkie inne zmiany wprowadzane w organizacji.

Zapisy żądania zmian można automatycznie powiązać z określonymi elementami w narzędziach planowania pracy (np. JIRA, Rally, LeanKit, ThoughtWorks Mingle), co pozwala stworzyć szerszy kontekst dla wprowadzanych zmian — na przykład powiązać

powinny być identyfikowane na początku budowania procesu zarządzania zmianami. W przeciwnym razie implementacja zarządzania zmianami może tworzyć niepotrzebnie wysoki stopień zadań administracyjnych i obciążenie dla procesu”.

je z defektami funkcjonalności, zdarzeniami produkcyjnymi lub historyjkami użytkownika. Można to osiągnąć w prosty sposób poprzez uwzględnienie numerów zleceń roboczych (ang. *tickets*) z narzędzi planowania w komentarzach powiązanych z opercjami ewidencjonowania w repozytorium kontroli wersji*. W ten sposób możemy śledzić wdrożenia do produkcji w połączeniu ze zmianami w repozytorium kontroli wersji, a na tej podstawie wiązać je z ticketami z narzędzi planowania pracy.

Tworzenie możliwości identyfikacji i kontekstu powinno być łatwe. Nie powinno to powodować zbyt wielkiego obciążenia dla inżynierów. Powiązanie z historyjkami użytkownika, wymaganiami lub defektami prawie zawsze wystarcza — wszelkie dalsze szczegóły, takie jak otwarcie zlecenia roboczego dla każdej operacji ewidencjonowania w repozytorium kontroli wersji, prawdopodobnie nie będą przydatne, a tym samym staną się niepotrzebne i niepożądane, głównie ze względu na wywoływanie zakłóceń codziennej pracy.

CO ROBIĆ, GDY ZMIANY SĄ SKLASYFIKOWANE JAKO NORMALNE

Zmiany, których nie można sklasyfikować jako standardowe, należy uznać za **zmiany normalne**. Przed wdrożeniem wymagają one zgody co najmniej kilku osób należących do komisji CAB. W takim przypadku celem powinno być zadanie o szybkie wdrożenie nawet wtedy, gdy nie jest ono w pełni zautomatyzowane.

Należy wówczas zadbać o to, aby wszelkie przekazywane żądania zmian były jak najbardziej kompletne i dokładne. Komisja CAB powinna otrzymać wszystko, co jest potrzebne do prawidłowej oceny zmiany — ostatecznie, jeśli żądanie zmiany ma nieprawidłową formę lub jest niekompletne, to zostanie ono odesłane do autora, co wydłuża czas potrzebny do wdrożenia zmian w produkcji i podważa prawidłowość rozumienia celów procesu zarządzania zmianami.

Tworzenie kompletnych i dokładnych specyfikacji RFC prawie na pewno można zautomatyzować. W wyniku tego procesu zlecenie robocze jest wypełniane szczegółowo danymi na temat tego, co dokładnie ma zostać zmienione. Na przykład można automatycznie utworzyć zlecenie robocze zmiany ServiceNow, zawierające link do historyjki użytkownika JIRA wraz z manifestami komplikacji i wynikiem testów z narzędzia z potoku wdrożeń oraz linkami do uruchamianych skryptów Puppet (Chef).

Ponieważ zaewidencjonowane zmiany zostaną ręcznie ocenione przez ludzi, to jeszcze bardziej istotne jest opisanie kontekstu zmiany. Obejmuje to określenie powodu, dla którego wprowadzamy zmianę (np. podanie linku do funkcjonalności, defektów lub incydentów), osób, których dotyczy zmiana, oraz co się zmienia.

* Termin *ticket* — dosł. „bilet” — jest powszechnie używany jako określenie identyfikowalnego w unikatowy sposób zlecenia roboczego.

Celem jest współdzielenie dowodów i artefaktów, które dadzą nam pewność działania zmian w produkcji w sposób zgodny z projektem. Mimo że specyfikacje RFC zazwyczaj zawierają pola tekstowe bez struktury, to należy dostarczyć linki do danych możliwych do odczytania przez komputery (np. do plików JSON). Dzięki nim inni będą mogli zintegrować i przetworzyć nasze dane.

W wielu zestawach narzędzi można to zrobić w sposób zapewniający zgodność z przepisami i w pełni zautomatyzowany. Na przykład narzędzia firmy ThoughtWorks Mingle i Go potrafią automatycznie łączyć ze sobą takie informacje, jak lista naprawionych defektów oraz zaimplementowanych nowych funkcjonalności powiązanych ze zmianą, i umieścić je w RFC.

Po przesłaniu dokumentu RFC wskazani członkowie CAB zapoznają się ze zmianami, poddają je odpowiedniemu procesowi i zatwierdzą je tak samo jak inne przesłane żądania zmian. Jeśli wszystko pójdzie dobrze, osoby odpowiedzialne za zmiany docenią dokładność i szczegółowość przesyłanych zmian, ponieważ pozwoliliśmy im szybko zweryfikować poprawność dostarczonych informacji (np. przeglądanie linków do artefaktów z narzędzi należących do potoku wdrożeń). Naszym celem powinno być jednak stałe prezentowanie popartego przykładami dowodu udanych zmian, tak aby ostatecznie zdobyć uznanie członków CAB oraz akceptację faktu, że wprowadzane automatycznie zmiany mogą być bezpiecznie sklasyfikowane jako zmiany standardowe.

Studium przypadku

Automatyczne zmiany infrastruktury jako zmiany standardowe w firmie Salesforce.com (2012)

Firma Salesforce została założona w 2000 roku. Jej misją było dostarczenie funkcjonalności zarządzania relacjami z klientami w postaci usługi. Oferta firmy Salesforce została powszechnie przyjęta na rynku, co doprowadziło do udanego debiutu na giełdzie w 2004 roku. Do roku 2007 firma miała ponad 59 000 klientów korporacyjnych, przetwarzających setki milionów transakcji dziennie, a roczne przychody osiągnęły poziom 497 mln dolarów.

Jednak mniej więcej w tym samym czasie pogorszyła się jej zdolność do rozwijania i publikowania nowych funkcjonalności. W 2006 roku zrealizowano cztery główne wydania dla klientów, ale w 2007 roku firma zdołała zrealizować zaledwie jedno takie wydanie pomimo zatrudnienia większej liczby inżynierów. W rezultacie liczba funkcjonalności dostarczanych przez zespoły stale malała, natomiast liczba dni pomiędzy głównymi wydaniami ciągle rosła.

Ponieważ rozmiar każdego wydania stawał się coraz większy, wyniki wdrażania również stawały się coraz gorsze. Karthik Rajan, który wówczas był wiceprezesem ds. infrastruktury inżynierowej, w prezentacji w 2013 roku raportował, że rok 2007

był „ostatnim rokiem, w którym oprogramowanie było wytwarzane i dostarczane z wykorzystaniem procesu kaskadowego i kiedy przełączliśmy się do bardziej przyrostowego procesu dostarczania oprogramowania”.

Dave Mangot i Reena Mathew na konferencji 2014 DevOps Enterprise Summit opisali wynikową wieloletnią transformację DevOps, która rozpoczęła się w 2009 roku. Według Mangota i Mathew do 2013 roku wdrożenie zasad i praktyk DevOps skróciło terminy realizacji wdrożeń z sześciu dni do pięciu minut. W rezultacie udało im się łatwiej skalować możliwości i przetwarzać ponad miliard transakcji na dzień.

Jednym z głównych aspektów transformacji Salesforce było powierzenie odpowiedzialności za inżynierię jakości wszystkim — niezależnie od tego, czy byli to inżynierowie działu Dev, Ops, czy Infosec. Aby osiągnąć ten cel, zintegrowano automatyczne testy ze wszystkimi fazami wytwarzania aplikacji i środowiska, a także z procesami ciągłej integracji i wdrażania. W ramach tych prac stworzono narzędzie open source Rouser do realizacji testów funkcjonalnych modułów Puppet.

Zaczęto również regularnie wykonywać **badania niszczące** — termin używany w produkcji, opisujący wykonywanie długotrwałych badań wytrzymałościowych w najtrudniejszych warunkach pracy do czasu zniszczenia komponentu poddawanego testom. Zespół Salesforce rozpoczął rutynowe testowanie usług, podając je coraz większym obciążeniom, do momentu wystąpienia awarii. Takie działanie pozwoliło mu zrozumieć tryby awarii i wprowadzić odpowiednie korekty. Nic dziwnego, że w efekcie uzyskano znacznie wyższą jakość usług w warunkach normalnych obciążzeń produkcyjnych.

Zadania bezpieczeństwa informacji zostały włączone do procesu inżynierii jakości na najwcześniejszych etapach projektu. Inżynierowie Infosec stale współpracowali w krytycznych fazach projektu, takich jak projektowanie architektury i testy, a także w procesie integracji narzędzi zabezpieczeń z automatycznym procesem testowania.

Zdaniem Mangot i Mathew jednym z kluczowych sukcesów wynikających z powtarzalności i rygoru zastosowanego w procesie była opinia grupy zarządzania zmianami, zgodnie z którą „zmiany w infrastrukturze wprowadzone za pośrednictwem narzędzia Puppet będą odtąd traktowane jako »zmiany standardowe«, wymagające znacznie mniej lub nawet niewymagające żadnych uzgodnień z komisją CAB”. Ponadto zauważono, że „ręczne zmiany w infrastrukturze nadal będą wymagały aprobaty”.

W ten sposób nie tylko zintegrowano proces DevOps z zarządzaniem zmianami, ale także stworzono dodatkową motywację do zautomatyzowania procesu zmian dla większej części infrastruktury.

ZMNIEJSZENIE ROLI PODZIAŁU OBOWIĄZKÓW

Przez dziesięciolecia stosowaliśmy podział obowiązków jako jeden z podstawowych mechanizmów zmniejszenia ryzyka oszustw lub błędów popełnianych w procesie rozwoju oprogramowania. Akceptowaną praktyką w większości cyklów życia wytwarzania systemów (ang. *System Development Life Cycle — SDLC*) było wymaganie od programisty przesyłania zmian do tzw. bibliotekarza kodu, który przeglądał i zatwierdzał zmianę przed wprowadzeniem jej do produkcji przez dział operacji IT.

Istnieje mnóstwo innych, mniej kontrowersyjnych przykładów podziału obowiązków w pracy Ops — na przykład administratorom serwerów przydzielano uprawnienia do wyświetlania logów, ale nie do ich modyfikowania, aby uniemożliwić osobie z uprzywilejowanym dostępem usuwanie dowodów oszustw lub innych problemów.

Kiedy wdrożenia do produkcji były wykonywane rzadziej (np. raz na rok) i kiedy wykonywane prace były mniej skomplikowane, podział pracy i przekazywanie jej było właściwym działaniem. Jednak wraz ze wzrostem złożoności i częstotliwości wdrażania skuteczna realizacja wdrożeń do produkcji coraz bardziej wymaga, aby wszystkie osoby uczestniczące w strumieniu wartości mogły szybko zobaczyć wyniki swojej pracy.

Podział obowiązków często może to utrudniać poprzez spowolnienie i zmniejszenie roli informacji zwrotnych odbieranych przez inżynierów na temat swojej pracy. To uniemożliwia inżynierom przyjęcie pełnej odpowiedzialności za jakość pracy i zmniejsza zdolność firmy do tworzenia systemu uczenia się.

W związku z tym tam, gdzie to możliwe, należy unikać podziału obowiązków jako mechanizmu zarządzania. Zamiast tego należy stosować takie techniki, jak programowanie w parach, ciągła kontrola operacji ewidencjonowania kodu oraz przeglądy kodu. Te mechanizmy mogą dać nam niezbędną pewność co do jakości wykonywanej pracy. Ponadto w przypadku zastosowania tych mechanizmów możemy pokazać, że osiągamy równoważne wyniki z sytuacją, w której był stosowany podział obowiązków.

Studium przypadku

Zgodność z normą PCI i historia ku przestrodze o podziale obowiązków w Etsy (2014)

Bill Massie jest menedżerem ds. rozwoju oprogramowania w Etsy, odpowiedzialnym za aplikację płacową (skrót od „I Can Haz Tokens”). Aplikacja ICHT przyjmuje zlecenia przelewu od klientów za pośrednictwem zbioru opracowanych wewnętrznie aplikacji przetwarzania płatności, które obsługują zlecenia online poprzez pobranie danych o kartach kredytowych od ich posiadaczy, tokenizację, skomunikowanie się z procesorem płatności i zrealizowanie transakcji*.

* Autorzy dziękują Billowi Massiemu i Johnowi Allspawowi za spędzenie całego dnia z Gene'em Kimem i podzielenie się z nim wiedzą na temat zgodności z przepisami.

Ponieważ według standardu **PCI DSS** (ang. *Payment Card Industry Data Security Standards*) środowisko danych posiadacza karty (ang. *Cardholder Data Environment* — **CDE**) to „ludzie, procesy i technologie, przechowywanie, przetwarzanie lub przesyłanie danych posiadacza karty lub wrażliwych danych uwierzytelniania”, w tym wszelkie podłączone komponenty systemu, aplikacja ICHT podlega standardowi PCI DSS.

W celu zapewnienia zgodności z PCI DSS aplikacja ICHT jest fizycznie i logicznie oddzielona od reszty organizacji Etsy i jest zarządzana przez całkowicie odrębny zespół programistów, inżynierów baz danych, inżynierów sieci i inżynierów operacyjnych. Każdy członek zespołu ma dwa laptopy: jeden dla aplikacji ICHT (jest skonfigurowany inaczej w celu spełnienia wymagań DSS, a gdy nie jest używany, jest zamknięty w sejfie) i drugi — do wykonywania pozostałych zadań w Etsy.

W ten sposób udało się oddzielić środowisko CDE od reszty organizacji Etsy i ograniczyć zakres regulacji PCI DSS do jednego, wydzielonego obszaru. Systemy, które tworzą środowisko CDE, są od siebie oddzielone (i inaczej zarządzane) niż reszta środowisk Etsy na poziomie fizycznym, sieci, kodu źródłowego i infrastruktury logicznej. Ponadto środowisko CDE jest zbudowane i eksploatowane przez interdyscyplinarny zespół odpowiedzialny wyłącznie za CDE.

W celu dostosowania do potrzeby zatwierdzenia kodu zespół ICHT musiał zmodyfikować swoje praktyki ciągłego dostarczania. Zgodnie z pkt 6.3.2 standardu PCI DSS w wersji 3.1 zespoły powinny dokonywać przeglądu całego niestandardowego kodu przed wprowadzeniem go do produkcji lub udostępnieniem klientom w celu zidentyfikowania wszelkich potencjalnych słabych punktów (przy użyciu procesów ręcznych lub zautomatyzowanych) pod kątem udzielenia odpowiedzi na następujące pytania:

- Czy zmiany w kodzie są weryfikowane przez inne osoby niż autor oraz osoby posiadające wiedzę na temat technik przeglądania kodu i praktyk bezpiecznego kodowania?
- Czy przeglądy kodu dają gwarancję, że kod jest tworzony zgodnie ze wskaźnikami bezpieczeństwa?
- Czy przed wydaniem są implementowane odpowiednie poprawki?
- Czy przed publikacją wyniki przeglądu kodu są odpowiednio sprawdzone i zatwierdzone przez kierownictwo?

W celu spełnienia tego wymagania zespół początkowo postanowił wyznaczyć Massiego na osobę zatwierdzającą kod, odpowiedzialną za wdrażanie zmian w środowisku produkcyjnym. Pożądane wdrożenia miały być oznaczone flagą w systemie JIRA. Massie miał zaznaczać je jako przejrzone i zatwierdzone, a następnie ręcznie wdrażać do produkcyjnej wersji ICHT.

Takie postępowanie pozwoliło firmie Etsy spełnić wymagania PCI DSS i uzyskać od asesorów podpis pod raportem zgodności. Zespół zaobserwował jednak istotne problemy.

Massie zauważył, że jednym z niepokojących efektów ubocznych „jest poziom »podziałów« w zespole ICHT, niespotykany wśród innych zespołów w firmie Etsy. Odkąd wdrożyliśmy podział obowiązków oraz inne mechanizmy wymagane w celu zachowania zgodności z PCI DSS, nikt w tym środowisku nie może być inżynierem pełnego stosu (ang. *full stack*)”.

Efekt jest taki, że podczas gdy reszta zespołów Dev i Ops w firmie Etsy ściśle ze sobą współpracuje, wdrażając zmiany, płynnie i z pełnym zaufaniem, to jak zaobserwował Massie: „W naszym środowisku PCI obecne są strach i niechęć do zadań wdrażania i utrzymywania, ponieważ nikt nie widzi niczego poza swoim fragmentem stosu oprogramowania. Wydaje się, że wprowadzone pozornie drobne zmiany do sposobu pracy stworzyły nieprzenikalny mur pomiędzy programistami a inżynierami operacyjnymi i niezaprzecjalnie wywołują napięcia, których nikt w Etsy nie obserwował od 2008 roku. Nawet jeśli masz zaufanie do swojej części stosu, to jest niemożliwe, abyś mógł mieć pewność, że to, co zmieni ktoś inny, nie spowoduje awarii twojej części stosu”.

Poniższe studium przypadku pokazuje, że organizacje stosujące DevOps mogą zachować zgodność z przepisami. To studium przypadku może być jednak historią ku przestrodze. Wszystkie cnoty, które przypisujemy wysokowydajnym zespołom DevOps, są kruche — nawet zespół, który ma doświadczenia z zasadami wysokiego zaufania i wspólnymi celami, może mieć problemy w sytuacji, gdy zaczną być stosowane mechanizmy niskiego zaufania.

ZADBANIE O DOKUMENTACJĘ I DOWODY DLA AUDYTORÓW I INSPEKTORÓW ODPOWIEDZIAŁNYCH ZA ZAPEWNIEŃ ZGODNOŚCI Z PRZEPISAMI

W miarę coraz powszechniejszego adaptowania wzorców DevOps przez organizacje techniczne powstaje coraz więcej napięć pomiędzy działami IT a audytorami. Nowe wzorce DevOps są wyzwaniem dla tradycyjnego sposobu myślenia o inspekcji, mechanizmach kontroli i łagodzeniu ryzyka.

Jak zaobserwował Bill Shinn, główny architekt rozwiązań zabezpieczeń w Amazon Web Services: „DevOps w całości dotyczy stworzenia pomostu pomiędzy Dev a Ops. W pewnym sensie wyzwanie stworzenia pomostu pomiędzy inżynierami DevOps a audytorami i inspektorami zgodności z przepisami jest nawet większe. Zastanówmy się, ilu audytorów potrafi czytać kod i jak wielu deweloperów przeczytało normę NIST 800-37 lub ustawę Gramm-Leach-Blileya. To tworzy luki wiedzy, a społeczność DevOps musi dążyć do stworzenia pomostu nad tymi lukami”.

Studium przypadku

Zapewnienie zgodności z przepisami w regulowanym środowisku

Pomoc klientom — dużym przedsiębiorstwom — w udowodnieniu, że mogą zastosować się do wszystkich obowiązujących przepisów i rozporządzeń, należy do obowiązków Billa Shinna, głównego architekta rozwiązań zabezpieczeń w Amazon Web Services. Bill Shinn poświęcił wiele lat pracy z ponad tysiącem klientów korporacyjnych, takich jak Hearst Media, GE, Phillips i Pacific Life, którzy wykorzystywali publiczne chmury w środowiskach o dużym stopniu regulacji prawnych.

Shinn zaobserwował: „Jeden z problemów polega na tym, że audytorzy są szkoleni w zakresie metod, które nie są odpowiednie dla wzorców pracy DevOps. Na przykład jeśli audytorzy mieli do sprawdzenia środowisko z 10 000 serwerów produkcyjnych, to tradycyjne podejście polegało na poddaniu analizie próbki tysiąca serwerów wraz z dowodami w postaci zrzutów ekranu na zastosowanie odpowiednich zasobów, ustawień kontroli dostępu, instalacji agentów, logów serwera i tak dalej”.

„Takie podejście sprawdzało się w odniesieniu do środowisk fizycznych”, kontynuował Shinn. „Ale w jaki sposób przygotować próbkę, gdy infrastruktura jest kodem i kiedy — dzięki automatycznemu skalowaniu — serwery przez cały czas pojawiają się i znikają? Te same problemy występują w przypadku stosowania potoku wdrożeń, który bardzo różni się od tradycyjnego procesu tworzenia oprogramowania, gdzie jedna grupa pisze kod, a inna grupa wdraża ten kod do produkcji”.

Shinn wyjaśnia: „Wśród audytorów najbardziej powszechną metodą zbierania dowodów nadal są zrzuty ekranów i pliki CSV wypełnione ustawieniami konfiguracji i logami. Naszym celem jest stworzenie alternatywnych metod prezentacji danych, które wyraźnie pokazują audytorom, że stosowane mechanizmy działają i są skuteczne”.

W celu stworzenia pomostu pomiędzy zespołami a audytorami obie grupy wspólnie pracują w procesie projektowania mechanizmów. Stosują metodę iteracyjną, przypisując jeden mechanizm w każdym sprincie w celu ustalenia tego, co jest potrzebne, jeśli chodzi o dowody kontroli. Takie podejście gwarantuje audytorom uzyskanie informacji, które są wymagane, gdy usługa jest w produkcji, całkowicie na żądanie.

Shinn stwierdza, że najlepszym sposobem osiągnięcia tego celu jest „wysyłanie wszystkich danych do systemów telemetrycznych, takich jak Splunk lub Kibana. Dzięki temu audytorzy mogą otrzymać to, czego potrzebują, w sposób całkowicie samoobsługowy. Nie muszą żądać próbki danych — zamiast tego logują się do systemu Kibana, a następnie szukają potrzebnych dowodów kontroli

dla określonego przedziału czasowego. W idealnej sytuacji mogą bardzo szybko się przekonać o istnieniu dowodów świadczących o skuteczności naszych mechanizmów”.

Shinn kontynuuje: „Dzięki nowoczesnym mechanizmom rejestrowania inspekcji, czatom i potokom wdrożeń jest doskonała widoczność i przejrzystość tego, co dzieje się w produkcji, zwłaszcza w porównaniu ze sposobem, do jakiego przyzwyczaili się inżynierowie działu operacji, a prawdopodobieństwo popełnienia błędów i wprowadzania luk w zabezpieczeniach jest znacznie mniejsze. Zatem prawdziwym wyzwaniem jest przekształcenie tych wszystkich dowodów w coś, co będzie znajome dla audytora”.

To wymaga wyprowadzenia wymagań inżynierijnych z rzeczywistych przepisów. Shinn wyjaśnia: „Aby odkryć, czego wymaga HIPAA z punktu widzenia bezpieczeństwa informacji, trzeba przyjrzeć się 45 przepisom CFR części 160. oraz przeanalizować podczęści A i C części 164. Następnie trzeba czytać dalej, by dojść do punktu »Zabezpieczenia techniczne i mechanizmy kontroli«. Dopiero z tego punktu można się dowiedzieć, że należy określić działania podlegające śledzeniu i kontroli w zakresie informacji zdrowotnych pacjenta, udokumentować i zaimplementować te mechanizmy, wybrać narzędzia, a następnie przejrzeć i przechwycić odpowiednie informacje”.

Shinn kontynuuje: „Sposób spełnienia tych wymagań powinien być uzgodniony pomiędzy inspektorami zgodności z przepisami a zespołami zabezpieczeń i DevOps — w szczególności w zakresie sposobów zapobiegania, wykrywania i korygowania problemów. Czasami wspomniane wymagania mogą być spełnione za pośrednictwem ustawień konfiguracji lub mechanizmów kontroli wersji. Innym razem należy zastosować mechanizmy monitorowania”.

Shinn podaje przykład: „Możemy zaimplementować jeden z tych mechanizmów za pomocą narzędzia AWS CloudWatch. Aby sprawdzić, czy mechanizm działa, wystarczy jeden wiersz polecenia. Ponadto musimy pokazać, gdzie są rejestrowane logi — w idealnej sytuacji magazynujemy je we frameworku rejestrowania, gdzie możemy powiązać dowody kontroli z rzeczywistymi wymaganiami”.

Aby pomóc rozwiązać ten problem, w podręczniku *DevOps Audit Defense Toolkit* zamieszczono wyczerpujący opis procesu kontroli zgodności z przepisami i audytu dla fikcyjnej organizacji (Parts Unlimited z projektu Phoenix). Opis zaczyna się od omówienia celów organizacji, procesów biznesowych, głównych zagrożeń i wynikowego środowiska kontroli, a także sposobu, w jaki kierownictwo może skutecznie udowodnić, że mechanizmy kontroli istnieją i są skuteczne. Zaprezentowano również zbiór przeszkód dla kontroli wraz ze sposobami ich przezwyciężenia.

Dokument opisuje sposób takiego zaprojektowania mechanizmów w potoku wdrożeń, aby złagodzić wskazane zagrożenia. Zamieszczono w nim również przykłady atestów tych mechanizmów i artefaktów wchodzących w ich skład w celu zademonstrowania skuteczności ich stosowania. Dokument miał być

ogólny dla wszystkich celów, włącznie ze wsparciem dla zgodności z przepisami dotyczącymi raportów finansowych (np. SOX-404, HIPAA, FedRAMP, UE Modelu Contracts oraz proponowanych przepisów Reg-SCI), zobowiązań umownych (np. PCI DSS, DOD DISA) oraz skuteczności i wydajności operacji.

Studium przypadku

Wykorzystanie telemetrii produkcji dla systemów ATM

Mary Smith (pseudonim) kieruje inicjatywą DevOps dla działu bankowości konsumenckiej dużej organizacji usług finansowych w Stanach Zjednoczonych. Zaobserwowała ona, że inżynierowie bezpieczeństwa informacji, audytorzy i inspektorzy zgodności z przepisami często zbyt dużo uwagi przykładają do przeglądów kodu jako mechanizmu wykrywania nadużyć. Zamiast tego, aby skutecznie ograniczyć zagrożenia związane z błędami i nadużyciami, powinni polegać na mechanizmach monitorowania produkcji w połączeniu z testami automatycznymi oraz przeglądami i zatwierdzeniami kodu.

Zauważyła, że:

Wiele lat temu mieliśmy programistę, który umieścił mechanizm backdoor w kodzie wdrożonym później do naszych bankomatów. Przestępcom udało się kilkakrotnie przełączyć bankomat do trybu konserwacji i podjąć gotówkę. Oszustwo udało nam się wykryć bardzo szybko i nie przyczyniło się do tego mechanizm przeglądu kodu. Tego typu backdoory są trudne lub nawet niemożliwe do wykrycia, gdy przestępca ma wystarczające środki, motywację i możliwości.

Jednakże udało się nam szybko wykryć oszustwo podczas regularnych spotkań przeglądu operacji, kiedy jedna z osób zauważyła, że bankomaty w mieście były wprowadzane w tryb konserwacji w nieplanowanych terminach. Oszustwo wykryliśmy jeszcze przed zaplanowanym procesem kontroli środków pieniężnych, podczas którego ilość gotówki jest uzgadniana z listą autoryzowanych transakcji.

W przypadku opisanym w tym studium do oszustwa doszło pomimo zastosowania podziału obowiązków pomiędzy działami Dev i Ops oraz stosowania procesu zatwierdzania zmian, ale zostało ono szybko wykryte i skorygowane dzięki skutecznym mechanizmom telemetrii produkcji.

PODSUMOWANIE

W całym niniejszym rozdziale omawialiśmy praktyki, dzięki którym bezpieczeństwo informacji staje się zadaniem wszystkich osób w strumieniu wartości, a wszystkie cele związane z bezpieczeństwem są zintegrowane w codziennej pracy. W ten sposób możemy znacznie poprawić skuteczność naszych mechanizmów. Dzięki temu jesteśmy w stanie lepiej zapobiegać naruszeniom zabezpieczeń, jak również szybciej je wykrywać i przywracać działanie. Ponadto możemy znacznie zmniejszyć ilość pracy związanej z przygotowaniem i uzyskaniem pomyślnych wyników z kontroli zgodności z przepisami.

PODSUMOWANIE CZĘŚCI VI

W poprzednich rozdziałach analizowaliśmy sposób zastosowania zasad DevOps do bezpieczeństwa informacji, aby pomóc w osiągnięciu celów i zadbać o to, aby bezpieczeństwo stało się częścią codziennej pracy każdego z nas. Dzięki lepszym mechanizmom zabezpieczeń jesteśmy w stanie lepiej się bronić i jesteśmy bardziej czujni. Możemy rozwiązywać problemy z zabezpieczeniami, zanim przybiorą one katastrofalne rozmiary, i — co ważniejsze — nasze systemy są bezpieczniejsze niż kiedykolwiek.

Wezwanie do działania Podsumowanie podręcznika DevOps

Dotarliśmy do końca szczegółowego omówienia zarówno zasad, jak i technicznych praktyk DevOps. W czasie, kiedy wyzwaniem każdego technicznego lidera jest zapewnienie bezpieczeństwa, niezawodności i elastyczności, oraz w czasach częstych naruszeń bezpieczeństwa dążenia do jak najszybszego dostarczania produktów na rynek i ogromnej transformacji technologicznych DevOps oferuje rozwiązanie. Mamy nadzieję, że ta książka dostarczyła Czytelnikom głębszej wiedzy na temat problemu i mapy drogowej pozwalającej stworzyć odpowiednie rozwiązania.

Zgodnie z tym, co powiedzieliśmy na początku książki *DevOps*, pomiędzy działami Dev i Ops istnieje nieodłączny konflikt, który pozostawiony bez kontroli prowadzi do pogarszających się problemów — coraz dłuższego czasu oczekiwania na wprowadzenie na rynek nowych produktów i funkcjonalności, niskiej jakości, coraz częstszego przestojów i powiększającego się dłużu technicznego, mniejszej wydajności produkcyjnej, a także coraz większego niezadowolenia pracowników i wypalenia zawodowego.

Zasady i wzorce DevOps pozwalają przełamać ten podstawowy, przewlekły konflikt. Mamy nadzieję, że po przeczytaniu tej książki Czytelnicy przekonali się, że transformacje DevOps pozwalają tworzyć dynamiczne, uczące się organizacje, osiągać wspaniałe rezultaty szybkiego przepływu i światowej klasy niezawodności i bezpieczeństwa, jak również poprawić konkurencyjność i zadowolenie pracowników.

DevOps potencjalnie potrzebuje stosowania nowych norm kulturowych i zmian w praktykach technicznych i architekturze. Wymaga to koalicji, która obejmuje kierownictwo biznesowe, zarządzanie produktem, działy Dev, QA, operacji IT, Infosec, a nawet marketingu, skąd wywodzi się wiele inicjatyw technicznych. Gdy wszystkie te zespoły pracują razem, można stworzyć bezpieczny system pracy, umożliwić małym zespołom szybkie i samodzielne rozwijanie i weryfikowanie poprawności kodu, który można bezpiecznie wdrażać dla klientów. Takie działania pozwalają zmaksymalizować wydajność pracy programistów, organizacyjne uczenie się, wysoki poziom satysfakcji pracowników i umiejętność wygrywania na rynku.

Celem tej książki było wystarczające skodyfikowanie zasad i praktyk DevOps, aby niesamowite wyniki osiągane w ramach społeczności DevOps mogły być replikowane przez innych. Mamy nadzieję, że przyczynimy się do przyspieszenia przyjęcia inicjatyw DevOps i wsparcia ich skutecznych implementacji, a jednocześnie obniżenia energii aktywacji niezbędnej do ich wdrożenia.

Znamy niebezpieczeństwa związane z odkładaniem usprawnień i stosowaniem obejść, a także trudności w zmianie sposobu nadawania priorytetów i realizowania codziennej pracy. Ponadto rozumiemy zagrożenia i wysiłek niezbędny do przyjęcia przez organizację nowych zasad pracy, a także postrzegania DevOps jako kolejnej moduł, która wkrótce będzie zastąpiona inną rewelacją.

Uważamy, że stosowanie DevOps spowoduje równie wielką transformację w sposób pracy organizacji technicznych, jak stosowanie Lean spowodowało transformację pracy organizacji produkcyjnych w latach 80. Organizacje, które przyjmą DevOps, zwyciężą na rynku kosztem tych, którzy tego nie zrobią. Przekształcać się w energiczne i stale uczące się organizacje, które pod względem wydajności i innowacyjności przewyższają swoich konkurentów.

Z tych powodów DevOps nie jest wyłącznie koniecznością technologiczną, ale również imperatywem organizacyjnym. Ostateczny wniosek brzmi: DevOps może być zastosowane i nadaje się dla wszelkich organizacji, które muszą zwiększyć przepływ planowanej pracy technicznej przy zachowaniu jakości, niezawodności i poczucia bezpieczeństwa klientów.

Oto nasze wezwanie do działania: niezależnie od tego, jaką funkcję pełnisz w swojej organizacji, zacznij wokół siebie poszukiwania ludzi, którzy chcą zmienić sposób wykonywania pracy. Pokaż tę książkę innym i utwórzcie koalicję ludzi o podobnych poglądach, którzy przerwą spiralę degradacji. Poproś menedżerów o wsparcie tych wysiłków albo jeszcze lepiej, o sponsoring i przewodnictwo tym staraniom.

Wreszcie, skoro już dotarłeś tak daleko, chcemy ujawnić Ci sekret. W wielu naszych studiach przypadku po osiągnięciu przełomowych wyników, które zostały przedstawione w książce, wielu agentów zmiany uzyskało awans, ale w niektórych przypadkach później nastąpiły zmiany w kierownictwie, co spowodowało odejście wielu zaangażowanych osób. W rezultacie wprowadzone przez nich zmiany organizacyjne zostały cofnięte.

Uważamy, że nie należy zapominać o takiej możliwości. Osoby zaangażowane w transformację DevOps z góry wiedziały, że to, co robią, jest obarczone dużym prawdopodobieństwem porażki, a pomimo to nie cofnęły się przed zmianami. W ten sposób, co być może jest najważniejsze, zainspirowały pozostałych z nas, pokazując co można zrobić. Innowacje nie są możliwe bez podejmowania ryzyka, a jeśli nie udało Ci się zdenerwować co najmniej kilku menedżerów, to prawdopodobnie nie starałeś się wystarczająco mocno. Nie pozwól, aby układ odpornościowy Twojej organizacji zapobiegł realizacji Twojej wizji lub zawrócił Cię z obranej drogi. Jak mawiał Jesse Robbins, były „mistrz katastrofy” w Amazonie: „Nie walcz głupio, czyn więcej rzeczy niezwykłych”.

DevOps przynosi korzyści wszystkim osobom w strumieniu wartości technologii — niezależnie od tego, czy reprezentujemy Dev, Ops, QA, Infosec, właściciela produktu, czy klientów. DevOps przywraca radość z tworzenia doskonałych produktów i ogranicza liczbę „marszów śmierci”. Sprzyja tworzeniu humanitarnych warunków pracy, w których jest mniej weekendów i świąt odebranych naszym bliskim. Umożliwia zespołom wspólną pracę, aby przetrwać, uczyć się, rozwijać, wzbudzać zachwyt klientów i pomaga organizacji odnieść sukces.

Mamy szczerą nadzieję, że *DevOps* pomoże Ci osiągnąć te cele.

MATERIAŁY DODATKOWE

DODATKI

DODATEK 1.

KONWERGENCJA DEVOPS Z INNYMI RUCHAMI

Uważamy, że DevOps korzysta z niezwykłej zbieżności z innymi ruchami w zarządzaniu, które się wzajemnie uzupełniają i mogą pomóc w stworzeniu potężnej koalicji zdolnej do przekształcenia sposobu rozwijania i dostarczania produktów i usług IT.

John Willis nazwał to „Konwergencją DevOps”. Poniżej opisano elementy tej konwergencji w przybliżonym porządku chronologicznym (zwróćmy uwagę, że nie ma to być opis wyczerpujący, lecz jedynie wzmianka, która prezentuje postęp w myśleniu i raczej mało prawdopodobne połączenia, które doprowadziły do powstania ruchu DevOps).

LEAN

Ruch Lean rozpoczął się w latach 80. jako próba kodyfikacji Toyota Production System dzięki popularyzacji takich technik, jak mapowanie strumienia wartości, tablice kanban i całkowite produktywne utrzymanie ruchu (ang. *Total Productive Maintenance — TPM*).

Lean przyjmował dwa główne założenia: głęboko zakorzenione przekonanie, że czas realizacji (tzn. czas potrzebny do konwersji surowców na wyroby gotowe) był najlepszym wskaźnikiem jakości, zadowolenia klientów i satysfakcji pracowników,

oraz twierdzenie, że jednym z najlepszymi predyktorów krótkich terminów realizacji były niewielkie rozmiary partii, a teoretycznym ideałem był „przepływ jednej sztuki” (czyli przepływ „1×1”: surowce 1, rozmiar partii 1).

Zasady metodyki Lean koncentrują się wokół tworzenia wartości dla klienta — na myśleniu systemowym, tworzeniu stałości celu, uwzględnianiu myśli naukowej, tworzeniu przepływu i zasady „pull” (w przeciwieństwie do „push”), zapewnieniu gwarancji jakości, kierowaniu z pokorą oraz poszanowaniem każdego człowieka.

RUCH AGILE

W 2001 r. 17 wiodących myślicieli w rozwoju oprogramowania ogłosiło manifest Agile. Celem manifestu było przekształcenie „lekkich metod” rozwijania oprogramowania, takich jak **DP** (ang. *Design Patterns*) i **DSDM** (ang. *Dynamic System Development Method*) w szerszy ruch, który mógłby zastąpić „ciężkie” procesy rozwoju oprogramowania, takie jak metoda kaskadowa oraz inne metodologie, na przykład **RUP** (ang. *Rational Unified Process*).

Kluczową zasadą było „dostarczanie działającego oprogramowania często — od kilku tygodni do kilku miesięcy, z preferencją dla krótszych ram czasowych”. Dwie inne zasady koncentrują się wokół potrzeby niewielkich, samomotywujących się zespołów, pracujących w modelu zarządzania wysokiego zaufania oraz z naciskiem na niewielkie partie. Ruch Agile wiąże się również z zestawem takich narzędzi i praktyk, jak Scrum, standup itd.

RUCH KONFERENCJI VELOCITY

W 2007 Steve Souders, John Allspaw i Jesse Robbins utworzyli konferencję Velocity jako centrum wymiany poglądów dla społeczności IT Operations i Web Performance. Na konferencji Velocity 2009 John Allspaw i Paul Hammond przedstawili słynną prezentację *10 Deploys per Day: Dev and Ops Cooperation at Flickr*.

RUCH INFRASTRUKTURA AGILE

Na konferencji Agile Toronto 2008 Patrick Dubois i Andrew Schafer przeprowadzili sesję „birds of a feather” dotyczącą zastosowania zasad Agile do infrastruktury zamiast do kodu aplikacji. Prezenterzy szybko zyskali zwolenników wśród osób o podobnych poglądach, w tym Johna Willisa. Później Dubois był tak podekscytowany prezentacją Allspawa i Hammonda *10 Deploys per Day: Dev and Ops Cooperation at Flickr*, że w 2009 roku zorganizował w Gandawie, w Belgii, pierwszą konferencję DevOpsDays, na której powstał termin „DevOps”.

RUCH CIĄGŁYCH DOSTAW

Opierając się na nurtach w wytwarzaniu oprogramowania — ciągłej komplikacji, ciągłym testowaniu i integracji — Jez Humble i David Farley rozszerzyli tę koncepcję o ideę ciągłego dostarczania, która obejmowała „potok wdrożeń”, mający zagwarantować ciągłość gotowości do wdrożenia kodu i infrastruktury oraz aby kod zaewidencjonowany na gałęzi master był zawsze wdrażany w środowisku produkcyjnym.

Pomysł ten po raz pierwszy zaprezentowano na konferencji Agile 2006. Został także niezależnie zaprojektowany przez Tima Fitza w artykule na blogu zatytułowanym „Continuous Deployment”.

RUCH TOYOTA KATA

W 2009 Mike Rother napisał książkę *Toyota Kata: Managing People for Improvement, Adaptiveness and Superior Results*, w której opisał wnioski z 20-letnich prac zmierzających do zrozumienia i skodyfikowania mechanizmów Toyota Production System. *Toyota Kata* opisuje „niewidzialne procedury zarządzania i myślenia, leżące u podstaw sukcesu Toyoty, z ciągłym doskonaleniem i adaptacją... oraz to, jak inne firmy mogą opracować podobne procedury i sposoby myślenia w swoich organizacjach”.

Mike Rother doszedł do wniosku, że społeczność Lean pominęła najważniejszą praktykę ze wszystkich, którą określił jako *Improvement Kata*. Wyjaśnił, że każda organizacja ma procedury pracy, a decydującym czynnikiem w firmie Toyota było stosowanie zwyczaju stałego doskonalenia pracy i włączanie udoskonalania do codziennej pracy wszystkich osób w organizacji. *Toyota Kata* instytucjonalizuje iteracyjne, przyrostowe, naukowe podejście do rozwiązywania problemów w dążeniu do wspólnego rozumienia „właściwego kierunku” dla całej organizacji.

RUCH LEAN STARTUP

W 2011 roku Eric Ries napisał książkę *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*, gdzie skodyfikował wiedzę zdobytą w firmie IMVU, startupie z Doliny Krzemowej, która swój sposób działania oparła na pracy Steve'a Blanka *The Four Steps to the Epiphany*, a także technikach ciągłego wdrażania. Eric Ries skodyfikował również inne powiązane praktyki i terminy, w tym *Minimum Viable Product*, cykl nauki „zbuduj-zmierz-naucz się” (ang. *build-measure-learn*) oraz wiele innych wzorców technicznych ciągłego wdrażania.

RUCH LEAN UX

W 2013 r. Jeff Gothelf napisał książkę *Lean UX: Applying Lean Principles to Improve User Experience*, w której zaprezentował, jak poprawić „rozmyte warstwy front end”, i wyjaśnił, w jaki sposób właściciele produktu mogą opakować hipotezy biznesowe i eksperymenty, a następnie zdobyć zaufanie do tych hipotez biznesowych przed zainwestowaniem czasu i zasobów w wynikowe funkcjonalności. Dzięki dodaniu Lean UX mamy teraz narzędzia do pełnego zoptymalizowania przepływu pomiędzy hipotezami biznesowymi, rozwojem funkcjonalności, testowaniem, wdrażaniem i dostarczaniem usług do klienta.

RUCH RUGGED COMPUTING

W 2011 roku Joshua Corman, David Rice i Jeff Williams sformułowali wniosek o wyraźnym braku sensu w zabezpieczaniu aplikacji i środowisk w późnych fazach cyklu życia. W wyniku tych prac stworzyli filozofię określającą jako *Rugged Computing*, która próbuje ująć w ramy niefunkcjonalne wymagania stabilności, skalowalności, dostępności, niezawodności, trwałości, bezpieczeństwa, możliwości wsparcia, zarządzania i ochrony przed zagrożeniami.

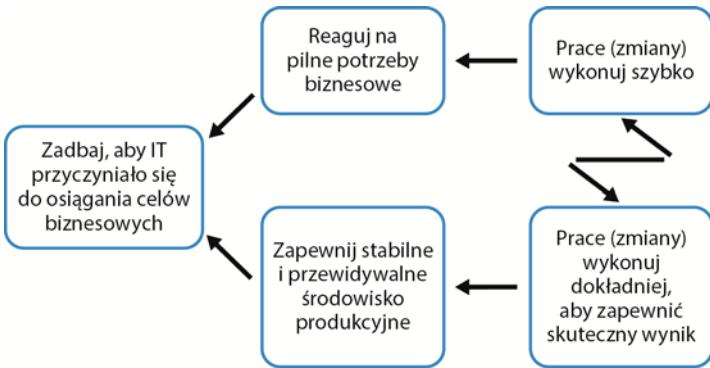
Ze względu na potencjalne wysokie tempo wydań DevOps może wywierać niesamowity nacisk na działy QA i Infosec. Kiedy tempo wdrożeń zmienia się z comiesięcznego lub cokwartalnego na setki lub tysiące dziennie, to dwutygodniowe czasy dostarczania wyników z działów Infosec lub QA nie mają już zastosowania. Ruch *Rugged Computing* zakłada, że obecne podejście do walki z zagrożeniami stosowane przez większość programów ochrony informacji jest bezskuteczne.

DODATEK 2.

TEORIA OGRANICZEŃ I PODSTAWOWE PRZEWLEKŁE KONFLIKTY

W teorii ograniczeń szeroko mówi się o tworzeniu chmur konfliktów podstawowych (często określanych jako „C3”). Chmurę konfliktów dla IT zaprezentowano na rysunku 46.

W latach 80. w firmach produkcyjnych obserwowano dobrze znany podstawowy przewlekły konflikt. Każdy kierownik działu produkcyjnego miał dwa ważne cele biznesowe: chronić sprzedaż i zmniejszać koszty. Problem polegał na tym, że w celu ochrony sprzedaży przedstawiciele kierownictwa sprzedaży byli zachęcani do zwiększenia zapasów, tak aby zawsze było możliwe spełnienie potrzeb klienta.



Rysunek 46. Podstawowy przewlekły konflikt w każdej organizacji IT

Z drugiej strony w celu zmniejszenia kosztów kierownictwo produkcji zachęcano do zmniejszania zapasów, aby prace w toku, które nie mogły być natychmiast dostarczone do klientów w formie realizacji sprzedaży, nie wiązały środków pieniężnych.

Konflikt udało się przełamać dzięki przyjęciu zasad Lean, takich jak redukcja rozmiaru partii, zmniejszenie ilości produkcji w toku oraz skrócenie i wzmacnienie pętli sprzężeń zwrotnych. To spowodowało dramatyczny wzrost produktywności fabryk, jakości produktów i zadowolenia klientów.

Zasady rządzące wzorcami pracy DevOps są takie same jak te, które doprowadziły do transformacji produkcji. Przyczyniło się to do zoptymalizowania strumienia wartości IT, przekształcenia potrzeb biznesowych na możliwości i usługi dostarczające wartość dla klientów.

DODATEK 3. TABELARYCZNA FORMA SPIRALI DEGRADACJI

W tabeli 4. zaprezentowano w formie dwóch kolumn spiralę degradacji opisaną w książce *Projekt Feniks*:

Tabela 4. Spirala degradacji

Punkt widzenia działu operacji IT	Punkt widzenia działu developmentu...
Kruche aplikacje są podatne na awarie	Kruche aplikacje są podatne na awarie
Długi czas wymagany do wykrycia, które fragmenty zostały zamienione miejscami	W kolejce są umieszczane pilne projekty, w których większe znaczenie mają terminy
Mechanizm wykrywania bazuje na sprzedawcy	Do produkcji przekazywany jest coraz bardziej kruchy kod (mniej bezpieczny)
Odtwarzanie usługi wymaga zbyt wiele czasu	Więcej wydań wiąże się z coraz bardziej kłopotliwymi instalacjami

Tabela 4. Spirala degradacji

Punkt widzenia działu operacji IT	Punkt widzenia działu developmentu...
Zbyt dużo „gaszenia pożarów” i niezaplanowanej pracy	Wydłużenie cyklu wydań w celu zamortyzowania kosztów wdrażania
Pilne przeróbki i naprawy zabezpieczeń	Awarie większych wdrożeń są coraz trudniejsze do zdiagnozowania
Nie można realizować planowej pracy w projekcie	W większości przypadków najbardziej doświadczony i mało liczebny personel operacji IT ma coraz mniej czasu na rozwiązywanie problemów proceduralnych
Sfrustrowani klienci odchodzą	Coraz większy stos zadań do wykonania w projekcie kluczowym dla celów biznesowych
Akcje idą w dół	Coraz większe napięcia pomiędzy działami operacji IT, Dev i Design
Firma nie jest w stanie sprostać zobowiązań na Wall Street	
Firma składa jeszcze więcej obietnic na Wall Street	

DODATEK 4. NIEBEZPIECZEŃSTWA ZWIĄZANE Z PRZEŁĄCZANIEM PRACY I KOLEJKAMI

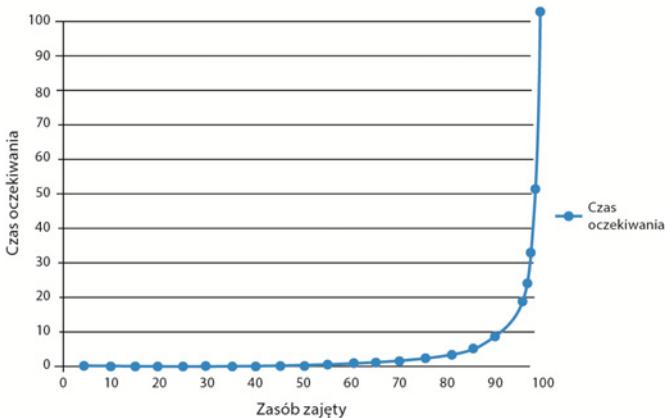
Problem z dużą ilością czasu w kolejkach pogłębia się, gdy istnieje potrzeba wielokrotnego przełączania pracy, ponieważ to właśnie w ten sposób dochodzi do powstawania kolejek. Na rysunku 47 pokazano czas oczekiwania w funkcji zajętości zasobu w ośrodku pracy. Asymptotyczna krzywa pokazuje, dlaczego „prosta zmiana zajmująca 30 minut” często trwa tygodniami — konkretni inżynierowie i gniazda produkcyjne często stają się problematycznymi wąskimi gardłami w przypadku intensywnego wykorzystywania. W miarę zbliżania się ośrodka pracy do 100% wykorzystania, wszelkie prace wymagane od ośrodka będą tkiły w kolejkach i nie będą realizowane bez przyspieszenia lub eskalacji.

Oś X na rysunku 47 reprezentuje procent zajętości dla zasobu w ośrodku pracy, natomiast oś Y określa przybliżony czas oczekiwania (lub — dokładniej — długość kolejki). Z kształtu krzywej wynika, że jeśli wykorzystanie zasobu przekroczy 80%, to czas oczekiwania zaczyna gwałtownie wzrastać.

W Projekcie Feniks Bill i jego zespół zdali sobie sprawę z katastrofalnych skutków tej właściwości na czas realizacji zobowiązań składanych do biura zarządzania projektami:

Powiem im to, co powiedział mi Erik na konferencji MRP-8 na temat zależności czasu oczekiwania od wykorzystania zasobów. „Czas oczekiwania to »procent czasu zajętości« podzielony przez »procent czasu beczynności«. Mówiąc inaczej, jeśli zasób jest w 50% zajęty, to jest w 50% beczynny. Czas oczekiwania

$\text{Czas oczekiwania} = (\% \text{ zajęty}) / (\% \text{ bezczynny})$



Rysunek 47. Rozmiar kolejki i czasy oczekiwania jako funkcja procentu wykorzystania
(źródło: Kim, Behr i Spafford, „The Phoenix Project”, wydanie eBook, s. 557)

wynosi 50% podzielone przez 50%, czyli jedną jednostkę czasu. Przyjmijmy, że to jest jedna godzina”.

Zatem średnio zadanie będzie czekać w kolejce przez jedną godzinę, zanim zacznie się jego realizacja.

„Z drugiej strony, jeśli zasób jest zajęty w 90%, to czas oczekiwania wynosi »90% podzielone przez 10%«, czyli 9 godzin. Innymi słowy, zadanie będzie czekać w kolejce 9 razy dłużej w porównaniu z przypadkiem, gdy zasób był bezczynny w 50%”.

Wyciągam zatem następujący wniosek: „... W przypadku projektu Feniks, przy założeniu przekazywania zadania 7 razy oraz zajętości każdego z tych zasobów na poziomie 90%, zadania będą przebywały w kolejce łącznie 9 godzin razy 7 kroków...”.

„Co? 63 godziny samego tylko oczekiwania?”. Wes mówi z niedowierzaniem: „To jest niemożliwe!”.

Patty mówi z uśmiechem: „Ależ oczywiście, że możliwe. Ponieważ to tylko 30 minut pisania”.

Bill wraz z zespołem zdają sobie sprawę, że ich „proste 30-minutowe zadanie” w rzeczywistości wymaga 7-krotnego przełączania (np. zespół serwerów, zespół sieci, zespół bazy danych, zespół wirtualizacji i co oczywiste, Brent, inżynier typu „gwiazda rocka”).

Jeśli zakładamy, że wszystkie ośrodki pracy były zajęte w 90%, to zgodnie z rysunkiem średni czas oczekiwania w każdym ośrodku roboczym wynosił 9 godzin, a ponieważ praca musi przejść przez 7 ośrodków roboczych, łączny czas pracy wyniesie 7-krotność tej wartości: 63 godziny.

Innymi słowy, całkowity procent wartości dodanego czasu (nieraz nazywany czasem przetwarzania) wynosił zaledwie 0,16% łącznego czasu realizacji (30 minut podzielone przez 63 godziny). To oznacza, że przez 99,8% całkowitego czasu realizacji praca po prostu oczekiwana w kolejce na realizację.

DODATEK 5. MITY NA TEMAT BEZPIECZEŃSTWA

Dziesięciolecia badań nad złożonymi systemami pokazują, że stosowane środki zaradczne bazują na kilku mitach. W książce Denisa Besnarda i Erika Hollnagela *Some Myths about Industrial Safety* zostały one zestawione w następujący sposób:

- **Mit 1.** „Najważniejszą przyczyną wypadków i incydentów jest błąd ludzki”.
- **Mit 2.** „Systemy będą bezpieczne, jeśli ludzie będą przestrzegali obowiązujących procedur”.
- **Mit 3.** „Bezpieczeństwo można poprawić przez bariery i zabezpieczenia; im więcej warstw ochrony tym większe bezpieczeństwo”.
- **Mit 4.** „Analiza wypadku pozwala zidentyfikować jego główną przyczynę (prawdę)”.
- **Mit 5.** „Badania wypadków polegają na logicznej i racjonalnej identyfikacji ich przyczyn na podstawie faktów”.
- **Mit 6.** „Bezpieczeństwo zawsze ma najwyższy priorytet. Ta zasada nigdy nie będzie zagrożona”.

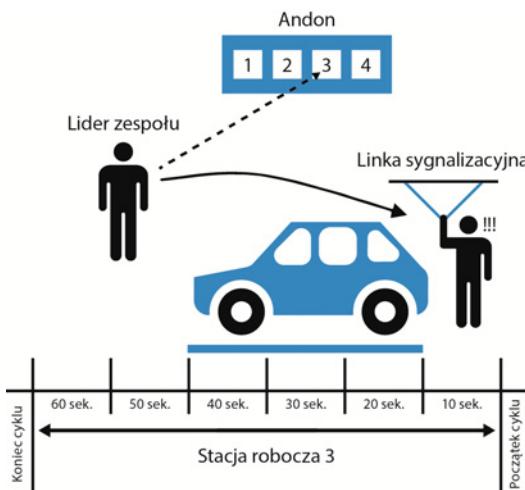
Różnice pomiędzy mitami a rzeczywistością przedstawiono w tabeli 5:

Tabela 5. Dwie historie

Mit	Rzeczywistość
Za przyczynę awarii uważa się błąd ludzki.	Ludzki błąd jest efektem słabości systemowych głębiej wewnętrz organizacji.
Powiedzenie ludziom, co powinni byli zrobić, jest zadowalającym sposobem opisania awarii.	Powiedzenie ludziom, co powinni byli zrobić, nie wyjaśnia, dlaczego było dla nich sensowne to, co faktycznie zrobili.
Powiedzenie ludziom, aby byli ostrożni, przyczyni się do usunięcia problemu.	Polepszenie bezpieczeństwa jest możliwe tylko w wyniku ciągłego poszukiwania słabości organizacji.

DODATEK 6. LINKA ANDON FIRMY TOYOTA

Wiele osób pyta, jak można ukończyć jakąkolwiek pracę, skoro liczba pociągnięć za linkę Andon (rysunek 48) sięga 5000 dziennie? Aby być precyzyjnym, nie każde pociągnięcie za linkę Andon powoduje zatrzymanie całej linii produkcyjnej. Jest raczej tak, że w przypadku pociągnięcia za linkę Andon lider zespołu nadzorujący pracę określonego ośrodka pracy ma 50 sekund na rozwiązywanie problemu. Jeśli problem nie zostanie rozwiązany po upływie 50 sekund, częściowo zmontowany pojazd przekracza fizycznie narysowaną linię na podłodze i linia montażowa zostaje zatrzymana.



Rysunek 48. Linka Andon w firmie Toyota

DODATEK 7. OPROGRAMOWANIE „Z PÓŁKI”

Obecnie, aby zaewidencjonować skomplikowane oprogramowanie **COTS** (ang. *commercial off-the-shelf*) (np. SAP, IBM WebSphere, Oracle WebLogic) w systemie kontroli wersji, trzeba wyeliminować użycie narzędzia instalacyjnego z graficznym interfejsem użytkownika. Aby to zrobić, trzeba wykryć, co robi dostarczony przez dostawcę instalator, przeprowadzić instalację na czystym obrazie serwera, sprawdzić różnice w systemie plików i umieścić dodane pliki w repozytorium kontroli wersji. Pliki, które nie zależą od środowiska, są umieszczane w jednym miejscu („instalacja podstawowa”), natomiast pliki specyficzne dla środowiska są umieszczane w osobnym katalogu („test” lub „produkcja”). W ten sposób operacje instalacji oprogramowania

stają się jedynie operacjami kontroli wersji, co umożliwia ich lepszą widoczność, powtarzalność i szybkość działania.

Czasami trzeba również przekształcić wszystkie ustawienia konfiguracji aplikacji w taki sposób, aby znalazły się w repozytorium kontroli wersji. Na przykład można przekształcić konfiguracje aplikacji przechowywane w bazie danych na pliki XML i na odwrót.

DODATEK 8. SPOTKANIA POST-MORTEM

Poniżej zaprezentowano przykładową agendę spotkania post-mortem:

- Gospodarz spotkania lub moderator wygłasza oświadczenie, w którym podkreśla, że spotkanie nie ma na celu wskazywania winnych oraz że nie będzie koncentrowało się na przeszłych wydarzeniach lub spekulacjach na temat „co powinno było zostać zrobione” lub „co mogło się wydarzyć”. Moderatorzy mogą odczytać dyrektywę „Retrospective Prime Directive” z witryny internetowej *Retrospective.com*.

Ponadto moderator przypomina wszystkim, że wszelkie środki zaradcze muszą być komuś przypisane, i jeżeli nie ma gwarancji, że działania naprawcze uzyskają po spotkaniu najwyższy priorytet, to znaczy, że to nie są dobre działania naprawcze (ma to uniemożliwić przekształcenie spotkania w zdarzenie generujące listę dobrych pomysłów, które nigdy nie zostaną zaimplementowane).

- Osoby obecne na spotkaniu uzgadniają kompletną oś czasu zdarzenia, łącznie z tym, kiedy i kto wykrył problem, jak został wykryty (np. automatyczne monitorowanie, wykrywanie ręczne, informacja od klienta), kiedy usługa została pomyślnie przywrócona itd. Na osi czasu zaznaczana jest także cała komunikacja zewnętrzna.

Kiedy używamy określenia „oś czasu”, to może ono przywoływać obraz liniowego zbioru kroków opisujących sposób zdobycia wiedzy na temat problemu i jego ostatecznego rozwiązania. W rzeczywistości, zwłaszcza w złożonych systemach, często jest wiele wydarzeń, które przyczyniły się do wypadku oraz wiele ścieżek rozwiązywania problemów i działań podejmowanych w procesie dążenia do jego wyeliminowania. W tym procesie staramy się notować wszystkie te wydarzenia oraz perspektywy uczestników, a tam, gdzie to możliwe, tworzyć hipotezy dotyczące przyczyn i skutków.

- Zespół tworzy listę wszystkich czynników, które przyczyniły się do incydentu, zarówno ludzkich, jak i technicznych. Następnie mogą one być posortowane na kategorie, takie jak „decyzja projektowa”, „korekta”, „wykrycie problemu” itd.

Zespół używa takich technik, jak burza mózgów oraz *infinite hows* (dosł. „nieskończone pytania »jak«”) w celu ustalenia czynników, które okazały się szczególnie ważne w procesie wykrywania głębszych poziomów przyczyn awarii. Należy uwzględnić i uszanować wszystkie punkty widzenia — nikt nie powinien mieć prawa podważania lub zaprzeczania realności czynnika przyczynowego zidentyfikowanego przez inną osobę. Moderator spotkania post-mortem powinien zadbać o to, aby czas poświęcony na tę aktywność był wystarczający oraz aby zespół nie próbował angażować się w zbieżne zachowania, takie jak próba identyfikacji jednej lub większej liczby „głównych przyczyn”.

- Osoby obecne na spotkaniu osiągną porozumienie w postaci listy działań korygujących, które po spotkaniu uzyskają najwyższy priorytet. Utworzenie tej listy wymaga burzy mózgów oraz wyboru najlepszych potencjalnych działań zapobiegających występowaniu problemu albo umożliwiających jego szybsze wykrywanie bądź przywracanie stanu prawidłowego. Na liście powinny również się znaleźć inne sposoby poprawy systemów.

Celem jest zidentyfikowanie jak najmniejszej liczby przyrostowych kroków w celu osiągnięcia pożądanych rezultatów, w przeciwieństwie do zmian typu „big bang”, których implementacja nie tylko trwa dłużej, ale także opóźnia potrzebne usprawnienia.

Należy również wygenerować osobną listę pomysłów o niższym priorytecie i przypisać im właściciela. Jeśli w przyszłości wystąpią podobne problemy, to te pomysły mogą służyć za podstawę wytwarzania przyszłych środków zaradczych.

- Osoby obecne na spotkaniu powinny osiągnąć porozumienie w sprawie parametrów incydentu oraz ich wpływu na organizację. Na przykład możemy zdecydować o zmierzeniu incydentów zgodnie z następującymi parametrami:
 - **Waga zdarzenia** — jak poważny był problem. Dotyczy to bezpośrednio wpływu na usługę oraz klientów.
 - **Całkowity czas przestoju** — jak długo klienci nie mogli używać usługi w żadnym stopniu.
 - **Czas do wykrycia** — ile zajęło nam lub naszym systemom uświadomienie sobie istnienia problemu.
 - **Czas do rozwiązania** — ile czasu od uzyskania wiedzy o problemie zajęło nam przywrócenie usługi.

Bethany Macri z firmy Etsy zaobserwowała: „Brak wskazywania winnych na spotkaniach post-mortem nie oznacza, że nikt nie przyjmuje odpowiedzialności za zdarzenia. Oznacza to, że chcemy dowiedzieć się, jakie były okoliczności, które pozwoliły wprowadzić zmianę lub doprowadzić do problemu. Jakie było większe środowisko...”

Idea jest taka, że usunięcie wskazywania winnych usuwa strach, a dzięki wyeliminowaniu strachu zyskujemy uczciwość”.

DODATEK 9. MAŁPIA ARMIA

Po awarii AWS EAST Outage w 2011 roku w firmie Netflix przeprowadzono liczne dyskusje na temat inżynierii systemów do automatycznej obsługi awarii. Dyskusje te wyewoluowały do usługi o nazwie „Chaos Monkey”.

Od tego czasu usługa Chaos Monkey przekształciła się w całą rodzinę narzędzi, nazywaną wewnętrz firmy Małpią Armią (ang. *Simian Army*), które symulują coraz bardziej katastrofalne błędy:

- **Chaos Gorilla** — symuluje awarię całej strefy dostępności AWS.
- **Chaos Kong** — symuluje awarię całych regionów AWS, takich jak Ameryka Północna albo Europa.

Oto inni członkowie „małpiej armii”:

- **Latency Monkey** — indukuje sztuczne opóźnienia lub przerwy w warstwie RESTful klient-serwer w celu zasymulowania degradacji usługi i zagwarantowania właściwych reakcji usług zależnych.
- **Conformity Monkey** — wyszukuje i zamknięte egzemplarze AWS, które nie są zgodne z najlepszymi praktykami (np. gdy egzemplarze nie należą do grupy automatycznego skalowania lub gdy nie ma w katalogu usług adresu e-mail inżyniera, któremu należy zgłaszać problemy).
- **Doctor Monkey** — podłącza się do testów kondycji uruchamianych na poszczególnych egzemplarzach. Wyszukuje egzemplarze o złej kondycji i proaktywnie je zamknięte, jeśli właściciele nie usuną przyczyny problemu na czas.
- **Janitor Monkey** — dba o to, aby środowisko chmury było pozbawione śmieci i odpadów; wyszukuje niewykorzystane zasoby i usuwa je.
- **Security Monkey** — rozszerzenie Conformity Monkey, wyszukuje i kończy działanie egzemplarzy naruszających zasady bezpieczeństwa lub ze słabymi punktami, takimi jak nieprawidłowo skonfigurowane grupy zabezpieczeń AWS.

DODATEK 10.

PRZEZROCZYSTY CZAS SPRAWNOŚCI

Lenny Rachitsky pisał o zaletach zjawiska określanego jako „przezroczysty czas sprawności”:

1. Jeśli użytkownicy będą w stanie samodzielnie zidentyfikować problemy systemowe bez kontaktowania się telefonicznie bądź e-mailem z działem pomocy technicznej, to koszty wsparcia technicznego spadną. Użytkownicy nie będą już zmuszeni zgadywać, czy ich problemy mają charakter lokalny, czy globalny, i będą w stanie szybciej ustalić główną przyczynę problemu, zanim skontaktują się z działem wsparcia.
2. Możliwości komunikacji z użytkownikami podczas przestojów są lepsze ze względu na rozgłoszeniowy charakter internetu w przeciwieństwie do komunikacji „jeden do jednego” w przypadku rozmowy telefonicznej lub wiadomości e-mail. Potrzeba mniej czasu na komunikację dotyczącą tej samej sprawy, a więcej na rozwiązywanie problemu.
3. Wyznaczamy jedno, oczywiste miejsce, do którego użytkownicy mogą się udać w przypadku wystąpienia przestojów. Oszczędzamy czas użytkowników poświęcanego na przeszukiwanie forów, Twittera lub bloga.
4. Zaufanie jest podstawą każdej udanej adaptacji SaaS. Od działania naszej usługi lub platformy zależy powodzenie biznesu naszych klientów oraz źródeł ich utrzymania. Zarówno obecni, jak i przyszli klienci wymagają zaufania do naszej usługi. Muszą mieć pewność, że w przypadku problemów z usługą nie pozostaną sami i bez informacji. Najlepszym sposobem na zbudowanie zaufania jest wgląd w nieoczekiwane zdarzenia w czasie rzeczywistym. Pozostawienie klientów samych i bez informacji już nie wchodzi w rachubę.
5. Jest tylko kwestią czasu, kiedy każdy poważny dostawca SaaS będzie oferował dostępny publicznie panel kondycji usługi. Użytkownicy będą się tego domagać.

ZASOBY DODATKOWE

- Wiele typowych problemów napotykanych przez organizacje IT omówiono w pierwszej części książki *Projekt Feniks. Powieść o IT, modelu DevOps i o tym, jak pomóc firmie w odniesieniu sukcesu* autorstwa Gene'a Kima, Kevina Behra i George'a Spafforda.
- Niżej wymieniony film prezentuje wykład Paula O'Neilla podczas jego kadencji jako CEO firmy Alcoa. Zawiera również opis śledztwa, jakie miało miejsce po śmiertelnym wypadku nastoletniego pracownika w jednej z fabryk Alcoa: https://www.youtube.com/watch?v=tC2ucDs_XJY.
- Więcej informacji na temat mapowania strumienia wartości można znaleźć w książce *Value Stream Mapping. How to Visualize Work and Align Leadership for Organizational Transformation* Karen Martin i Mike'a Osterlinga.
- Więcej informacji na temat systemów ORM można znaleźć w serwisie Stack Overflow: <http://stackoverflow.com/questions/1279613/what-is-an-orm-and-where-can-i-learn-more-about-it>.
- Doskonałym źródłem informacji na temat rytuałów zwanego wytwarzania oprogramowania oraz stosowania ich w pracy operacji IT można znaleźć w serii postów napisanych na blogu „Agile Admin”: <http://theagileadmin.com/2011/02/21/scrum-for-operations-what-is-scrum/>.

- Więcej informacji na temat architektury umożliwiającej szybkie komplikacje można znaleźć w poście na blogu Daniela Worthingtona-Bodarta „Crazy Fast Build Times (or When 10 Seconds Starts to Make You Nervous)”: <http://dan.bodar.com/2012/02/28/crazy-fast-build-times-or-when-10-seconds-starts-to-make-you-nervous/>.
- Więcej informacji na temat testowania wydajności na Facebooku wraz z niektórymi szczegółowymi danymi dotyczącymi procesu publikacji na Facebooku można znaleźć w prezentacji Chuka Rossiego: „The Facebook Release Process”: <http://www.infoq.com/presentations/Facebook-Release-Process>.
- Wiele dodatkowych wariantów techniki dark launching można znaleźć w rozdziale 8. książki Thomasa A. Limoncelliego, Strata R. Chalupa i Christina J. Hogana *The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems*, tom 2.
- Doskonały opis techniczny przełączników funkcji można znaleźć pod tym linkiem: <http://martinfowler.com/articles/feature-toggles.html>.
- Proces publikacji został omówiony bardziej szczegółowo w książkach Thomasa A. Limoncelliego, Strata R. Chalupa i Christina J. Hogana *The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems*, tom 2; *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation* Jęza Humble'a i Davida Farleya oraz *Release It! Design and Deploy Production-Ready Software* Michaela T. Nygarda.
- Opis wzorca *circuit breaker* można znaleźć tutaj: <http://martinfowler.com/bliki/CircuitBreaker.html>.
- Więcej informacji na temat kosztów opóźnień można znaleźć w książce Donalda G. Reinertsena *Second Generation Lean Product Development*.
- Dodatkowy opis profilaktyki zapobiegania awariom dla usługi Amazon S3 można znaleźć tutaj: https://qconsf.com/sf2010/dl/qcon-sanfran-2009/slides/JasonMcHugh_AmazonS3ArchitectingForResiliencyInTheFaceOfFailures.pdf.
- Doskonały przewodnik prowadzenia badań użytkowników można znaleźć w książce Jeffa Gothelfa i Joshua Seidena: *Applying Lean Principles to Improve User Experience*.
- Which Test Won? to witryna, na której są prezentowane setki rzeczywistych testów A/B wraz z pytaniami do przeglądającego o odgadnięcie wariantu, który zapewnił lepszą wydajność. Witryna wzmacnia przekonanie, że jeśli faktycznie czegoś nie przetestujemy, to możemy tylko zgadywać. Witryna jest dostępna pod tym adresem: <http://whichtestwon.com/>.

- Listę wzorców architektonicznych można znaleźć w książce Michaela T. Nygara *Release It! Design and Deploy Production-Ready Software*.
- Przykład opublikowanych notatek ze spotkania post-mortem w firmie Chef można znaleźć pod adresem: <https://www.chef.io/blog/2014/08/14/cookbook-dependency-api-postmortem/>. Nagranie wideo ze spotkania można znaleźć tutaj: <https://www.youtube.com/watch?v=Rmi1Tn5oWfI>.
- Bieżący harmonogram nadchodzących konferencji DevOpsDays można znaleźć w witrynie internetowej DevOpsDays: <http://www.devopsdays.org/>. Instrukcje dotyczące organizowania nowych konferencji DevOpsDays można znaleźć na stronie DevOpsDay Organizing Guide: <https://www.devopsdays.org/pages/organizing/>.
- Więcej informacji na temat narzędzi do zarządzania sekretami można znaleźć w poście na blogu Noah Kantrowitza „Secrets Management and Chef”: <https://coderanger.net/chef-secrets/>.
- James Wickett i Gareth Rushgrove umieścili wszystkie swoje przykłady bezpiecznych potoków wdrożeń w witrynie GitHub: <https://github.com/secure-pipeline>.
- Witrynę bazy danych słabych punktów National Vulnerability Database oraz źródeł danych XML można znaleźć pod adresem: <https://nvd.nist.gov/>.
- Konkretny scenariusz opisujący integrację narzędzi Puppet oraz Go and Mingle firmy ThoughtWorks (aplikacja do zarządzania projektami) można znaleźć w poście na blogu „Puppet Labs” autorstwa Andrew Cunninghama i Andrew Myersa, edytowanego przez Jeza Humble'a: <https://puppetlabs.com/blog/a-deployment-pipeline-for-infrastructure>.
- Przygotowania i przebieg audytów zgodności z przepisami omówiono w prezentacji Jasona Chana z 2015 roku SEC310: *Splitting the Check on Compliance and Security: Keeping Developers and Auditors Happy in the Cloud*: https://www.youtube.com/watch?v=Io00_K4v12Y&feature=youtu.be.
- Historia transformacji konfiguracji aplikacji przez Jeza Humble'a i Davida Farleya dla Oracle WebLogic została opisana w książce *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Mirco Hering opisał ogólne podejście do tego procesu pod tym adresem: <http://notafactoryanymore.com/2015/10/19/devops-for-systems-of-record-a-new-hope-preview-of-does-talk/>.
- Listę przykładowych wymagań operacyjnych DevOps można znaleźć tutaj: <http://blog.devopsguys.com/2013/12/19/the-top-ten-devops-operational-requirements/>.

PRZYPISY KOŃCOWE

WPROWADZENIE

- 18 Przed rewolucją... Eliyahu M. Goldratt, *Beyond the Goal: Eliyahu Goldratt Speaks on the Theory of Constraints (Your Coach in a Box)* (Prince Frederick, Maryland, Gildan Media, 2005), audiobook.
- 19 Można to ująć jeszcze zwięzlej... Jeff Immelt, GE CEO *Jeff Immelt: Let's Finally End the Debate over Whether We Are in a Tech Bubble*, „Business Insider”, 9 grudnia 2015, <http://www.businessinsider.com/ceo-of-ge-lets-finally-end-the-debate-over-whether-we-are-in-a-tech-bubble-2015-12?IR=T>.
- 20 Można też zacytować Jeffreya Snovera... *Weekly Top 10: Your DevOps Flavor*, „Electric Cloud”, 1 kwietnia 2016, <http://electric-cloud.com/blog/2016/04/weekly-top-10-devops-flavor/>.
- 21 Dr Eliyahu M. Goldratt... Goldratt, Beyond the Goal.
- 23 Jak powiedział Christopher Little... osobista korespondencja Christophera Little z Gene’em Kimem, 2010.
- 23 Jak zauważyl Steven J. Spear... Steven J. Spear, *The High-Velocity Edge: How Market Leaders Leverage Operational Excellence to Beat the Competition* (Nowy Jork, NY, McGraw Hill Education), wydanie Kindle edition, rozdział 3.

- 23 W 2013 roku... Chris Skinner, *Banks have bigger development shops than Microsoft*, blog Chrisa Skinnera, dostęp 28 lipca 2016, <http://thefinanser.com/2011/09/banks-have-bigger-development-shops-than-microsoft.html/>.
- 23 Dla projektów zarząd zwykle... Nico Stehr i Reiner Grundmann, *Knowledge: Critical Concepts, Volume 3* (Londyn, Routledge, 2005), s. 139.
- 24 Na przykład dr Vernon Richardson... A. Masli, V. Richardson, M. Widenmier i R. Zmud, *Senior Executive's IT Management Responsibilities: Serious IT Deficiencies and CEO-CFO Turnover*, „MIS Quarterly” (opublikowany elektronicznie, 21 czerwca 2016).
- 24 Weźmy pod uwagę poniższe obliczenia... IDC Forecasts Worldwide IT Spending to Grow 6% in 2012, Despite Economic Uncertainty, „Business Wire”, 10 września 2012, <http://www.businesswire.com/news/home/20120910005280/en/IDC-Forecasts-Worldwide-Spending-Grow-6-2012>.
- 28 Pierwszą niespodziankę... Nigel Kersten, IT Revolution i PwC, *2015 State of DevOps Report* (Portland, OR: Puppet Labs, 2015), https://puppet.com/resources/white-paper/2015-state-of-devops-report?_ga=1.6612658.168869.1464412647&link=%2blog.
- 29 Problem ten został wskazany... Frederick P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition* (Upper Saddle River, NJ, Addison-Wesley, 1995).
- 29 Jak zaobserwował Randy Shoup... Gene Kim, Gary Gruver, Randy Shoup i Andrew Phillips, *Exploring the Uncharted Territory of Microservices*, XebiaLabs.com, webinar, 20 lutego 2015, <https://xebialabs.com/community/webinars/exploring-the-uncharted-territory-of-microservices/>.
- 29 W raporcie *State of DevOps Report* z 2015 roku... Kersten, IT Revolution i PwC, *2015 State of DevOps Report*.
- 29 Innym, bardziej skrajnym przykładem... *Velocity 2011: Jon Jenkins*, „Velocity Culture”, nagranie YouTube, 15:13, opublikowane przez O'Reilly, 20 czerwca 2011, <https://www.youtube.com/watch?v=dxk8b9rSKOo>; *Transforming Software Development*, nagranie YouTube, 40:57, opublikowane przez Amazon Web Service, 10 kwietnia 2015, <https://www.youtube.com/watch?v=YCrhemssYuI&feature=youtu.be>.
- 30 W późniejszym czasie trwania kariery... Eliyahu M. Goldratt, *Beyond the Goal*.
- 31 Podobnie jak w przypadku... JGFLL, recenzja książki Gene'a Kima, Kevina Behra i George'a Spafforda *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win*, 4 marca 2013, <http://www.amazon.com/review/R1KSSPTEGLWJ23>; Mark L. Townsend, recenzja książki Gene'a Kima, Kevina Behra i George'a Spafforda *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win*, recenzja Amazon.com, 2 marca 2013, http://uedata.amazon.com/gp/customer-reviews/R1097DFODM12VD/ref=cm_cr_getr_d_rvw_ttl?ie=UTF8&ASIN=%B00VATFAMI; Scott Van Den Elzen, recenzja książki Gene'a Kima, Kevina Behra

i George'a Spafforda *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win*, 13 marca 2013, http://uedata.amazon.com/gp/customer-reviews/R2K95XEH5OL3Q5/ref=cm_cr_getr_d_rvw_ttl?ie=UTF8&ASIN=B00VATFAMI.

CZĘŚĆ I. WPROWADZENIE

- 36 *Jedną z kluczowych zasad było...* Kent Beck et al., *Twelve Principles of Agile Software*, AgileManifesto.org, 2001, <http://agilemanifesto.org/principles.html>.
- 38 *Doszedł do wniosku, że...* Mike Rother, *Toyota Kata: Managing People for Improvement, Adaptiveness and Superior Results* (Nowy Jork, McGraw Hill, 2010), wydanie Kindle, część III.

ROZDZIAŁ 1.

- 39 *Karen Martin i Mike Osterling...* Karen Martin i Mike Osterling, *Value Stream Mapping: How to Visualize Work and Align Leadership for Organizational Transformation* (Nowy Jork, McGraw Hill, 2013), wydanie Kindle, rozdział 1.
- 41 *Będziemy w tej książce...* Tamże, rozdział 3.
- 43 *Karen Martin i...* Tamże.

ROZDZIAŁ 2.

- 50 *Badania wykazały...* Joshua S. Rubinstein, David E. Meyer i Jeffrey E. Evans, *Executive Control of Cognitive Processes in Task Switching*, *Journal of Experimental Psychology: Human Perception and Performance* 27, nr 4 (2001): 763 – 797, doi: 10.1037//0096-1523.27.4.763, <http://www.umich.edu/~bcalab/documents/Rubinstein%26MeyerEvans2001.pdf>.
- 50 *Dominica DeGrandis, jeden...* DOES15 — Dominica DeGrandis — *The Shape of Uncertainty*, nagranie YouTube, 22:54, opublikowane na konferencji DevOps Enterprise Summit, 5 listopada 2015, <https://www.youtube.com/watch?v=Gp05i0d34gg>.
- 50 *Taiichi Ohno porównał...* Sami Bahri, *Few Patients-In-Process and Less Safety Scheduling; Incoming Supplies are Secondary*, blog The Deming Institute, 22 sierpnia 2013, <https://blog.deming.org/2013/08/fewer-patients-in-process-and-less-safety-scheduling-incoming-supplies-are-secondary/>.
- 50 *Innymi słowy...* Spotkanie pomiędzy Davidem J. Andersenem i zespołem firmy Motorola a Danielem S. Vacantim, 24 lutego 2004; historia opowiadana na konferencji USC CSSE Research Review Barry'emu Boehmowi w marcu 2004 roku.
- 51 *Olbrzymie różnice...* James P. Womack i Daniel T. Jones, *Lean Thinking: Banish Waste and Create Wealth in Your Corporation* (Nowy Jork, Free Press, 2010), wydanie Kindle, rozdział 1.

- 52 Istnieje wiele technik... Eric Ries, *Work in small batches*, StartupLessonsLearned.com, 20 lutego 2009, <http://www.startuplessonslearned.com/2009/02/work-in-small-batches.html>.
- 54 Dr Goldratt w książce... Goldratt, *Beyond the Goal*.
- 54 Jako rozwiązywanie... Eliyahu M. Goldratt, *The Goal: A Process of Ongoing Improvement* (Great Barrington, MA, North River Press, 2014), wydanie Kindle, „Five Focusing Steps”.
- 55 Shigeo Shingo, jeden... Shigeo Shingo, *A Study of the Toyota Production System: From an Industrial Engineering Viewpoint* (Londyn, Productivity Press, 1989); „The 7 Wastes (Seven forms of Muda)”, BeyondLean.com, dostęp 28 lipca 2016, <http://www.beyondlean.com/7-wastes.html>.
- 56 Mary i Tom Poppendieck w książce... Mary Poppendieck i Tom Poppendieck, *Implementing Lean Software: From Concept to Cash* (Upper Saddle River, NJ, Addison-Wesley, 2007), s. 74.
- 56 W książce wymieniono... zaczerpnięte z książki Damona Edwardsa *DevOps Kaizen: Find and Fix What Is Really Behind Your Problems*, Slideshare.net, opublikowane przez dev2ops, 4 maja 2015, <http://www.slideshare.net/dev2ops/dev-ops-kaizen-damon-edwards>.

ROZDZIAŁ 3.

- 60 Dr Charles Perrow... Charles Perrow, *Normal Accidents: Living with High Risk Technologies* (Princeton, NJ, Princeton University Press, 1999).
- 60 Dr Sidney Dekker... Sidney Dekker, *The Field Guide to Understanding Human Error* (Lund University, Szwecja, Ashgate, 2006).
- 60 Po zdekodowaniu... Spear, *The High-Velocity Edge*, rozdział 8.
- 60 Dr Spear rozszerzył... Tamże.
- 61 Dr Peter Senge... Peter M. Senge, *The Fifth Discipline: The Art & Practice of the Learning Organization* (Nowy Jork, Doubleday, 2006), wydanie Kindle, rozdział 5.
- 61 W jednym z dobrze udokumentowanych... NUMMI, „This American Life”, 26 marca 2010, <http://www.thisamericanlife.org/radio-archives/episode/403/transcript>.
- 62 Jak powiedziała Elisabeth Hendrickson... DOES15 — Elisabeth Hendrickson — *Its All About Feedback*, nagranie YouTube, 34:47, opublikowane przez DevOps Enterprise Summit, 5 listopada 2015, <https://www.youtube.com/watch?v=r2BFTXBundQ>.
- 63 Mówi, że... Spear, *The High-Velocity Edge*, rozdział 1.
- 64 Jak zauważa dr Spear... Tamże, rozdział 4.
- 65 Oto kilka przykładów nieskutecznych... Jez Humble, Joanne Molesky i Barry O'Reilly, *Lean Enterprise: How High Performance Organizations Innovate at Scale* (Sebastopol, CA, O'Reilly Media, 2015), wydanie Kindle, część IV.

- 65 W XVIII wieku... Dr Thomas Sowell, *Knowledge and Decisions* (Nowy Jork, Basic Books, 1980), s. 222.
- 66 Jak zauważył Gary Gruver... Gary Gruver, osobista korespondencja z Gene'em Kimem, 2014.

ROZDZIAŁ 4.

- 69 Na przykład w zakładzie GM Fremont... Paul Adler, „Time-and-Motion Regained”, *Harvard Business Review*, styczeń – luty 1993, <https://hbr.org/1993/01/time-and-motion-regained>.
- 70 Postępowanie „wskazać, obarczyć winą i zdyskredytować”... Dekker, *The Field Guide to Understanding Human Error*, rozdział 1.
- 70 Dr Sidney Dekker... *Just Culture: Balancing Safety and Accountability*, Lund University, Human Factors & System Safety website, 6 listopada 2015, <http://www.humanfactors.lth.se/sidney-dekker/books/just-culture/>.
- 71 Zauważył, że w organizacjach... Ron Westrum, *The study of information flow: A personal journey*, materiały z konferencji Safety Science 67 (sierpień 2014), s. 58 – 63, https://www.researchgate.net/publication/261186680_The_study_of_information_flow_A_personal_journey.
- 72 Odkrycia dr. Westruma... Nicole Forsgren Velasquez, Gene Kim, Nigel Kersten i Jez Humble, raport 2014 *State of DevOps* (Portland, OR, Puppet Labs, IT Revolution Press i ThoughtWorks, 2014), <http://puppetlabs.com/2014-devops-report>.
- 72 Jak powiedziała Bethany Macri... Bethany Macri, *Morgue: Helping Better Understand Events by Building a Post Mortem Tool — Bethany Macri*, nagranie w serwisie Vimeo, 33:34, opublikowane przez info@devopsdays.org, 18 października 2013, <http://vimeo.com/77206751>.
- 72 Dr Spear zaobserwował... Spear, *The High-Velocity Edge*, rozdział 1.
- 72 W książce *The Fifth*... Senge, *The Fifth Discipline*, rozdział 1.
- 72 Mike Rother w Toyota Kata zaobserwował... Mike Rother, *Toyota Kata*, s. 12.
- 72 Dlatego właśnie Mike Orzen... Mike Orzen, osobista korespondencja z Gene'em Kimem, 2012.
- 73 Rozważmy opisany poniżej... Paul O'Neill, „Forbes”, 11 października 2001, <http://www.forbes.com/2001/10/16/poneill.html>.
- 73 W 1987 roku statystyki wypadków... Spear, *The High-Velocity Edge*, rozdział 4.
- 73 Jak napisał dr Spear... Tamże.
- 74 Znakomitym przykładem... Tamże, rozdział 5.5.
- 75 Nassim Nicholas Taleb... Nassim Nicholas Taleb, *Antifragile: Things That Gain from Disorder* (Incerto), (Nowy Jork, Random House, 2012).

- 76 Według Womacka... Jim Womack, *Gemba Walks* (Cambridge, MA, Lean Enterprise Institute, 2011), wydanie Kindle, lokalizacja 4113.
- 76 Mike Rother sformalizował... Rother, *Toyota Kata*, część IV.
- 77 Mike Rother powiedział... Tamże, „Podsumowanie”.

ROZDZIAŁ 5.

- 83 W związku z tym trzeba... Michael Rembetsy i Patrick McDonnell, *Continuously Deploying Culture [at Etsy]*, Slideshare.net, 4 października 2012, opublikowane przez Patricka McDonnela, <http://www.slideshare.net/mcdonmps/continuously-deploying-culture-scaling-culture-at-etsy-14588485>.
- 83 W 2015 roku roczne przychody... Profil firmy Nordstrom, Inc. na Vault com, <http://www.vault.com/company-profiles/retail/nordstrom,-inc/company-overview.aspx>.
- 83 „Scena” dla przekształceń... DOES14 — Courtney Kissler — Christian Dior — Transforming to a Culture of Continuous Improvement, nagranie YouTube 29:59, opublikowane przez DevOps Enterprise Summit 2014, 29 października 2014, <https://www.youtube.com/watch?v=0ZAcslZBSlo>.
- 84 Firmy te czasami określano... Tom Gardner, Barnes & Noble, *Blockbuster, Borders: The Killer B's Are Dying*, „The Motley Fool”, 21 lipca 2010, <http://www.fool.com/investing/general/2010/07/21/barnes-noble-blockbuster-borders-the-killer-bs-are.aspx>.
- 84 Jak opisała Kissler... Kissler, DOES14 — Courtney Kissler — Nordstrom.
- 84 Jak powiedziała Kissler... Tamże; wstawki do cytatów dokonane przez Courtney Kissler w prywatnej korespondencji z Gene'em Kimem, 2016.
- 85 Jak powiedziała Kissler... Tamże; wstawki do cytatów dokonane przez Courtney Kissler w prywatnej korespondencji z Gene'em Kimem, 2016.
- 85 W 2015 roku Kissler... Tamże.
- 86 W dalszej części wypowiedzi kontynuowała... Tamże.
- 86 Kissler stwierdziła na koniec: „Z perspektywy... Tamże.
- 87 Przykładem projektu DevOps... Ernest Mueller, *Business model driven cloud adoption: what NI Is doing in the cloud*, Slideshare.net, 28 czerwca 2011, opublikowane przez Ernesta Muellera, <https://www.slideshare.net/mxyzplk/business-model-driven-cloud-adoption-what-ni-is-doing-in-the-cloud>.
- 87 Chociaż wiele osób uważa... Niepublikowane obliczenia Gene'a Kima po szczytcie 2014 DevOps Enterprise Summit. Zgodnie z jednym z wniosków... Kersten, IT Revolution i PwC, 2015 *State of DevOps Report*.
- 88 CSG (2013): W... Prugh, DOES14: Scott Prugh, CSG — *DevOps i Lean in Legacy Environments*, Slideshare.net, 14 listopada 2014, opublikowane przez DevOps Enterprise Summit, <http://www.slideshare.net/DevOpsEnterpriseSummit/scott-prugh>.

- 88 *Etsy* (2009): W... Rembetsy i McDonnell, *Continuously Deploying Culture [at Etsy]*.
- 88 *Firma badawcza Gartner*... Bernard Golden, *What Gartner's Bimodal IT Model Means to Enterprise CIOs*, „CIO Magazine”, 27 stycznia 2015, <http://www.cio.com/article/2875803/cio-role/what-gartner-s-bimodal-it-model-means-to-enterprise-cios.html>.
- 89 *Systemy SoR*... Tamże.
- 89 *Systemy SoA*... Tamże.
- 89 *Dane z raportów*... Kersten, IT Revolution i PwC, *2015 State of DevOps Report*.
- 89 *Scott Prugh, wiceprezesa*... Scott Prugh, osobista korespondencja z Gene’em Kimem, 2014.
- 89 *Geoffrey A. Moore*... Geoffrey A. Moore i Regis McKenna, *Crossing the Chasm: Marketing and Selling High-Tech Products to Mainstream Customers* (Nowy Jork, HarperCollins, 2009), 11.
- 90 *Transformacje „z hukiem”*... *Four Pillars of PayPal’s »Big Bang« Agile Transformation*, „TechTarget”, sierpień 2014, <http://searchcio.techtarget.com/feature/Four-pillars-of-PayPals-big-bang-Agile-transformation>.
- 91 *Na poniższej liście*... *Creating High Velocity Organizations*, opis kursu Roberta Fernandeza i Steve'a Speara, witryna WWW MIT Sloan Executive Education, dostęp 30 maja 2016, <http://executive.mit.edu/openenrollment/program/organizational-development-high-velocity-organizations>.
- 91 *Ale jak powiedział Ron van Kemenade*... Ron van Kemenade, *Nothing Beats Engineering Talent: The Agile Transformation at ING*, prezentacja na konferencji DevOps Enterprise Summit, Londyn, Wielka Brytania, 30 czerwca – 1 lipca 2016.
- 92 *Peter Drucker*... Leigh Buchanan, *The Wisdom of Peter Drucker from A to Z*, Inc., 19 listopada 2009, <http://www.inc.com/articles/2009/11/drucker.html>.

ROZDZIAŁ 6.

- 93 *Z biegiem lat*... Kissler, „DOES14 — Courtney Kissler — Nordstrom”.
- 94 *Kissler wyjaśniła*... Ross Clanton i Michael Dudy, wywiad z Courtney Kissler i Jasonem Josephym, *Continuous Improvement at Nordstrom*, „The Goat Farm”, nagranie podcast, 25 czerwca 2015, <http://goatcan.do/2015/06/25/the-goat-farm-episode-7-continuous-improvement-at-nordstrom/>.
- 94 *Kissler z dumą stwierdziła*... Tamże.
- 95 *Menedżerowie techniczni lub*... Brian Maskell, *What Does This Guy Do? Role of Value Stream Manager*, „Maskell”, 3 lipca 2015, <http://blog.maskell.com/?p=2106http://www.lean.org/common/display/?o=221>.

- 96 *Damon Edwards, współgospodarz... Damon Edwards, DevOps Kaizen: Find and Fix What Is Really Behind Your Problems*, Slideshare.net, opublikowane przez dev2ops, 4 maja 2015, <http://www.slideshare.net/dev2ops/dev-ops-kaizen-damon-edwards>.
- 98 *Dr Vijay Govindarajan i dr Chris Trimble... Vijay Govindarajan i Chris Trimble, The Other Side of Innovation: Solving the Execution Challenge* (Boston, MA, Harvard Business Review, 2010), wydanie Kindle.
- 99 *Na podstawie swoich... Tamże, część I.*
- 102 *Po „doświadczeniach bliskich śmierci”... Marty Cagan, Inspired: How to Create Products Customers Love* (Saratoga, CA, SVPG Press, 2008), s. 12.
- 102 *Cagan zauważył, że... Tamże.*
- 103 *Sześć miesięcy po... Ashlee Vance, LinkedIn: A Story About Silicon Valley’s Possibly Unhealthy Need for Speed, „Bloomberg”, 30 kwietnia 2013, <http://www.bloomberg.com/bw/articles/2013-04-29/linkedin-a-story-about-silicon-valleys-possibly-unhealthy-need-for-speed>.*
- 103 *Firma LinkedIn została utworzona... LinkedIn started back in 2003 — Scaling LinkedIn — A Brief History*, Slideshare.net, opublikowane przez Joshua Clemmę, 9 listopada 2015, http://www.slideshare.net/joshclemm/how-linkedin-scaled-a-brief-history/3-LinkedIn_started_back_in_2003.
- 103 *Rok później... Jonas Klit Nielsen, 8 Years with LinkedIn — Looking at the Growth [Infographic], MindJumpers.com, 10 maja 2011, <http://www.mindjumpers.com/blog/2011/05/linkedin-growth-infographic/>.*
- 103 *W listopadzie 2015 roku... LinkedIn started back in 2003*, Slideshare.net.
- 103 *Problem polegał na... From a Monolith to Microservices + REST: The Evolution of LinkedIn’s Architecture*, Slideshare.net, opublikowany przez Karana Parikha, 6 listopada 2014, <http://www.slideshare.net/parikhk/restli-and-deco>.
- 103 *Josh Clemm... LinkedIn started back in 2003*, Slideshare.net.
- 103 *W 2013 r. publicystka... Vance, LinkedIn: A Story About, „Bloomberg”.*
- 104 *Scott rozpoczął operację... How I Structured Engineering Teams at LinkedIn and AdMob for Success, „First Round Review”, 2015, <http://firstround.com/review/how-i-structured-engineering-teams-at-linkedin-and-admob-for-success/>.*
- 104 *Scott opisał jeden... Ashlee Vance, Inside Operation InVersion, the Code Freeze that Saved LinkedIn, „Bloomberg”, 11 kwietnia 2013, <http://www.bloomberg.com/news/articles/2013-04-10/inside-operation-inversion-the-code-freeze-that-saved-linkedin>.*
- 104 *Vance opisał jednak... Vance, LinkedIn: A Story About, „Bloomberg”.*
- 104 *Zgodnie z tym, co napisał Josh Clemm... LinkedIn started back in 2003*, Slideshare.net.
- 104 *Kevin Scott stwierdził... How I Structured Engineering Teams, „First Round Review”.*

- 105 *Jak powiedział Christopher Little... Osobista korespondencja Christophera Little'a z Gene'em Kimem, 2011.*
- 106 *Jak powiedział Ryan Martens... Ryan Martens, osobista korespondencja z Gene'em Kimem, 2013.*

ROZDZIAŁ 7.

- 107 *Zauważył, że „po... Dr Melvin E. Conway, How Do Committees Invent? Mel-Conway.com, <http://www.melconway.com/research/committees.html>, wcześniej opublikowany w magazynie „Datamation”, kwiecień 1968.*
- 107 *Te obserwacje doprowadziły... Tamże.*
- 107 *Eric S. Raymond, autor... Eric S. Raymond, Conway's Law, catb.org, dostęp 31 maja 2016, <http://catb.org/~esr/jargon/>.*
- 108 *Transformacje DevOps w Etsy... Sarah Buhr, Etsy Closes Up 86 Percent on First Day of Trading, „Tech Crunch”, 16 kwietnia 2015, <http://techcrunch.com/2015/04/16/etsy-stock-surges-86-percent-at-close-of-first-day-of-trading-to-30-per-share/>.*
- 108 *Jak podczas prezentacji... Scaling Etsy: What Went Wrong, What Went Right, Slideshare.net, prezentacja opublikowana przez Rossa Snydera, 5 października 2011, <http://www.slideshare.net/beamrider9/scaling-etsy-what-went-wrong-what-went-right>.*
- 108 *Jak zaobserwował Snyder... Tamże.*
- 108 *Innymi słowy... Sean Gallagher, When „Clever” Goes Wrong: How Etsy Overcame Poor Architectural Choices, „Arstechnica”, 3 października 2011, <http://arstechnica.com/business/2011/10/when-clever-goes-wrong-how-etsy-overcame-poor-architectural-choices/>.*
- 109 *Snyder wyjaśnił, że... Scaling Etsy Slideshare.net.*
- 109 *W firmie Etsy początkowo były... Tamże.*
- 109 *Na wiosnę 2009... Tamże.*
- 109 *Jak opisywał Snyder... Ross Snyder, Surge 2011 — Scaling Etsy: What Went Wrong, What Went Right, nagranie YouTube, opublikowane przez Surge Conference, 23 grudnia 2011, <https://www.youtube.com/watch?v=eenrfm50mXw>.*
- 110 *Snyder powiedział ... Tamże.*
- 110 *Sprouter był jedną z... Continuously Deploying Culture: Scaling Culture at Etsy — Velocity Europe 2012, Slideshare.net, opublikowane przez Patricka McDonnella, 4 października 2012, <http://www.slideshare.net/mcdonmps/continuously-deploying-culture-scaling-culture-at-etsy-14588485>.*
- 110 *Zostały one zdefiniowane przez... Creating High Velocity Organizations, opis kursu Roberta Fernandeza i Stevena Speara.*
- 112 *Adrian Cockcroft zauważył... Adrian Cockcroft, osobista korespondencja z Gene'em Kimem, 2014.*

- 114 *Podczas ruchu Lean...* Spear, *The High-Velocity Edge*, rozdział 8.
- 114 *Jak napisał Mike Rother...* Rother, Toyota Kata, 250.
- 114 *Podczas rozwijań nad...* DOES15 — Jody magnezu — *DevOps in the Enterprise: A Transformation Journey*, nagranie YouTube, 28:22, opublikowane przez DevOps Enterprise Summit, 5 listopada 2015, <https://www.youtube.com/watch?v=USYrJDaPEFtM>.
- 115 *Jody Mulkey kontynuuje...* Tamże.
- 115 *Pedro Canahuati, dyrektor...* Pedro Canahuati, *Growing from the Few to the Many: Scaling the Operations Organization at Facebook*, „InfoQ”, 16 grudnia 2013, <http://www.infoq.com/presentations/scaling-operations-facebook>.
- 115 *Nadmierna specjalizacja działów...* Spear, *The High-Velocity Edge*, rozdział 1.
- 116 *Scott Prugh opisał...* Scott Prugh, *Continuous Delivery*, „Scaled Agile Framework”, aktualizacja 14 lutego 2013, <http://www.scaledagileframework.com/continuous-delivery/>.
- 116 „Dzięki interdyscyplinarnemu szkoleniu... Tamże.
- 116 „Tradycyjni menedżerowie często... Tamże.
- 116 *Ponadto, jak zauważyl Prugh...* Tamże.
- 117 *Gdy cenimy ludzi...* Dr Carol Dweck, *Carol Dweck Revisits the „Growth Mindset”*, „Education Week”, 22 września 2015, <http://www.edweek.org/ew/articles/2015/09/23/carol-dweck-revisits-the-growth-mindset.html>.
- 117 *Jak opisuje Jason Cox...* *Disney DevOps: To Infinity and Beyond*, prezentacja DevOps Enterprise Summit 2014, San Francisco, CA, październik 2014.
- 117 *Jak powiedział John Lauderbach...* John Lauderbach, osobista rozmowa z Gene'em Kimem, 2001.
- 119 *Te właściwości mają również...* Tony Mauro, *Adopting Microservices at Netflix: Lessons for Architectural Design*, NGINX, 19 lutego 2015, <https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/>; Adam Wiggins, *The Twelve-Factor App*, 12Factor.net, 30 stycznia 2012, <http://12factor.net/>.
- 119 *Randy Shoup, były...* *Exploring the Uncharted Territory of Microservices*, nagranie YouTube, 56:50, opublikowane przez XebiaLabs, Inc., 20 lutego 2015, <https://www.youtube.com/watch?v=MRa21icSIQk>.
- 120 *W ramach inicjatywy...* Humble, O'Reilly i Molesky, *Lean Enterprise*, część III.
- 120 *W kulturze firmy Netflix...* Reed Hastings, *Netflix Culture: Freedom and Responsibility*, Slideshare.net, 1 sierpnia 2009, <http://www.slideshare.net/reed2001/culture-1798664>.
- 120 *W 2005 roku CTO firmy Amazon...* Larry Dignan, *Little Things Add Up, „Baseline”*, 19 października 2005, <http://www.baselinemag.com/c/a/ProjectsManagement/Profiles-Lessons-From-the-Leaders-in-the-iBaselinei500/3>.

- 121 *Target jest szóstą...* Heather Mickman i Ross Clanton, *DOES15 — Heather Mickman & Ross Clanton — (Re)building an Engineering Culture: DevOps at Target*, nagranie YouTube, 33:39, opublikowane przez DevOps Enterprise Summit, 5 listopada 2015, <https://www.youtube.com/watch?v=7s-VbB1fG5o>.
- 121 *Jak opisuje Mickman...* Tamże.
- 122 *W 2012 roku, próbując...* Tamże.
- 122 *Ponieważ nasz zespół...* Tamże.
- 122 *W kolejnych dwóch latach...* Tamże.
- 122 *Wprowadzone zmiany przyniosły...* Tamże.
- 123 *Zespół API Enablement...* Tamże.

ROZDZIAŁ 8.

- 125 *W firmie Big Fish...* *Big Fish Celebrates 11th Consecutive Year of Record Growth*, BigFishGames.com, 28 stycznia 2014, <http://pressroom.bigfishgames.com/2014-01-28-Big-Fish-Celebrates-11th-Consecutive-Year-of-Record-Growth>.
- 126 *Jak zauważył: „Gdy...* Paul Farrall, osobista korespondencja z Gene'em Kimem, styczeń 2015.
- 126 *Farrall zdefiniował dwa...* Tamże, 2014.
- 126 *Jak stwierdził: „Model...* Tamże.
- 127 *Ernest Mueller zaobserwował...* Ernest Mueller, osobista korespondencja z Gene'em Kimem, 2014.
- 127 *Jak zauważył Damon Edwards...* Edwards, *DevOps Kaizen*.
- 128 *Dianne Marsh dyrektor...* *Dianne Marsh „Introducing Change while Preserving Engineering Velocity”*, nagranie YouTube, 17:37, opublikowane przez Flowcon, 11 listopada 2014, <https://www.youtube.com/watch?v=eW3ZxY67fnc>.
- 129 *Jason Cox powiedział...* Jason Cox, „Disney DevOps”.
- 130 *W firmie Etsy taki model...* *devopsdays Minneapolis 2015 — Katherine Daniels — DevOps: The Missing Pieces*, nagranie YouTube, 33:26, opublikowane przez DevOps Minneapolis, 13 lipca 2015, <https://www.youtube.com/watch?v=LNJkVw93yTU>.
- 131 *Jak zaobserwował Ernest Mueller...* Ernest Mueller, osobista korespondencja z Gene'em Kimem, 2015.
- 132 *Scrum to metodologia zwinnego...* Hirotaka Takeuchi i Ikujiro Nonaka, *New Product Development Game*, „Harvard Business Review” (styczeń 1986): 137 – 146.

ROZDZIAŁ 9.

- 141 *W swojej prezentacji... Em Campbell-Pretty, DOES14 — Em Campbell — Pretty — How a Business Exec Led Agile, Lead, CI/CD*, nagranie YouTube, 29:47, opublikowane przez DevOps Enterprise Summit, 20 kwietnia 2014, <https://www.youtube.com/watch?v=-4pIMMTbtwE>.
- 141 *Campbell-Pretty została... Tamże.*
- 142 *W firmie stworzono... Tamże.*
- 142 *Campbell-Pretty zaobserwowała... Tamże.*
- 142 *Campbell-Pretty opisała... Tamże.*
- 145 *Pierwszym systemem kontroli wersji... Version Control History*, PlasticSCM.com, dostęp 31 maja 2016, <https://www.plasticscm.com/version-control-history.html>.
- 145 *System kontroli wersji... Jennifer Davis i Katherine Daniels, Effective DevOps: Building a Culture of Collaboration, Affinity, and Tooling at Scale* (Sebastopol, CA, O'Reilly Media, 2016), 37.
- 147 *Bill Baker, uznany... Simon Sharwood, Are Your Servers PETS or CATTLE?, „The Register”, 18 marca 2013, http://www.theregister.co.uk/2013/03/18/servers_pets_or_cattle_cern/.*
- 148 *W Netflix średni... Jason Chan, OWASP AppSecUSA 2012: Real World Cloud Application Security, nagranie YouTube, 37:45, opublikowane przez Christiaan008, 10 grudnia 2012, <https://www.youtube.com/watch?v=daNA0jXDvYk>.*
- 148 *Drugi wzorzec jest... Chad Fowler, Trash Your Servers and Burn Your Code: Immutable Infrastructure and Disposable Components*, ChadFowler.com, 23 czerwca 2013, <http://chadfowler.com/2013/06/23/immutable-deployments.html>.
- 149 *Cały stos aplikacji... John Willis, Docker and the Three Ways of DevOps Part 1: The First Way — Systems Thinking, „Docker”, 26 maja 2015, <https://blog.docker.com/2015/05/docker-three-ways-devops/>.*

ROZDZIAŁ 10.

- 153 *Gary Gruver zauważył... Gary Gruver, osobista korespondencja z Gene'em Kimem, 2014.*
- 154 *Występowały różne problemy... DOES15 — Mike Bland — Pain Is Over, If You Want It*, Slideshare.net, opublikowane przez Gene'a Kima, 18 listopada 2015, <http://www.slideshare.net/ITRevolution/does15-mike-bland-painis-over-if-you-want-it-55236521>.
- 154 *Bland opisuje, w jaki sposób... Tamże.*
- 154 *Bland pisał, że... Tamże.*

- 154 *Jak opisuje Bland...* Tamże.
- 154 *Jak zauważał Bland...* Tamże.
- 155 *W ciągu najbliższych...* Tamże.
- 155 Eran Messeri, *inżynier...* Eran Messeri, *What Goes Wrong When Thousands of Engineers Share the Same Continuous Build?*, prezentacja na konferencji GOTO, Aarhus, Dania, 2 października 2013.
- 155 *Messeri wyjaśnia: „W Google...* Tamże.
- 156 *Cały ich kod...* Tamże.
- 156 *Oto niektóre inne...* Tamże.
- 156 *Dla działu Dev ciągła integracja...* Jez Humble i David Farley, osobista korespondencja z Gene'em Kimem, 2012.
- 157 *Potok wdrożeń...* Jez Humble i David Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* (Upper Saddle River, NJ, Addison-Wesley, 2011), s. 3
- 157 *Jez Humble i David Farley...* Tamże, s. 188.
- 161 *Jak zaobserwowali Humble i Farley...* Tamże, s. 258.
- 162 *Martin Fowler zauważał...* Martin Fowler, *Continuous Integration*, MartinFowler.com, 1 maja 2006, <https://www.martinfowler.com/articles/continuousIntegration.html>.
- 163 *Martin Fowler opisał...* Martin Fowler, *TestPyramid*, MartinFowler.com, 1 maja 2012, <http://martinfowler.com/bliki/TestPyramid.html>.
- 165 *Technika ta została...* Martin Fowler, *Test Driven Development*, MartinFowler.com, 5 marca 2005, <http://martinfowler.com/bliki/TestDrivenDevelopment.html>.
- 165 *Nachi Nagappan, E. Michael...* Nachiappan Nagappan, E. Michael Maximilien, Thirumalesh Bhat i Laurie Williams, *Realizing quality improvement through test driven development: results and experiences of four industrial teams*, „Empir Software Engineering”, 13, (2008): 289 – 302, http://research.microsoft.com/en-us/groups/ese/nagappan_tdd.pdf.
- 165 *Elisabeth Hendrickson na konferencji...* *On the Care and Feeding of Feedback Cycles*, Slideshare.net, opublikowane przez Elisabeth Hendrickson, 1 listopada 2013, <http://www.slideshare.net/ehendrickson/care-and-feeding-of-feedback-cycles>.
- 166 *Jednak samo zautomatyzowanie...* *Decreasing false positives in automated testing*, Slideshare.net, opublikowane przez Sauce Labs, 24 marca 2015, <http://www.slideshare.net/saucelabs/decreasing-false-positives-in-automated-testing>; Martin Fowler, *Eradicating Non-determinism in Tests*, MartinFowler.com, 14 kwietnia 2011, <http://martinfowler.com/articles/nonDeterminism.html>.
- 166 *Jak powiedział Gary Gruver...* Gary Gruver, *DOES14 — Gary Gruver — Macy's — Transforming Traditional Enterprise Software Development Processes*, nagranie

- YouTube, 27:24, opublikowane przez DevOps Enterprise Summit 2014, 29 października 2014, <https://www.youtube.com/watch?v=-HSSGiYXA7U>.
- 169 *Randy Shoup, były... Randy Shoup, The Virtuous Cycle of Velocity: What I Learned About Going Fast at eBay and Google by Randy Shoup*, nagranie YouTube, 30:05, opublikowane przez Flowcon, 26 grudnia 2013, <https://www.youtube.com/watch?v=4EwLBoRyXTOI>.
- 171 *Czasami tę sytuację... David West, Water scrum-fall is-reality_of_agile_for_most*, Slideshare.net, opublikowane przez harsoft, 22 kwietnia 2013, <http://www.slideshare.net/harsoft/water-scrumfall-isrealityofagileformost>.

ROZDZIAŁ 11.

- 174 *Gary Gruver dyrektor... Gene Kim, The Amazing DevOps Transformation of the HP LaserJet Firmware Team (Gary Gruver)*, ITRevolution.com, 2013, <http://itrevolution.com/the-amazing-devops-transformation-of-the-hp-laserjet-firmware-team-gary-gruver/>.
- 174 *Gruver opisał ten problem... Tamże.*
- 175 *Do włączania i wyłączania... Tamże.*
- 175 *Gruver przyznaje, że tworzenie... Gary Gruver i Tommy Mouser, Leading the Transformation: Applying Agile and DevOps Principles at Scale* (Portland, OR, IT Revolution Press), s. 60.
- 175 *Gruver zauważył: „Bez... Kim, The Amazing DevOps Transformation*, ITRevolution.com.
- 177 *Jeff Atwood, założyciel... Jeff Atwood, Software Branching and Parallel Universes*, CodingHorror.com, 2 października 2007, <http://blog.codinghorror.com/software-branching-and-parallel-universes/>.
- 178 *Ward Cunningham, programista... Ward Cunningham, Ward Explains Debt Metaphor*, c2.com, 2011, <http://c2.com/cgi/wiki?WardExplainsDebtMetaphor>.
- 179 *Ernest Mueller, który pomógł... Ernest Mueller, 2012: A Release Odyssey*, Slideshare.net, opublikowane przez Ernesta Muellera, 12 marca 2014, <http://www.slideshare.net/mxyzplk/2012-a-release-odyssey>.
- 179 *W tamtym okresie... Bazaarvoice, Inc. Announces Its Financial Results for the Fourth Fiscal Quarter and Fiscal Year Ended April 30, 2012*, BazaarVoice.com, 6 czerwca 2012, <http://investors.bazaarvoice.com/releasedetail.cfm?ReleaseID=680964>.
- 179 *Jak zaobserwował Mueller, „nie... Ernest Mueller, DOES15 — Ernest Mueller — DevOps Transformations At National Instruments and... nagranie YouTube*, 34:14, opublikowane przez DevOps Enterprise Summit, 5 listopada 2015, <https://www.youtube.com/watch?v=6Ry40h1UAyE>.
- 180 *„Dzięki ciągłemu uruchamianiu... Tamże.*
- 181 *Mueller opisywał kolejne sukcesy... Tamże.*
- 181 *Jednak dane... Kersten, IT Revolution i PwC, 2015 State of DevOps Report.*

ROZDZIAŁ 12.

- 183 W 2012 roku Rossi... Chuck Rossi, *Release engineering and push karma: Chuck Rossi*, post na stronie facebookowej Chucka Rossiego, 5 kwietnia 2012, <https://www.facebook.com/notes/facebook-engineering/release-engineering-and-push-karma-chuck-rossi/10150660826788920>.
- 183 Tuż przed wykonaniem... Ryan Paul, *Exclusive: a behind-the-scenes look at Facebook release engineering*, „Ars Technica”, 5 kwietnia 2012, [http://arstechnica.com/business/2012/04/exclusive-a-behind-the-scenes-look-at-facebook-release-engineering/1/](http://arstechnica.com/business/2012/04/exclusive-a-behind-the-scenes-look-at-facebook-release-engineering/).
- 183 Dalej Rossi pisał: „Jeśli... Chuck Rossi, *Release engineering and push karma*.
- 183 Baza kodu warstwy frontend Facebooka... Paul, *Exclusive: a behind-the-scenes look at Facebook release engineering*, „Ars Technica”.
- 184 Wyjaśnił, że... Chuck Rossi, *Ship early and ship twice as often* post na stronie Chucka Rossiego na Facebooku, 3 sierpnia 2012, <https://www.facebook.com/notes/facebook-engineering/ship-early-and-ship-twice-as-often/10150985860363920>.
- 184 Kent Beck, twórca... Kent Beck, *Slow Deployment Causes Meetings*, post na stronie Kenta Becka na Facebooku, 19 listopada 2015, https://www.facebook.com/notes/kent-beck/slow-deployment-causes-meetings/1055427371156793?_rdr=p.
- 187 Scott Prugh, ich... Prugh, *DOES14: Scott Prugh, CSG — DevOps and Lean in Legacy Environments*.
- 187 Prugh zaobserwował: „Było... Tamże.
- 188 Prugh pisał: „Środowiska... Tamże.
- 188 Prugh zauważył również... Tamże.
- 188 Eksperymenty udowodniły, że... *Puppet Labs and IT Revolution Press, 2013 State of DevOps Report* (Portland, OR, Puppet Labs, 2013), <http://exin.vanharen.net/Player/eKnowledge/2013-state-of-devops-report.pdf>.
- 188 Jak informował Prugh... Scott Prugh and Erica Morrison, *DOES15 — Scott Prugh & Erica Morrison — Conway & Taylor Meet the Strangler (v2.0)*, nagranie YouTube 29:39, opublikowane przez DevOps Enterprise Summit, 5 listopada 2015, <https://www.youtube.com/watch?v=tKdIHCL0DUg>.
- 189 Rozważmy poniższy cytat... Tim Tischler, osobista rozmowa z Gene’em Kimem, FlowCon 2013.
- 191 Z raportu *State od DevOps*... Velasquez, Kim, Kersten i Humble, *2014 State of DevOps Report*.
- 192 Proces wdrażania w firmie Etsy... Chad Dickerson, *Optimizing for developer happiness*, CodeAsCraft.com, 6 czerwca 2011, <https://codeascraft.com/2011/06/06/optimizing-for-developer-happiness/>.
- 192 Jak napisał Noah Sussman... Noah Sussman i Laura Beth Denker, *Divide and Conquer*, CodeAsCraft.com, 20 kwietnia 2011, <https://codeascraft.com/2011/04/20/divide-and-concur/>.

- 193 Sussman pisze: „Metodą... Tamże.
- 193 Gdyby wszystkie testy... Tamże.
- 193 Kiedy nadchodzi kolejny... Erik Kastner, *Quantum of Deployment*, CodeasCraft.com, 20 maja 2010, <https://codeascraft.com/2010/05/20/quantum-of-deployment/>.
- 194 W praktyce terminy... Puppet Labs and IT Revolution Press, 2013 *State of DevOps Report*.
- 198 Proces oddzielenia zmian... Timothy Fitz, *Continuous Deployment at IMVU: Doing the impossible fifty times a day*, TimothyFitz.com, 10 lutego 2009, <http://timothyfitz.com/2009/02/10/continuous-deployment-at-imvu-doing-the-impossible-fifty-times-a-day/>.
- 198 Ten wzorzec jest również... Fitz, *Continuous Deployment*, TimothyFitz.com.; Michael Hrenko, „DOES15 — Michael Hrenko — DevOps Insured By Blue Shield of California”, nagranie YouTube, 42:24, opublikowane przez DevOps Enterprise Summit, 5 listopada 2015, <https://www.youtube.com/watch?v=NlgrOT24UDw>.
- 198 Dan North i Dave Farley... Humble i Farley, *Continuous Delivery*, s. 265.
- 200 Wzorzec CIS... Eric Ries, *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses* (Nowy Jork, Random House, 2011), audiobook.
- 202 Zaawansowanym przykładem... Andrew „Boz” Bosworth, *Building and testing at Facebook*, post na stronie Boza na Facebooku, 8 sierpnia 2012, <https://www.facebook.com/notes/facebook-engineering/building-and-testing-at-facebook/10151004157328920>; *Etsy's Feature flagging API used for operational rampups and A/B testing*, GitHub.com, <https://github.com/etsy/feature>, GitHub.com, <https://github.com/Netflix/archaius>.
- 204 W 2009 roku, kiedy... John Allspaw, *Convincing management that cooperation and collaboration was worth it*, KitchenSoap.com, 5 stycznia 2012, <http://www.kitchensoap.com/2012/01/05/convincing-management-that-cooperation-and-collaboration-was-worth-it/>.
- 204 W podobnym tonie wypowiedział się... Rossi, *Release engineering and push karma*.
- 204 Przez prawie dekadę... Emil Protalinski, *Facebook passes 1.55B monthly active users and 1.01B daily active users*, „Venture Beat”, 4 listopada 2015, <http://venturebeat.com/2015/11/04/facebook-passes-1-55b-monthly-active-users-and-1-01-billion-daily-active-users/>.
- 204 Do roku 2015 Facebook... Tamże.
- 205 Eugene Letuchy, inżynier... Eugene Letuchy, *Chat na Facebooku*, post na stronie Eugene'a Letuchy'ego na Facebooku, 3 maja 2008, http://www.facebook.com/note.php?note_id=14218138919&id=944554719.
- 205 Implementacja tej intensywnej obliczeniowo... Tamże.
- 206 Jak napisał Letuchy... Tamże.

- 206 *Jednak w 2015 roku... Jez Humble, osobista korespondencja z Gene'em Kimem, 2014.*
- 206 *Jego zaktualizowane definicje... Tamże.*
- 207 *W firmach Amazon i... Tamże.*

ROZDZIAŁ 13.

- 209 *To jest zasada... Jez Humble, What is Continuous Delivery, ContinuousDelivery.com, dostęp 28 maja 2016, <https://continuousdelivery.com/>.*
- 209 *Zaobserwował, że... Kim, Gruver, Shoup i Phillips, Exploring the Uncharted Territory of Microservices.*
- 209 *Dalej mówi: „Patrząc... Tamże.*
- 209 *Architektura w serwisie eBay... Shoup, From Monolith to Microservices.*
- 211 *Charles Betz, autor... Charles Betz, Architecture and Patterns for IT Service Management, Resource Planning, and Governance: Making Shoes for the Cobbler's Children (Witham, MA, Morgan Kaufmann, 2011), s. 300.*
- 211 *Jak opisuje Randy Shoup... Randy Shoup, From the Monolith to Microservices, Slideshare.net, opublikowane przez Randy'ego Shoupa, 8 października 2014, <http://www.slideshare.net/RandyShoup/goto-aarhus2014-enterprisearchitecturemicroservices>.*
- 212 *Shoup zauważyl: „Organizacje... Tamże.*
- 212 *Jak zauważyl Randy Shoup... Tamże.*
- 214 *Jedną z najczęściej badanych... Werner Vogels, A Conversation with Werner Vogels, „acmqueue” 4, nr 4 (2006), 14 – 22, <http://queue.acm.org/detail.cfm?id=1142065>.*
- 214 *Vogels powiedział Grayowi... Tamże.*
- 214 *Opisując proces myślowy... Tamże.*
- 214 *Vogels zauważyl: „Duże... Tamże.*
- 215 *W 2011 roku firma Amazon... John Jenkins, Velocity 2011: Jon Jenkins, „Velocity Culture”, nagranie wideo na YouTube, 15:13, opublikowane przez O'Reilly, 20 czerwca 2011, <https://www.youtube.com/watch?v=dxk8b9rSKOo>.*
- 215 *Do 2015 r. liczba wdrożeń... Ken Exner, Transforming Software Development, nagranie YouTube, 40:57, opublikowane przez Amazon Web Services, 10 kwietnia 2015, <https://www.youtube.com/watch?v=YCrhemssYuI&feature=youtu.be>.*
- 215 *Określenie strangler application... Martin Fowler, StranglerApplication, MartinFowler.com, 29 czerwca 2004, <http://www.martinfowler.com/bliki/StranglerApplication.html>.*
- 215 *Gdy implementujemy... Boris Lublinsky, Versioning in SOA, The Architecture Journal, kwiecień 2007, <https://msdn.microsoft.com/en-us/library/bb491124.aspx>.*

- 216 Wzorzec aplikacji-dusiciela... Paul Hammant, *Introducing Branch by Abstraction*, PaulHammant.com, 26 kwietnia 2007, http://paulhammant.com/blog/branch_by_abstraction.html.
- 216 Martin Fowler wskazuje... Martin Fowler, *StranglerApplication*, MartinFowler.com, 29 czerwca 2004, <http://www.martinfowler.com/bliki/StranglerApplication.html>.
- 216 Firma Blackboard Inc. jest... Gregory T. Huang, *Blackboard CEO Jay Bhatt on the Global Future of Edtech*, „Xconomy”, 2 czerwca 2014, <http://www.xconomy.com/boston/2014/06/02/blackboard-ceo-jay-bhatt-on-the-global-future-of-edtech/>.
- 217 Jak zaobserwował David Ashman... David Ashman, *DOES14 — David Ashman — Blackboard Learn — Keep Your Head in the Clouds*, nagranie YouTube, 30:43, opublikowane przez DevOps Enterprise Summit 2014, 28 października 2014, <https://www.youtube.com/watch?v=SSmixnMpsI4>.
- 217 W 2010 roku Ashman... Tamże.
- 217 Wpływ złożoności na wydajność... David Ashman, osobista korespondencja z Gene'em Kimem, 2014.
- 217 Ashman stwierdził: „Stało się... Tamże.
- 218 „W rzeczywistości — opisywał Ashman... Tamże.
- 219 Ashman zakończył: „Umożliwienie... Tamże.

ROZDZIAŁ 14.

- 225 W dziale Ops... Kim, Behr i Spafford, *The Visible Ops Handbook: Implementing ITIL in 4 Practical and Auditable Steps* (Eugene, OR, IT Process Institute, 2004), wydanie Kindle, „Wprowadzenie”.
- 225 Natomiast z badania... Tamże.
- 225 Innymi słowy... Tamże.
- 226 Aby skorzystać z tego... Telemetry, Wikipedia, strona ostatnio zmodyfikowana 5 maja 2016, <https://en.wikipedia.org/wiki/Telemetry>.
- 226 McDonnell opisał zagrożenia... Michael Rembetsky i Patrick McDonnell, *Continuously Deploying Culture: Scaling Culture at Etsy — Velocity Europe 2012*, Slideshare.net, opublikowane przez Patricka McDonnella, 4 października 2012, <http://www.slideshare.net/mcdonnp/continuously-deploying-culture-scaling-culture-at-etsy-14588485>.
- 226 McDonnell wyjaśniał dalej... Tamże.
- 226 Do roku 2011 firma Etsy... John Allspaw, osobista rozmowa z Gene'em Kimem, 2014.
- 227 Jak zażartował Ian Malpass... Ian Malpass, *Measure Anything, Measure Everything*, CodeAsCraft.com, 15 lutego 2011, <http://codeascraft.com/2011/02/15/measure-anything-measure-everything/>.

- 227 Z raportu *State of DevOps*... Kersten, IT Revolution i PwC, 2015 *State of DevOps Report*.
- 227 Dwie najważniejsze praktyki... 2014 *State Of DevOps Findings!* Velocity Conference, Slideshare.net, opublikowane przez Gene'a Kima, 30 czerwca 2014, <http://www.slideshare.net/realgenekim/2014-state-of-devops-findings-velocity-conference>.
- 228 James Turnbull w książce... James Turnbull, *The Art of Monitoring* (Seattle, WA, Amazon Digital Services, 2016), wydanie Kindle, „Wprowadzenie”.
- 229 ta funkcja umożliwia... *Monitorama — Please, no more Minutes, Milliseconds, Monoliths or Monitoring Tools*, Slideshare.net, opublikowane przez Adriana Cockcrofta, 5 maja 2014, <http://www.slideshare.net/adriancockcroft/monitorama-please-no-more>.
- 231 Scott Prugh, główny architekt... Prugh, DOES14: *Scott Prugh, CSG — DevOps and Lean in Legacy Environments*.
- 232 Do obsługi wymienionych... Brice Figureau, *The 10 Commandments of Logging*, blog Mastersena, 13 stycznia 2013, <http://www.masterzen.fr/2013/01/13/the-10-commandments-of-logging/>.
- 232 Wybór odpowiedniego poziomu... Dan North, osobista korespondencja z Gene'em Kimem, 2016.
- 232 Aby mieć pewność... Anton Chuvakin, *LogLogic/Chuvakin Log Checklist*, opublikowane za zgodą, 2008, <http://juliusdavies.ca/logging/llclc.html>.
- 233 W 2004 roku Gene Kim... Kim, Behr i Spafford, *The Visible Ops Handbook*, „Wprowadzenie”.
- 234 Taka filozofia... Dan North, *Ops and Operability*, SpeakerDeck.com, 25 lutego 2016, <https://speakerdeck.com/tastapod/ops-and-operability>.
- 234 Jak opisywał John Allspaw... John Allspaw, osobista korespondencja z Gene'em Kimem, 2011.
- 234 Często jest stosowana... *Information Radiators*, AgileAlliance.com, dostęp 31 maja 2016, <https://www.agilealliance.org/glossary/incremental-radiators/>.
- 237 Chociaż może istnieć pewien opór... Ernest Mueller, osobista korespondencja z Gene'em Kimem, 2014.
- 237 Prachi Gupta, dyrektor inżynierii... Prachi Gupta, *Visualizing LinkedIn's Site Performance*, Blog LinkedIn Engineering, 13 czerwca 2011, <https://engineering.linkedin.com/25/visualizing-linkedin-s-site-performance>.
- 238 W ten sposób rozpoczął się... Eric Wong, *Eric the Intern: the Origin of InGraphs*, LinkedIn, 30 czerwca 2011, <http://engineering.linkedin.com/32/eric-intern-origin-ingraphs>.
- 238 Wong pisał: „Żeby... Tamże.
- 238 W tamtym czasie... Tamże.
- 238 Pisząc o efektach... Gupta, *Visualizing LinkedIn's Site Performance*.

- 240 *Ed Blankenship, starszy... Ed Blankenship, osobista korespondencja z Gene'em Kimem*, 2016.
- 242 *Jednak coraz częściej te... Mike Burrows, The Chubby lock service for loosely-coupled distributed systems*, OSDI'06: Seventh Symposium on Operating System Design and Implementation, listopad 2006, <http://static.googleusercontent.com/media/research.google.com/en//archive/chubby-osdi06.pdf>.
- 243 *Na szczególną uwagę zasługuje narzędzie Consul... Jeff Lindsay, Consul Service Discovery with Docker*, Progrium.com, 20 sierpnia 2014, <http://progrium.com/blog/2014/08/20/consul-service-discovery-with-docker>.
- 243 *Jak zaobserwował Jody Mulkey... Jody Mulkey, DOES15 — Jody Mulkey — DevOps in the Enterprise: A Transformation Journey*, nagranie YouTube, 28:22, opublikowane przez DevOps Enterprise Summit, 5 listopada 2015, <https://www.youtube.com/watch?v=USYrDaPEFtM>.

ROZDZIAŁ 15.

- 245 *W 2015 roku firma Netflix... List do akcjonariuszy firmy Netflix*, 19 stycznia 2016, http://files.shareholder.com/downloads/NFLX/2432188684/x0x870685/C6213FF9-5498-4084-A0FF-74363CEE35A1/Q4_15_Letter_to_Shareholders_-_COMBINED.pdf.
- 245 *Roy Rapoport opisał... Roy Rapoport, osobista korespondencja z Gene'em Kimem*, 2014.
- 246 *Jedną z technik statystycznych... Victoria Hodge i Jim Austin, A Survey of Outlier Detection Methodologies*, „Artificial Intelligence Review 22”, nr 2 (październik 2004): 85 – 126, http://www.geo.upm.es/postgrado/CarlosLopez/papers/Hodge+Austin_OutlierDetection_AIRE381.pdf.
- 246 *Rapoport wyjaśnił, że... Roy Rapoport, osobista korespondencja z Gene'em Kimem*, 2014.
- 246 *Rapoport kontynuował: „Możemy... Tamże.*
- 246 *Rapoport stwierdził, że... Tamże.*
- 247 *Jak zaobserwował John Vincent... Toufic Boubez, Simple math for anomaly detection toufic boubez — metafor software — monitorama pdx 2014-05-05*, Slideshare.net, opublikowane przez tboubez, 6 maja 2014, <http://www.slideshare.net/tboubez/simple-math-for-anomaly-detection-toufic-boubez-metafor-software-monitorama-pdx-20140505>.
- 248 *Tom Limoncelli, współautor... Tom Limoncelli, Stop monitoring whether or not your service is up!*, EverythingSysAdmin.com, 27 listopada 2013, <http://everythingsysadmin.com/2013/11/stop-monitoring-if-service-is-up.html>.
- 249 *Jak zauważył dr Toufic Boubez... Toufic Boubez, Simple math for anomaly detection toufic boubez — metafor software — monitorama pdx 2014-05-05*, Slideshare.net,

- opublikowane przez tboubez, 6 maja 2014, <http://www.slideshare.net/tboubez/simple-math-for-anomaly-detection-toufic-boubez-metafor-software-monitorama-pdx-20140505>.
- 250 250 *Jak wyjaśnia dr Nicole Forsgren...* Dr Nicole Forsgren, osobista korespondencja z Gene'em Kimem, 2015.
- 251 251 *Zasada działania narzędzia Scryer...* Daniel Jacobson, Danny Yuan i Neeraj Joshi, *Scryer: Netflix's Predictive Auto Scaling Engine — „The Netflix Tech Blog”*, 5 listopada 2013, <http://techblog.netflix.com/2013/11/scryer-netflixs-predictive-auto-scaling.html>.
- 253 253 *Techniki te są...* Varun Chandola, Arindam Banerjee i Vipin Kumar, *Anomaly detection: A survey*, „ACM Computing Surveys 41”, nr 3 (lipiec 2009), artykuł nr 15, <http://doi.acm.org/10.1145/1541880.1541882>.
- 253 253 *Tarun Reddy, wiceprezes...* Tarun Reddy, osobista rozmowa z Gene'em Kimem, centrala firmy Rally, Boulder, CO, 2014.
- 255 255 *W 2014 roku dr Toufic Boubez...* Test Kołmogorowa-Smirnowa, Wikipedia, ostatnio zmodyfikowano 11 marca 2017, https://pl.wikipedia.org/wiki/Test_Ko%C5%82moga-Smirnowa
- 256 256 *Mówiąc „Kołmogorow-Smirnow”... Simple math for anomaly detection toufic boubez — metafor software — monitorama pdx 2014-05-05*, Slideshare.net, opublikowane przez tboubez, 6 maja 2014, <http://www.slideshare.net/tboubez/simple-math-for-anomaly-detection-toufic-boubez-metafor-software-monitorama-pdx-20140505>.

ROZDZIAŁ 16.

- 259 259 *W 2006 roku, Nick...* Mark Walsh, *Ad Firms Right Media, AdInterax Sell To Yahoo, „MediaPost”*, 18 października 2006, <http://www.mediapost.com/publications/article/49779/ad-firms-right-media-adinterax-sell-to-yahoo.html?edition=>.
- 259 259 *Galbreath opisywał...* Nick Galbreath, osobista rozmowa z Gene'em Kimem, 2013.
- 260 260 *Galbreath zaobserwował jednak...* Nick Galbreath, *Continuous Deployment — The New #1 Security Feature, from BSidesLA 2012*, Slideshare.net, opublikowane przez Nicka Galbreatha, 16 sierpnia 2012, <http://www.slideshare.net/nickgsuperstar/continuous-deployment-the-new-1-security-feature>.
- 260 260 *Galbreath, który obserwował...* Tamże.
- 261 261 *Galbreath zauważał, że...* Tamże.
- 263 263 *Jak w 2011 roku zaobserwował Patrick Lightbody...* Volocity 2011: Patrick Lightbody, „From Inception to Acquisition”, nagranie YouTube, 15:28, opublikowane przez O'Reilly, 17 czerwca 2011, <https://www.youtube.com/watch?v=ShmPod8JecQ>.
- 264 264 *Jak zaobserwował podczas prezentacji...* Arup Chakrabarti, *Common Ops Mistakes*, prezentacja w Heavy Bit Industries, 3 czerwca 2014, <http://www.heavybit.com/library/video/common-ops-mistakes/>.

- 265 *Później Jeff Sussna... From Design Thinking to DevOps and Back Again: Unifying Design & Operations*, nagranie w serwisie Vimeo 21:19, opublikowane przez Williama Evansa, 5 czerwca 2015, <https://vimeo.com/129939230>.
- 266 *Jak kiedyś powiedział anonimowy...* osobista rozmowa anonimowego inżyniera z Gene'em Kimem, 2005.
- 267 *Wytyczne i wymagania uruchamiania...* Tom Limoncelli, *SRE@Google: Thousands Of DevOps Since 2004*, nagranie YouTube USENIX Association Talk, NYC, opublikowane przez USENIX, 45:57, 12 stycznia 2012, <http://www.youtube.com/watch?v=iIuTnhdTzK0>.
- 269 *Jak powiedział Treynor Sloss...* Ben Treynor, *Keys to SRE* (prezentacja Usenix SREcon14, Santa Clara, CA, 30 maja 2014), <https://www.usenix.org/conference/srecon14/technical-sessions/presentation/keys-sre>.
- 269 *Treynor Sloss opierał się...* Tamże.
- 270 *Nawet wtedy, gdy nowe...* Limoncelli, „SRE@Google”.
- 270 *Tom Limoncelli podczas...* Tamże.
- 271 *Limoncelli zauważył: „W...* Tamże.
- 271 *Ponadto, jak zaobserwował Limoncelli...* Tom Limoncelli, osobista korespondencja z Gene'em Kimem 2016.
- 271 *Limoncelli wyjaśnił: „Pomoc...* Tamże, 2015.

ROZDZIAŁ 17.

- 273 *Ogólnie rzecz biorąc, jak...* Humble, O'Reilly i Molesky, *Lean Enterprise*, część II.
- 274 *W 2012 r. firma osiągnęła...* Intuit, Inc., *2012 Annual Report: Form 10-K*, 31 lipca 2012, http://s1.q4cdn.com/018592547/files/doc_financials/2012/INTU_2012_7_31_10K_r230_at_09_13_12_FINAL_and_Camera_Ready.pdf.
- 274 *Cook wyjaśniał, że...* Scott Cook, *Leadership in an Agile Age: wywiad ze Scottem Cookiem*, Intuit.com, 20 kwietnia 2011, <https://web.archive.org/web/201602052050418/http://network.intuit.com/2011/04/20/leadership-in-the-agile-age/>.
- 274 *Kontynuował: „Dzięki...* Tamże.
- 275 *W poprzednich epokach...* *Marketing bezpośredni*, Wikipedia, strona ostatnio zmodyfikowana 12 września 2014, https://pl.wikipedia.org/wiki/Marketing_bezp%C5%99Bredni.
- 275 *Co ciekawe, technikę tę...* Freakonomics, *Fighting Poverty With Actual Evidence*, pełna transkrypcja, Freakonomics.com, 27 listopada 2013, <http://freakonomics.com/2013/11/27/fighting-poverty-with-actual-evidence-full-transcript/>.
- 276 *Ronny Kohavi, posiadacz tytułu...* Ron Kohavi, Thomas Crook i Roger Longbotham, *Online Experimentation at Microsoft* (referat zaprezentowany na XV Międzynarodowej Konferencji ACM SIGKDD XV na temat wykrywania

wiedzy i ekstrakcji danych, Paryż, Francja, 2009), http://www.exp-platform.com/documents/exp_dmcasesstudies.pdf.

- 276 *Kohavi zauważał, że... Tamże.*
- 277 *Jak zażartował Jez Humble... Jez Humble, osobista korespondencja z Gene'em Kimem, 2015.*
- 277 *W 2014 r. w wywiadzie... Wang, Kendrick, *Etsy's Culture Of Continuous Experimentation and A/B Testing Spurs Mobile Innovation*, Apptimize.com, 30 stycznia 2014, <http://apptimize.com/blog/2014/01/etsy-continuous-innovation-ab-testing/>.*
- 278 *Barry O'Reilly, współautor... Barry O'Reilly, *How to Implement Hypothesis-Driven Development*, BarryOReilly.com, 21 października 2013, <http://barryoreilly.com/2013/10/21/how-to-implement-hypothesis-driven-development/>.*
- 278 *W 2009 roku Jim... Gene Kim, *Organizational Learning and Competitiveness: Revisiting the „Allspaw/Hammond 10 Deploys Per Day at Flickr” Story*, ITRevolution.com, 2015, <http://itrevolution.com/organizational-learning-and-competitiveness-a-different-view-of-the-allspawhammond-10-deploys-per-day-at-flickr-story/>.*
- 279 *Stoneham zauważał, że... Tamże.*
- 279 *Dalej pisał: „Te... Tamże.*
- 279 *Dzięki zdumiewającym wynikom... Tamże.*
- 280 *Stoneham zakończył: „Właśnie... Tamże.*

ROZDZIAŁ 18.

- 282 *Po wysłaniu żądania ściągnięcia... Scott Chacon, *Github Flow*, ScottChacon.com, 31 sierpnia 2011, <http://scottchacon.com/2011/08/31/github-flow.html>.*
- 283 *Na przykład w 2012 roku... Jake Douglas, *Deploying at Github*, GitHub.com, 29 sierpnia 2012, <https://github.com/blog/1241-deploying-at-github>.*
- 283 *Błąd wdrażania i brak możliwości... John Allspaw, *Counterfactual Thinking, Rules, and the Knight Capital Accident*, KitchenSoap.com, 29 października 2013, <http://www.kitchensoap.com/2013/10/29/counterfactuals-knight-capital/>.*
- 285 *Zgodnie z jednym z podstawowych przekonań... Bradley Staats i David M. Upton, *Lean Knowledge Work*, Harvard Business Review, październik 2011, <https://hbr.org/2011/10/lean-knowledge-work>.*
- 285 *W raporcie Puppet Labs... Velasquez, Kim, Kersten i Humble, 2014 State of DevOps Report.*
- 287 *Jak zauważył Randy Shoup... Randy Shoup, osobista rozmowa z Gene'em Kimem, 2015.*
- 287 *Giray Özil napisał na Twitterze... Giray Özil, post na Twitterze, 27 lutego 2013, 10:42:00, <https://twitter.com/girayozil/status/306836785739210752>.*

- 289 *Jak wspomniano wcześniej... Eran Messeri, What Goes Wrong When Thousands of Engineers Share the Same Continuous Build?, (2013), <http://scribes.tweetscriber.com/realgenekim/206>.*
- 289 *W 2010 roku było... John Thomas i Ashish Kumar, Welcome to the Google Engineering Tools Blog, blog Google Engineering Tools, opublikowano 3 maja 2011, <http://google-engtools.blogspot.com/2011/05/welcome-to-google-engineering-tools.html>.*
- 289 *Wymagało to znaczącej... Ashish Kumar, Development at the Speed and Scale of Google (prezentacja na konferencji QCon, San Francisco, CA, 2010), https://qconsf.com/sf2010/dl/qcon-sanfran-2010/slides/AshishKumar_DevelopingProductsattheSpeedandScaleofGoogle.pdf.*
- 289 *Jak powiedział: „Pracowałem... Randy Shoup, osobista korespondencja z Gene'em Kimem, 2014.*
- 291 *Jeff Atwood, jeden z... Jeff Atwood, Pair Programming vs. Code Reviews, CodingHorror.com, 18 listopada 2013, <http://blog.codinghorror.com/pair-programming-vs-code-reviews/>.*
- 291 *Dalej pisał: „Większość... Tamże.*
- 292 *Dr Laurie Williams w 2001 roku... Pair Programming, strona Wiki ALICE ostatnio zmodyfikowana 4 kwietnia 2014, http://euler.math.uga.edu/wiki/index.php?title=Pair_programming.*
- 292 *Twierdziła, że... Elisabeth Hendrickson, DOES15 — Elisabeth Hendrickson — Its All About Feedback, nagranie YouTube, 34:47, opublikowane przez DevOps Enterprise Summit, 5 listopada 2015, <https://www.youtube.com/watch?v=r2BFTXBundQ>.*
- 292 *W swojej prezentacji... Tamże.*
- 293 *W przypadku przeglądów kodu... Tamże.*
- 293 *Co gorsza, utalentowani programiści... Tamże.*
- 293 *Hendrickson ubolewał nad tym, że... Tamże.*
- 294 *W ten sposób opisano rzeczywiste... Ryan Tomayko i Shawn Davenport, osobista rozmowa z Gene'em Kimem, 2013.*
- 294 *Miało wiele stron... Tamże.*
- 294 *Czytając to żądanie... Tamże.*
- 295 *Jak zaobserwował Adrian Cockcroft... Wywiad z Adrianem Cockcroftem przeprowadzony przez Michaela Duncy'ego i Rossa Clantona, Adrian Cockcroft of Battery Ventures — the Goat Farm — Episode 8, „The Goat Farm”, podcast audio, 31 lipca 2015, <http://goatcan.do/2015/07/31/adrian-cockcroft-of-battery-ventures-the-goat-farm-episode-8/>.*
- 295 *W podobnym tonie wypowiedział się... Tapabrata Pal, Banking on Innovation & DevOps, nagranie YouTube, 32:57, opublikowane przez DevOps Enterprise Summit, 4 stycznia 2016, <https://www.youtube.com/watch?v=bbWFCKGhxOs>.*

- 295 Jason Cox, główny... Jason Cox, *Disney DevOps*.
- 295 W firmie Target w... Ross Clanton i Heather Mickman, *DOES14 — Ross Clanton i Heather Mickman — DevOps at Target*, nagranie YouTube, 29:20, opublikowane przez DevOps Enterprise Summit 2014, 29 października 2014, <https://www.youtube.com/watch?v=exrjV9V9vhY>.
- 296 „W czasie realizacji procesu... Tamże.
- 296 Dodała: „Chciałabym... Tamże.
- 296 Weźmy pod uwagę historię... John Allspaw i Jez Humble, osobista korespondencja z Gene'em Kimem, 2014.

ROZDZIAŁ 19.

- 303 W efekcie powstaje coś... Spear, *The High-Velocity Edge*, rozdział 1.
- 303 „Dla takich organizacji... Tamże, rozdział 10.
- 303 Uderzający przykład... Julianne Pepitone, *Amazon EC2 Outage Downs Reddit, Quora, „CNN Money”*, 22 kwietnia 2011, http://money.cnn.com/2011/04/21/technology/amazon_server_outage.
- 304 W styczniu 2013... Timothy Prickett Morgan, *A Rare Peek Into The Massive Scale of AWS, „Enterprise Tech”*, 14 listopada 2014, <http://www.enterprisotech.com/2014/11/14/rare-peek-massive-scale-aws/>.
- 304 Jednak w poście na blogu... Adrian Cockcroft, Cory Hicks i Greg Orzell, *Lessons Netflix Learned from the AWS Outage, „The Netflix Tech Blog”*, 29 kwietnia 2011, <http://techblog.netflix.com/2011/04/lessonsnetflix-learned-from-aws-outage.html>.
- 304 Zrobiono to ze względu... Tamże.
- 305 Dr Sidney Dekker... Sidney Dekker, *Just Culture: Balancing Safety and Accountability* (Lund University, Szwecja: Ashgate Publishing Company, 2007), s. 152.
- 305 Twierdzi, że... *DevOpsDays Brisbane 2014 — Sidney Decker Who's to Blame?*, nagranie wideo w serwisie Vimeo, 1:07:38, opublikowane przez info@devopsdays.org, 2014, <https://vimeo.com/102167635>.
- 306 Jak powiedział John Allspaw... Wywiad Jenna Webba z Johnem Allspawem, *Post-Mortems, Sans Finger-Pointing, „The O'Reilly Radar Postcast”*, podcast audio, 21 sierpnia 2014, <http://radar.oreilly.com/2014/08/postmortems-sans-finger-pointing-the-oreilly-radar-podcast.html>.
- 306 Analizy post-mortem bez szukania winnych... John Allspaw, *Blameless PostMortems and a Just Culture*, CodeAsCraft.com, 22 maja 2012, <http://codeascraft.com/2012/05/22/blameless-postmortems/>.
- 308 Ian Malpass, inżynier... Ian Malpass, *DevOpsDays Minneapolis 2014 — Ian Malpass, Fallible humans*, nagranie YouTube, 35:48, opublikowane przez DevOps Minneapolis, 20 lipca 2014, <https://www.youtube.com/watch?v=5NY-SrQFrBU>.

- 308 *Dan Milstein, jeden...* Dan Milstein, *Post-Mortems at HubSpot: What I Learned from 250 Whys*, HubSpot, 1 czerwca 2011, <http://product.hubspot.com/blog/bid/64771/Post-Mortems-at-HubSpot-What-I-Learned-From-250-Whys>.
- 309 *Randy Shoup, były...* Randy Shoup, osobista korespondencja z Gene'em Kimem, 2014.
- 309 *Można także rozszerzyć... Post-Mortem for February 24, 2010 Outage*, witryna WWW Google App Engine, 4 marca 2010, <https://groups.google.com/forum/#!topic/google-appengine/p2QKJ0OSLc8>; *Summary of the Amazon DynamoDB Service Disruption and Related Impacts in the US-East Region*, witryna internetowa Amazon Web Services, dostęp 28 maja 2016, <https://aws.amazon.com/message/5467D2/>.
- 309 *Dążenie do prowadzenia...* Bethany Macri, *Morgue: Helping Better Understand Events by Building a Post Mortem Tool* — Bethany Macri, nagranie w serwisie Vimeo, 33:34, opublikowane przez info@devopsdays.org, 18 października 2013, <http://vimeo.com/77206751>.
- 310 *Na przykład zgodnie z opisem...* Spear, *The High-Velocity Edge*, rozdział 4.
- 311
- 310 *Dr Amy C. Edmondson...* Amy C. Edmondson, *Strategies for Learning from Failure*, kwiecień 2011, <https://hbr.org/2011/04/strategies-for-learning-from-failure>.
- 311 *Dr Spear, podsumowując...* Tamże.
- 311 *Teraz wiemy, że...* Tamże, rozdział 3.
- 311 *Jednak przed ponownym...* Michael Roberto, Richard M.J. Bohmer i Amy C. Edmondson, *Facing Ambiguous Threats*, „Harvard Business Review”, listopad 2006, <https://hbr.org/2006/11/facing-ambiguous-threats/ar/1>.
- 311 *Opisywali oni, że...* Tamże.
- 311 *Zaobserwowali oni: „Firmy...* Tamże.
- 311 *Autorzy wyciągnęli wniosek...* Tamże.
- 312 *Roy Rapoport z Netflix...* Roy Rapoport, osobista korespondencja z Gene'em Kimem, 2012.
- 312 *Dalej pisał: „Rozmawiałem...* Tamże.
- 312 *Na koniec stwierdził...* Tamże.
- 313 *Jak skomentował Michael Nygard...* Michael T. Nygard, *Release It!: Design and Deploy Production-Ready Software* (Pragmatic Bookshelf: Raleigh, NC, 2007), wydanie Kindle, część I.
- 313 *Jeszcze bardziej interesujący...* Jeff Barr, *EC2 Maintenance Update*, Blog AWS, 25 września 2014, <https://aws.amazon.com/blogs/aws/ec2-maintenance-update/>.
- 313 *Jak wspomina Christos Kalantzis...* Bruce Wong i Christos Kalantzis, *A State of Xen — Chaos Monkey & Cassandra*, blog techniczny Netflix, 2 października 2014, <http://techblog.netflix.com/2014/10/a-state-of-xen-chaos-monkey-cassandra.html>.

- 313 Ale — kontynuował Kalantzis... Tamże.
- 313 Jak napisali Kalantzis i Bruce Wong... Tamże.
- 314 Jeszcze bardziej zaskakujące... Roy Rapoport, osobista korespondencja z Gene'em Kimem, 2015.
- 314 Specyficzne wzorce architektoniczne... Adrian Cockcroft, osobista korespondencja z Gene'em Kimem, 2012.
- 314 W tym podroziale... Jesse Robbins, *GameDay: Creating Resiliency Through Destruction* — LISA11, Slideshare.net, opublikowane przez Jesse'ego Robbinsa, 7 grudnia 2011, <http://www.slideshare.net/jesserobbins/ameday-creating-resiliency-through-destruction>.
- 314 Robbins zdefiniował inżynierię... Tamże.
- 314 Robbins zauważył, że... Jesse Robbins, Kripa Krishnan, John Allspaw i Tom Moncelli, *Resilience Engineering: Learning to Embrace Failure*, „amcqueue” 10, nr 9 (13 września 2012): <https://queue.acm.org/detail.cfm?id=2371297>.
- 315 Jak żartuje Robbins... Tamże.
- 315 Jak opisuje Robbins... Tamże.
- 315 Robbins wyjaśnia: „Możemy... Tamże.
- 315 W tym czasie... Kripa Krishnan: „Learning Continuously From Failures” at Google, nagranie YouTube, 21:35, opublikowane przez Flowcon, 11 listopada 2014, <https://www.youtube.com/watch?v=KqqS3wgQum0>.
- 316 Jak napisał Krishnan... Kripa Krishnan, „Weathering the Unexpected”, *Communications of the ACM* 55, nr 11 (listopad 2012): 48 – 52, <http://cacm.acm.org/magazines/2012/11/156583-weathering-the-unexpected/abstract>.
- 316 Podczas awarii zdobyto... Tamże.
- 316 Jak mawia Peter Senge... Powszechnie przypisywane Peterowi Senge'emu.

ROZDZIAŁ 20.

- 318 Jak opisuje Jesse Newland... Jesse Newland, *ChatOps w GitHub*, SpeakerDeck.com, 7 lutego 2013, <https://speakerdeck.com/jnewland/chatops-at-github>.
- 318 Jak przypomina sobie Mark Imbriaco... Mark Imbriaco, osobista korespondencja z Gene'em Kimem, 2015.
- 318 W narzędziu Hubot uwzględniono... Newland, „ChatOps o GitHub”.
- 319 Hubot często wykonywał zadania... Tamże.
- 319 Newland zauważył, że... Tamże.
- 320 Zamiast umieszczać... Leon Osterweil, *Software processes are software too*, artykuł zaprezentowany na konferencji International Conference on Software Engineering, Monterey, CA, 1987, <http://www.cs.unibo.it/cianca/wwwpages/ids/lettura/Osterweil.pdf>.

- 320 *Justin Arbuckle, były... Justin Arbuckle, What Is ArchOps: Executive Chef Round-table* (2013).
- 320 *System ten „pozwolił... Tamże.*
- 320 *Arbuckle zakończył... Tamże.*
- 320 *W 2015 r. firma Google... Google Is 2 Billion Lines of Code — and It's All in One Place*, Wired, 16 września 2015, <http://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/>.
- 321 *Projekty Chrome i... Tamże.*
- 321 *Rachel Potvin, menedżer... Tamże.*
- 321 *Ponadto Eran Messeri... Eran Messeri, What Goes Wrong When Thousands of Engineers Share the Same Continuous Build? (2013), <http://scribes.tweetscriber.com/realgenekim/206>.*
- 321 *Jak opisuje Randy Shoup... Randy Shoup, osobista korespondencja z Gene'em Kimem, 2014.*
- 321 *Tom Limoncelli jest współautorem... Tom Limoncelli, Yes, you can really work from HEAD, EverythingSysAdmin.com, 15 marca 2014, <http://everythingsysadmin.com/2014/03/yes-you-really-can-work-from-head.html>.*
- 326 *Jak opisuje Tom Limoncelli... Tom Limoncelli, Python is better than Perl6, EverythingSysAdmin.com, 10 stycznia 2011, <http://everythingsysadmin.com/2011/01/python-is-better-than-perl6.html>.*
- 326 *W Google używano języka C++... Which programming languages does Google use internally?, Quora.com forum, dostęp 29 maja 2016, <https://www.quora.com/Which-programming-languages-does-Google-use-internally>; When will Google permit languages other than Python, C++, Java and Go to be used for internal projects?, forum Quora.com, dostęp 29 maja 2016, <https://www.quora.com/When-will-Google-permit-languages-other-than-Python-C-Java-and-Go-to-be-used-for-internal-projects/answer/Neil-Kandalgaonkar>.*
- 326 *Olivier Jacques i Rafael Garcia... Ralph Loura, Olivier Jacques i Rafael Garcia, DOES15 — Ralph Loura, Olivier Jacques, & Rafael Garcia — Breaking Traditional IT Paradigms to..., nagranie YouTube, 31:07, opublikowane przez DevOps Enterprise Summit, 16 listopada 2015, https://www.youtube.com/watch?v=q9nNqqie_sM.*
- 326 *W wielu organizacjach... Michael Rembetsky i Patrick McDonnell, Continuously Deploying Culture: Scaling Culture at Etsy — Velocity Europe 2012, Slideshare.net, opublikowane przez Patricka McDonnella, 4 października 2012, <http://www.slideshare.net/mcdonnps/continuously-deploying-culture-scaling-culture-at-etsy-14588485>.*
- 327 *W tym czasie w Etsy... Tamże.*
- 327 *Dan McKinley, programista... Dan McKinley, Why MongoDB Never Worked Out at Etsy, McFunley.com, 26 grudnia 2012, <http://mcfunley.com/why-mongodb-never-worked-out-at-etsy>.*

ROZDZIAŁ 21.

- 329 Jedna z praktyk... Kaizen, Wikipedia, strona ostatnio zmodyfikowana 12 maja 2016, <https://en.wikipedia.org/wiki/Kaizen>.
- 329 Dr Spear wyjaśnia... Spear, *The High-Velocity Edge*, rozdział 8.
- 329 Spear zaobserwował, że... Tamże.
- 330 Clanton opisuje: „Mamy... Mickman i Clanton, *(Re)building an Engineering Culture*.
- 330 Ravi Pandey, menedżer... Ravi Pandey, osobista korespondencja z Gene'em Kimem, 2015.
- 330 Clanton dodaje... Mickman i Clanton, *(Re)building an Engineering Culture*.
- 331 Oprócz terminów pochodzących z Lean... Queue Inversion Week, „Righteous IT”, 12 lutego 2009, <https://righteousit.wordpress.com/2009/02/12/queue-inversion-week/>.
- 332 Jak zauważył dr Spear... Spear, *The High-Velocity Edge*, rozdział 3.
- 332 W rozmowie z Jessicą... Jessica Stillman, *Hack Days: Not Just for Facebookers*, Inc., 3 lutego 2012, <http://www.inc.com/jessica-stillman/hack-days-not-just-for-facebookers.html>.
- 332 W 2008 r. Facebook... AP, *Number of active users at Facebook over the years*, „Yahoo! News”, 1 maja 2013, <https://www.yahoo.com/news/number-active-users-facebook-over-230449748.html?ref=gs>.
- 332 W czasie trwania dnia hack day... Haiping Zhao, *HipHop for PHP: Move Fast*, post na stronie Haipinga Zhao na Facebooku, 2 lutego 2010, <https://www.facebook.com/notes/facebook-engineering/hiphop-for-phpmove-fast/280583813919>.
- 332 W wywiadzie dla Cade'a... Cade Metz, *How Three Guys Rebuilt the Foundation of Facebook*, „Wired”, 10 czerwca 2013, <http://www.wired.com/wiredenterprise/2013/06/facebook-hhvm-saga/all/>.
- 333 Steve Farley, wiceprezes... Steve Farley, osobista korespondencja z Gene'em Kimem, 5 lutego 2016.
- 333 Karthik Gaekwad, który... Agile 2013 Talk: *How DevOps Change Everything*, „Slideshare.net”, opublikowano przez Karthika Gaekwada, 7 sierpnia 2013, <http://www.slideshare.net/karthequian/howdevopschangeseverythingagile2013-karthikgaekwad/>.
- 334 Jak zażartował Glenn O'Donnell... Glenn O'Donnell, *DOES14 — Glenn O'Donnell — Forrester — Modern Services Demand a DevOps Culture Beyond Apps*, nagranie YouTube, 12:20, opublikowane przez DevOps Enterprise Summit 2014, 5 listopada 2014, https://www.youtube.com/watch?v=pvPWKuO4_48.
- 335 W roku 2014 r. firma... Nationwide, 2014 Annual Report, <https://www.nationwide.com/about-us/nationwide-annual-report-2014.jsp>.
- 335 Steve Farley, wiceprezes... Steve Farley, osobista korespondencja z Gene'em Kimem, 2016.

- 335 *Capital One, jeden z... DOES15 — Tapabrata Pal — Banking on Innovation & DevOps*, nagranie YouTube, 32:57, opublikowane przez DevOps Enterprise Summit, 4 stycznia 2016, <https://www.youtube.com/watch?v=bbWFCKGhxOs>.
- 335 *Dr Tapabrata Pal...* Tapabrata Pal, osobista korespondencja z Gene'em Kimem, 2015.
- 335 *Firma Target jest szóstym... „Przedsiębiorstwo Fact Sheet”*, witryna internetowa firmy Target, dostęp 9 czerwca 2016, <https://corporate.target.com/press/corporate>.
- 335 *Nawiasem mówiąc, pierwsze... Evelijn Van Leeuwen i Kris Buytaert, DOES15 — Evelijn Van Leeuwen i Kris Buytaert — Turning Around the Containership*, nagranie YouTube, 30:28, opublikowane przez DevOps Enterprise Summit, 21 grudnia 2015, <https://www.youtube.com/watch?v=0GId4AMKvPc>.
- 336 *Clanton opisuje: „2015... Mickman i Clanton, (Re)building an Engineering Culture.*
- 336 *W Capital One... DOES15 — Tapabrata Pal — Banking on Innovation & DevOps*, nagranie w serwisie YouTube, 32:57, opublikowane przez DevOps Enterprise Summit, 4 stycznia 2016, <https://www.youtube.com/watch?v=bbWFCKGhxOs>.
- 336 *Bland powiedział, że... Bland, DOES15 — Mike Bland — Pain Is Over, If You Want It.*
- 336 *Mimo że grouplet... Tamże.*
- 336 *Wykorzystano kilka mechanizmów... Tamże.*
- 336 *Bland powiedział... Tamże.*
- 337 *Bland kontynuował: „Jeden... Tamże.*
- 337 *Zgodnie z tym, co opisuje Bland... Tamże.*
- 337 *Dalej Bland mówił... Tamże.*
- 337 *Inną ważną konstrukcją... Tamże.*
- 337 *Bland opisuje wydarzenia Fixit... Mike Bland, Fixits, or I Am the Walrus, „Mike-Bland.com”, 4 października 2011, <https://mike-bland.com/2011/10/04/fixits.html>.*
- 338 *Z wydarzeń Fixit, zgodnie... Tamże.*

ROZDZIAŁ 22.

- 343 *Jedno z najczęstszych... James Wickett, Attacking Pipelines—Security meets Continuous Delivery*, „Slideshare.net”, opublikowane przez Jamesa Wicketta, 11 czerwca 2014, <https://www.slideshare.net/wickett/attacking-pipelinessecurity-meets-continuous-delivery>.
- 343 *James Wickett, jeden... Tamże.*
- 344 *Podobne pomysły zostały... Tapabrata Pal, DOES15 — Tapabrata Pal — Banking on Innovation & DevOps*, nagranie YouTube 32:57, opublikowane przez DevOps

Enterprise Summit, 4 stycznia 2016, <https://www.youtube.com/watch?v=→bbWFCKGhxOs>.

- 344 *Justin Arbuckle, były... Justin Arbuckle, osobista rozmowa z Gene'em Kimem, 2015.*
- 344 *Kontynuując tę myśl, stwierdził: „Dzięki... Tamże.*
- 345 *To pomogło... Snehal Antani, IBM Innovate DevOps Keynote, nagranie YouTube, 47:57, opublikowane przez IBM DevOps, 12 czerwca 2014, <https://www.youtube.com/watch?v=s0M1P05-6Io>.*
- 345 *Podczas prezentacji... Nick Galbreath, DevOpsSec: Applying DevOps Principles to Security, DevOpsDays Austin 2012, prezentacja Slideshare, opublikowana przez Nicka Galbreatha, 12 kwietnia 2012, <http://www.slideshare.net/nickgsuperstar/devopssec-apply-devops-principles-to-security>.*
- 345 *Ponadto stwierdził: „Zawsze... Tamże.*
- 350 *Ponadto należy zdefiniować... OWASP Cheat Sheet Series, OWASP.org, ostatnio zmodyfikowano 2 marca 2016, https://www.owasp.org/index.php/OWASP_Cheat_Sheet_Series.*
- 351 *Skala wzrostu liczby użytkowników... Justin Collins, Alex Smolen i Neil Matatall, Putting to your Robots to Work V1.1, prezentacja Slideshare.net, opublikowana przez Neila Matatalla, 24 kwietnia 2012, <https://www.slideshare.net/xplodersuv/sf-2013-robots>.*
- 351 *Na początku 2009 roku... What Happens to Companies That Get Hacked? FTC Cases, forum Giant Bomb, opublikowane przez „SuicidalSnowman”, lipiec 2012, <http://www.giantbomb.com/forums/off-topic-31/what-happens-to-companies-that-get-hacked-ftc-case-540466>.*
- 352 *We wcześniej wspomnianej prezentacji... Collins, Smolen i Matatall, Putting to your Robots to Work V1.1.*
- 353 *Pierwszy wielki przełom... Twitter Engineering, „Hack Week @ Twitter”, blog na Twitterze, 25 stycznia 2012, <https://blog.twitter.com/2012/hack-week-twitter>.*
- 353 *Josh Corman zauważył... Josh Corman i John Willis, Immutable Awesomeness — Josh Corman and John Willis at DevOps Enterprise Summit 2015, nagranie YouTube, 34:25, opublikowane przez Sonatype, 21 października 2015, <https://www.youtube.com/watch?v=-S8-lrm3iV4>.*
- 354 *W raporcie za rok 2014... Verizon, 2014 Data Breach Investigations Report, („Verizon Enterprise Solutions”, 2014), https://dti.delaware.gov/pdfs/rp_Verizon-DBIR-2014_en_xg.pdf.*
- 354 *W 2015 roku... 2015 State of the Software Supply Chain Report: Hidden Speed Bumps on the Way to „Continuous” („Fulton”, MD, Sonatype, Inc., 2015), http://cdn2.hubspot.net/hubfs/1958393/White_Papers/2015_State_of_the_Software_Supply_Chain_Report-.pdf?t=1466775053631.*

- 354 Ostatnią statystykę... Dan Geer i Joshua Corman, *Almost Too Big to Fail; login:: The Usenix Magazine*, 39, no. 4 (sierpień 2014), 66 – 68, https://www.usenix.org/system/files/login/articles/15_geer_0.pdf.
- 355 Agencje rządu federalnego USA... Wyatt Kash, *New details released on proposed 2016 IT spending*, FedScoop, 4 lutego 2015, <http://fedscoop.com/what-top-agencies-would-spend-on-it-projects-in-2016>.
- 356 Mike Bland wyjaśnia... Bland, *DOES15 — Mike Bland — Pain Is Over, If You Want It.*
- 356 Cloud.gov działa obecnie... Mossadeq Zia, Gabriel Ramírez, Noah Kunin, *Compliance Masonry: Building a risk management platform, brick by brick*, „18F”, 15 kwietnia 2016, <https://18f.gsa.gov/2016/04/15/compliance-masonry-building-a-risk-management-platform/>.
- 357 Marcus Sachs, jeden... Marcus Sachs, osobista korespondencja z Gene’em Kimem, 2010.
- 358 Oto przykłady... VPC Best Configuration Practices, „blog Flux7”, 23 stycznia 2014, <http://blog.flux7.com/blogs/aws/vpc-best-configuration-practices>.
- 359 W 2010 roku Nick... Nick Galbreath, *Fraud Engineering, from Merchant Risk Council Annual Meeting 2012*, prezentacja Slideshare.net, opublikowana przez Nicka Galbreatha, 3 maja 2012, <http://www.slideshare.net/nickgsuperstar/fraud-engineering>.
- 359 Szczególnie interesowaliśmy się... Nick Galbreath, *DevOpsSec: Applying DevOps Principles to Security, DevOpsDays Austin 2012*, prezentacja SlideShare.net, opublikowana przez Nicka Galbreatha, 12 kwietnia 2013, <http://www.slideshare.net/nickgsuperstar/devopssec-apply-devops-principles-to-security>.
- 359 Zawsze szukaliśmy... Tamże.
- 359 To był śmiesznie prosty test... Tamże.
- 359 Jak zaobserwował Galbreath... Tamże.
- 360 Galbreath zauważyl... Tamże.
- 360 Jak zaobserwował Jonathan Claudius... Jonathan Claudius, *Attacking Cloud Services with Source Code*, Speakerdeck.com, opublikowano przez Jonathana Claudiusa, 16 kwietnia 2013, <https://speakerdeck.com/claudijd/attacking-cloud-services-with-source-code>.

ROZDZIAŁ 23.

- 364 ITIL definiuje narzędzie... Axelos, *ITIL Service Transition (ITIL Lifecycle Suite)* (Belfast, Ireland: TSO, 2011), 48.
- 367 Firma Salesforce została założona... Reena Mathew i Dave Mangot, *DOES14 — Reena Mathew i Dave Mangot — Salesforce*, Slideshare.net, opublikowane przez

ITRevolution, 29 października 2014, <https://www.slideshare.net/ITRevolution/does14-reena-matthew-and-dave-mangot-salesforce>.

- 367 Do roku 2007... Dave Mangot i Karthik Rajan, *Agile.2013. effecting.a.dev ops.transformation.at.salesforce*, prezentacja Slideshare.net, opublikował Dave Mangot, 12 sierpnia 2013, <https://www.slideshare.net/dmangot/agile2013effectingadevopstransformationatsalesforce>.
- 367 Karthik Rajan, który wówczas... Tamże.
- 368 Dave Mangot i Reena Mathew... Mathew i Mangot, „DOES14 — Salesforce”.
- 368 Według Mangot i Matthew... Tamże.
- 368 Ponadto zauważono, że... Tamże.
- 369 Bill Massie jest... Bill Massie, osobista korespondencja z Gene'em Kimem, 2014.
- 370 Ponieważ według standardu... „Glossary”, witryna WWW PCI Security Standards Council, dostęp 30 maja 2016, https://www.pcisecuritystandard.org/pci_security/glossary.
- 370 Czy przed publikacją wyniki przeglądu kodu... PCI Security Standards Council, *Payment Card Industry (PCI) Data Security Stands: Requirements and Security Assessment Procedures, Version 3.1* (PCI Security Standards Council, 2015), punkt 6.3.2. <https://webcache.googleusercontent.com/search?q=cache:hpRe2COzzdAJ>; https://www.pcisecuritystandards.org/documents/PCI_DSS_v3-1_SAQ_D_Merchant_rev1-1.docx ↩+&cd=2&hl=en&ct=clnk&gl=us
- 370 W celu spełnienia tego wymagania... Bill Massie, osobista korespondencja z Gene'em Kimem, 2014.
- 371 Massie zauważył, że... Tamże.
- 371 Efekt jest taki, że... Tamże.
- 371 Jak zaobserwował Bill Shinn... Bill Shinn, *DOES15 — Bill Shinn — Prove it! The Last Mile for DevOps in Regulated Organizations*, prezentacja Slideshare.net, opublikowana przez ITRevolution, 20 listopada 2015, <http://www.slideshare.net/ITRevolution/does15-bill-shinn-prove-it-the-last-mile-for-devops-in-regulated-organizations>.
- 372 Pomoc klientom... Tamże.
- 372 Shinn zaobserwował: „Jeden... Tamże.
- 372 „Takie podejście sprawdzało się... Tamże.
- 372 Shinn wyjaśnia: „Wśród... Tamże.
- 372 Shinn stwierdza, że... Tamże.
- 373 Shinn kontynuuje: „Dzięki... Tamże.
- 373 To wymaga wyprowadzenia... Tamże.
- 373 Shinn kontynuuje: „Sposób... Tamże.
- 373 Shinn podaje przykład... Tamże.

- 373 Aby pomóc rozwiązać... James DeLuccia, Jeff Gallimore, Gene Kim i Byron Miller, *DevOps Audit Defense Toolkit* (Portland, OR, IT Revolution, 2015), <http://itrevolution.com/devops-and-auditors -the-devops-audit-defense-toolkit>.
- 374 Zaobserwowała ona, że... Mary Smith (pseudonim), osobista korespondencja z Gene'em Kimem, 2013.
- 374 Zauważała, że... Tamże, 2014.

PODSUMOWANIE

- 378 Jak mawiał Jesse Robbins... *Hacking Culture at VelocityConf*, prezentacja Slide-share.net, opublikowana przez Jesse'ego Robbinsa, 28 czerwca 2012, <http://www.slideshare.net/jesserobbins/hacking-culture-at-velocityconf>.

DODATEK

- 383 Ruch Lean rozpoczął się... Ries, *The Lean Startup*.
- 384 Kluczową zasadą było... Kent Beck et al., *Twelve Principles of Agile Software*, AgileManifesto.org, 2001, <http://agilemanifesto.org/principles.html>.
- 385 Opierając się na... Humble i Farley, *Continuous Delivery*.
- 385 Pomysł ten po raz pierwszy... Fitz, *Continuous Deployment at IMVU*.
- 385 Toyota Kata opisuje... Rother, Toyota Kata, „Wprowadzenie”.
- 385 Mike Rother doszedł do wniosku... Tamże.
- 385 W 2011 roku Eric... Ries, *The Lean Startup*.
- 388 W Projekcie Feniks... Kim, Behr i Spafford, *The Phoenix Project*, s. 365.
- 390 Mit 1. „Najważniejszą... Denis Besnard i Erik Hollnagel, *Some Myths about Industrial Safety* (Paryż, Centre De Recherche Sur Les Risques Et Les Crises Mines, 2012), s. 3, http://gswong.com/?wpfb_dl=31.
- 390 Mit 2. „Systemy... Tamże, s. 4.
- 390 Mit 3. „Bezpieczeństwo... Tamże, s. 6.
- 390 Mit 4. „Analiza... Tamże, s. 8.
- 390 Mit 5. „Badania... Tamże, s. 9.
- 390 Mit 6. „Bezpieczeństwo... Tamże, s. 11.
- 391 Jest raczej tak, że... John Shook, *Five Missing Pieces in Your Standardized Work (Part 3 of 3)*, Lean.org, 27 października 2009, <http://www.lean.org/shook/DisplayObject.cfm?o=1321>.
- 393 Czas do rozwiązywania... *Post Event Retrospective — Part 1*, Rally Blogs, dostęp 31 maja 2016, <https://www.rallydev.com/blog/engineering/post-event-retrospective-part-i>.

- 393 *Bethany Macri z firmy... Morgue: Helping Better Understand events by Building a Post Mortem Tool* — Bethany Macri, nagranie Vimeo 33:34, opublikowane przez info@devopsdays.org, 18 października 2013, <http://vimeo.com/77206751>.
- 394 *Dyskusje te... Cockcroft, Hicks i Orzell, „Lessons Netflix Learned”.*
- 394 *Od tego czasu usługa Chaos... Tamże.*
- 395 *Lenny Rachitsky pisał... Lenny Rachitsky, 7 Keys to a Successful Public Health Dashboard, „Transparent Uptime”, 1 grudnia 2008, <http://www.transparentuptime.com/2008/11/rules-for-successful-public-health.html>.*

SKOROWIDZ

A

analiza
dynamiczna, 349
post-mortem, 306, 309
statyczna, 349
analizowanie telemetrii, 245
antykruchosć, 75
archetypy architektoniczne, 212
architektura, 55, 107
ewolucyjna, 214
luźnych sprzężeń, 118
systemu, 267
ATDD, 165
ATO, authority to operate, 355
audytor, 371
automatyczne
skalowanie, 251
testy, 160
zmiany infrastruktury, 367
automatyzacja, 183
procesu wdrożeń, 185
standardowych procesów, 319
awaria, 306
potoku wdrożeń, 169
awarie produkcyjne, 313

B

badania niszczące, 368
baza danych zarządzania konfiguracją, 146
bezpieczeństwo, 70, 211
aplikacji, 349
informacji, 343, 357
łańcucha dostaw oprogramowania, 353
pracy, 118
środowiska, 355
wdrażania kodu, 259
bezpieczne wdrożenia, 261
bezpieczny system pracy, 61
biblioteka DML, 146
błędy, 162
budowanie nowej wiedzy, 62

C

C/A, complete and accurate, 43
CAB, change advisory board, 364
CDE, Cardholder Data Environment, 370
cele organizacji, 325
chat roomy, 317
ciągła integracja, 173, 178, 179

ciągle

- budowanie, 156
- dostarczanie, 139, 206
- uczenia się, 32, 299
- wdrażanie, 206

coaching kata, 76

COTS, commercial off-the-shelf, 391

czas

- przetwarzania, 41
- realizacji wdrażania, 43

czasy realizacji wdrażania, 42

D

DAA, Designated Approving Authority,

355

definiowanie czasu realizacji, 41

dług techniczny, 101, 330

DML, Definitive Media Library, 146

dni gier, 314

dodatkowe

funkcje, 56

procesy, 56

dokumentacja, 322, 371

dostarczanie wartości do klienta, 93

dostęp do telemetrii, 236

druga droga, 59

dzielenie się doświadczeniami, 334

dzień gry, 75

E

eksperymentowanie, 32, 69, 299

eliminowanie

ograniczeń, 54

procesów biurokratycznych, 295

trudności, 55

etyka Devops, 25

F

finansowanie, 117

G

gated commits, 178

generaliści, 116

globalne usprawnienia, 317

GRC, governance, risk, and compliance, 345

grouplet, 336

H

heroizm, 57

higiena produkcji, 267

historyjki użytkownika, 324

HRR, Hand-Off Readiness Review, 270

I

IDE, integrated development environment,

158

idealne środowisko Devops, 43

identyfikowanie ograniczeń, 54

implementacja przełączników funkcji, 202

infrastruktura, 147

telemetrii, 228

innowacje, 89

innowatorzy, 90

instalacje kodu, 54

instrumentacja, 248

środowiska, 359

integracja

mechanizmów zabezpieczeń, 363

technik wytwarzania oprogramowania,
273

zabezpieczeń, 347

zadań wdrażania, 190

integralność kodu źródłowego, 350

integrowanie

inżynierów OPS, 131

kodu, 156

środowisk, 156

inżynieria odporności, 314

inżynierowie

DEV, 263

OPS, 263

J

jakość, 64

K

- kaizen blitzes, 73
- kolejki, 388
- komentarze, 282
- kompilacja, 190
- konfiguracja testów, 54
- konflikty, 20
- kontrola
 - jakości, 65
 - zmian, 284
- koordynacja, 281, 286
- koszty, 24
- kultura uczenia się, 70, 76

L

- Lean, 383
- liczba
 - defektów, 267
 - przełączeń, 53
- limit WIP, 51
- limity pracy w toku, 50
- linka Andon, 170, 391
- LRR, launch readiness review, 270
- luki w telemetrii, 239
- luźno sprzężone usługi, 119

M

- małpia armia, 394
- manifest Agile, 384
- mapa strumienia wartości, 93, 96
- marketing
 - bezpośredni, 275
 - masowy, 275
- mechanizm
 - kontroli zabezpieczeń, 346
 - kontroli zmian, 284
 - odsyfania usługi, 268
- menedżer strumienia wartości, 95
- menedżerowie
 - techniczni, 95
 - wydania, 95
- mikrousługi, 212
- mity, 11, 390

model

- eksperymentalny, 311
- standardowy, 311
- monolity, 212
- MVP, minimal viable product, 330

N

- narzędzia, 288
- natywna chmura, 304
- norma PCI, 369

O

- obliczanie średnich, 246
- obserwator, 291
- ocena efektywności procesów, 293
- ochrona potoku wdrożeń, 360, 363
- oczekiwanie, 56
- odchylenie standardowe, 246
- oddzielenie wdrożeń od wydań, 194
- ograniczenie pracy, 49
- oprogramowanie bazujące na rewizji
 - master, 175
- oprogramowanie
 - COTS, 391
 - sterowane hipotezami, 273
- optymalizacja, 66
 - kosztów, 111
 - szybkości, 112
- organizacje
 - funkcjonalne, 110
 - macierzowe, 110
 - rynkowe, 110
- orientacja funkcyjonalna, 113
- OWASP, 349

P

- parametr konieczności wykonywania
 - przeróbek, 43
- parametry
 - infrastruktury, 242
 - poziomu aplikacji, 240
 - produkci, 234

- pierwsza droga, 47
- pionierzy, 90
- planowanie
 - analiz post-mortem, 306
 - cech funkcjonalnych, 278
 - poprawy, 100
 - zmian, 286
- podejście
 - big bang, 149
 - rozwojowe, 117
 - sztywne, 117
- podział obowiązków, 369
- poprawa widoczności pracy, 105
- poprawka w przód, 262
- POS, point of sale, 330
- potok wdrożeń, 141, 169, 190, 347, 363
- poziom
 - aplikacji, 239, 248
 - bazy danych, 248
 - biznesowy, 239
 - DEBUG, 232
 - ERROR, 232
 - FATAL, 232
 - INFO, 232
 - infrastruktury, 239
 - oprogramowania klienckiego, 239
 - potoku wdrożeń, 239
 - sieci, 248
 - systemu operacyjnego, 248
 - WARN, 232
- praca parami, 291
- prace
 - niestandardowe, 57
 - wykonane częściowo, 56
- praktyki UX, 265
- prawo Conwaya, 107, 118
- prezentowanie realizacji zadań, 344
- problemy, 20, 61
- proces
 - wdrażania, 267
 - zatwierdzania zmian, 283
- produktywne utrzymanie ruchu, 383
- program
 - Hubot, 318
 - NR, 74
- programowanie
 - sterowane testami, 291
 - w parach, 288, 291, 292
- projektowanie
 - granic zespołów, 118
 - organizacji, 107
- promienniki informacji, 236
- przegląd, 281
 - zmian, 287
- przełączanie
 - pomiędzy zadaniami, 56
 - pracy, 388
- przełączenia, 53
- przełącznik cech funkcjonalnych, 202
- przenoszenie pracy, 48
- przepływ, 32, 45, 47
 - pierwsza droga, 47
- przewlekłe konflikty, 386
- przezroczysty czas sprawności, 395
- przydzielenie łączników ops, 130
- publikowanie, 277

R

- raport DBIR, 354
- rejestrowanie zdarzeń, 231
- repozytorium
 - kodu źródłowego, 289, 320, 346
 - prawdy, 145
- retrospektywy, 133
- rewizja master, 175–178
- rewizje, 145
- RFC, request for change, 364
- router zdarzeń, 229
- rozkład Gaussa, 247, 249
- rozmiar
 - partii, 51
 - zespołów, 120
- rozpozyszczanie
 - praktyk, 336
 - wiedzy, 322
- rozwiązywanie problemów, 62, 225, 233
- rozwijanie metodyki, 91
- ruch, 56
 - Agile, 384
 - ciągłych dostaw, 385

- konferencji Velocity, 384
Lean, 383
Lean Startup, 385
Lean UX, 386
Rugged Computing, 386
Toyota Kata, 385
RUP, rational unified process, 384
- S**
- scrum, 132
SDLC, system development life cycle, 369
silosy, 115
skalowanie
 wydajności, 29
 zależności, 350
skuteczność orientacji funkcjonalnej, 113
słabe sygnały awarii, 311
SOA, service-oriented architectures, 119
SOE, system of engagement, 86
SOR, system of record, 86
SOT, shared operations team, 187
specjalisci, 116
spirala degradacji, 21, 25, 387
spotkania post-mortem, 392
sprint, 149
sprzężenia zwrotne, 32, 59, 259
SRE, site reliability engineers, 269
standardyzacja stosu technologii, 326
standupy, 132
strangler application, 215
strangler application pattern, 210
strumień wartości
 identyfikacja zespołów, 95
 inżynier pełnego stosu, 116
 mapa, 96
 produkcji, 39
 technologii, 40
 wybór, 83
swarming, 62
system
 SoE, 86, 88
 SoR, 86, 88
 uczenia się, 329
systemy CIS, 200
szybka transformata Fouriera, 252

- Ś**
- śledzenie defektów, 345
ślepe uruchomienia, 203
średnia, 247
środowisko zbliżone do produkcyjnego, 149

- T**
- tablica kanban, 134
TBD, trunk-based development, 175
TC, Test Certified, 337
TDD, test-driven development, 165, 184, 291
techniczne praktyki przepływu, 137
techniki TDD, 184
telemetria, 225, 233, 239
 produkci, 357
 zabezpieczeń, 358
teoria
 ograniczeń, 386
 zgnilego jabłka, 305
test, 54
 A/B, 275
 weryfikacyjny, 160
testowalność, 211
testowanie, 114, 156, 190
 automatyczne, 153
 cechy funkcjonalnej, 276
 statycznych zabezpieczeń, 351
 wydajności, 167
 wymagań niefunkcjonalnych, 168
testy
 akceptacyjne, 161
 integracji, 161
 jednostkowe, 161
 ręczne, 165
TPM, Total Productive Maintenance, 383
trendy, 19
trenowanie próbnych awarii, 314
trzecia droga, 69
tworzenie
 danych, 198
 infrastruktury, 147
parametrów produkcji, 234
repozytorium prawdy, 145

tworzenie
samoobsługowych parametrów, 237
środowisk na żądanie, 143
środowiska, 54
telemetrii, 225
telemetrii rejestrowania zdarzeń, 231
typ alertów, 267

U

uczenie się, 76, 303, 305, 329, 333
uniwersalność rozwiązania, 30
uruchamianie testów, 54
usługa Chaos Monkey, 394
usługi
brownfield, 86
greenfield, 86
niezmienialne, 215
wersjonowane, 215
współdzielone, 346
usprawnienia, 72, 317
globalne, 74
utrzymywanie spójnych środowisk, 187
uzgadnianie wspólnego celu, 100

W

wady ukryte, 315
wartość biznesowa metodyki DevOps, 28
wchodzenie z hukiem, 90
wdrażanie, 40, 186, 190
wdrożenia, 195
automatyczne, 189
produkcyjne, 263
samoobsługowe, 189
wewnętrzne konsultacje, 336
WIP, Work In Progress, 50
właściciel produktu, 95
wprowadzenie inżynierów do zespołów, 129
wskaźnik procentu C/A, 43
wskaźniki biznesowe, 240
współbieżność, 164
wstrzykiwanie awarii produkcyjnych, 313
wybór strumieni wartości, 83
wydajność, 211
wydanie, 195
wyglądzanie, 253

wykrywanie
anomalii, 253, 255
 błędów, 162
odstających, 246
wymagania niefunkcjonalne, 101
ujednolicone, 323

wytwarzanie oprogramowania

małymi partiami, 177

sterowane testami, 165

wyuczona bezradność, 24

wzmacnianie

kultury uczenia się, 76

pożądanych zachowań, 105

wzorce

odporności, 75

wydań bazujące na aplikacjach, 196

wydań bazujące na środowisku, 195

wydań kanarkowych, 200

wzorzec

aplikacji dusiciela, 216

strangler application, 210, 215

wdrażania niebieskie-zielone, 196

Z

zabezpieczenia, 114
zagrożenia, 290
zakres monitorowania, 267
zamrażanie zmian, 290
zarządzanie
usługą produkcyjną, 266
zmianami, 339
zasada
ciągłego uczenia się, 69
przepływu, 47
sprzężenia zwrotnego, 59
zbieranie danych, 229
zespoły rynkowe, 112
zespół
deweloperów, 95
operacji współdzielonych, 187
operacyjny, 95
transformacji, 98
walidacji, 95
zabezpieczeń, 95
zarządzania zmianami, 288

zgodność z przepisami, 339, 363, 371
zielona kompilacja, 160
zintegrowane środowisko programisty, 158
zintegrowanie zadań, 125
złożone systemy, 60
zmiany
 niskiego ryzyka, 365
 normalne, 364, 366
 pilne, 365
 standardowe, 364
zmniejszenie
 liczby przełączeń, 53
 wielkości partii, 50
zwiększać wydajności pracy, 127

PODZIĘKOWANIA

JEZ HUMBLE

Napisanie tej książki było spełnieniem marzeń przede wszystkim dla Gene'a. Praca z Gene'em i pozostałymi współautorami, Johnem i Patem, a także Toddem, Anną i Robynem, zespołem redakcyjnym i produkcyjnym IT Revolution była dla mnie ogromnym przywilejem i przyjemnością — dziękuję. Chcę również podziękować Nicole Forsgren, której praca w ciągu ostatnich trzech lat wraz z Gene'em, Alanną Brown, Nigelem Kerstenem i ze mną nad raportem *State of DevOps* przyczyniła się do rozwoju, przetestowania i doprecyzowania wielu koncepcji zaprezentowanych w tej książce. Moja żona Rani i moje dwie córki Amrita i Reshma dały mi bezgraniczną miłość i wsparcie w czasie mojej pracy nad tą książką, a także w każdej innej części mojego życia. Dziękuję Wam. Kocham Was. Na koniec chciałbym powiedzieć, że czuję się niesamowicie szczęśliwy, że jestem częścią społeczności DevOps, która prawie bez wyjątku podąża ścieżką empatii i rozwijającej się kultury szacunku i uczenia się. Dziękuję Wam wszystkim i każdemu z osobna.

JOHN WILLIS

Po pierwsze i najważniejsze, muszę podziękować mojej świętej żonie za to, że wytrzymuje z moją szaloną karierą. Wyrażenie tego, jak wiele nauczyłem się od współautorów tej książki, Patricka, Gene'a i Jeza, zajęłoby osobną książkę. Innymi bardzo ważnymi

doradcami w mojej podróży są Mark Hinkle, Mark Burgess, Andrew Clay Shafer i Michael Cote. Chcę również podziękować Adamowi Jacobowi za zatrudnienie mnie w firmie Chef i danie mi wolnej ręki w zakresie poznawania w pierwszych dniach tego czegoś, co nazywamy DevOps. Na koniec, co nie umniejsza jego zasług, dziękuję mojemu „współnikowi w przestępstwie” — współgospodarzowi *DevOps Cafe*, Damonowi Edwardsowi.

PATRICK DEBOIS

Chciałbym podziękować wszystkim osobom, które towarzyszyły mi w tej podróży. Chcę wyrazić Wam wszystkim ogromną wdzięczność.

GENE KIM

Wielkie podziękowania należą się Margueritte, mojej kochającej żonie, za prawie jedenasto niesamowitych lat. To wystarczająco długo, aby wspierać mnie podczas ponad pięciu lat, kiedy ciągle byłem zajęty. Dziękuję również moim synom Reidowi, Parke-rowi i Grantowi. Oczywiście dziękuję moim rodzicom Benowi i Gaili Kim za pomoc w zostaniu nerdem we wczesnym okresie mojego życia. Chcę również podziękować moim kolegom współautorom za wszystko, czego się od nich dowiedziałem, a także Annie Noak, Aly’emu Hoffmanowi, Robynowi Crummer-Olsenowi, Toddowi Satter-stenowi i pozostałym członkom zespołu IT Revolution za doprowadzenie projektu tej książki do jego zakończenia.

Jestem bardzo wdzięczny wszystkim ludziom, którzy nauczyli mnie wielu rzeczy tworzących podstawę niniejszej książki: Johnowi Allspawowi (Etsy), Alannie Brown (Puppet), Adrianowi Cockcroftowi (Battery Ventures), Justinowi Collinsowi (Brake-man Pro), Joshowi Cormanowi (Atlantic Council), Jasonowi Coxowi (The Walt Disney Company), Dominice DeGrandis (LeanKit), Damonowi Edwardsowi (DTO Solutions), dr Nicole Forsgren (Chef), Gary’emu Gruverowi, Samowi Guckenheimerowi (Micro-soft), Elisabeth Hendrickson (Pivotal Software), Nickowi Galbreathowi (Signal Scien-ces), Tomowi Limoncelliemu (Stack Exchange), Chrisowi Little’owi, Ryanowi Marten-sowi, Ernestowi Muellerowi (AlienVault), Mike’owi Orzenowi, Scottowi Prughowi (CSG International), Royowi Rapoportowi (Netflix), Tarunowi Reddy’emu (CA/Rally), Jesse’emu Robbinsowi (Orion Labs), Benowi Rockwoodowi (Chef), Andrew Shaferowi (Pivotal), Randy’emu Shoupowi (Stitch Fix), Jamesowi Turnbullowi (Kickstarter) i Jamesowi Wickettowi (Signal Sciences).

Chcę także podziękować wielu osobom, na których przykładzie badaliśmy niezwy-kłe historie podróży DevOps, w tym Justinowi Arbuckle’owi, Davidowi Ashmanowi, Charliemu Betzowi, Mike’owi Blandowi, dr. Touficowi Boubezowi, Em Campbell-Pretty, Jasonowi Chanowi, Pete’owi Cheslockowi, Rossowi Clantonowi, Jonathanowi Claudiusowi, Shawnowi Davenportowi, Jamesowi DeLuccia, Robowi Englandowi,

Johnowi Esserowi, Jamesowi Frymanowi, Paulowi Farrallowi, Nathenowi Harveyowi, Mirco Heringowi, Adamowi Jacobowi, Luke'owi Kaniesowi, Kaimarowi Karu, Nigelowi Kerstenowi, Courtney Kissler, Bethany Macri, Simonowi Morrisowi, Ianowi Malpassowi, Dianne Marsh, Normanowi Marksowi, Billowi Massiemu, Neilowi Mata-tallowi, Michaelowi Nygardowi, Patrickowi McDonnellowi, Eranowi Messeriemu, Heatherowi Mickmanowi, Jody'emu Mulkeyowi, Paulowi Mullerowi, Jesse'emu Newlandowi, Danowi Northowi, dr. Tapabracie Palowi, Michaelowi Rembetsy'emu, Mike'owi Rotherowi, Paulowi Stackowi, Garethowi Rushgrove'owi, Markowi Schwartzowi, Nathanowi Shimekowi, Billowi Shinnowi, JP Schneiderowi, dr. Stevenowi Spearowi, Laurence'owi Sweeneyowi, Jimowi Stonehamowi i Ryanowi Tomayko.

Jestem głęboko wdzięczny wielu recenzentom, których fantastyczne opinie pozwoliły nadać ksztalt tej książce: Willowi Albenziemu, JT Armstrongowi, Paulowi Auclairowi, Edowi Bellisowi, Danielowi Blanderowi, Mattowi Brenderowi, Alannie Brown, Brandenowi Burtonowi, Rossowi Clantonowi, Adrianowi Cockcroftowi, Jennifer Davis, Jessica DeVita, Stephenowi Feldmanowi, Martinowi Fisherowi, Stephenowi Fishmanowi, Jeffowi Gallimore'owi, Becky Hartman, Mattowi Hatchowi, Williamowi Hertlingowi, Robowi Hirschfeldowi, Timowi Hunterowi, Steinowi Inge'owi Morisbakiowi, Markowi Kleinowi, Alanowi Kraftowi, Bridget Kromhaut, Chrisowi Leavory'emu, Chrisowi Leavoyowi, Jenny'emu Madorsky'emu, Dave'owi Mangotowi, Chrisowi McDevittowi, Chrisowi McEniry'emu, Mike'owi McGarrowi, Thomasowi McGonagle'owi, Samowi McLeodowi, Byronowi Millerowi, Davidowi Mortmanowi, Chivasowi Nambiarowi, Charlesowi Nellesowi, Johnowi Osborne'owi, Mattowi O'Keefe'owi, Manuelowi Paisowi, Gary'emu Pedrettiemu, Danowi Piessensowi, Brianowi Prince'owi, Dennisowi Ravenelle'owi, Pete'owi Reidowi, Markosowi Rendellowi, Trevorowi Robertsowi, Jr Frederickowi Schollowi, Matthew Selheimerowi, Davidowi Severskiemu, Samiowi Shahowi, Paulowi Stackowi, Scottowi Stocktonowi, Dave'owi Temperowi, Toddowi Varlandowi, Jeremy'emu Voorhisowi i Brandenowi Williamsowi.

Kilka osób przekazało mi niewiarygodne spojrzenie na to, jak będzie wyglądała przyszłość tworzenia z wykorzystaniem nowoczesnych kompilatorów. Wśród tych osób wymienię Andrew Odewahna (O'Reilly Media), który pozwolił nam używać fantastycznej platformy przeglądów kodu Chimera, Jamesa Turnbulla (Kickstarter) za pomoc przy tworzeniu mojego pierwszego zestawu narzędzi do renderowania publikacji i Scotta Chacona (GitHub) za pracę nad systemem GitHub Flow przeznaczonym dla autorów.

Biogramy autorów



Gene Kim jest wielokrotnie nagradzanym CTO, naukowcem i autorem książek *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win** oraz *The Visible Ops Handbook*. Jest założycielem firmy IT Revolution i gospodarzem konferencji DevOps Enterprise Summit.



Jez Humble jest współautorem nagrodzonej nagrodą Jolta książki *Continuous Delivery* oraz przełomowej książki *Lean Enterprise*. Koncentruje się na udzielaniu firmom pomocy w częstym dostarczaniu cennego, wysokiej jakości oprogramowania dzięki implementacji skutecznych praktyk inżynierijnych.



Patrick Debois jest niezależnym konsultantem IT, który wypełnia lukę pomiędzy projektami a operacjami przy użyciu technik Agile w zarządzaniu projektami, rozwijaniu ich i administrowaniu systemem.



John Willis pracuje w branży zarządzania IT ponad 35 lat. Jest autorem sześciu książek z serii IBM Redbooks, a także założycielem firmy Chain Bridge Systems i jej głównym architektem. Obecnie pracuje w firmie Evangelist at Docker, Inc.

* Wydanie polskie: *Projekt Feniks. Powieść o IT, modelu DevOps i o tym, jak pomóc firmie w odniesieniu sukcesu*, Helion 2016.

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJE

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>