

Le Manuel DevOps : Comment Créer une Agilité, une Fiabilité et une Sécurité de Classe Mondiale dans les Organisations Technologiques

Préface

Eurêka !

Le chemin pour achever Le Manuel DevOps a été long—il a commencé par des appels Skype hebdomadaires entre les co-auteurs en février 2011, avec la vision de créer un guide prescriptif qui servirait de compagnon au livre alors inachevé *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win*.

Plus de cinq ans plus tard, après plus de deux mille heures de travail, Le Manuel DevOps est enfin là. Compléter ce livre a été un processus extrêmement long, bien que très gratifiant et rempli d'apprentissages incroyables, avec une portée bien plus large que ce que nous avions initialement envisagé.

Tout au long du projet, tous les co-auteurs partageaient la conviction que le DevOps est véritablement important, formée lors d'un moment personnel de révélation bien plus tôt dans chacune de nos carrières professionnelles, auquel, je suppose, beaucoup de nos lecteurs s'identifieront.

Gene Kim

J'ai eu le privilège d'étudier les organisations technologiques performantes depuis 1999, et l'une des premières constatations était que l'interaction entre les différents groupes fonctionnels des opérations IT, de la sécurité de l'information et du développement était cruciale pour le succès. Mais je me souviens encore de la première fois où j'ai vu l'ampleur de la spirale descendante qui se produit lorsque ces fonctions travaillent vers des objectifs opposés.

C'était en 2006, et j'ai eu l'opportunité de passer une semaine avec le groupe qui gérait les opérations IT externalisées d'un grand service de réservation de compagnies aériennes. Ils décrivaient les conséquences en aval de leurs grandes mises à jour logicielles annuelles : chaque mise à jour causait un chaos immense et des perturbations pour le sous-traitant ainsi

que pour les clients ; il y avait des pénalités de SLA (accord de niveau de service) à cause des interruptions affectant les clients ; il y avait des licenciements des membres du personnel les plus talentueux et expérimentés en raison des déficits de profit qui en résultait ; il y avait beaucoup de travail imprévu et de gestion de crises, de sorte que le personnel restant ne pouvait pas travailler sur les arriérés de demandes de service toujours croissants provenant des clients ; le contrat était maintenu par les actes héroïques de la gestion intermédiaire ; et tout le monde avait l'impression que le contrat serait condamné à être remis en appel d'offres dans trois ans.

Le sentiment d'impuissance et de futilité qui en résultait a créé pour moi les prémisses d'une croisade morale. Le développement semblait toujours être considéré comme stratégique, mais les opérations IT étaient vues comme tactiques, souvent déléguées ou entièrement externalisées, pour revenir cinq ans plus tard dans un état pire que celui dans lequel elles avaient été confiées.

Pendant de nombreuses années, beaucoup d'entre nous savaient qu'il devait y avoir une meilleure façon de faire. Je me souviens d'avoir vu les présentations de la conférence Velocity de 2009, décrivant des résultats étonnantes rendus possibles par l'architecture, les pratiques techniques et les normes culturelles que nous connaissons maintenant sous le nom de DevOps. J'étais tellement excité, car cela indiquait clairement la meilleure voie que nous recherchions tous. Et aider à diffuser cette idée a été l'une de mes motivations personnelles pour co-écrire The Phoenix Project. Vous pouvez imaginer à quel point il a été incroyablement gratifiant de voir la communauté plus large réagir à ce livre, décrivant comment il les a aidés à atteindre leurs propres moments de révélation.

Jez Humble

Mon moment de révélation concernant le DevOps a eu lieu dans une start-up en 2000 - mon premier emploi après avoir obtenu mon diplôme. Pendant un certain temps, nous n'étions que deux membres du personnel technique. Je faisais tout : réseau, programmation, support, administration des systèmes. Nous déployions le logiciel en production par FTP directement depuis nos postes de travail.

Puis, en 2004, j'ai obtenu un poste chez ThoughtWorks, une société de conseil où ma première mission était de travailler sur un projet impliquant environ soixante-dix personnes. J'étais dans une équipe de huit ingénieurs dont le travail à temps plein consistait à déployer notre logiciel dans un environnement de type production. Au début, c'était vraiment stressant. Mais au bout de quelques mois, nous sommes passés de déploiements manuels prenant deux semaines à un déploiement automatisé prenant une heure, où nous pouvions avancer et reculer en quelques millisecondes en utilisant le modèle de déploiement blue-green pendant les heures normales de bureau.

Ce projet a inspiré bon nombre des idées dans le livre Continuous Delivery (Addison-Wesley, 2000) et dans celui-ci. Une grande partie de ce qui me motive, ainsi que d'autres travaillant dans ce domaine, est la connaissance que, quelles que soient vos contraintes, nous pouvons toujours faire mieux, et le désir d'aider les gens dans leur parcours.

Patrick Debois

Pour moi, c'était une collection de moments. En 2007, je travaillais sur un projet de migration de centre de données avec quelques équipes Agile. J'étais jaloux qu'elles aient une productivité si élevée - capables de faire tant de choses en si peu de temps.

Pour ma mission suivante, j'ai commencé à expérimenter le Kanban dans les opérations et j'ai vu comment la dynamique de l'équipe a changé. Plus tard, à la conférence Agile Toronto 2008, j'ai présenté mon article IEEE sur ce sujet, mais j'ai eu l'impression qu'il n'a pas trouvé un large écho dans la communauté Agile. Nous avons créé un groupe d'administration système Agile, mais j'ai négligé l'aspect humain des choses.

Après avoir vu la présentation "10 Deploys per Day" de John Allspaw et Paul Hammond lors de la conférence Velocity 2009, j'étais convaincu que d'autres pensaient de la même manière. J'ai donc décidé d'organiser les premiers DevOpsDays, coinçant accidentellement le terme DevOps.

L'énergie de l'événement était unique et contagieuse. Lorsque les gens ont commencé à me remercier parce que cela avait changé leur vie pour le mieux, j'ai compris l'impact. Je n'ai pas cessé de promouvoir le DevOps depuis.

John Willis

En 2008, je venais de vendre une entreprise de conseil spécialisée dans les pratiques IT à grande échelle et les opérations de gestion de la configuration et de la surveillance (Tivoli) lorsque j'ai rencontré pour la première fois Luke Kanies (le fondateur de Puppet Labs). Luke faisait une présentation sur Puppet lors d'une conférence O'Reilly sur la gestion de la configuration (CM).

Au début, je traînais simplement au fond de la salle, tuant le temps et pensant : « Qu'est-ce que ce jeune de vingt ans pourrait m'apprendre sur la gestion de la configuration ? » Après tout, j'avais littéralement passé toute ma vie à travailler dans certaines des plus grandes entreprises du monde, les aidant à concevoir des solutions de gestion de la configuration et d'autres solutions de gestion des opérations. Cependant, environ cinq minutes après le début de sa session, je me suis avancé au premier rang et j'ai réalisé que tout ce que j'avais fait pendant les

vingt dernières années était faux. Luke décrivait ce que j'appelle maintenant la gestion de la configuration de deuxième génération.

Après sa session, j'ai eu l'occasion de m'asseoir et de prendre un café avec lui. J'étais totalement convaincu par ce que nous appelons maintenant l'infrastructure en tant que code. Cependant, pendant que nous prenions un café, Luke a commencé à aller encore plus loin, expliquant ses idées. Il a commencé à me dire qu'il croyait que les opérations allaient devoir commencer à se comporter comme des développeurs de logiciels. Ils allaient devoir conserver leurs configurations dans un contrôle de version et adopter des modèles de livraison CI/CD pour leur flux de travail. Étant à l'époque un ancien de l'exploitation IT, je pense que je lui ai répondu quelque chose comme : « Cette idée va sombrer comme un plomb avec les gars des Ops. » (J'avais clairement tort.)

Puis, environ un an plus tard, en 2009, lors d'une autre conférence O'Reilly, Velocity, j'ai vu Andrew Clay Shafer faire une présentation sur l'Agile Infrastructure. Dans sa présentation, Andrew montrait cette image emblématique d'un mur entre les développeurs et les opérations avec une représentation métaphorique du travail jeté par-dessus le mur. Il a appelé cela « le mur de la confusion ». Les idées qu'il exprimait dans cette présentation ont codifié ce que Luke essayait de me dire un an plus tôt. Ce fut l'ampoule pour moi. Plus tard cette année-là, j'étais le seul Américain invité aux premiers DevOpsDays à Gand. À la fin de cet événement, ce que nous appelons DevOps était clairement dans mon sang.

Il est clair que les co-auteurs de ce livre sont tous parvenus à une épiphanie similaire, même s'ils y sont arrivés par des chemins très différents. Mais il y a maintenant une multitude de preuves que les problèmes décrits ci-dessus se produisent presque partout, et que les solutions associées au DevOps sont presque universellement applicables.

L'objectif de l'écriture de ce livre est de décrire comment reproduire les transformations DevOps dont nous avons fait partie ou que nous avons observées, ainsi que de dissiper bon nombre des mythes sur les raisons pour lesquelles le DevOps ne fonctionnerait pas dans certaines situations. Voici quelques-uns des mythes les plus courants que nous entendons au sujet du DevOps.

Mythes

Le DevOps est uniquement pour les start-ups

Bien que les pratiques DevOps aient été initiées par les entreprises web-scale, les "licornes" de l'Internet comme Google, Amazon, Netflix et Etsy, chacune de ces organisations a, à un moment de leur histoire, risqué de faire faillite à cause des problèmes associés aux organisations plus traditionnelles "cheval de trait" : des mises à jour de code très dangereuses sujettes à des échecs catastrophiques, l'incapacité de publier des fonctionnalités assez rapidement pour battre la concurrence, des préoccupations de conformité, une incapacité à évoluer, des niveaux élevés de méfiance entre le Développement et les Opérations, et ainsi de suite.

Cependant, chacune de ces organisations a été capable de transformer leur architecture, leurs pratiques techniques et leur culture pour créer les résultats étonnantes que nous associons au DevOps. Comme l'a plaisanté le Dr. Branden Williams, un dirigeant en sécurité de l'information : « Qu'on ne parle plus de licornes ou de chevaux DevOps, mais seulement de pur-sang et de chevaux qui vont à la fabrique de colle. »

Le DevOps remplace l'Agile

Les principes et pratiques DevOps sont compatibles avec Agile, beaucoup observant que le DevOps est une continuation logique du parcours Agile commencé en 2001. Agile sert souvent de facilitateur efficace du DevOps, en raison de son accent sur les petites équipes livrant continuellement du code de haute qualité aux clients.

De nombreuses pratiques DevOps émergent si nous continuons à gérer notre travail au-delà de l'objectif du « code potentiellement livrable » à la fin de chaque itération, en l'étendant à un code toujours en état de déploiement, avec des développeurs intégrant leur code quotidiennement, et en démontrant nos fonctionnalités dans des environnements semblables à la production.

Le DevOps est incompatible avec ITIL

Beaucoup voient le DevOps comme une réaction contre ITIL ou ITSM (IT Service Management), qui a été publié pour la première fois en 1989. ITIL a largement influencé plusieurs générations de praticiens des Ops, y compris un des co-auteurs, et est une bibliothèque de pratiques en constante évolution visant à codifier les processus et les pratiques qui sous-tendent les opérations IT de classe mondiale, couvrant la stratégie, la conception et le support des services.

Les pratiques DevOps peuvent être rendues compatibles avec les processus ITIL. Cependant, pour supporter les délais plus courts et les fréquences de déploiement plus élevées associés au DevOps, de nombreux aspects des processus ITIL deviennent entièrement automatisés, résolvant de nombreux problèmes associés aux processus de gestion de la configuration et des mises en production (par exemple, maintenir à jour la base de données de gestion de

configuration et les bibliothèques logicielles définitives). Et parce que le DevOps nécessite une détection rapide et une récupération lors des incidents de service, les disciplines ITIL de conception de service, de gestion des incidents et des problèmes restent aussi pertinentes que jamais.

Le DevOps est incompatible avec la sécurité de l'information et la conformité

L'absence de contrôles traditionnels (par exemple, la séparation des tâches, les processus d'approbation des changements, les revues de sécurité manuelles à la fin du projet) peut inquiéter les professionnels de la sécurité de l'information et de la conformité.

Cependant, cela ne signifie pas que les organisations DevOps n'ont pas de contrôles efficaces. Au lieu que les activités de sécurité et de conformité soient effectuées uniquement à la fin du projet, les contrôles sont intégrés à chaque étape du travail quotidien dans le cycle de vie du développement logiciel, ce qui entraîne de meilleurs résultats en termes de qualité, de sécurité et de conformité.

Mythe - Le DevOps signifie l'élimination des opérations IT, ou "NoOps"

Beaucoup interprètent à tort le DevOps comme l'élimination complète de la fonction des opérations IT. Cependant, cela est rarement le cas. Bien que la nature du travail des opérations IT puisse changer, il reste aussi important que jamais. Les opérations IT collaborent beaucoup plus tôt dans le cycle de vie du logiciel avec le Développement, qui continue de travailler avec les opérations IT bien après que le code a été déployé en production.

Au lieu que les opérations IT effectuent des travaux manuels provenant de tickets de travail, elles permettent la productivité des développeurs grâce à des API et des plateformes en libre-service qui créent des environnements, testent et déploient le code, surveillent et affichent la télémétrie de production, etc. En faisant cela, les opérations IT deviennent plus semblables au Développement (tout comme le QA et l'Infosec), engagées dans le développement de produits, où le produit est la plateforme que les développeurs utilisent pour tester, déployer et exécuter en toute sécurité, rapidement et en toute sécurité leurs services IT en production.

Le DevOps est juste "l'infrastructure en tant que code" ou l'automatisation

Bien que de nombreux modèles DevOps présentés dans ce livre nécessitent de l'automatisation, le DevOps exige également des normes culturelles et une architecture permettant d'atteindre les objectifs partagés tout au long de la chaîne de valeur IT. Cela va bien au-delà de l'automatisation. Comme l'a écrit Christopher Little, un dirigeant technologique et l'un des premiers chroniqueurs du DevOps : « Le DevOps n'est pas seulement l'automatisation, tout comme l'astronomie n'est pas seulement les télescopes. »

Le DevOps est uniquement pour les logiciels open source

Bien que de nombreuses histoires de réussite DevOps se déroulent dans des organisations utilisant des logiciels tels que la pile LAMP (Linux, Apache, MySQL, PHP), la réalisation des résultats DevOps est indépendante de la technologie utilisée. Des réussites ont été obtenues avec des applications écrites en Microsoft.NET, COBOL, et en code assembleur pour mainframe, ainsi qu'avec SAP et même des systèmes embarqués (par exemple, le firmware HP LaserJet).

PROPAGER LE MOMENT AHA!

Chacun des auteurs a été inspiré par les innovations incroyables se produisant dans la communauté DevOps et par les résultats qu'elles créent : des systèmes de travail sécurisés et permettant à de petites équipes de développer et de valider rapidement et indépendamment du code qui peut être déployé en toute sécurité chez les clients. Étant donné notre conviction que le DevOps est la manifestation de la création d'organisations dynamiques et apprenantes qui renforcent continuellement des normes culturelles de haute confiance, il est inévitable que ces organisations continuent d'innover et de gagner sur le marché.

Nous espérons sincèrement que The DevOps Handbook servira de ressource précieuse pour de nombreuses personnes de différentes manières : un guide pour planifier et exécuter des transformations DevOps, un ensemble d'études de cas pour rechercher et apprendre, une chronique de l'histoire de DevOps, un moyen de créer une coalition englobant les Product Owners, l'Architecture, le Développement, le QA, les Opérations IT et la Sécurité de l'information pour atteindre des objectifs communs, un moyen d'obtenir le soutien des plus hauts niveaux de leadership pour les initiatives DevOps, ainsi qu'une impérative morale de changer notre façon de gérer les organisations technologiques pour permettre une meilleure efficacité et efficience, tout en permettant un environnement de travail plus heureux et plus humain, aidant chacun à devenir des apprenants à vie - cela aide non seulement chacun à atteindre ses objectifs les plus élevés en tant qu'êtres humains, mais aussi à aider leurs organisations à réussir.

Avant-propos

Dans le passé, de nombreux domaines de l'ingénierie ont connu une sorte d'évolution notable, "montant de niveau" dans leur compréhension de leur propre travail. Bien qu'il existe des programmes universitaires et des organisations de soutien professionnel situées au sein de disciplines spécifiques de l'ingénierie (civile, mécanique, électrique, nucléaire, etc.), le fait est que la société moderne a besoin que toutes les formes d'ingénierie reconnaissent les avantages de travailler de manière multidisciplinaire.

Pensez à la conception d'un véhicule haute performance. Où s'arrête le travail d'un ingénieur mécanique et où commence celui d'un ingénieur électrique ? Où (et comment, et quand) une personne possédant des connaissances en aérodynamique (qui aurait certainement des opinions bien arrêtées sur la forme, la taille et l'emplacement des fenêtres) devrait-elle collaborer avec un expert en ergonomie des passagers ? Qu'en est-il des influences chimiques du mélange de carburant et de l'huile sur les matériaux du moteur et de la transmission au cours de la durée de vie du véhicule ? Il y a d'autres questions que nous pouvons poser sur la conception d'une automobile, mais le résultat final est le même : le succès des entreprises techniques modernes nécessite absolument la collaboration de multiples perspectives et expertises.

Pour qu'un domaine ou une discipline progresse et mûrisse, il doit atteindre un point où il peut réfléchir de manière réfléchie à ses origines, rechercher un ensemble diversifié de perspectives sur ces réflexions et placer cette synthèse dans un contexte utile pour la vision que la communauté a de l'avenir.

Ce livre représente une telle synthèse et doit être considéré comme une collection essentielle de perspectives sur le domaine (je soutiendrai, encore émergent et en rapide évolution) de l'ingénierie logicielle et des opérations.

Peu importe l'industrie dans laquelle vous évoluez ou le produit ou service que votre organisation fournit, cette façon de penser est primordiale et nécessaire à la survie de chaque dirigeant d'entreprise et de technologie.

John Allspaw, CTO, Etsy

Brooklyn, NY, août 2016

Table des matières

Le Manuel DevOps : Comment Créer une Agilité, une Fiabilité et une Sécurité de Classe Mondiale dans les Organisations Technologiques	1
Préface.....	1
Gene Kim	1
Jez Humble	2
Patrick Debois.....	3
John Willis.....	3
Mythes.....	5
Le DevOps est incompatible avec ITIL	5
Mythe - Le DevOps signifie l'élimination des opérations IT, ou "NoOps".....	6
PROPAGER LE MOMENT AHA!	7
Avant-propos	8
Imaginez un Monde Où Dev et Ops Deviennent DevOps	17
Le problème : Quelque chose dans votre organisation doit nécessairement être amélioré (sinon vous ne lirez pas ce livre)	20
La Spirale descendante en trois actes.....	21
Les coûts : humains et économiques.....	24
L'Éthique de DevOps : Il existe une meilleure façon	25
Briser la spirale descendante avec DevOps	25
La valeur commerciale du DevOps	27
Le DevOps aide à échelonner la productivité des développeurs.....	28
L'universalité de la solution	30
Le manuel DevOps : un guide essentiel.....	31
Partie I - Les trois voies	33
La première voie : Les principes du flux	42
Rendre notre travail visible	42
Limiter le travail en cours (WIP)	44
Réduire les tailles de lot.....	45
Réduire le nombre de transmissions	48
Identifier et éliminer en continu nos contraintes	49
Éliminer les difficultés et le gaspillage dans le flux de valeur	50
La deuxième voie : Les principes de retour de l'information	53
Travailler en toute sécurité dans les systèmes complexes	53
Voir les problèmes lorsqu'ils se produisent	54
Réunir et résoudre les problèmes pour construire de nouvelles connaissances	56

Pousser la qualité plus près de la source.....	58
Permettre l'optimisation pour les centres de travail en aval	59
La troisième voie : Les principes d'apprentissage continu et d'expérimentation.....	61
Favoriser l'apprentissage organisationnel et une culture de sécurité	62
Institutionnaliser l'amélioration du travail quotidien	64
Transformer les découvertes locales en améliorations globales	65
Injecter des modèles de résilience dans notre travail quotidien.....	66
Les leaders renforcent une culture d'apprentissage	67
Partie II - Par où commencer	71
Sélectionner le Flux de Valeur par lequel Commencer.....	72
Sélection du Flux de Valeur par lequel Commencer	74
Services Greenfield vs. Brownfield	75
Exemples de Transformations Brownfield Réussies	76
Considérer les Systèmes d'Enregistrement et les Systèmes d'Engagement.....	77
Commencer avec les Groupes les Plus Sympathiques et Innovants	78
Étendre le DevOps dans toute notre organisation	79
Comprendre le travail dans notre flux de valeur, le rendre visible et l'étendre à l'ensemble de l'organisation.....	81
Identification des équipes soutenant notre flux de valeur.....	82
Créer une carte de flux de valeur pour voir le travail	83
Créer une équipe de transformation dédiée	85
S'accorder sur un objectif commun.....	87
Garder des horizons de planification courts.....	88
Réserver 20 % des cycles aux exigences non fonctionnelles et à la réduction de la dette technique	88
Étude de Cas - L'opération InVersion chez LinkedIn (2011)	90
Augmenter la visibilité du travail.....	92
Utiliser les outils pour renforcer les comportements souhaités	92
Comment concevoir notre organisation et notre architecture en tenant compte de la loi de Conway	95
Archétypes organisationnels.....	97
Problèmes souvent causés par une orientation fonctionnelle excessive (optimisation des coûts)	98
Permettre aux équipes orientées marché (optimisation de la vitesse)	99
Faire fonctionner l'orientation fonctionnelle	100
Tester, opérations et sécurité comme le travail de tous, tous les jours	101
Permettre à chaque membre de l'équipe d'être un généraliste.....	102

Financer non pas des projets, mais des services et des produits	104
Concevoir les limites des équipes selon la loi de Conway	105
Créer des architectures découplées pour favoriser la productivité et la sécurité des développeurs	105
Maintenir la taille des équipes petites (la règle de l'équipe de deux pizzas)	107
Étude de cas : La mise en place d'API chez Target (2015).....	108
Comment obtenir d'excellents résultats en intégrant les opérations dans le travail quotidien du développement.....	110
Créer des services partagés pour améliorer la productivité des équipes de développement	112
Intégrer des ingénieurs Ops dans nos équipes de service.....	114
Assigner un Liaison Ops à Chaque Équipe de Service	115
Intégrer les Ops dans les rituels Dev.....	115
Rendre le Travail des Ops Visible sur des Tableaux Kanban Partagés.....	118
Partie III - La Première Voie, les Pratiques Techniques du Flux.....	120
Créer les Fondations de Notre Pipeline de Déploiement	121
Activer la création à la demande d'environnements de développement, de test et de production	122
Créer notre répertoire unique de vérité pour l'ensemble du système.....	124
Faire de l'infrastructure plus facile à reconstruire qu'à réparer	126
Modifier notre définition du « DONE » de développement pour inclure l'exécution dans des environnements de type production.....	127
Activer des tests automatisés rapides et fiables	130
Construire, tester et intégrer continuellement notre code et nos environnements	132
Construire une suite de tests de validation automatisée rapide et fiable	135
Attraper les Erreurs le Plus Tôt Possible dans Nos Tests Automatisés	137
Assurer que les tests s'exécutent rapidement (en parallèle, si nécessaire).....	139
Écrire nos tests automatisés avant d'écrire le code (Développement piloté par les tests)	140
Automatiser le plus possible nos tests manuels	141
Intégrer les tests de performance dans notre suite de tests.....	142
Intégrer les tests des exigences non fonctionnelles dans notre suite de tests.....	143
Actionner notre corde Andon lorsque le pipeline de déploiement est interrompu	144
Pourquoi nous devons tirer le cordon Andon.....	145
Activer et pratiquer l'intégration continue.....	147
Développement en petits lots et ce qui se passe lorsque nous validons du code dans le tronc peu fréquemment.....	150
Adopter les pratiques de développement basées sur le tronc.....	151

Étude de cas - Intégration continue chez Bazaarvoice (2012).....	152
Automatiser et permettre des déploiements à faible risque	155
AUTOMATISER NOTRE PROCESSUS DE DÉPLOIEMENT	157
Étude de Cas : Déploiements quotidiens chez CSG International (2013)	159
Automatiser les déploiements en libre-service	161
Intégrer le déploiement de code dans le pipeline de déploiement	162
Étude de cas – Etsy - Déploiement par les développeurs en libre-service, un exemple de déploiement continu (2014).....	164
Découpler les déploiements des lancements	165
Modèles de lancement basés sur l'environnement	167
Étude de cas - Dixons Retail - Déploiement blue-green pour le système de point de vente (2008)	169
Modèles basés sur l'application pour permettre des déploiements plus sûrs.....	171
Étude de cas- Lancement en mode sombre de Facebook Chat (2008)	174
Enquête sur la livraison continue et le déploiement continu en pratique	176
Architecturer pour des mises en production à faible risque	178
Une architecture qui favorise la productivité, la testabilité et la sécurité	180
Archétypes architecturaux : Monolithes vs. Microservices.....	181
Étude de cas - Architecture évolutive chez Amazon (2002)	183
Utiliser le modèle de l'application Strangler pour faire évoluer notre architecture d'entreprise en toute sécurité	184
Étude de cas - Modèle Strangler chez Blackboard Learn (2011)	185
Partie IV - La deuxième voie - Les pratiques du feedback.....	189
Créer de la télémétrie pour permettre de voir et de résoudre les problèmes	191
Créer notre infrastructure de télémétrie centralisée.....	194
Créer une télémétrie de journalisation des applications qui aide à la production	197
Utiliser la télémétrie pour guider la résolution de problèmes.....	199
Activer la création de métriques de production comme partie du travail quotidien	200
Créer un accès en libre-service à la télémétrie et aux radiateurs d'information	201
Étude de cas - Créer des métriques en libre-service chez LinkedIn (2011).....	203
Trouver et combler les lacunes de télémétrie.....	204
Métriques d'applications et d'affaires.....	205
Métriques d'infrastructure	207
Analyser la télémétrie pour mieux anticiper les problèmes et atteindre les objectifs	210
Utiliser les moyennes et les écarts types pour détecter les problèmes potentiels	211
Instrumenter et alerter sur les résultats indésirables	213

Problèmes qui surviennent lorsque nos données de télémétrie ont une distribution non gaussienne	214
Étude de cas : Auto-Scaling Capacity chez Netflix (2012)	216
Utilisation des techniques de détection d'anomalies	217
Étude de cas : Détection avancée des anomalies (2014)	219
Permettre des retours pour que le développement et les opérations puissent déployer le code en toute sécurité.....	223
Utiliser la télémétrie pour rendre les développements plus sûrs.....	225
Partager les tâches de rotation de téléavertisseur (Pager) entre Dev et Ops.....	226
Faire suivre le travail en aval par les développeurs	228
Faire en sorte que les développeurs gèrent initialement eux-mêmes leur service en production	229
Étude de cas - La révision de préparation au lancement et au transfert chez Google (2010)	232
Intégrer le développement piloté par hypothèses et les tests A/B dans notre travail quotidien	236
Une brève histoire des tests A/B.....	237
Intégrer les tests A/B dans nos tests de fonctionnalité	238
Intégrer les tests A/B dans notre planification des fonctionnalités	239
Étude de cas - Doublement de la croissance du revenu grâce à un cycle de mise en œuvre rapide	240
Créer des processus de révision et de coordination pour améliorer la qualité de notre travail actuel.....	243
Les dangers des processus d'approbation des changements	245
Dangers des « Changements trop contrôlés »	246
Permettre la coordination et la planification des changements	248
Permettre la révision par les pairs des changements.....	249
Étude de cas - Revues de code chez Google (2010).....	250
Potentiels dangers de faire plus de tests manuels et de gel des modifications	252
Utiliser la programmation en binôme pour améliorer tous nos changements	252
Étude de cas - Le Pair Programming remplace les processus de revue de code défaillants chez Pivotal Labs (2011)	254
Évaluer l'efficacité des processus de pull request.....	255
Couper sans peur les procédures bureaucratiques.....	256
Partie V - La troisième voie - Les pratiques d'apprentissage continu et d'expérimentation.....	259
Activer et injecter l'apprentissage dans le travail quotidien	260
Établir une culture juste et apprenante.....	261
Planifier les réunions de post-mortem sans blâme après les accidents	263

Publier nos post-mortems le plus largement possible	265
Diminuer les tolérances aux incidents pour détecter des signaux de défaillance toujours plus faible	266
Redéfinir l'échec et encourager la prise de risques calculés.....	268
Injecter des échecs en production pour permettre la résilience et l'apprentissage	268
Instituer des journées de jeu pour répéter les défaillances	270
Convertir les découvertes locales en améliorations globales	273
Utiliser des salles de discussion et des bots de discussion pour automatiser et capturer les connaissances organisationnelles	273
Automatiser les processus standardisés dans les logiciels pour la réutilisation	275
Créer un dépôt de code source unique et partagé pour toute notre organisation	276
Diffuser les connaissances en utilisant les tests automatisés comme documentation et les communautés de pratique.....	278
Concevoir pour les opérations à travers des exigences non fonctionnelles codifiées.....	279
Construire des histoires utilisateurs d'opérations réutilisables dans le développement ..	279
Veiller à ce que les choix techniques aident à atteindre les objectifs opérationnels.....	280
ÉTUDE DE CAS - Standardisation d'une nouvelle pile technologique chez Etsy (2010)	282
Réserver du temps pour créer un apprentissage et une amélioration organisationnels	283
Institutionnaliser des rituels pour rembourser la dette technique	284
Permettre à chacun d'enseigner et d'apprendre	286
Partagez vos expériences des conférences DevOps	287
Étude de cas - Conférences technologiques internes chez Nationwide Insurance, Capital One et Target (2014).....	288
Créer un système de consultation et de coaching interne pour propager les pratiques	289
Partie VI - Les pratiques d'intégration de la sécurité de l'information, de la gestion du changement et de la conformité.....	293
La sécurité de l'information comme tâche de tous, tous les jours	293
Intégrer la sécurité dans les démonstrations d'itération de développement	294
Intégrer la sécurité dans le suivi des défauts et les retours d'expérience	295
Intégrer les contrôles de sécurité préventifs dans les répertoires de code source partagés les services partagés	295
Intégrer la sécurité dans notre pipeline de déploiement	297
Assurer la sécurité de l'application	298
Étude de Cas - Tests de Sécurité Statique chez Twitter (2009).....	300
Assurer la sécurité de notre chaîne d'approvisionnement logiciel.....	302
Assurer la sécurité de l'environnement.....	304
Étude de cas - 18F : Automatisation de la conformité pour le gouvernement fédéral avec Compliance Masonry	304

Intégrer la sécurité de l'information dans la télémétrie de production.....	305
Créer une télémétrie de sécurité dans nos applications.....	306
Créer une télémétrie de sécurité dans notre environnement	307
Étude de cas - Instrumenter l'environnement chez Etsy (2010)	307
Protéger notre pipeline de déploiement.....	309
Protection de la pipeline de déploiement	311
Intégrer la sécurité et la conformité dans les processus d'approbation des changements	311
Recatégoriser la majorité de nos changements à faible risque comme des changements standards	312
Que faire lorsque les changements sont classés comme des changements normaux	313
Étude de cas - Changements d'infrastructure automatisés en tant que changements standards chez Salesforce.com (2012)	314
Réduire la dépendance à la séparation des tâches	316
Étude de cas - Conformité PCI et un conte de précaution sur la séparation des tâches chez Etsy (2014)	317
Assurer la documentation et les preuves pour les auditeurs et les agents de conformité.	318
Étude de Cas - Prouver la Conformité dans les Environnements Réglementés (2015)	319
Étude de Cas - S'appuyer sur la Télémétrie de Production pour les Systèmes de Distributeurs Automatiques de Billets	321
Un Appel à l'Action - Conclusion du DevOps Handbook	323
Annexes	326
Annexe 1 : la convergence du Devops	326
Le mouvement LEAN	326
Le mouvement agile	326
Le mouvement de la conférence Velocity	327
Le mouvement de l'infrastructure agile	327
Le mouvement de la livraison continue	327
Le mouvement Toyota Kata	327
Le mouvement Lean Startup	328
Le mouvement Lean UX	328
Le mouvement Rugged Computing	328
Annexe 2 : Théorie des contraintes et conflits nucléaires, chroniques	329
Annexe 3 – Formule tabulaire de la spirale descendante	330
Annexe 4 – Les dangers des passages de relais et des files d'attente	331
Annexe 5 – Mythes de la sécurité industrielle	333
Annexe 6 – Le cordon Andon de Toyota.....	334

Annexe 7 - Logiciels commerciaux.....	335
Annexe 8 - Réunions Post-Mortem	336
Annexe 9 - L'armée Simienne	338
Annexe 10 - Disponibilité transparente.....	339

Imaginez un Monde Où Dev et Ops Deviennent DevOps

Une introduction à The DevOps Handbook

Imaginez un monde où les propriétaires de produits, le développement, la QA, les opérations IT et la sécurité de l'information travaillent ensemble, non seulement pour s'aider mutuellement, mais aussi pour garantir le succès global de l'organisation. En travaillant vers un objectif commun, ils permettent un flux rapide de travail planifié en production (par exemple, réaliser des dizaines, des centaines, voire des milliers de déploiements de code par jour), tout en atteignant une stabilité, une fiabilité, une disponibilité et une sécurité de classe mondiale.

Dans ce monde, des équipes interfonctionnelles testent rigoureusement leurs hypothèses sur les fonctionnalités qui raviront le plus les utilisateurs et feront avancer les objectifs organisationnels. Elles ne se contentent pas de mettre en œuvre des fonctionnalités pour les utilisateurs, mais s'assurent activement que leur travail circule de manière fluide et fréquente à travers toute la chaîne de valeur sans causer de chaos et de perturbations aux opérations IT ou à tout autre client interne ou externe.

Simultanément, la QA, les opérations IT et la sécurité de l'information travaillent toujours à réduire les frictions pour l'équipe, en créant des systèmes de travail qui permettent aux développeurs d'être plus productifs et d'obtenir de meilleurs résultats. En ajoutant l'expertise de la QA, des opérations IT et de la sécurité de l'information aux équipes de livraison et en utilisant des outils et des plateformes automatisés en libre-service, les équipes peuvent utiliser cette expertise dans leur travail quotidien sans dépendre d'autres équipes.

Cela permet aux organisations de créer un système de travail sûr, où de petites équipes peuvent rapidement et indépendamment développer, tester et déployer du code et de la valeur rapidement, en toute sécurité, de manière fiable et sécurisée aux clients. Cela permet aux organisations de maximiser la productivité des développeurs, de favoriser l'apprentissage organisationnel, de créer une grande satisfaction des employés et de gagner sur le marché.

Ce sont les résultats que produit DevOps. Pour la plupart d'entre nous, ce n'est pas le monde dans lequel nous vivons. Plus souvent qu'autrement, le système dans lequel nous travaillons est défaillant, ce qui entraîne des résultats extrêmement médiocres bien en deçà de notre véritable potentiel. Dans notre monde, le développement et les opérations IT sont des adversaires ; les activités de test et de sécurité de l'information se produisent uniquement à la fin d'un projet, trop tard pour corriger les problèmes trouvés ; et presque toute activité critique nécessite trop d'efforts manuels et trop de transferts, nous laissant toujours en attente. Non seulement cela contribue à des délais extrêmement longs pour accomplir quoi que ce soit, mais la qualité de notre travail, en particulier les déploiements en production, est également problématique et chaotique, entraînant des impacts négatifs pour nos clients et notre entreprise.

En conséquence, nous sommes loin de nos objectifs, et toute l'organisation est insatisfaite de la performance de l'IT, ce qui entraîne des réductions de budget et des employés frustrés et malheureux qui se sentent impuissants face à un processus et des résultats immuables.

La solution ? Nous devons changer notre manière de travailler ; DevOps nous montre la meilleure voie à suivre.

Pour mieux comprendre le potentiel de la révolution DevOps, regardons la révolution industrielle des années 1980. En adoptant les principes et les pratiques Lean, les organisations manufacturières ont considérablement amélioré la productivité des usines, les délais de livraison des clients, la qualité des produits et la satisfaction des clients, leur permettant de gagner sur le marché.

Avant la révolution, les délais moyens de commande des usines manufacturières étaient de six semaines, avec moins de 70 % des commandes livrées à temps. En 2005, avec la mise en œuvre généralisée des pratiques Lean, les délais moyens des produits étaient tombés à moins de trois semaines, et plus de 95 % des commandes étaient livrées à temps. Les organisations qui n'ont pas mis en œuvre les pratiques Lean a perdu des parts de marché, et beaucoup ont fait faillite.

De même, la barre a été relevée pour la livraison de produits et services technologiques—ce qui était suffisant dans les décennies précédentes ne l'est plus maintenant. Depuis les quatre dernières décennies, le coût et le temps nécessaires pour développer et déployer des capacités et des fonctionnalités stratégiques pour les entreprises ont chuté de plusieurs ordres de grandeur. Dans les années 1970 et 1980, la plupart des nouvelles fonctionnalités nécessitaient de un à cinq ans pour être développées et déployées, souvent au coût de dizaines de millions de dollars.

Dans les années 2000, grâce aux avancées technologiques et à l'adoption des principes et pratiques Agile, le temps nécessaire pour développer de nouvelles fonctionnalités avait diminué à des semaines ou des mois, mais le déploiement en production nécessitait encore des semaines ou des mois, souvent avec des résultats catastrophiques.

Et en 2010, avec l'introduction de DevOps et l'évolution incessante du matériel, des logiciels, et maintenant du cloud, les fonctionnalités (et même des entreprises entières en démarrage) pouvaient être créées en semaines, déployées rapidement en production en quelques heures ou minutes - pour ces organisations, le déploiement est finalement devenu routinier et à faible risque. Ces organisations peuvent réaliser des expériences pour tester des idées commerciales, découvrant lesquelles créent le plus de valeur pour les clients et l'organisation dans son ensemble, lesquelles sont ensuite développées en fonctionnalités qui peuvent être rapidement et en toute sécurité déployées en production.

Tableau 1. La tendance toujours accélérante vers une livraison de logiciels plus rapide, moins coûteuse et à faible risque

	1970s–1980s	1990s	2000s–Present
Era	Mainframes	Client/Server	Commoditization and Cloud
Representative technology of era	COBOL, DB2 on MVS, etc.	C++, Oracle, Solaris, etc.	Java, MySQL, Red Hat, Ruby on Rails, PHP, etc.
Cycle time	1–5 years	3–12 months	2–12 weeks
Cost	\$1M–\$100M	\$100k–\$10M	\$10k–\$1M
At risk	The whole company	A product line or division	A product feature
Cost of failure	Bankruptcy, sell the company, massive layoffs	Revenue miss, CIO's job	Negligible

(Source: Adrian Cockcroft, “Velocity and Volume (or Speed Wins),” presentation at FlowCon, San Francisco, CA, November 2013.)

Aujourd’hui, les organisations adoptant les principes et pratiques DevOps déploient souvent des changements des centaines, voire des milliers de fois par jour. À une époque où l’avantage compétitif nécessite un temps de mise sur le marché rapide et une expérimentation incessante, les organisations qui ne parviennent pas à reproduire ces résultats sont destinées à perdre sur le marché face à des concurrents plus agiles et pourraient même risquer de faire faillite, à l’instar des organisations manufacturières qui n’ont pas adopté les principes Lean.

De nos jours, quelle que soit l’industrie dans laquelle nous sommes en concurrence, la manière dont nous acquérons des clients et leur livrons de la valeur dépend du flux de valeur technologique. Pour le dire encore plus simplement, comme l’a déclaré Jeffrey Immelt, PDG de General Electric : "Toute industrie et toute entreprise qui n’intègrent pas le logiciel au cœur de leur activité seront perturbées". Ou comme l’a dit Jeffrey Snover, Fellow Technique chez Microsoft : "Dans les ères économiques précédentes, les entreprises créaient de la valeur en déplaçant des atomes. Maintenant, elles créent de la valeur en déplaçant des bits."

Il est difficile de surestimer l'ampleur de ce problème : il affecte chaque organisation, quelle que soit l'industrie dans laquelle nous opérons, la taille de notre organisation, que nous soyons à but lucratif ou non lucratif. Plus que jamais, la manière dont le travail technologique est géré et exécuté prédit si nos organisations gagneront sur le marché, voire survivront. Dans de nombreux cas, nous devrons adopter des principes et des pratiques qui semblent très différentes de celles qui nous ont guidés avec succès au cours des dernières décennies. Voir Annexe 1.

Maintenant que nous avons établi l'urgence du problème résolu par DevOps, prenons un moment pour explorer plus en détail la symptomatologie du problème, pourquoi il se produit, et pourquoi, sans intervention dramatique, le problème s'aggrave avec le temps.

Le problème : Quelque chose dans votre organisation doit nécessairement être amélioré (sinon vous ne lirez pas ce livre)

La plupart des organisations ne peuvent pas déployer des changements en production en quelques minutes ou heures, mais nécessitent plutôt des semaines ou des mois. Elles ne sont pas non plus capables de déployer des centaines ou des milliers de changements en production par jour ; au lieu de cela, elles luttent pour effectuer des déploiements mensuels voire trimestriels. De plus, les déploiements en production ne sont pas routiniers, impliquant des pannes et des situations chroniques de gestion de crises et d'héroïsme.

À une époque où l'avantage concurrentiel nécessite une rapidité de mise sur le marché, des niveaux de service élevés et une expérimentation incessante, ces organisations sont à un désavantage concurrentiel significatif. Cela est en grande partie dû à leur incapacité à résoudre un conflit fondamental et chronique au sein de leur organisation technologique.

Le conflit fondamental et chronique

Dans presque toutes les organisations informatiques, il existe un conflit inhérent entre le Développement et les Opérations IT, créant une spirale descendante, aboutissant à des délais de mise sur le marché de plus en plus longs pour les nouveaux produits et fonctionnalités, une qualité réduite, des pannes accrues et, pire encore, une dette technique croissante.

Le terme "dette technique" a été inventé par Ward Cunningham. Analogique à la dette financière, la dette technique décrit comment les décisions que nous prenons mènent à des problèmes de plus en plus difficiles à résoudre avec le temps, réduisant continuellement nos options disponibles à l'avenir - même lorsque cette dette est contractée judicieusement, nous en subissons les intérêts.

Un facteur qui contribue à cela est les objectifs souvent concurrents du Développement et des Opérations IT. Les organisations IT sont responsables de nombreuses choses, parmi lesquelles les deux objectifs suivants, qui doivent être poursuivis simultanément :

- Répondre à un paysage concurrentiel en évolution rapide
- Fournir un service stable, fiable et sécurisé aux clients

Fréquemment, le Développement sera responsable de répondre aux changements du marché, en déployant des fonctionnalités et des changements en production aussi rapidement que possible. Les Opérations IT seront responsables de fournir aux clients un service IT stable, fiable et sécurisé, rendant difficile voire impossible pour quiconque d'introduire des changements en production qui pourraient compromettre cette stabilité.

Configurées de cette manière, le Développement et les Opérations IT ont des objectifs et des incitations diamétralement opposés.

Le Dr. Eliyahu M. Goldratt, l'un des fondateurs du mouvement de gestion de la fabrication, a appelé ce type de configuration "le conflit fondamental et chronique" - lorsque les mesures et les incitations organisationnelles à travers différents silos empêchent l'atteinte des objectifs globaux de l'organisation.

Ce conflit crée une spirale descendante si puissante qu'elle empêche l'atteinte des résultats commerciaux souhaités, tant à l'intérieur qu'à l'extérieur de l'organisation IT. Ces conflits chroniques placent souvent les travailleurs technologiques dans des situations qui mènent à une qualité logicielle et de service médiocre, et à de mauvais résultats pour les clients, ainsi qu'à un besoin quotidien de contournements, de gestion de crises et d'héroïsme, que ce soit dans la gestion des produits, le développement, la QA, les opérations IT ou la sécurité de l'information.

La Spirale descendante en trois actes

La spirale descendante dans l'IT se déroule en trois actes qui sont probablement familiers à la plupart des praticiens de l'IT.

Premier Acte : La situation dans les Opérations IT

Le premier acte commence dans les Opérations IT, où notre objectif est de maintenir les applications et les infrastructures en fonctionnement afin que notre organisation puisse fournir de la valeur aux clients. Dans notre travail quotidien, beaucoup de nos problèmes sont dus à des applications et des infrastructures complexes, mal documentées et incroyablement fragiles. C'est la dette technique et les solutions de contournement quotidiennes avec

lesquelles nous vivons constamment, toujours promettant que nous allons réparer le désordre lorsque nous aurons un peu plus de temps. Mais ce temps ne vient jamais.

De manière alarmante, nos artefacts les plus fragiles supportent soit nos systèmes générateurs de revenus les plus importants, soit nos projets les plus critiques. En d'autres termes, les systèmes les plus sujets à l'échec sont également les plus importants et se trouvent au centre de nos changements les plus urgents. Lorsque ces changements échouent, ils mettent en péril nos promesses organisationnelles les plus importantes, telles que la disponibilité pour les clients, les objectifs de revenus, la sécurité des données des clients, la précision des rapports financiers, etc.

Deuxième Acte : La compensation pour les promesses non tenues

Le deuxième acte commence lorsque quelqu'un doit compenser la dernière promesse non tenue. Cela pourrait être un chef de produit promettant une fonctionnalité plus grande et plus audacieuse pour éblouir les clients ou un dirigeant d'entreprise fixant un objectif de revenu encore plus élevé. Ensuite, ignorant ce que la technologie peut ou ne peut pas faire, ou quels facteurs ont conduit à manquer notre engagement précédent, ils engagent l'organisation technologique à tenir cette nouvelle promesse.

En conséquence, le Développement est chargé d'un autre projet urgent qui nécessite inévitablement de résoudre de nouveaux défis techniques et de faire des compromis pour respecter la date de livraison promise, augmentant ainsi notre dette technique - bien sûr, avec la promesse que nous allons réparer tous les problèmes résultants lorsque nous aurons un peu plus de temps.

Troisième Acte : L'accroissement des difficultés

Cela prépare le terrain pour le troisième et dernier acte, où tout devient un peu plus difficile, petit à petit - tout le monde devient un peu plus occupé, le travail prend un peu plus de temps, les communications deviennent un peu plus lentes, et les files d'attente de travail s'allongent un peu plus. Notre travail devient plus étroitement couplé, les petites actions provoquent de plus grandes défaillances, et nous devenons plus craintifs et moins tolérants aux changements. Le travail nécessite plus de communication, de coordination et d'approbations ; les équipes doivent attendre un peu plus longtemps pour que leur travail dépendant soit terminé ; et notre qualité continue de se détériorer. Les rouages commencent à tourner plus lentement et nécessitent plus d'efforts pour continuer à tourner.

Bien qu'il soit difficile de le voir sur le moment, la spirale descendante est évidente lorsque l'on prend du recul. Nous remarquons que les déploiements de code en production prennent de plus en plus de temps, passant de minutes à des heures, puis à des jours et des semaines. Et pire, les résultats des déploiements deviennent encore plus problématiques, entraînant un nombre croissant de pannes affectant les clients et nécessitant plus d'héroïsme et de gestion

de crises dans les Opérations, les privant encore davantage de leur capacité à rembourser la dette technique.

En conséquence, nos cycles de livraison de produits continuent de ralentir, moins de projets sont entrepris, et ceux qui le sont, sont moins ambitieux. De plus, les retours sur le travail de chacun deviennent plus lents et plus faibles, surtout les signaux de retour de nos clients. Et, peu importe ce que nous essayons, les choses semblent empirer. Nous ne sommes plus en mesure de répondre rapidement à notre paysage concurrentiel changeant, ni de fournir un service stable et fiable à nos clients. En conséquence, nous perdons finalement sur le marché.

Maintes et maintes fois, nous apprenons que lorsque l'IT échoue, toute l'organisation échoue. Comme Steven J. Spear l'a noté dans son livre *The High-Velocity Edge*, que les dégâts "se déroulent lentement comme une maladie qui s'étend" ou rapidement "comme un crash enflammé... la destruction peut être tout aussi complète."

Pourquoi cette spirale descendante se produit partout

Depuis plus d'une décennie, les auteurs de ce livre ont observé cette spirale destructrice se produire dans d'innombrables organisations de tous types et tailles. Nous comprenons mieux que jamais pourquoi cette spirale descendante se produit et pourquoi elle nécessite des principes DevOps pour être atténuée. Premièrement, comme décrit plus tôt, chaque organisation IT a deux objectifs opposés, et deuxièmement, chaque entreprise est une entreprise technologique, qu'elle le sache ou non.

Comme l'a dit Christopher Little, un cadre de logiciel et l'un des premiers chroniqueurs du DevOps, "Chaque entreprise est une entreprise technologique, peu importe le secteur dans lequel elle pense se trouver. Une banque est juste une entreprise IT avec une licence bancaire."

Pour nous convaincre que c'est le cas, considérons que la grande majorité des projets d'investissement ont une dépendance vis-à-vis de l'IT. Comme le dit l'adage, "Il est pratiquement impossible de prendre une décision commerciale qui ne se traduit pas par au moins un changement IT."

Dans le contexte des affaires et des finances, les projets sont essentiels car ils servent de principal mécanisme de changement au sein des organisations. Les projets sont généralement ce que la direction doit approuver, budgétiser et être tenue pour responsable ; par conséquent, ils sont le mécanisme qui permet d'atteindre les objectifs et les aspirations de l'organisation, que ce soit pour croître ou même rétrécir.

Les projets sont généralement financés par des dépenses en capital (c'est-à-dire des usines, des équipements et des projets majeurs, et les dépenses sont capitalisées lorsque le retour sur

investissement est attendu sur des années), dont 50 % sont désormais liés à la technologie. Cela est même vrai dans les secteurs dits "low tech" avec les dépenses historiques les plus faibles en technologie, tels que l'énergie, le métal, l'extraction de ressources, l'automobile et la construction. En d'autres termes, les dirigeants d'entreprises sont bien plus dépendants de la gestion efficace de l'IT pour atteindre leurs objectifs qu'ils ne le pensent.

Les coûts : humains et économiques

Lorsque les personnes sont piégées dans cette spirale descendante pendant des années, en particulier celles qui se trouvent en aval du développement, elles se sentent souvent coincées dans un système qui pré determine l'échec et les laisse impuissantes face aux résultats. Cette impuissance est souvent suivie d'un épuisement professionnel, avec des sentiments associés de fatigue, de cynisme, et même de désespoir.

De nombreux psychologues affirment que créer des systèmes qui engendrent des sentiments d'impuissance est l'une des choses les plus dommageables que nous puissions faire à nos semblables : nous privons les autres de leur capacité à contrôler leurs propres résultats et créons même une culture où les gens craignent de faire la bonne chose par crainte de punition, d'échec ou de mettre en péril leur subsistance. Cela peut créer les conditions d'une impuissance apprise, où les gens deviennent réticents ou incapables d'agir pour éviter le même problème à l'avenir.

Pour nos employés, cela signifie de longues heures, du travail le week-end et une diminution de la qualité de vie, non seulement pour l'employé, mais pour tous ceux qui dépendent de lui, y compris la famille et les amis. Il n'est pas surprenant que, lorsque cela se produit, nous perdions nos meilleurs éléments (sauf ceux qui se sentent obligés de rester par sens du devoir ou d'obligation).

En plus de la souffrance humaine associée à la manière actuelle de travailler, le coût d'opportunité de la valeur que nous pourrions créer est stupéfiant : les auteurs estiment que nous manquons environ 2,6 trillions de dollars de création de valeur par an, ce qui équivaut, au moment de la rédaction de ce texte, à la production économique annuelle de la France, la sixième économie mondiale.

Considérons le calcul suivant : IDC et Gartner ont estimé qu'en 2011, environ 5 % du produit intérieur brut mondial (3,1 trillions de dollars) étaient dépensés en IT (matériel, services et télécommunications). Si nous estimons que 50 % de ces 3,1 trillions de dollars étaient consacrés aux coûts d'exploitation et à la maintenance des systèmes existants, et qu'un tiers de ces 50 % était dépensé en travail urgent et non planifié ou en refactorisation, environ 520 milliards de dollars étaient gaspillés.

Si l'adoption de DevOps nous permettait, par une meilleure gestion et une excellence opérationnelle accrue, de réduire de moitié ce gaspillage et de redéployer ce potentiel humain dans quelque chose qui vaut cinq fois la valeur (une proposition modeste), nous pourrions créer 2,6 trillions de dollars de valeur par an.

L'Éthique de DevOps : Il existe une meilleure façon

Dans les sections précédentes, nous avons décrit les problèmes et les conséquences négatives du statu quo en raison du conflit chronique de base, de l'incapacité à atteindre les objectifs organisationnels, aux dommages que nous infligeons à nos semblables. En résolvant ces problèmes, DevOps permet de manière étonnante d'améliorer simultanément les performances organisationnelles, d'atteindre les objectifs de tous les rôles technologiques fonctionnels (Développement, QA, IT Operations, Infosec), et d'améliorer la condition humaine.

Cette combinaison excitante et rare peut expliquer pourquoi DevOps a suscité tant d'enthousiasme en si peu de temps, y compris chez les leaders technologiques, les ingénieurs et une grande partie de l'écosystème logiciel dans lequel nous évoluons.

Briser la spirale descendante avec DevOps

Idéalement, de petites équipes de développeurs mettent en œuvre leurs fonctionnalités de manière indépendante, valident leur exactitude dans des environnements de type production, et leur code est déployé en production rapidement, en toute sécurité et de manière sécurisée. Les déploiements de code sont routiniers et prévisibles. Au lieu de commencer les déploiements à minuit le vendredi et de passer tout le week-end à les terminer, les déploiements se font pendant la journée de travail, quand tout le monde est déjà au bureau, et sans que nos clients ne s'en rendent compte - sauf lorsqu'ils voient de nouvelles fonctionnalités et corrections de bogues qui les ravissent. Et, en déployant du code au milieu de la journée de travail, pour la première fois depuis des décennies, les opérations IT travaillent pendant les heures normales de bureau comme tout le monde.

En créant des boucles de rétroaction rapides à chaque étape du processus, tout le monde peut immédiatement voir les effets de ses actions. Chaque fois que des changements sont validés dans le contrôle de version, des tests automatisés rapides sont exécutés dans des environnements de type production, donnant une assurance continue que le code et les environnements fonctionnent comme prévu et sont toujours dans un état sécurisé et déployable.

Les tests automatisés aident les développeurs à découvrir rapidement leurs erreurs (généralement en quelques minutes), ce qui permet des corrections plus rapides ainsi qu'un véritable apprentissage - un apprentissage impossible lorsque les erreurs sont découvertes six mois plus tard lors des tests d'intégration, lorsque les souvenirs et le lien entre cause et effet ont longtemps disparu. Au lieu d'accumuler de la dette technique, les problèmes sont corrigés dès

qu'ils sont trouvés, mobilisant toute l'organisation si nécessaire, car les objectifs globaux l'emportent sur les objectifs locaux.

Une télémétrie de production omniprésente dans notre code et nos environnements de production garantit que les problèmes sont détectés et corrigés rapidement, confirmant que tout fonctionne comme prévu et que les clients obtiennent de la valeur du logiciel que nous créons.

Dans ce scénario, tout le monde se sent productif - l'architecture permet à de petites équipes de travailler en toute sécurité et de manière découpée du travail des autres équipes qui utilisent des plateformes en libre-service tirant parti de l'expérience collective des opérations et de la sécurité de l'information. Au lieu que tout le monde attende tout le temps, avec de grandes quantités de travail urgent tardif, les équipes travaillent de manière indépendante et productive en petits lots, livrant rapidement et fréquemment de nouvelles valeurs aux clients.

Même les lancements de produits et de fonctionnalités à haute visibilité deviennent routiniers en utilisant des techniques de lancement sombre. Bien avant la date de lancement, nous mettons tout le code nécessaire pour la fonctionnalité en production, invisible pour tout le monde sauf pour les employés internes et de petits cohortes d'utilisateurs réels, ce qui nous permet de tester et d'évoluer la fonctionnalité jusqu'à ce qu'elle atteigne l'objectif commercial souhaité.

Et, au lieu de lutter contre les incendies pendant des jours ou des semaines pour faire fonctionner la nouvelle fonctionnalité, nous changeons simplement un basculement de fonctionnalité ou un paramètre de configuration. Ce petit changement rend la nouvelle fonctionnalité visible à des segments de clients de plus en plus larges, se rétablissant automatiquement si quelque chose ne va pas. En conséquence, nos lancements sont contrôlés, prévisibles, réversibles et sans stress.

Ce ne sont pas seulement les lancements de fonctionnalités qui sont plus calmes - toutes sortes de problèmes sont détectés et corrigés tôt, lorsqu'ils sont plus petits, moins coûteux et plus faciles à corriger. À chaque correction, nous générerons également des apprentissages organisationnels, nous permettant de prévenir le problème de se reproduire et nous permettant de détecter et de corriger plus rapidement des problèmes similaires à l'avenir.

De plus, tout le monde apprend constamment, favorisant une culture basée sur les hypothèses où la méthode scientifique est utilisée pour garantir que rien n'est tenu pour acquis - nous ne faisons rien sans mesurer et traiter le développement de produits et l'amélioration des processus comme des expériences.

Parce que nous valorisons le temps de chacun, nous ne passons pas des années à construire des fonctionnalités que nos clients ne veulent pas, à déployer du code qui ne fonctionne pas, ou à corriger quelque chose qui n'est pas réellement la cause de notre problème.

Parce que nous nous soucions d'atteindre des objectifs, nous créons des équipes à long terme qui sont responsables de les atteindre. Au lieu d'équipes de projets où les développeurs sont réaffectés et redistribués après chaque lancement, ne recevant jamais de retours sur leur travail, nous gardons les équipes intactes afin qu'elles puissent continuer à itérer et à s'améliorer, utilisant ces apprentissages pour mieux atteindre leurs objectifs. Cela est également vrai pour les équipes de produits qui résolvent des problèmes pour nos clients externes, ainsi que pour nos équipes de plateformes internes qui aident les autres équipes à être plus productives, sûres et sécurisées.

Au lieu d'une culture de la peur, nous avons une culture de haute confiance et de collaboration, où les gens sont récompensés pour prendre des risques. Ils peuvent parler sans crainte des problèmes au lieu de les cacher ou de les mettre en arrière-plan - après tout, nous devons voir les problèmes pour les résoudre.

Lorsqu'un problème survient, nous menons des post-mortems sans blâme, non pas pour punir quelqu'un, mais pour mieux comprendre ce qui a causé l'incident et comment le prévenir à l'avenir. Ce rituel renforce notre culture d'apprentissage. Nous organisons également des conférences internes sur la technologie pour éléver nos compétences et garantir que tout le monde enseigne et apprend en permanence.

Parce que nous nous soucions de la qualité, nous injectons même des défauts dans notre environnement de production afin de comprendre comment notre système échoue de manière planifiée. Nous menons des exercices planifiés pour pratiquer les défaillances à grande échelle, tuant aléatoirement des processus et des serveurs de calcul en production, et injectant des latences réseau et autres actes néfastes pour nous assurer de devenir toujours plus résilients. En faisant cela, nous favorisons une meilleure résilience ainsi que l'apprentissage et l'amélioration organisationnels.

Dans ce monde, chacun possède son travail, quel que soit son rôle dans l'organisation technologique. Ils ont confiance que leur travail compte et contribue de manière significative aux objectifs organisationnels, prouvé par leur environnement de travail peu stressant et le succès de leur organisation sur le marché. Leur preuve est que l'organisation gagne effectivement sur le marché.

La valeur commerciale du DevOps

Nous avons des preuves décisives de la valeur commerciale du DevOps. De 2013 à 2016, dans le cadre du rapport annuel "State of DevOps" de Puppet Labs, auquel ont contribué les auteurs

Jez Humble et Gene Kim, nous avons collecté des données auprès de plus de vingt-cinq mille professionnels de la technologie, dans le but de mieux comprendre la santé et les habitudes des organisations à tous les stades de l'adoption du DevOps.

La première surprise révélée par ces données a été la différence de performance entre les organisations hautement performantes utilisant des pratiques DevOps et leurs homologues moins performantes dans les domaines suivants :

- Métriques de débit
- Déploiements de code et de changements (trente fois plus fréquents)
- Temps de déploiement du code et des changements (deux cents fois plus rapide)
- Métriques de fiabilité
- Déploiements en production (taux de succès des changements soixante fois plus élevé)
- Temps moyen de restauration du service (168 fois plus rapide)
- Métriques de performance organisationnelle
- Objectifs de productivité, de part de marché et de rentabilité (deux fois plus susceptibles d'être dépassés)
- Croissance de la capitalisation boursière (50% plus élevée sur trois ans)

En d'autres termes, les organisations hautement performantes étaient à la fois plus agiles et plus fiables, fournissant des preuves empiriques que le DevOps nous permet de surmonter le conflit chronique central. Les organisations hautement performantes déployaient du code trente fois plus fréquemment, et le temps nécessaire pour passer de "code validé" à "code opérationnel en production" était deux cents fois plus rapide : leurs temps de déploiement étaient mesurés en minutes ou en heures, tandis que les organisations moins performantes avaient des temps de déploiement mesurés en semaines, mois, voire trimestres.

De plus, les organisations hautement performantes étaient deux fois plus susceptibles de dépasser leurs objectifs de rentabilité, de part de marché et de productivité. Et, pour celles qui ont fourni un symbole boursier, nous avons constaté que leur croissance de la capitalisation boursière était 50% plus élevée sur trois ans. Elles avaient également une meilleure satisfaction au travail, des taux d'épuisement professionnel plus faibles et leurs employés étaient 2,2 fois plus susceptibles de recommander leur organisation à leurs amis comme un excellent lieu de travail.

Les organisations hautement performantes avaient également de meilleurs résultats en matière de sécurité de l'information. En intégrant les objectifs de sécurité à toutes les étapes des processus de développement et d'opérations, elles passaient 50% moins de temps à remédier aux problèmes de sécurité.

Le DevOps aide à échelonner la productivité des développeurs

Lorsque nous augmentons le nombre de développeurs, la productivité individuelle des développeurs diminue souvent de manière significative en raison des frais généraux de

communication, d'intégration et de test. Cela est mis en évidence dans le célèbre livre de Frederick Brook, *The Mythical Man-Month*, où il explique que lorsque les projets sont en retard, ajouter plus de développeurs non seulement diminue la productivité individuelle des développeurs, mais diminue également la productivité globale.

D'un autre côté, le DevOps nous montre que lorsque nous avons la bonne architecture, les bonnes pratiques techniques et les bonnes normes culturelles, de petites équipes de développeurs sont capables de développer, intégrer, tester et déployer rapidement, en toute sécurité et de manière indépendante des changements en production. Comme l'a observé Randy Shoup, ancien directeur de l'ingénierie chez Google, les grandes organisations utilisant DevOps "ont des milliers de développeurs, mais leur architecture et leurs pratiques permettent à de petites équipes d'être incroyablement productives, comme si elles étaient une startup."

Le rapport State of DevOps 2015 a examiné non seulement le nombre de "déploiements par jour", mais aussi le nombre de "déploiements par jour par développeur". Nous avons émis l'hypothèse que les organisations hautement performantes seraient capables d'échelonner leur nombre de déploiements à mesure que la taille des équipes augmentait.

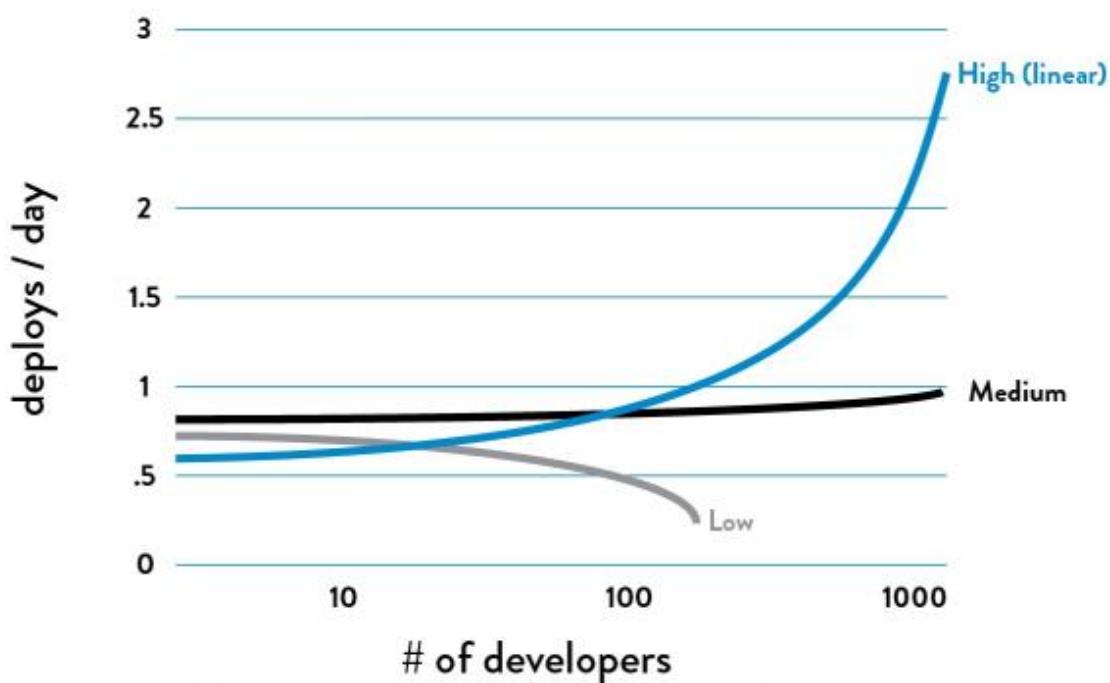


Figure 1. Deployments/day vs. number of developers (Source: Puppet Labs, 2015 State Of DevOps Report.)[#]

En effet, c'est ce que nous avons constaté. La figure 1 montre que chez les organisations à faible performance, le nombre de déploiements par jour par développeur diminue à mesure que la taille de l'équipe augmente, reste constant pour les organisations à performance moyenne et augmente linéairement pour les organisations à haute performance.

En d'autres termes, les organisations adoptant DevOps sont capables d'augmenter linéairement le nombre de déploiements par jour à mesure qu'elles augmentent le nombre de développeurs, tout comme l'ont fait Google, Amazon et Netflix.

L'universalité de la solution

L'un des livres les plus influents dans le mouvement Lean manufacturing est *The Goal: A Process of Ongoing Improvement*, écrit par le Dr Eliyahu M. Goldratt en 1984. Il a influencé toute une génération de directeurs d'usine professionnels dans le monde entier. Ce livre est un roman qui raconte l'histoire d'un directeur d'usine devant résoudre ses problèmes de coûts et de délais de production en quatre-vingt-dix jours, sous peine de voir son usine fermée. Plus tard dans sa carrière, le Dr Goldratt a décrit les lettres qu'il recevait en réponse à *The Goal*. Ces lettres commençaient souvent par : « Vous avez évidemment été caché dans notre usine, car vous avez décrit ma vie [en tant que directeur d'usine] exactement... » Plus important encore, ces lettres montraient que les gens étaient capables de reproduire les percées en matière de performance décrites dans le livre dans leur propre environnement de travail.

The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win, écrit par Gene Kim, Kevin Behr et George Spafford en 2013, a été étroitement modelé sur *The Goal*. Ce roman suit un leader informatique confronté à tous les problèmes typiques endémiques aux organisations informatiques : un projet dépassant le budget et en retard sur le calendrier qui doit être mis sur le marché pour que l'entreprise survive. Il rencontre des déploiements catastrophiques, des problèmes de disponibilité, de sécurité et de conformité, et ainsi de suite. Finalement, lui et son équipe utilisent les principes et les pratiques DevOps pour surmonter ces défis, aidant leur organisation à réussir sur le marché. De plus, le roman montre comment les pratiques DevOps ont amélioré l'environnement de travail pour l'équipe, créant moins de stress et une plus grande satisfaction grâce à une implication accrue des praticiens tout au long du processus.

Comme pour *The Goal*, il existe de nombreuses preuves de l'universalité des problèmes et des solutions décrites dans *The Phoenix Project*. Considérons certaines déclarations trouvées dans les critiques Amazon : « Je me retrouve dans les personnages de *The Phoenix Project*... J'ai probablement rencontré la plupart d'entre eux au cours de ma carrière », « Si vous avez déjà travaillé dans un aspect de l'informatique, du DevOps ou de la sécurité de l'information, vous pourrez certainement vous identifier à ce livre », ou « Il n'y a pas un seul personnage dans *The Phoenix Project* avec lequel je ne m'identifie pas moi-même ou que je ne connais pas dans la vie réelle... sans parler des problèmes auxquels ces personnages sont confrontés et qu'ils surmontent. »

Dans le reste de ce livre, nous décrirons comment reproduire la transformation décrite dans *The Phoenix Project*, et fournirons également de nombreuses études de cas sur la manière dont

d'autres organisations ont utilisé les principes et les pratiques DevOps pour reproduire ces résultats.

Le manuel DevOps : un guide essentiel

Le but du Manuel DevOps est de vous fournir la théorie, les principes et les pratiques nécessaires pour démarrer avec succès votre initiative DevOps et atteindre vos objectifs désirés. Ces conseils sont basés sur des décennies de théorie de gestion solide, l'étude d'organisations technologiques performantes, notre travail d'aide à la transformation des organisations, et des recherches qui valident l'efficacité des pratiques DevOps prescrites. Ainsi que des entretiens avec des experts pertinents dans le domaine et des analyses de près de cent études de cas présentées lors du sommet des entreprises DevOps.

Divisé en six parties, ce livre couvre les théories et les principes DevOps en utilisant les Trois Voies, une vision spécifique de la théorie sous-jacente initialement introduite dans The Phoenix Project. Le Manuel DevOps s'adresse à tous ceux qui réalisent ou influencent le travail dans le flux de valeur technologique (qui comprend généralement la Gestion de Produit, le Développement, l'Assurance Qualité, les Opérations IT et la Sécurité de l'Information), ainsi qu'aux dirigeants commerciaux et marketing, d'où proviennent la plupart des initiatives technologiques.

Le lecteur n'est pas censé avoir une connaissance approfondie de l'un de ces domaines, ni de DevOps, Agile, ITIL, Lean, ou de l'amélioration des processus. Chacun de ces sujets est introduit et expliqué dans le livre au fur et à mesure que cela devient nécessaire.

Notre intention est de créer une connaissance pratique des concepts essentiels dans chacun de ces domaines, à la fois pour servir de guide de base et pour introduire le langage nécessaire pour aider les praticiens à travailler avec tous leurs pairs à travers tout le flux de valeur IT, et pour encadrer des objectifs communs.

Ce livre sera utile aux dirigeants et aux parties prenantes de l'entreprise qui sont de plus en plus dépendants de l'organisation technologique pour la réalisation de leurs objectifs.

De plus, ce livre est destiné aux lecteurs dont les organisations pourraient ne pas rencontrer tous les problèmes décrits dans le livre (par exemple, les longs délais de déploiement ou les déploiements douloureux). Même les lecteurs dans cette position chanceuse bénéficieront de la compréhension des principes DevOps, en particulier ceux liés aux objectifs partagés, à la rétroaction et à l'apprentissage continu.

Dans la Partie I, nous présentons un bref historique du DevOps et introduisons la théorie sous-jacente ainsi que les thèmes clés des corpus de connaissances pertinents qui s'étendent sur des décennies. Nous présentons ensuite les principes de haut niveau des Trois Voies : Flux, Rétroaction, et Apprentissage et Expérimentation continus.

La Partie II décrit comment et où commencer, et présente des concepts tels que les flux de valeur, les principes et modèles de conception organisationnelle, les modèles d'adoption organisationnelle, et des études de cas.

La Partie III décrit comment accélérer le Flux en construisant les fondations de notre pipeline de déploiement : permettre des tests automatisés rapides et efficaces, une intégration continue, une livraison continue, et une architecture pour des déploiements à faible risque.

La Partie IV discute comment accélérer et amplifier la Rétroaction en créant une télémétrie de production efficace pour voir et résoudre les problèmes, mieux anticiper les problèmes et atteindre les objectifs, permettre la rétroaction pour que le Dev et les Ops puissent déployer les changements en toute sécurité, intégrer les tests A/B dans notre travail quotidien, et créer des processus de révision et de coordination pour augmenter la qualité de notre travail.

La Partie V décrit comment accélérer l'Apprentissage Continu en établissant une culture juste, en convertissant les découvertes locales en améliorations globales, et en réservant correctement du temps pour créer un apprentissage et des améliorations organisationnels.

Enfin, dans la Partie VI, nous décrivons comment intégrer correctement la sécurité et la conformité dans notre travail quotidien, en intégrant des contrôles de sécurité préventifs dans les référentiels de code source et les services partagés, en intégrant la sécurité dans notre pipeline de déploiement, en améliorant la télémétrie pour mieux détecter et récupérer, en protégeant le pipeline de déploiement, et en atteignant les objectifs de gestion des changements.

En codifiant ces pratiques, nous espérons accélérer l'adoption des pratiques DevOps, augmenter le succès des initiatives DevOps, et réduire l'énergie d'activation nécessaire pour les transformations DevOps.

Partie I - Les trois voies

Introduction

Dans la première partie du livre "The DevOps Handbook", nous explorerons comment la convergence de plusieurs mouvements importants en matière de gestion et de technologie a préparé le terrain pour le mouvement DevOps. Nous décrivons les flux de valeur, comment DevOps est le résultat de l'application des principes Lean au flux de valeur technologique, et les Trois Voies : le Flux, le Retour d'Information et l'Apprentissage et l'Expérimentation Continue.

Les points principaux abordés dans ces chapitres incluent :

Les principes du Flux, qui accélèrent la livraison du travail de Développement à Opérations à nos clients.

Les principes du Retour d'Information, qui nous permettent de créer des systèmes de travail toujours plus sûrs.

Les principes de l'Apprentissage et de l'Expérimentation Continue, qui favorisent une culture de haute confiance et une approche scientifique de l'amélioration organisationnelle en prenant des risques faisant partie de notre travail quotidien.

Une brève histoire

DevOps et ses pratiques techniques, architecturales et culturelles résultantes représentent une convergence de nombreux mouvements philosophiques et de gestion. Alors que de nombreuses organisations ont développé ces principes de manière indépendante, comprendre que DevOps résulte d'un large éventail de mouvements, un phénomène décrit par John Willis (l'un des co-auteurs de ce livre) comme la "convergence de DevOps", montre une progression de la pensée étonnante et des connexions improbables. Il y a des décennies d'expérience tirées de la fabrication, des organisations à haute fiabilité, des modèles de gestion à haute confiance, et d'autres qui nous ont amenés aux pratiques DevOps que nous connaissons aujourd'hui.

DevOps est le résultat de l'application des principes les plus éprouvés du domaine de la fabrication physique et du leadership au flux de valeur IT. DevOps s'appuie sur des corpus de connaissances issus de Lean, de la Théorie des Contraintes, du Système de Production Toyota, de l'ingénierie de la résilience, des organisations apprenantes, de la culture de la sécurité, des facteurs humains, et bien d'autres encore. D'autres contextes précieux dont DevOps s'inspire comprennent les cultures de gestion à haute confiance, le leadership servant, et la gestion du changement organisationnel. Le résultat est une qualité, une fiabilité, une stabilité et une sécurité de classe mondiale à un coût et un effort toujours plus bas ; et un flux et une fiabilité accélérés tout au long du flux de valeur technologique, y compris la Gestion de Produit, le Développement, l'Assurance Qualité, les Opérations IT, et la Sécurité Informatique.

Bien que les fondements de DevOps puissent être considérés comme dérivés du Lean, de la Théorie des Contraintes, et du mouvement Toyota Kata, beaucoup considèrent également DevOps comme la continuation logique du voyage Agile commencé en 2001.

Le mouvement Lean

Des techniques telles que la Cartographie des Flux de Valeur, les Tableaux Kanban, et la Maintenance Productive Totale ont été codifiées pour le Système de Production Toyota dans les années 1980. En 1997, le Lean Enterprise Institute a commencé à rechercher des applications du Lean à d'autres flux de valeur, tels que l'industrie des services et la santé.

Deux des principaux principes du Lean comprennent la croyance profonde que le temps de fabrication nécessaire pour convertir les matières premières en biens finis était le meilleur prédicteur de la qualité, de la satisfaction client, et du bonheur des employés, et que l'un des meilleurs prédicteurs de courts délais était les petites tailles de lots de travail.

Les principes Lean se concentrent sur la façon de créer de la valeur pour le client grâce à la pensée systémique en créant une constance de but, en adoptant une pensée scientifique, en créant un flux et un tirage (plutôt que de pousser), en assurant la qualité à la source, en menant avec humilité, et en respectant chaque individu.

Le manifeste Agile

Le Manifeste Agile a été créé en 2001 par dix-sept des principaux penseurs du développement logiciel. Ils voulaient créer un ensemble de valeurs et de principes légers par rapport aux processus lourds de développement logiciel tels que le développement en cascade, et les méthodologies telles que le Processus Unifié Rational.

Un principe clé était de "livrer fréquemment un logiciel opérationnel, de quelques semaines à quelques mois, avec une préférence pour les délais plus courts", mettant l'accent sur le désir de petites tailles de lots, de versions incrémentielles plutôt que de grandes versions en cascade. D'autres principes ont souligné la nécessité d'équipes petites et motivées, travaillant dans un modèle de gestion à haute confiance.

Agile est crédité pour avoir considérablement augmenté la productivité de nombreuses organisations de développement. Et curieusement, de nombreux moments clés de l'histoire de DevOps se sont également produits au sein de la communauté Agile ou lors de conférences Agile, comme décrit ci-dessous.

Mouvement de l'infrastructure Agile et de la vitesse

Lors de la conférence Agile de 2008 à Toronto, Canada, Patrick Debois et Andrew Schafer ont organisé une session "birds of a feather" sur l'application des principes Agile à l'infrastructure plutôt qu'au code d'application. Bien qu'ils aient été les seules personnes présentes, ils ont rapidement trouvé des adeptes aux idées similaires, dont le co-auteur John Willis.

Plus tard, lors de la conférence Velocity de 2009, John Allspaw et Paul Hammond ont présenté la présentation séminale "10 déploiements par jour : Coopération Dev et Ops chez Flickr", où ils

ont décrit comment ils ont créé des objectifs communs entre le Développement et les Opérations et utilisé des pratiques d'intégration continue pour faire du déploiement une partie du travail quotidien de chacun. Selon des témoignages directs, tous ceux présents à la présentation ont immédiatement compris qu'ils étaient en présence de quelque chose de profond et d'une importance historique.

Patrick Debois n'était pas présent, mais il était tellement excité par l'idée d'Allspaw et Hammond qu'il a créé le premier DevOpsDays à Gand, en Belgique (où il vivait) en 2009. C'est là que le terme "DevOps" a été créé.

Le mouvement de la livraison continue

S'appuyant sur la discipline du développement continu de la construction, du test et de l'intégration, Jez Humble et David Farley ont étendu le concept à la livraison continue, qui définissait le rôle d'un "pipeline de déploiement" pour garantir que le code et l'infrastructure soient toujours dans un état de déploiement, et que tout le code vérifié dans le tronc puisse être déployé en toute sécurité en production. Cette idée a été présentée pour la première fois lors de la conférence Agile de 2006, et a également été développée indépendamment en 2009 par Tim Fitz dans un article de blog sur son site intitulé "Déploiement continu".

Toyota Kata

En 2009, Mike Rother a écrit "Toyota Kata : Gérer les personnes pour l'amélioration, l'adaptabilité et des résultats supérieurs", qui a encadré son voyage de vingt ans pour comprendre et codifier le Système de Production Toyota. Il avait été l'un des étudiants diplômés qui avaient voyagé avec des cadres de GM pour visiter les usines de Toyota et avait aidé à développer la boîte à outils Lean, mais il était perplexe lorsque aucune des entreprises adoptant ces pratiques ne répliquait le niveau de performance observé dans les usines de Toyota.

Il a conclu que la communauté Lean avait raté la pratique la plus importante de toutes, qu'il appelait le kata d'amélioration. Il explique que chaque organisation a des routines de travail, et le kata d'amélioration exige de créer une structure pour la pratique quotidienne et habituelle du travail d'amélioration, car c'est la pratique quotidienne qui améliore les résultats. Le cycle constant d'établissement d'états futurs désirés, de définition d'objectifs hebdomadaires et l'amélioration continue du travail quotidien est ce qui a guidé l'amélioration chez Toyota.

Ce qui précède décrit l'histoire de DevOps et des mouvements pertinents sur lesquels il s'appuie. Tout au long du reste de la Partie I, nous examinerons les flux de valeur, comment les principes Lean peuvent être appliqués au flux de valeur technologique, et les Trois Voies du Flux, du Retour d'Information, et de l'Apprentissage et de l'Expérimentation Continuels.

Agile, livraison continue et les trois voies

Dans ce chapitre, une introduction à la théorie sous-jacente du Lean Manufacturing est présentée, ainsi que les Trois Voies, les principes à partir desquels tous les comportements DevOps observés peuvent être dérivés. Notre attention se porte principalement ici sur la théorie et les principes, décrivant de nombreuses décennies d'expérience tirée de la fabrication, des organisations à haute fiabilité, des modèles de gestion à haute confiance, et d'autres encore, à partir desquels les pratiques DevOps ont été dérivées. Les principes et modèles concrets résultants, ainsi que leur application pratique au flux de valeur technologique, sont présentés dans les chapitres restants du livre.

Le flux de valeur en fabrication

L'un des concepts fondamentaux du Lean est le flux de valeur. Nous le définirons d'abord dans le contexte de la fabrication, puis extrapolerons comment il s'applique à DevOps et au flux de valeur technologique. Karen Martin et Mike Osterling définissent le flux de valeur dans leur livre "Value Stream Mapping: How to Visualize Work and Align Leadership for Organizational Transformation" comme "la séquence d'activités qu'une organisation entreprend pour répondre à une demande du client", ou "la séquence d'activités nécessaires pour concevoir, produire et livrer un bien ou un service à un client, comprenant les flux d'informations et de matériaux".

Dans les opérations de fabrication, le flux de valeur est souvent facile à voir et à observer : il commence lorsque une commande client est reçue et que les matières premières sont libérées sur le sol de l'usine. Pour permettre des délais d'exécution rapides et prévisibles dans n'importe quel flux de valeur, il y a généralement un focus implacable sur la création d'un flux de travail lisse et régulier, en utilisant des techniques telles que les petites tailles de lots, la réduction du travail en cours (WIP), la prévention des retours pour s'assurer que nous ne transmettons pas de défauts aux centres de travail en aval, et l'optimisation constante de notre système vers nos objectifs globaux.

Le flux de valeur technologique

Les mêmes principes et modèles qui permettent le flux rapide du travail dans les processus physiques s'appliquent également de manière égale au travail technologique (et, pour tout dire, à tout travail de connaissance). En DevOps, nous définissons généralement notre flux de valeur technologique comme le processus requis pour convertir une hypothèse commerciale en un service technologique qui offre de la valeur au client.

L'entrée dans notre processus est la formulation d'un objectif commercial, d'un concept, d'une idée ou d'une hypothèse, et commence lorsque nous acceptons le travail en Développement, l'ajoutant à notre liste de travail engagée.

À partir de là, les équipes de développement qui suivent un processus Agile ou itératif typique transformeront probablement cette idée en histoires utilisateur et en une sorte de spécification

de fonctionnalités, qui seront ensuite mises en œuvre en code dans l'application ou le service en cours de construction. Le code est ensuite vérifié dans le référentiel de contrôle de version, où chaque changement est intégré et testé avec le reste du système logiciel.

Pour le reste de ce livre, notre attention se portera sur le délai de déploiement, un sous-ensemble du flux de valeur décrit ci-dessus. Ce flux de valeur commence lorsque n'importe quel ingénieur dans notre flux de valeur (qui comprend Développement, Assurance Qualité, Opérations IT, et Infosec) enregistre un changement dans le contrôle de version et se termine lorsque ce changement fonctionne avec succès en production, fournissant de la valeur au client et générant des retours et des téléméasures utiles.

Définir le délai d'exécution par rapport au temps de traitement

Dans la communauté Lean, le délai d'exécution est l'une des deux mesures couramment utilisées pour mesurer la performance dans les flux de valeur, l'autre étant le temps de traitement (parfois appelé temps de manipulation ou temps de tâche). Alors que l'horloge de délai d'exécution démarre lorsque la demande est faite et se termine lorsque celle-ci est satisfaite, l'horloge de temps de traitement ne démarre que lorsque nous commençons le travail sur la demande du client - spécifiquement, elle exclut le temps pendant lequel le travail est en file d'attente, en attente d'être traité.

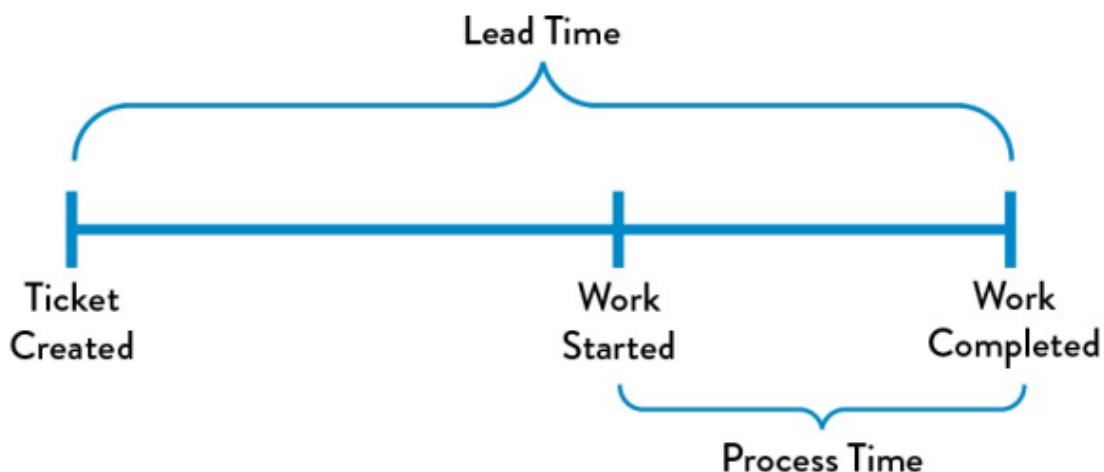


Figure 2. Lead time vs. process time of a deployment operation

Parce que le délai d'exécution est ce que le client ressent, nous concentrons généralement notre attention sur l'amélioration des processus là-bas plutôt que sur le temps de traitement. Cependant, la proportion de temps de traitement par rapport au délai d'exécution sert de mesure importante d'efficacité - obtenir un flux rapide et de courts délais d'exécution nécessite presque toujours de réduire le temps pendant lequel notre travail attend dans les files d'attente.

Le scénario courant : Des délais de déploiement requérant des mois

Dans la situation habituelle, nous nous retrouvons souvent dans des situations où nos délais de déploiement nécessitent des mois. C'est particulièrement courant dans les grandes organisations complexes qui travaillent avec des applications monolithiques étroitement couplées, souvent avec des environnements de test d'intégration rares, des délais de mise en production et de test longs, une forte dépendance aux tests manuels, et plusieurs processus d'approbation requis. Lorsque cela se produit, notre flux de valeur peut ressembler à la figure 3 :

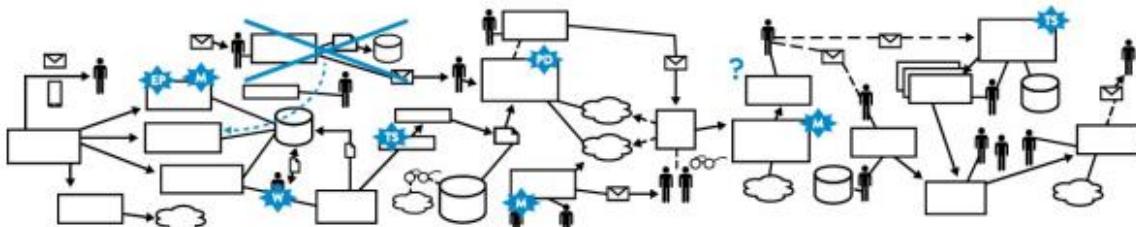


Figure 3: A technology value stream with a deployment lead time of three months
(Source: Damon Edwards, "DevOps Kaizen," 2015.)

Lorsque nous avons de longs délais de déploiement, des prouesses sont nécessaires à presque chaque étape du flux de valeur. Nous pouvons découvrir que rien ne fonctionne à la fin du projet lorsque nous fusionnons toutes les modifications de l'équipe de développement, ce qui entraîne un code qui ne se construit plus correctement ou ne passe aucun de nos tests. Résoudre chaque problème nécessite des jours ou des semaines d'investigation pour déterminer qui a cassé le code et comment il peut être réparé, et aboutit toujours à de mauvais résultats pour le client.

Notre idéal DevOps : des délais de déploiement de minutes

Dans l'idéal DevOps, les développeurs reçoivent rapidement et constamment des retours sur leur travail, ce qui leur permet de mettre en œuvre, d'intégrer et de valider rapidement et indépendamment leur code, et de le déployer dans l'environnement de production (soit en déployant le code eux-mêmes, soit par d'autres).

Nous atteignons cela en vérifiant continuellement de petits changements de code dans notre référentiel de contrôle de version, en effectuant des tests automatisés et exploratoires dessus, et en le déployant en production. Cela nous permet d'avoir un haut degré de confiance que nos changements fonctionneront comme prévu en production et que tout problème peut être rapidement détecté et corrigé.

Cela est le plus facilement réalisé lorsque nous avons une architecture modulaire, bien encapsulée et faiblement couplée, de sorte que de petites équipes puissent travailler avec un haut degré d'autonomie, les échecs étant petits et contenus, et sans causer de perturbations globales.

Dans ce scénario, notre délai de déploiement est mesuré en minutes, ou, dans le pire des cas, en heures. Notre carte de flux de valeur résultante devrait ressembler à quelque chose comme la figure 4 :

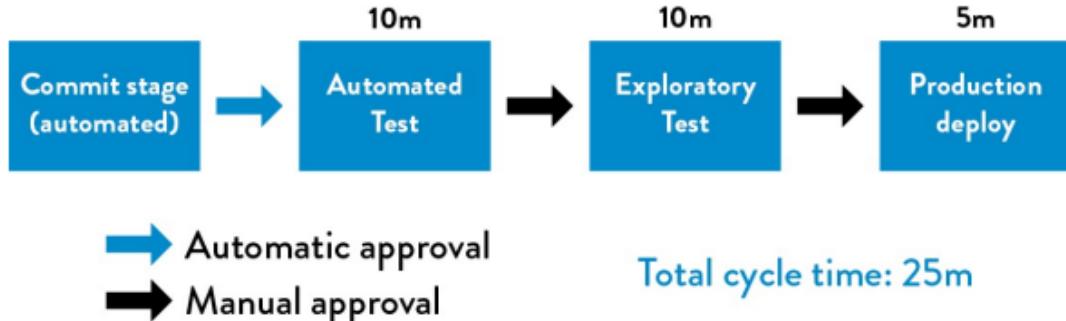


Figure 4: A technology value stream with a lead time of minutes

Observation du "%C/A" comme mesure de travail de retour

En plus des délais d'exécution et des temps de traitement, la troisième métrique clé dans le flux de valeur technologique est le pourcentage de complétude et d'exactitude (%C/A). Cette métrique reflète la qualité de la sortie de chaque étape de notre flux de valeur. Karen Martin et Mike Osterling affirment que "le %C/A peut être obtenu en demandant aux clients en aval quel pourcentage du temps ils reçoivent un travail 'utilisable tel quel', ce qui signifie qu'ils peuvent effectuer leur travail sans avoir à corriger les informations qui leur ont été fournies, ajouter des informations manquantes qui auraient dû être fournies, ou clarifier des informations qui auraient pu être plus claires."

Les trois voies : les principes sous-jacents du DevOps

Le Projet Phoenix présente les Trois Voies comme l'ensemble des principes sous-jacents à partir desquels tous les comportements et modèles DevOps observés sont dérivés (figure 5). La Première Voie permet un flux rapide de gauche à droite du travail du Développement aux Opérations jusqu'au client. Afin de maximiser le flux, nous devons rendre le travail visible, réduire nos tailles de lots et intervalles de travail, construire la qualité en empêchant les défauts d'être transmis aux centres de travail en aval, et optimiser constamment pour les objectifs globaux.

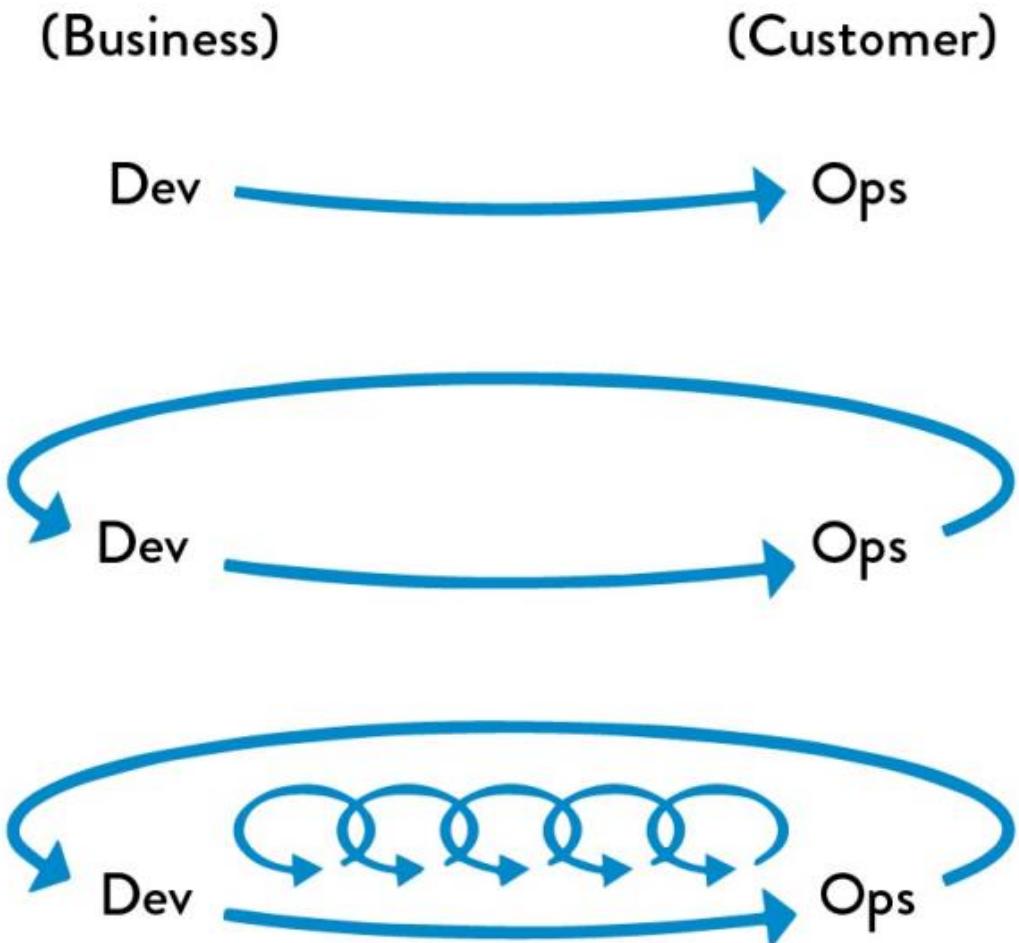


Figure 5: The Three Ways (Source: Gene Kim, “*The Three Ways: The Principles Underpinning DevOps*,” *IT Revolution Press blog*, accessed August 9, 2016, <http://itrevolution.com/the-three-ways-principles-underpinning-devops/>.)

En accélérant le flux à travers le flux de valeur technologique, nous réduisons le délai d'exécution nécessaire pour répondre aux demandes internes ou des clients, en particulier le temps nécessaire pour déployer du code dans l'environnement de production. En faisant cela, nous améliorons la qualité du travail ainsi que notre débit, et renforçons notre capacité à surpasser la concurrence par l'expérimentation.

Les pratiques qui en résultent comprennent les processus de construction, d'intégration, de test et de déploiement continus ; la création d'environnements à la demande ; la limitation du travail en cours (WIP) ; et la construction de systèmes et d'organisations qui sont sûrs à modifier.

La Deuxième Voie permet le flux rapide et constant des retours de droite à gauche à toutes les étapes de notre flux de valeur. Cela exige que nous amplifions les retours pour prévenir les problèmes de se reproduire, ou pour permettre une détection et une récupération plus rapides.

En faisant cela, nous créons de la qualité à la source et générerons ou intégrons des connaissances là où elles sont nécessaires - cela nous permet de créer des systèmes de travail de plus en plus sûrs où les problèmes sont trouvés et corrigés bien avant qu'une défaillance catastrophique ne se produise.

En voyant les problèmes au fur et à mesure qu'ils surviennent et en les traitant jusqu'à ce que des contre-mesures efficaces soient en place, nous raccourcissons et amplifions continuellement nos boucles de rétroaction, un principe de base de pratiquement toutes les méthodologies modernes d'amélioration des processus. Cela maximise les opportunités pour notre organisation d'apprendre et de s'améliorer.

La Troisième Voie permet la création d'une culture génératrice et de haute confiance qui soutient une approche dynamique, disciplinée et scientifique de l'expérimentation et de la prise de risque, facilitant la création d'un apprentissage organisationnel, à la fois à partir de nos succès et de nos échecs. De plus, en raccourcissant et en amplifiant continuellement nos boucles de rétroaction, nous créons des systèmes de travail de plus en plus sûrs et sommes mieux en mesure de prendre des risques et de réaliser des expériences qui nous aident à apprendre plus rapidement que notre concurrence et à réussir sur le marché.

Dans le cadre de la Troisième Voie, nous concevons également notre système de travail de manière à pouvoir multiplier les effets des nouvelles connaissances, transformant les découvertes locales en améliorations globales. Peu importe où quelqu'un effectue son travail, il le fait avec l'expérience cumulative et collective de tous dans l'organisation.

Conclusion

Dans ce chapitre, nous avons décrit les concepts de flux de valeur, le délai d'exécution comme l'une des mesures clés de l'efficacité tant pour les flux de valeur en fabrication que pour la technologie, et les concepts de haut niveau derrière chacune des Trois Voies, les principes qui sous-tendent DevOps.

Dans les chapitres suivants, les principes pour chacune des Trois Voies sont décrits plus en détail. Le premier de ces principes est le Flux, qui est axé sur la manière dont nous créons le flux rapide du travail dans n'importe quel flux de valeur, que ce soit dans la fabrication ou le travail technologique. Les pratiques qui permettent un flux rapide sont décrites dans la Partie III.

La première voie : Les principes du flux

Dans le flux de valeur technologique, le travail se déplace typiquement de Développement à Opérations, couvrant les zones fonctionnelles entre notre entreprise et nos clients. La Première Voie nécessite un flux rapide et fluide du travail du Développement aux Opérations, pour livrer de la valeur aux clients rapidement. Nous optimisons cet objectif global plutôt que des objectifs locaux, tels que les taux de compléction des fonctionnalités du Développement, les ratios de détection/correction des tests, ou les mesures de disponibilité des Opérations.

Nous augmentons le flux en rendant le travail visible, en réduisant les tailles de lots et les intervalles de travail, et en construisant la qualité dès le départ, en empêchant les défauts d'être transmis aux centres de travail en aval. En accélérant le flux à travers le flux de valeur technologique, nous réduisons le temps de traitement nécessaire pour répondre aux demandes des clients internes et externes, augmentant encore la qualité de notre travail tout en nous rendant plus agiles et capables de surpasser la concurrence par l'expérimentation.

Notre objectif est de réduire le temps nécessaire pour que les modifications soient déployées en production et d'augmenter la fiabilité et la qualité de ces services. Les indices sur la manière dont nous faisons cela dans le flux de valeur technologique peuvent être tirés de la manière dont les principes Lean ont été appliqués au flux de valeur de la fabrication.

Rendre notre travail visible

Une différence significative entre les flux de valeur technologique et de fabrication est que notre travail est invisible. Contrairement aux processus physiques, dans le flux de valeur technologique, nous ne pouvons pas facilement savoir où le flux est entravé ou quand le travail s'accumule devant des centres de travail contraints. Le transfert de travail entre les centres de travail est généralement très visible et lent car l'inventaire doit être déplacé physiquement. Cependant, dans le travail technologique, le déplacement peut être fait d'un simple clic, par exemple en réaffectant un ticket de travail à une autre équipe. Parce que c'est si facile, le travail peut rebondir entre les équipes sans fin en raison d'informations incomplètes, ou le travail peut être transmis aux centres de travail en aval avec des problèmes qui restent complètement invisibles jusqu'à ce que nous soyons en retard dans la livraison de ce que nous avons promis au client ou que notre application échoue en environnement de production.

Pour nous aider à voir où le travail circule bien et où il est en attente ou bloqué, nous devons rendre notre travail aussi visible que possible. L'une des meilleures méthodes pour y parvenir est d'utiliser des tableaux de travail visuels, tels que des tableaux kanban ou des tableaux de planification de sprint, où nous pouvons représenter le travail sur des cartes physiques ou électroniques. Le travail commence à gauche (souvent tiré d'un backlog), est tiré de centre de travail en centre de travail (représenté en colonnes), et se termine lorsqu'il atteint le côté droit du tableau, généralement dans une colonne étiquetée "fait" ou "en production".

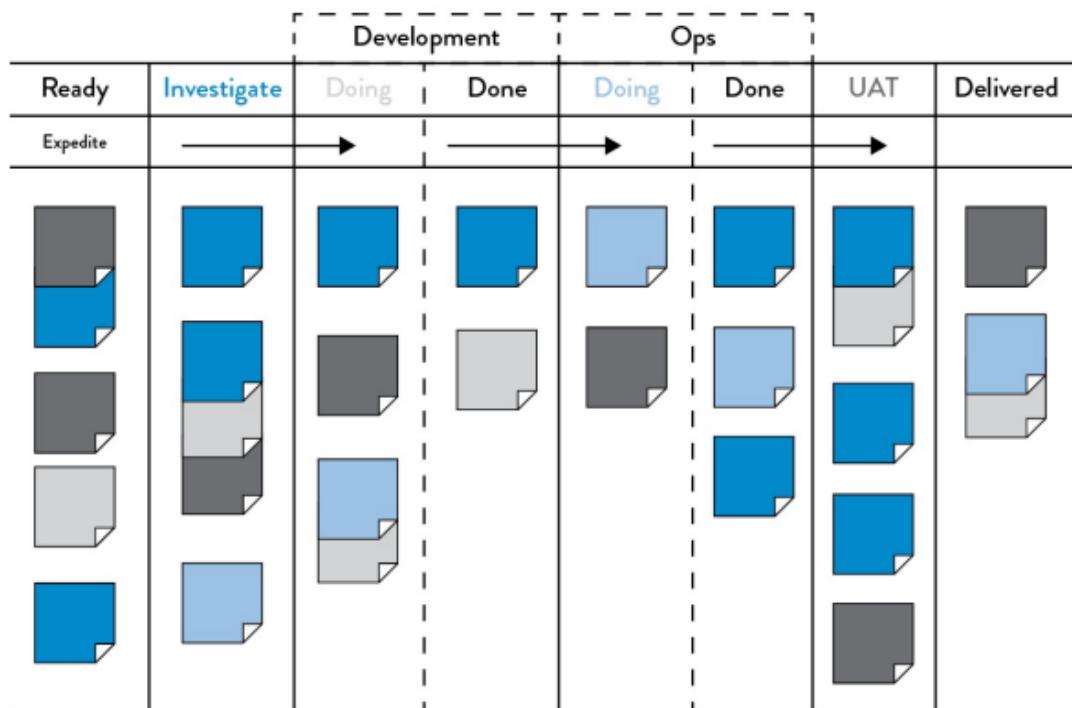


Figure 6: An example kanban board, spanning Requirements, Dev, Test, Staging, and In Production (Source: David J. Andersen and Dominica DeGrandis, Kanban for ITOps, training materials for workshop, 2012.)

Non seulement notre travail devient visible, mais nous pouvons également le gérer de manière qu'il circule de gauche à droite aussi rapidement que possible. De plus, nous pouvons mesurer le délai de traitement depuis le moment où une carte est placée sur le tableau jusqu'à ce qu'elle soit déplacée dans la colonne "Terminé".

Idéalement, notre tableau kanban couvrira l'ensemble du flux de valeur, définissant le travail comme terminé seulement lorsqu'il atteint le côté droit du tableau (figure 6). Le travail n'est pas terminé lorsque le développement complète l'implémentation d'une fonctionnalité, mais seulement lorsque notre application fonctionne avec succès en production, apportant de la valeur au client.

En mettant tout le travail de chaque centre de travail dans des files d'attente et en le rendant visible, tous les intervenants peuvent plus facilement prioriser le travail dans le contexte des objectifs globaux. Cela permet à chaque centre de travail de se concentrer sur le travail de la plus haute priorité jusqu'à ce qu'il soit terminé, augmentant ainsi le débit.

Limiter le travail en cours (WIP)

En fabrication, le travail quotidien est généralement dicté par un calendrier de production qui est généré régulièrement (par exemple, quotidiennement, hebdomadairement), établissant quelles tâches doivent être exécutées en fonction des commandes clients, des dates de livraison prévues, des pièces disponibles, etc.

En technologie, notre travail est généralement beaucoup plus dynamique, surtout dans les services partagés, où les équipes doivent satisfaire les demandes de nombreux intervenants différents. Par conséquent, le travail quotidien est souvent dominé par la priorité du jour, souvent avec des demandes de travail urgent venant par tous les moyens de communication possibles, y compris les systèmes de tickets, les appels de panne, les courriels, les appels téléphoniques, les salles de chat et les escalades de gestion.

Les interruptions dans la fabrication sont également très visibles et coûteuses, nécessitant souvent de casser la tâche actuelle et de mettre au rebut tout travail en cours incomplet pour commencer la nouvelle tâche. Ce niveau d'effort élevé décourage les interruptions fréquentes.

Cependant, interrompre les travailleurs en technologie est facile, car les conséquences sont invisibles pour presque tout le monde, même si l'impact négatif sur la productivité peut être beaucoup plus grand que dans la fabrication. Par exemple, un ingénieur affecté à plusieurs projets doit passer d'une tâche à l'autre, ce qui entraîne tous les coûts liés à la réinitialisation du contexte, ainsi que les règles et objectifs cognitifs.

Des études ont montré que le temps nécessaire pour accomplir même des tâches simples, comme trier des formes géométriques, se dégrade considérablement lors du multitâche. Évidemment, notre travail dans le flux de valeur technologique est beaucoup plus complexe que de trier des formes géométriques, donc les effets du multitâche sur le temps de traitement sont bien pires.

Nous pouvons limiter le multitâche en utilisant un tableau kanban pour gérer notre travail, par exemple en codifiant et en appliquant des limites de WIP (travail en cours) pour chaque colonne ou centre de travail, ce qui impose une limite supérieure au nombre de cartes pouvant être dans une colonne.

Par exemple, nous pouvons fixer une limite de WIP de trois cartes pour les tests. Lorsqu'il y a déjà trois cartes dans la colonne de test, aucune nouvelle carte ne peut être ajoutée à la colonne sauf si une carte est terminée ou retirée de la colonne "en cours" et remise en file d'attente (c'est-à-dire en remettant la carte dans la colonne de gauche). Rien ne peut être travaillé tant qu'il n'est pas représenté d'abord par une carte de travail, renforçant ainsi le fait que tout le travail doit être rendu visible.

Dominica DeGrandis, l'une des principales expertes dans l'utilisation des kanbans dans les flux de valeur DevOps, note que "contrôler la taille des files d'attente [WIP] est un outil de gestion extrêmement puissant, car c'est l'un des rares indicateurs avancés du délai de traitement - avec la plupart des éléments de travail, nous ne savons pas combien de temps cela prendra jusqu'à ce qu'ils soient effectivement terminés."

Limiter le WIP permet également de voir plus facilement les problèmes qui empêchent l'achèvement du travail.

Par exemple, lorsque nous limitons le WIP, nous constatons que nous n'avons peut-être rien à faire parce que nous attendons quelqu'un d'autre. Bien qu'il soit tentant de commencer un nouveau travail (c'est-à-dire "il vaut mieux faire quelque chose que rien"), une action bien meilleure serait de découvrir ce qui cause le retard et d'aider à résoudre ce problème.

Le mauvais multitâche se produit souvent lorsque les gens sont affectés à plusieurs projets, ce qui entraîne de nombreux problèmes de priorisation.

En d'autres termes, comme l'a dit David J. Andersen, auteur de "Kanban: Successful Evolutionary Change for Your Technology Business", "**arrêtez de commencer. Commencez à terminer.**"

Réduire les tailles de lot

Un autre composant clé pour créer un flux fluide et rapide est de réaliser le travail en petites tailles de lot. Avant la révolution Lean dans la fabrication, il était courant de fabriquer en grandes tailles de lot, surtout pour les opérations où la configuration des tâches ou le passage d'une tâche à une autre était long ou coûteux. Par exemple, produire de grands panneaux de carrosserie de voiture nécessite de placer de grands et lourds moules sur des machines de pressage de métal, un processus qui pouvait prendre des jours. Lorsque le coût de changement est si élevé, nous tamponnions souvent autant de panneaux que possible à la fois, créant de grands lots afin de réduire le nombre de changements.

Cependant, les grandes tailles de lot entraînent des niveaux de WIP qui montent en flèche et des niveaux élevés de variabilité dans le flux qui se répercutent sur l'ensemble de l'usine de fabrication. Le résultat est des délais de traitement longs et une mauvaise qualité - si un problème est trouvé dans un panneau de carrosserie, tout le lot doit être mis au rebut.

Une des leçons clés du Lean est que pour réduire les délais de traitement et augmenter la qualité, nous devons nous efforcer de réduire continuellement les tailles de lot. La limite

théorique inférieure pour la taille de lot est le flux en pièce unique, où chaque opération est réalisée une unité à la fois.

Les différences dramatiques entre les grandes et petites tailles de lot peuvent être vues dans la simple simulation de mailing de newsletter décrite dans *Lean Thinking: Banish Waste and Create Wealth in Your Corporation* de James P. Womack et Daniel T. Jones.

Supposons que dans notre propre exemple, nous ayons dix brochures à envoyer et que chaque envoi nécessite quatre étapes : plier le papier, insérer le papier dans l'enveloppe, sceller l'enveloppe et tamponner l'enveloppe.

La stratégie de grand lot (c'est-à-dire la "production de masse") consisterait à effectuer séquentiellement une opération sur chacune des dix brochures. En d'autres termes, nous plierions d'abord les dix feuilles de papier, puis nous insérerions chacune d'elles dans des enveloppes, puis nous scellerions les dix enveloppes, et enfin nous tamponnerions les dix enveloppes.

D'un autre côté, dans la stratégie de petit lot (c'est-à-dire le "flux en pièce unique"), toutes les étapes nécessaires pour compléter chaque brochure sont effectuées séquentiellement avant de commencer la suivante. En d'autres termes, nous plions une feuille de papier, l'insérons dans l'enveloppe, la scellons et la tamponnons - ce n'est qu'ensuite que nous recommençons le processus avec la feuille de papier suivante.

La différence entre l'utilisation de grandes et petites tailles de lot est dramatique (voir figure 7). Supposons que chacune des quatre opérations prenne dix secondes pour chacune des dix enveloppes. Avec la stratégie de grande taille de lot, la première enveloppe complétée et tamponnée est produite seulement après 310 secondes.

Pire encore, supposons que nous découvrions pendant l'opération de scellage des enveloppes que nous avons commis une erreur lors de la première étape de pliage - dans ce cas, le plus tôt où nous découvririons l'erreur serait à deux cents secondes, et nous devrions replier et réinsérer toutes les dix brochures de notre lot encore une fois.

Large Batches



Single-Piece Flow



Figure 7: Simulation of “envelope game” (fold, insert, seal, and stamp the envelope)

(Source: Stefan Luyten, “Single Piece Flow: Why mass production isn’t the most efficient way of doing ‘stuff’,” Medium.com, August 8, 2014, <https://medium.com/@stefanluyten/single-piece-flow-5d2c2bec845b#.9o7sn74ns.>)

En revanche, dans la stratégie des petits lots, la première enveloppe tamponnée complétée est produite en seulement quarante secondes, soit huit fois plus rapidement que la stratégie des grands lots. Et, si nous avons commis une erreur lors de la première étape, nous n'avons qu'à refaire une seule brochure dans notre lot. Les petites tailles de lot entraînent moins de travail en cours (WIP), des délais plus courts, une détection plus rapide des erreurs et moins de retouches.

Les résultats négatifs associés aux grandes tailles de lot sont tout aussi pertinents pour le flux de valeur technologique que pour la fabrication. Considérons le cas où nous avons un calendrier annuel pour les sorties de logiciels, où une année entière de code sur lequel le développement a travaillé est déployée en production.

Comme dans la fabrication, cette sortie en grand lot crée soudainement des niveaux élevés de WIP et des perturbations massives dans tous les centres de travail en aval, entraînant un mauvais flux et des résultats de mauvaise qualité. Cela confirme notre expérience courante selon laquelle plus le changement introduit en production est important, plus il est difficile de diagnostiquer et de corriger les erreurs de production, et plus il faut de temps pour les remédier.

Dans un article sur Startup Lessons Learned, Eric Ries déclare : « La taille du lot est l'unité à laquelle les produits de travail se déplacent entre les étapes d'un processus de développement [ou DevOps]. Pour les logiciels, le lot le plus facile à voir est le code. Chaque fois qu'un ingénieur valide du code, il regroupe une certaine quantité de travail. Il existe de nombreuses techniques pour contrôler ces lots, allant des petits lots nécessaires pour le déploiement continu aux

méthodes de développement plus traditionnelles basées sur des branches, où tout le code de plusieurs développeurs travaillant pendant des semaines ou des mois est regroupé et intégré ensemble. »

L'équivalent du flux pièce unique dans le flux de valeur technologique est réalisé avec le déploiement continu, où chaque changement validé dans le contrôle de version est intégré, testé et déployé en production. Les pratiques qui permettent cela sont décrites dans la partie IV.

Réduire le nombre de transmissions

Dans le flux de valeur technologique, chaque fois que nous avons des délais de déploiement longs, mesurés en mois, c'est souvent parce qu'il y a des centaines (voire des milliers) d'opérations nécessaires pour déplacer notre code depuis le contrôle de version jusqu'à l'environnement de production. Pour transmettre le code à travers le flux de valeur, plusieurs départements doivent travailler sur une variété de tâches, y compris les tests fonctionnels, les tests d'intégration, la création d'environnements, l'administration des serveurs, l'administration du stockage, la mise en réseau, l'équilibrage de charge et la sécurité de l'information.

Chaque fois que le travail passe d'une équipe à une autre, toutes sortes de communications sont nécessaires : demandes, spécifications, signalisations, coordinations, et souvent priorisations, planifications, désamorçages de conflits, tests et vérifications. Cela peut nécessiter l'utilisation de différents systèmes de gestion des tickets ou de gestion de projet ; la rédaction de documents de spécifications techniques ; la communication via des réunions, des courriels ou des appels téléphoniques ; et l'utilisation de partages de fichiers, de serveurs FTP et de pages Wiki.

Chacune de ces étapes est une file d'attente potentielle où le travail attend lorsque nous dépendons de ressources partagées entre différents flux de valeur (par exemple, les opérations centralisées). Les délais de réponse pour ces demandes sont souvent si longs qu'il y a une escalade constante pour que le travail soit effectué dans les délais nécessaires.

Même dans les meilleures circonstances, une partie des connaissances est inévitablement perdue à chaque transmission. Avec un nombre suffisant de transmissions, le travail peut complètement perdre le contexte du problème à résoudre ou l'objectif organisationnel soutenu. Par exemple, un administrateur de serveur peut voir un nouveau ticket demandant la création de comptes utilisateurs, sans savoir pour quelle application ou service c'est, pourquoi cela doit être créé, quelles sont toutes les dépendances, ou si c'est un travail récurrent.

Pour atténuer ces types de problèmes, nous nous efforçons de réduire le nombre de transmissions, soit en automatisant une grande partie du travail, soit en réorganisant les équipes afin qu'elles puissent elles-mêmes fournir de la valeur au client, au lieu de devoir

constamment dépendre des autres. En conséquence, nous augmentons le flux en réduisant le temps que notre travail passe en file d'attente, ainsi que le temps non ajoutant de la valeur.

Identifier et éliminer en continu nos contraintes

Pour réduire les délais de traitement et augmenter le débit, nous devons continuellement identifier les contraintes de notre système et améliorer sa capacité de travail. Dans "Beyond the Goal", le Dr. Goldratt déclare : "Dans tout flux de valeur, il y a toujours une direction de flux, et il n'y a toujours qu'une seule contrainte ; toute amélioration non faite à cette contrainte est une illusion." Si nous améliorons un centre de travail situé avant la contrainte, le travail s'accumulera encore plus rapidement au goulot d'étranglement, en attendant d'être traité par le centre de travail embouteillé.

D'un autre côté, si nous améliorons un centre de travail situé après le goulot d'étranglement, il restera affamé, en attendant que le travail passe le goulot d'étranglement. Comme solution, le Dr. Goldratt a défini les "cinq étapes de focalisation" :

- Identifier la contrainte du système.
- Décider comment exploiter la contrainte du système.
- Subordonner tout le reste aux décisions ci-dessus.
- Élever la contrainte du système.

Si lors des étapes précédentes une contrainte a été brisée, revenir à l'étape une, mais ne pas permettre à l'inertie de causer une contrainte du système.

Dans les transformations DevOps typiques, à mesure que nous passons de délais de déploiement mesurés en mois ou trimestres à des délais mesurés en minutes, la contrainte suit généralement cette progression :

Création d'environnement : Nous ne pouvons pas réaliser des déploiements à la demande si nous devons toujours attendre des semaines ou des mois pour des environnements de production ou de test. La contre-mesure consiste à créer des environnements à la demande et entièrement auto-gérés, afin qu'ils soient toujours disponibles lorsque nous en avons besoin.

Déploiement de code : Nous ne pouvons pas réaliser des déploiements à la demande si chacun de nos déploiements de code en production prend des semaines ou des mois à effectuer (c'est-à-dire que chaque déploiement nécessite 1 300 étapes manuelles et sujettes à des erreurs, impliquant jusqu'à trois cents ingénieurs). La contre-mesure consiste à automatiser nos déploiements autant que possible, avec pour objectif d'être complètement automatisés afin qu'ils puissent être effectués en libre-service par n'importe quel développeur.

Configuration et exécution des tests : Nous ne pouvons pas réaliser des déploiements à la demande si chaque déploiement de code nécessite deux semaines pour configurer nos environnements de test et ensembles de données, et encore quatre semaines pour exécuter

manuellement tous nos tests de régression. La contre-mesure consiste à automatiser nos tests afin de pouvoir exécuter les déploiements en toute sécurité et de les paralléliser afin que le taux de test puisse suivre notre rythme de développement de code.

Architecture trop contraignante : Nous ne pouvons pas réaliser des déploiements à la demande si une architecture trop contraignante signifie qu'à chaque fois que nous voulons faire un changement de code, nous devons envoyer nos ingénieurs à de nombreuses réunions de comité pour obtenir la permission de faire nos changements. Notre contre-mesure consiste à créer une architecture plus faiblement couplée afin que les changements puissent être effectués en toute sécurité et avec plus d'autonomie, augmentant ainsi la productivité des développeurs.

Une fois que toutes ces contraintes auront été brisées, notre contrainte sera probablement le développement ou les propriétaires de produit. Parce que notre objectif est de permettre à de petites équipes de développeurs de développer, tester et déployer indépendamment de la valeur pour les clients rapidement et de manière fiable, c'est là que nous voulons que notre contrainte soit. Les performants, qu'ils soient ingénieurs en développement, QA, Ops ou Infosec, déclarent que leur objectif est d'aider à maximiser la productivité des développeurs.

Lorsque la contrainte est ici, nous sommes limités uniquement par le nombre de bonnes hypothèses commerciales que nous créons et notre capacité à développer le code nécessaire pour tester ces hypothèses avec de vrais clients.

La progression des contraintes listée ci-dessus est une généralisation des transformations typiques - les techniques pour identifier la contrainte dans les flux de valeur réels, comme par le biais de la cartographie des flux de valeur et des mesures, sont décrites plus loin dans ce livre.

Éliminer les difficultés et le gaspillage dans le flux de valeur

Shigeo Shingo, l'un des pionniers du système de production Toyota, croyait que le gaspillage constituait la plus grande menace pour la viabilité des entreprises. La définition couramment utilisée dans Lean est "l'utilisation de tout matériau ou ressource au-delà de ce que le client nécessite et est prêt à payer". Il a défini sept types principaux de gaspillage dans la fabrication : l'inventaire, la surproduction, les traitements supplémentaires, le transport, l'attente, les mouvements et les défauts.

Des interprétations plus modernes du Lean ont noté que "l'élimination des gaspillages" peut avoir un contexte dévalorisant et déshumanisant ; au lieu de cela, l'objectif est reformulé pour réduire les difficultés et la monotonie dans notre travail quotidien grâce à un apprentissage continu afin d'atteindre les objectifs de l'organisation. Pour le reste de ce livre, le terme gaspillage impliquera cette définition plus moderne, car elle correspond plus étroitement aux idéaux et aux résultats souhaités de DevOps.

Dans le livre *Implementing Lean Software Development: From Concept to Cash*, Mary et Tom Poppendieck décrivent le gaspillage et les difficultés dans le flux de développement logiciel comme tout ce qui cause un retard pour le client, telles que des activités qui peuvent être contournées sans affecter le résultat.

Les catégories suivantes de gaspillage et de difficultés proviennent de *Implementing Lean Software Development* sauf indication contraire :

Travail partiellement fait : Cela inclut tout travail dans le flux de valeur qui n'a pas été complété (par exemple, des documents de requêtes ou des ordres de modification non encore examinés) et le travail en attente (par exemple, en attente de révision QA ou de ticket d'administration de serveur). Le travail partiellement fait devient obsolète et perd de la valeur avec le temps.

Processus supplémentaires : Tout travail supplémentaire effectué dans un processus qui n'ajoute pas de valeur au client. Cela peut inclure de la documentation non utilisée dans un centre de travail en aval, ou des examens ou approbations qui n'ajoutent pas de valeur au résultat. Les processus supplémentaires ajoutent des efforts et augmentent les délais.

Fonctionnalités supplémentaires : Des fonctionnalités intégrées au service qui ne sont pas nécessaires pour l'organisation ou le client (par exemple, "plaqué or"). Les fonctionnalités supplémentaires ajoutent de la complexité et des efforts pour tester et gérer la fonctionnalité.

Changement de tâches : Lorsque les personnes sont affectées à plusieurs projets et flux de valeur, les obligeant à changer de contexte et à gérer les dépendances entre les travaux, ajoutant des efforts et du temps supplémentaires dans le flux de valeur.

Attente : Tout retard entre les travaux nécessitant que les ressources attendent jusqu'à ce qu'elles puissent terminer le travail en cours. Les retards augmentent le temps de cycle et empêchent le client de recevoir de la valeur.

Mouvement : La quantité d'efforts nécessaires pour déplacer des informations ou des matériaux d'un centre de travail à un autre. Le gaspillage de mouvement peut être créé lorsque les personnes qui doivent communiquer fréquemment ne sont pas localisées ensemble. Les transmissions créent également du gaspillage de mouvement et nécessitent souvent une communication supplémentaire pour résoudre les ambiguïtés.

Défauts : Les informations, matériaux ou produits incorrects, manquants ou peu clairs créent du gaspillage, car des efforts sont nécessaires pour résoudre ces problèmes. Plus le temps entre la création du défaut et sa détection est long, plus il est difficile de résoudre le défaut.

Travail non standard ou manuel : La dépendance à l'égard du travail non standard ou manuel des autres, comme l'utilisation de serveurs non reconstruits, d'environnements de test et de configurations. Idéalement, toutes les dépendances sur les opérations devraient être automatisées, auto-gérées et disponibles à la demande.

Héroïsme : Pour qu'une organisation atteigne ses objectifs, des individus et des équipes sont mis dans une position où ils doivent accomplir des actes déraisonnables, qui peuvent même devenir une partie de leur travail quotidien (par exemple, des problèmes en production à 2 heures du matin, la création de centaines de tickets de travail dans le cadre de chaque version logicielle).

Notre objectif est de rendre ces gaspillages et difficultés visibles, et de faire systématiquement ce qui est nécessaire pour alléger ou éliminer ces fardeaux et difficultés afin d'atteindre notre objectif de flux rapide.

Conclusion

Améliorer le flux dans le flux de valeur technologique est essentiel pour atteindre les résultats de DevOps. Nous faisons cela en rendant le travail visible, en limitant le WIP, en réduisant les tailles de lot et le nombre de transmissions, en identifiant et en évaluant continuellement nos contraintes, et en éliminant les difficultés dans notre travail quotidien.

Les pratiques spécifiques qui permettent un flux rapide dans le flux de valeur DevOps sont présentées dans la Partie IV. Dans le prochain chapitre, nous présentons La Deuxième Voie : Les Principes de Feedback.

La deuxième voie : Les principes de retour de l'information

Tandis que la Première Voie décrit les principes permettant un flux de travail rapide de gauche à droite, la Deuxième Voie décrit les principes permettant un retour d'information rapide et constant de droite à gauche à toutes les étapes du flux de valeur. Notre objectif est de créer un système de travail toujours plus sûr et résilient.

Ceci est particulièrement important lorsque l'on travaille dans des systèmes complexes, où la première occasion de détecter et de corriger les erreurs se présente souvent lorsqu'un événement catastrophique est en cours, comme un accident de travail ou une fusion d'un réacteur nucléaire.

Dans le domaine de la technologie, notre travail se déroule presque entièrement dans des systèmes complexes avec un risque élevé de conséquences catastrophiques. Comme dans la fabrication, nous découvrons souvent des problèmes uniquement lorsque de grandes défaillances sont en cours, comme une panne de production massive ou une violation de la sécurité entraînant le vol de données clients.

Nous rendons notre système de travail plus sûr en créant un flux d'information rapide, fréquent et de haute qualité à travers notre flux de valeur et notre organisation, ce qui inclut des boucles de rétroaction et d'anticipation. Cela nous permet de détecter et de remédier aux problèmes lorsqu'ils sont plus petits, moins coûteux et plus faciles à résoudre ; d'éviter les problèmes avant qu'ils ne causent des catastrophes ; et de créer un apprentissage organisationnel que nous intégrons dans les travaux futurs. Lorsque des défaillances et des accidents surviennent, nous les considérons comme des occasions d'apprentissage, plutôt que comme des causes de punition et de blâme. Pour atteindre tout cela, explorons d'abord la nature des systèmes complexes et comment ils peuvent être rendus plus sûrs.

Travailler en toute sécurité dans les systèmes complexes

Une des caractéristiques définissant un système complexe est qu'il échappe à la capacité de toute personne à voir le système dans son ensemble et à comprendre comment toutes les pièces s'assemblent. Les systèmes complexes ont généralement un degré élevé d'interconnexion de composants étroitement couplés, et le comportement au niveau du système ne peut pas être expliqué simplement en termes de comportement des composants du système.

Le Dr Charles Perrow a étudié la crise de Three Mile Island et a observé qu'il était impossible pour quiconque de comprendre comment le réacteur se comporterait dans toutes les circonstances et comment il pourrait échouer. Lorsqu'un problème se produisait dans un

composant, il était difficile de l'isoler des autres composants, s'écoulant rapidement par les chemins de moindre résistance de manière imprévisible.

Le Dr Sidney Dekker, qui a également codifié certains des éléments clés de la culture de sécurité, a observé une autre caractéristique des systèmes complexes : faire la même chose deux fois ne mènera pas nécessairement ou de manière prévisible au même résultat. C'est cette caractéristique qui rend les listes de contrôle statiques et les meilleures pratiques, bien que précieuses, insuffisantes pour prévenir les catastrophes.

Par conséquent, puisque l'échec est inhérent et inévitable dans les systèmes complexes, nous devons concevoir un système de travail sûr, que ce soit dans la fabrication ou la technologie, où nous pouvons travailler sans peur, en étant confiants que toute erreur sera détectée rapidement, bien avant qu'elle ne cause des résultats catastrophiques, tels que des blessures aux travailleurs, des défauts de produit ou des impacts négatifs sur les clients.

Après avoir déchiffré le mécanisme causal derrière le Système de Production Toyota dans le cadre de sa thèse doctorale à la Harvard Business School, le Dr Steven Spear a déclaré que concevoir des systèmes parfaitement sûrs est probablement au-delà de nos capacités, mais nous pouvons rendre plus sûr le travail dans des systèmes complexes lorsque les quatre conditions suivantes sont remplies :

- Le travail complexe est géré de manière à révéler les problèmes dans la conception et les opérations.
- Les problèmes sont rapidement entourés et résolus, aboutissant à la construction rapide de nouvelles connaissances
- Les nouvelles connaissances locales sont exploitées globalement dans toute l'organisation
- Les dirigeants créent d'autres dirigeants qui développent continuellement ces types de capacités

Chacune de ces capacités est nécessaire pour travailler en toute sécurité dans un système complexe. Dans les sections suivantes, les deux premières capacités et leur importance sont décrites, ainsi que la manière dont elles ont été créées dans d'autres domaines et quelles pratiques les permettent dans le flux de valeur technologique. (Les troisième et quatrième capacités sont décrites au chapitre 4.)

Voir les problèmes lorsqu'ils se produisent

Dans un système de travail sûr, nous devons constamment tester nos hypothèses de conception et d'exploitation. Notre objectif est d'augmenter le flux d'information dans notre système provenant de nombreuses zones le plus tôt, le plus rapidement, le moins coûteusement et avec autant de clarté entre la cause et l'effet que possible. Plus nous pouvons

invalider d'hypothèses, plus nous pouvons trouver et résoudre rapidement les problèmes, augmentant notre résilience, notre agilité et notre capacité à apprendre et à innover.

Nous faisons cela en créant des boucles de rétroaction et d'anticipation dans notre système de travail. Le Dr Peter Senge, dans son livre *The Fifth Discipline: The Art & Practice of the Learning Organization*, a décrit les boucles de rétroaction comme une partie critique des organisations apprenantes et de la pensée systémique. Les boucles de rétroaction et d'anticipation font que les composants au sein d'un système se renforcent ou se contrebalancent mutuellement.

Dans la fabrication, l'absence de rétroaction efficace contribue souvent à des problèmes majeurs de qualité et de sécurité. Dans un cas bien documenté à l'usine de fabrication de General Motors à Fremont, il n'y avait pas de procédures efficaces en place pour détecter les problèmes pendant le processus d'assemblage, ni de procédures explicites sur ce qu'il fallait faire lorsque des problèmes étaient trouvés. En conséquence, il y a eu des instances où des moteurs étaient installés à l'envers, des voitures manquaient de volant ou de pneus, et des voitures devaient même être remorquées hors de la chaîne de montage car elles ne démarraient pas.

En revanche, dans des opérations de fabrication de haute performance, il y a un flux d'information rapide, fréquent et de haute qualité à travers toute la chaîne de valeur - chaque opération de travail est mesurée et surveillée, et tout défaut ou écart significatif est rapidement détecté et corrigé. Ce sont les fondations qui permettent la qualité, la sécurité, et l'apprentissage et l'amélioration continus.

Dans le flux de valeur technologique, nous obtenons souvent de mauvais résultats en raison de l'absence de retour d'information rapide. Par exemple, dans un projet logiciel en cascade, nous pouvons développer du code pendant toute une année et ne recevoir aucun retour d'information sur la qualité jusqu'à ce que nous commençons la phase de test - ou pire, lorsque nous livrons notre logiciel aux clients. Lorsque le retour d'information est si retardé et peu fréquent, il est trop lent pour nous permettre de prévenir des résultats indésirables.

En revanche, notre objectif est de créer des boucles de rétroaction et d'anticipation rapides partout où le travail est effectué, à toutes les étapes du flux de valeur technologique, englobant la gestion de produit, le développement, l'assurance qualité, la sécurité de l'information et les opérations. Cela inclut la création de processus de build, d'intégration et de test automatisés, afin que nous puissions détecter immédiatement lorsqu'un changement nous fait sortir d'un état de fonctionnement et de déploiement correct.

Nous créons également une télémétrie omniprésente afin que nous puissions voir comment tous nos composants du système fonctionnent dans l'environnement de production, de sorte que nous puissions détecter rapidement lorsqu'ils ne fonctionnent pas comme prévu. La

télémétrie nous permet également de mesurer si nous atteignons nos objectifs prévus et, idéalement, est diffusée à l'ensemble de la chaîne de valeur afin que nous puissions voir comment nos actions affectent les autres parties du système dans son ensemble.

Les boucles de rétroaction permettent non seulement une détection rapide et une récupération des problèmes, mais elles nous informent également sur la manière de prévenir ces problèmes à l'avenir. Faire cela augmente la qualité et la sécurité de notre système de travail, et crée un apprentissage organisationnel.

Comme l'a dit Elisabeth Hendrickson, VP of Engineering chez Pivotal Software, Inc. et auteur de *Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing*, "Quand je dirigeais l'ingénierie de la qualité, je décrivais mon travail comme 'créer des cycles de retour d'information'. Le retour d'information est crucial car c'est ce qui nous permet de diriger. Nous devons constamment valider entre les besoins des clients, nos intentions et nos implémentations. Les tests ne sont qu'un type de retour d'information."

Réunir et résoudre les problèmes pour construire de nouvelles connaissances

Il ne suffit pas de simplement détecter quand l'inattendu se produit. Lorsque des problèmes surviennent, nous devons les traiter immédiatement en mobilisant toutes les ressources nécessaires pour les résoudre.

Selon le Dr. Spear, l'objectif de cette approche est de contenir les problèmes avant qu'ils ne puissent se propager, et de diagnostiquer et traiter le problème de manière qu'il ne puisse pas se reproduire. « Ce faisant, » dit-il, « ils construisent une connaissance toujours plus profonde sur la gestion des systèmes de travail, convertissant l'ignorance initiale inévitable en connaissance. »

Le modèle de ce principe est le cordon Andon de Toyota. Dans une usine de fabrication Toyota, au-dessus de chaque centre de travail se trouve un cordon que chaque travailleur et manager est formé à tirer lorsqu'un problème survient ; par exemple, lorsqu'une pièce est défectueuse, lorsqu'une pièce requise n'est pas disponible, ou même lorsque le travail prend plus de temps que prévu.

Lorsque le cordon Andon est tiré, le chef d'équipe est alerté et travaille immédiatement à résoudre le problème. Si le problème ne peut être résolu dans un temps spécifié (par exemple, cinquante-cinq secondes), la ligne de production est arrêtée pour que toute l'organisation puisse être mobilisée pour aider à la résolution du problème jusqu'à ce qu'une contre-mesure réussie ait été développée.

Au lieu de contourner le problème ou de programmer une correction « lorsque nous aurons plus de temps, » nous nous réunissons pour le résoudre immédiatement - ce qui est presque à l'opposé du comportement observé à l'usine GM de Fremont décrite précédemment. Réunir des ressources est nécessaire pour les raisons suivantes :

- Cela empêche le problème de progresser en aval, où le coût et l'effort pour le réparer augmentent exponentiellement et la dette technique s'accumule.
- Cela empêche le centre de travail de commencer un nouveau travail, ce qui introduirait probablement de nouvelles erreurs dans le système.
- Si le problème n'est pas résolu, le centre de travail pourrait potentiellement rencontrer le même problème lors de la prochaine opération (par exemple, cinquante-cinq secondes plus tard), nécessitant plus de corrections et de travail.

Cette pratique de mobilisation semble contraire aux pratiques de gestion courantes, car nous permettons délibérément à un problème local de perturber les opérations globales. Cependant, cette approche favorise l'apprentissage. Elle empêche la perte d'informations critiques dues à l'oubli ou aux circonstances changeantes. Cela est particulièrement crucial dans les systèmes complexes, où de nombreux problèmes surviennent en raison d'une interaction inattendue et idiosyncratique de personnes, de processus, de produits, de lieux et de circonstances - au fur et à mesure que le temps passe, il devient impossible de reconstituer exactement ce qui se passait lorsque le problème est survenu.

Comme le note le Dr. Spear, se réunir est une partie du « cycle discipliné de reconnaissance, de diagnostic... et de traitement des problèmes en temps réel (contre-mesures ou mesures correctives dans le langage de la fabrication). C'est la discipline du cycle de Shewhart - planifier, faire, vérifier, agir - popularisé par W. Edwards Deming, mais accélérée à la vitesse de la lumière.
»

C'est uniquement par la mobilisation autour de problèmes toujours plus petits découverts toujours plus tôt dans le cycle de vie que nous pouvons dévier les problèmes avant qu'une catastrophe ne survienne. En d'autres termes, lorsque le réacteur nucléaire fond, il est déjà trop tard pour éviter les pires conséquences.

Pour permettre un retour d'information rapide dans le flux de valeur technologique, nous devons créer l'équivalent d'un cordon Andon et de la réponse de mobilisation associée. Cela nécessite également de créer une culture qui rend sûr, et même encourage, de tirer le cordon Andon lorsque quelque chose ne va pas, que ce soit lors d'un incident de production ou lorsque des erreurs surviennent plus tôt dans le flux de valeur, comme lorsque quelqu'un introduit un changement qui casse nos processus de build ou de test continus.

Lorsque des conditions déclenchent le tirage du cordon Andon, nous nous réunissons pour résoudre le problème et empêcher l'introduction de nouveaux travaux jusqu'à ce que le problème soit résolu. Cela fournit un retour d'information rapide à tout le monde dans le flux de valeur (en particulier à la personne qui a causé l'échec du système), nous permet d'isoler et de

diagnostiquer rapidement le problème, et empêche les facteurs de complication supplémentaires qui peuvent obscurcir la cause et l'effet.

Empêcher l'introduction de nouveaux travaux permet une intégration et un déploiement continu, qui représentent le flux de pièce unique dans le flux de valeur technologique. Tous les changements qui passent nos tests de build et d'intégration continu sont déployés en production, et tous les changements qui causent un échec de test déclenchent notre cordon Andon et sont traités jusqu'à résolution.

Pousser la qualité plus près de la source

Nous perpétuons parfois des systèmes de travail dangereux en raison de notre façon de répondre aux accidents et incidents. Dans les systèmes complexes, ajouter plus d'étapes d'inspection et de processus d'approbation augmente en réalité la probabilité d'échecs futurs. L'efficacité des processus d'approbation diminue à mesure que nous éloignons la prise de décision du lieu où le travail est effectué. Cela non seulement réduit la qualité des décisions, mais augmente également notre temps de cycle, diminuant ainsi la force du retour d'information entre la cause et l'effet, et réduisant notre capacité à apprendre des succès et des échecs.

Cela peut se voir même dans des systèmes plus petits et moins complexes. Lorsque les systèmes de commande et de contrôle bureaucratiques de haut en bas deviennent inefficaces, c'est généralement parce que la variance entre « qui devrait faire quelque chose » et « qui fait réellement quelque chose » est trop grande, en raison d'un manque de clarté et de rapidité.

Exemples de contrôles de qualité inefficaces :

- Exiger qu'une autre équipe effectue des tâches fastidieuses, sujettes aux erreurs et manuelles qui pourraient être facilement automatisées et exécutées selon les besoins par l'équipe qui a besoin du travail effectué
- Exiger des approbations de personnes occupées et éloignées du travail, les forçant à prendre des décisions sans une connaissance adéquate du travail ou des implications potentielles, ou simplement à tamponner leurs approbations
- Créer de gros volumes de documentation de détail douteux qui deviennent obsolètes peu de temps après avoir été écrits
- Pousser de gros lots de travail vers des équipes et des comités spéciaux pour approbation et traitement puis attendre des réponses

Au lieu de cela, nous avons besoin que tout le monde dans notre flux de valeur trouve et corrige les problèmes dans leur domaine de contrôle dans le cadre de notre travail quotidien. En faisant cela, nous rapprochons les responsabilités et la prise de décision en matière de qualité et de sécurité là où le travail est effectué, au lieu de compter sur des approbations de cadres éloignés.

Nous utilisons des revues par les pairs de nos modifications proposées pour obtenir les assurances nécessaires que nos modifications fonctionneront comme prévu.

Nous automatisons autant que possible les vérifications de qualité typiquement effectuées par un département de QA ou de sécurité de l'information. Au lieu que les développeurs aient besoin de demander ou de programmer un test, ces tests peuvent être effectués à la demande, permettant aux développeurs de tester rapidement leur propre code et même de déployer ces changements en production eux-mêmes.

En faisant cela, nous faisons de la qualité la responsabilité de tout le monde au lieu qu'elle soit uniquement celle d'un département séparé. La sécurité de l'information n'est pas seulement la responsabilité du département de la sécurité de l'information, tout comme la disponibilité n'est pas seulement la responsabilité des opérations.

Faire en sorte que les développeurs partagent la responsabilité de la qualité des systèmes qu'ils construisent améliore non seulement les résultats, mais accélère également l'apprentissage. Cela est particulièrement important pour les développeurs, car ils sont typiquement l'équipe la plus éloignée du client. Gary Gruver observe, « Il est impossible pour un développeur d'apprendre quoi que ce soit lorsque quelqu'un lui crie dessus pour quelque chose qu'il a cassé six mois plus tôt - c'est pourquoi nous devons fournir un retour d'information à tout le monde aussi rapidement que possible, en minutes, pas en mois. »

Permettre l'optimisation pour les centres de travail en aval

Dans les années 1980, les principes de la "Conception pour la Fabricabilité" (Designing for Manufacturability) visaient à concevoir des pièces et des processus permettant de créer des produits finis au coût le plus bas, avec la plus haute qualité et le flux le plus rapide. Par exemple, concevoir des pièces extrêmement asymétriques pour éviter qu'elles ne soient montées à l'envers, ou concevoir des vis impossibles à serrer trop fort.

Cela différait de la conception typique qui se concentrait principalement sur les clients externes, négligeant souvent les parties prenantes internes, telles que les personnes impliquées dans la fabrication.

Le Lean définit deux types de clients pour lesquels nous devons concevoir :

- Le client externe, qui paie pour le service que nous offrons.
- Le client interne, qui reçoit et traite le travail immédiatement après nous.

Selon le Lean, notre client le plus important est l'étape suivante en aval.

Optimiser notre travail pour le client en aval implique de développer une empathie pour leurs problèmes afin d'identifier les problèmes de conception qui empêchent un flux rapide et fluide. Dans le flux de valeur technologique, nous optimisons pour les centres de travail en aval en concevant pour les opérations, où les exigences non fonctionnelles opérationnelles (par exemple, l'architecture, la performance, la stabilité, la testabilité, la configurabilité et la sécurité) sont aussi prioritaires que les fonctionnalités utilisateur. En faisant cela, nous créons la qualité à la source, ce qui se traduit probablement par un ensemble d'exigences non fonctionnelles codifiées que nous pouvons intégrer de manière proactive dans chaque service que nous construisons.

Conclusion

Créer un retour d'information rapide est essentiel pour atteindre la qualité, la fiabilité et la sécurité dans le flux de valeur technologique. Nous y parvenons en :

- **Identifiant les problèmes au fur et à mesure qu'ils se produisent** : La détection rapide des problèmes est cruciale pour les traiter avant qu'ils ne s'aggravent.
- **Réunissant des ressources et résolvant les problèmes pour construire de nouvelles connaissances** : Mobiliser immédiatement les ressources nécessaires pour résoudre les problèmes empêche leur propagation et permet de transformer l'ignorance initiale en connaissance approfondie.
- **Poussant la qualité plus près de la source** : Responsabiliser les équipes locales pour trouver et corriger les problèmes dans leur domaine de contrôle améliore la qualité et la sécurité sans nécessiter de contrôles bureaucratiques éloignés.
- **Optimisant continuellement pour les centres de travail en aval** : Conception proactive en tenant compte des besoins opérationnels et des exigences non fonctionnelles pour assurer un flux de travail fluide et efficace.

Les pratiques spécifiques qui permettent un flux rapide dans le flux de valeur DevOps sont présentées dans la Partie IV. Le prochain chapitre présentera la Troisième Voie : Les principes de retour d'information.

La troisième voie : Les principes d'apprentissage continu et d'expérimentation

Alors que la Première Voie traite du flux de travail de gauche à droite et que la Deuxième Voie traite du retour d'information rapide et constant de droite à gauche, la Troisième Voie se concentre sur la création d'une culture d'apprentissage et d'expérimentation continus. Ce sont les principes qui permettent la création constante de connaissances individuelles, qui sont ensuite transformées en connaissances d'équipe et organisationnelles.

Dans les opérations de fabrication présentant des problèmes systémiques de qualité et de sécurité, le travail est généralement rigoureusement défini et appliqué. Par exemple, dans l'usine GM de Fremont décrite dans le chapitre précédent, les travailleurs avaient peu de capacité à intégrer des améliorations et des apprentissages dans leur travail quotidien, et les suggestions d'amélioration étaient "susceptibles de rencontrer un mur d'indifférence".

Dans ces environnements, il y a souvent aussi une culture de la peur et une faible confiance, où les travailleurs qui commettent des erreurs sont punis, et ceux qui font des suggestions ou signalent des problèmes sont considérés comme des dénonciateurs et des fauteurs de troubles. Lorsque cela se produit, la direction supprime activement, voire punit, l'apprentissage et l'amélioration, perpétuant ainsi les problèmes de qualité et de sécurité.

En revanche, les opérations de fabrication performantes exigent et promeuvent activement l'apprentissage. Au lieu que le travail soit rigoureusement défini, le système de travail est dynamique, avec des travailleurs de ligne effectuant des expériences dans leur travail quotidien pour générer de nouvelles améliorations, rendues possibles par la normalisation rigoureuse des procédures de travail et la documentation des résultats.

Dans le flux de valeur technologique, notre objectif est de créer une culture de haute confiance, renforçant que nous sommes tous des apprenants à vie qui doivent prendre des risques dans notre travail quotidien. En appliquant une approche scientifique à l'amélioration des processus et au développement de produits, nous apprenons de nos succès et de nos échecs, identifiant les idées qui ne fonctionnent pas et renforçant celles qui fonctionnent. De plus, tout apprentissage local est rapidement transformé en améliorations globales, afin que les nouvelles techniques et pratiques puissent être utilisées par l'ensemble de l'organisation.

Nous réservons du temps pour l'amélioration du travail quotidien afin d'accélérer et d'assurer l'apprentissage. Nous introduisons constamment des stress dans nos systèmes pour forcer l'amélioration continue. Nous simulons même des échecs et injectons des pannes dans nos services de production dans des conditions contrôlées pour augmenter notre résilience.

En créant ce système d'apprentissage continu et dynamique, nous permettons aux équipes de s'adapter rapidement et automatiquement à un environnement en constante évolution, ce qui nous aide finalement à gagner sur le marché.

Favoriser l'apprentissage organisationnel et une culture de sécurité

Lorsque nous travaillons au sein d'un système complexe, par définition, il est impossible pour nous de prévoir parfaitement tous les résultats de toute action que nous entreprenons. Cela contribue à des résultats inattendus, voire catastrophiques, et à des accidents dans notre travail quotidien, même lorsque nous prenons des précautions et travaillons soigneusement.

Lorsque ces accidents affectent nos clients, nous cherchons à comprendre pourquoi cela s'est produit. La cause principale est souvent considérée comme une erreur humaine, et la réponse managériale trop courante est de "nommer, blâmer et faire honte" à la personne qui a causé le problème. Et, soit subtilement, soit explicitement, la direction laisse entendre que la personne coupable de l'erreur sera punie. Ils créent ensuite plus de processus et d'approbations pour empêcher que l'erreur ne se reproduise.

Le Dr Sidney Dekker, qui a codifié certains des éléments clés de la culture de sécurité et a inventé le terme "culture juste", a écrit : "Les réponses aux incidents et aux accidents qui sont perçues comme injustes peuvent entraver les enquêtes de sécurité, promouvoir la peur plutôt que la pleine conscience chez les personnes qui effectuent des travaux critiques pour la sécurité, rendre les organisations plus bureaucratiques plutôt que plus prudentes, et cultiver le secret professionnel, l'évasion et l'autoprotection."

Ces problèmes sont particulièrement problématiques dans le flux de valeur technologique : notre travail est presque toujours effectué au sein d'un système complexe, et la manière dont la direction choisit de réagir aux échecs et aux accidents conduit à une culture de la peur, ce qui rend peu probable que les problèmes et les signaux d'échec soient jamais signalés. Le résultat est que les problèmes restent cachés jusqu'à ce qu'une catastrophe survienne.

Le Dr Ron Westrum a été l'un des premiers à observer l'importance de la culture organisationnelle sur la sécurité et la performance. Il a observé que dans les organisations de soins de santé, la présence de cultures "générationnelles" était l'un des principaux prédicteurs de la sécurité des patients. Le Dr Westrum a défini trois types de culture :

- Les organisations pathologiques se caractérisent par de grandes quantités de peur et de menace. Les gens accumulent souvent des informations, les retiennent pour des raisons politiques ou les déforment pour se faire mieux paraître. Les échecs sont souvent cachés.
- Les organisations bureaucratiques se caractérisent par des règles et des processus, souvent pour aider les départements individuels à maintenir leur "territoire". L'échec est traité par un système de jugement, aboutissant soit à une punition, soit à la justice et la clémence.
- Les organisations génératives se caractérisent par la recherche active et le partage d'informations pour mieux permettre à l'organisation d'atteindre sa mission. Les responsabilités sont partagées tout au long du flux de valeur, et l'échec se traduit par une réflexion et une enquête sincère.

Pathological	Bureaucratic	Generative
Information is hidden	Information may be ignored	Information is actively sought
Messengers are “shot”	Messengers are tolerated	Messengers are trained
Responsibilities are shirked	Responsibilities are compartmented	Responsibilities are shared
Bridging between teams is discouraged	Bridging between teams is allowed but discouraged	Bridging between teams is rewarded
Failure is covered up	Organization is just and merciful	Failure causes inquiry
New ideas are crushed	New ideas create problems	New ideas are welcomed

Figure 8: The Westrum organizational typology model: how organizations process information (Source: Ron Westrum, “A typology of organisation culture,” BMJ Quality & Safety 13, no. 2 (2004), doi:10.1136/qshc.2003.009522.)

Tout comme le Dr Westrum l'a constaté dans les organisations de soins de santé, une culture génératrice de haute confiance prédit également la performance informatique et organisationnelle dans les flux de valeur technologiques.

Dans le flux de valeur technologique, nous établissons les fondations d'une culture génératrice en nous efforçant de créer un système de travail sûr. Lorsque des accidents et des échecs se produisent, au lieu de rechercher l'erreur humaine, nous cherchons comment nous pouvons redessiner le système pour empêcher l'accident de se reproduire.

Par exemple, nous pouvons mener une autopsie sans blâme après chaque incident afin de mieux comprendre comment l'accident s'est produit et de convenir des meilleures contre-mesures pour améliorer le système, idéalement pour empêcher le problème de se reproduire et permettre une détection et une récupération plus rapides.

En faisant cela, nous créons un apprentissage organisationnel. Comme l'a déclaré Bethany Macri, une ingénierie chez Etsy qui a dirigé la création de l'outil Morgue pour aider à l'enregistrement des autopsies : "En éliminant le blâme, vous éliminez la peur ; en éliminant la peur, vous permettez l'honnêteté ; et l'honnêteté permet la prévention."

Le Dr Spear observe que le résultat de l'élimination du blâme et de la mise en place de l'apprentissage organisationnel est que "les organisations deviennent de plus en plus auto-diagnostiquantes et auto-améliorantes, habiles à détecter les problèmes [et] à les résoudre."

Beaucoup de ces attributs ont également été décrits par le Dr Senge comme des attributs des organisations apprenantes. Dans *The Fifth Discipline*, il a écrit que ces caractéristiques aident les clients, garantissent la qualité, créent un avantage concurrentiel et une main-d'œuvre énergisée et engagée, et découvrent la vérité.

Institutionnaliser l'amélioration du travail quotidien

Les équipes ne sont souvent pas capables ou ne veulent pas améliorer les processus dans lesquels elles opèrent. Le résultat est non seulement qu'elles continuent à souffrir de leurs problèmes actuels, mais que leur souffrance s'aggrave également avec le temps. Mike Rother a observé dans *Toyota Kata* qu'en l'absence d'améliorations, les processus ne restent pas les mêmes - en raison du chaos et de l'entropie, les processus se dégradent en fait avec le temps.

Dans le flux de valeur technologique, lorsque nous évitons de résoudre nos problèmes, en comptant sur des solutions de contournement quotidiennes, nos problèmes et notre dette technique s'accumulent jusqu'à ce que tout ce que nous faisons soit des solutions de contournement, essayant d'éviter le désastre, sans aucun cycle restant pour effectuer un travail productif. C'est pourquoi Mike Orzen, auteur de *Lean IT*, a observé : "Encore plus important que le travail quotidien est l'amélioration du travail quotidien."

Nous améliorons le travail quotidien en réservant explicitement du temps pour rembourser la dette technique, corriger les défauts, et refactoriser et améliorer les zones problématiques de notre code et de nos environnements - nous faisons cela en réservant des cycles dans chaque intervalle de développement, ou en programmant des blitz kaizen, qui sont des périodes où les ingénieurs s'auto-organisent en équipes pour travailler sur la résolution de tout problème qu'ils souhaitent.

Le résultat de ces pratiques est que tout le monde trouve et résout des problèmes dans leur domaine de contrôle, tout le temps, dans le cadre de leur travail quotidien. Lorsque nous résolvons enfin les problèmes quotidiens que nous avons contournés pendant des mois (ou des années), nous pouvons éradiquer de notre système les problèmes moins évidents. En détectant et en répondant à ces signaux de défaillance de plus en plus faibles, nous résolvons les problèmes quand il est non seulement plus facile et moins coûteux de le faire, mais aussi quand les conséquences sont moindres.

Considérez l'exemple suivant qui a amélioré la sécurité sur le lieu de travail chez Alcoa, un fabricant d'aluminium avec 7,8 milliards de dollars de revenus en 1987. La fabrication d'aluminium nécessite des températures extrêmement élevées, des pressions élevées et des

produits chimiques corrosifs. En 1987, Alcoa avait un bilan de sécurité effrayant, avec 2 % des quatre-vingt-dix mille employés blessés chaque année - soit sept blessures par jour. Lorsque Paul O'Neill a commencé en tant que PDG, son premier objectif était d'avoir zéro blessure pour les employés, les contractants et les visiteurs.

O'Neill voulait être informé dans les vingt-quatre heures de toute personne blessée au travail - non pas pour punir, mais pour s'assurer et promouvoir que les apprentissages étaient générés et incorporés pour créer un lieu de travail plus sûr. Au cours des dix années suivantes, Alcoa a réduit son taux de blessures de 95 %.

La réduction des taux de blessures a permis à Alcoa de se concentrer sur des problèmes plus petits et des signaux de défaillance plus faibles - au lieu de notifier O'Neill seulement en cas de blessures, ils ont commencé à signaler également tout incident évité de justesse.

En faisant cela, ils ont amélioré la sécurité sur le lieu de travail au cours des vingt années suivantes et possèdent l'un des bilans de sécurité les plus enviables de l'industrie.

Comme l'écrit le Dr Spear : "Les employés d'Alcoa ont progressivement cessé de contourner les difficultés, les inconvénients et les obstacles qu'ils rencontraient. Faire face, éteindre les incendies et se débrouiller ont été progressivement remplacés dans toute l'organisation par une dynamique d'identification des opportunités d'amélioration des processus et des produits. Lorsque ces opportunités ont été identifiées et que les problèmes ont été investigués, les poches d'ignorance qu'ils reflétaient ont été converties en pépites de connaissance." Cela a aidé l'entreprise à obtenir un avantage concurrentiel plus important sur le marché.

De même, dans le flux de valeur technologique, à mesure que nous rendons notre système de travail plus sûr, nous trouvons et résolvons des problèmes à partir de signaux de défaillance de plus en plus faibles. Par exemple, nous pouvons initialement effectuer des autopsies sans blâme uniquement pour les incidents ayant un impact sur les clients. Avec le temps, nous pouvons les réaliser pour des incidents ayant un impact moindre sur l'équipe et les incidents évités de justesse.

Transformer les découvertes locales en améliorations globales

Lorsque de nouveaux apprentissages sont découverts localement, il doit également y avoir un mécanisme permettant au reste de l'organisation d'utiliser et de bénéficier de ces connaissances. En d'autres termes, lorsque des équipes ou des individus ont des expériences qui créent une expertise, notre objectif est de convertir ces connaissances tacites (c'est-à-dire des connaissances difficiles à transférer à une autre personne par écrit ou verbalement) en

connaissances explicites et codifiées, qui deviennent l'expertise de quelqu'un d'autre grâce à la pratique.

Cela garantit que lorsque quelqu'un d'autre effectue un travail similaire, il le fait avec l'expérience cumulative et collective de tous ceux dans l'organisation qui ont déjà fait le même travail. Un exemple remarquable de transformation des connaissances locales en connaissances globales est le programme de propulsion nucléaire de la marine américaine (également connu sous le nom de "NR" pour "Naval Reactors"), qui a plus de 5 700 années-réacteurs d'opération sans un seul accident lié au réacteur ou fuite de radiation.

Le NR est connu pour son engagement intense envers les procédures scriptées et le travail standardisé, et la nécessité de rapports d'incident pour toute déviation par rapport à la procédure ou aux opérations normales afin d'accumuler les apprentissages, peu importe la faiblesse du signal de défaillance - ils mettent constamment à jour les procédures et les conceptions des systèmes en fonction de ces apprentissages.

Le résultat est que lorsqu'un nouvel équipage prend la mer pour sa première mission, lui et ses officiers bénéficient des connaissances collectives de 5 700 années-réacteurs sans accident. Il est tout aussi impressionnant que leurs propres expériences en mer soient ajoutées à ces connaissances collectives, aidant les futurs équipages à accomplir leurs propres missions en toute sécurité.

Dans le flux de valeur technologique, nous devons créer des mécanismes similaires pour créer des connaissances globales, comme rendre tous nos rapports d'autopsie sans blâme consultables par les équipes essayant de résoudre des problèmes similaires, et créer des dépôts de code source partagés couvrant toute l'organisation, où le code partagé, les bibliothèques et les configurations qui incarnent les meilleures connaissances collectives de l'ensemble de l'organisation peuvent être facilement utilisés. Tous ces mécanismes aident à convertir l'expertise individuelle en artefacts que le reste de l'organisation peut utiliser.

Injecter des modèles de résilience dans notre travail quotidien

Les organisations manufacturières moins performantes se protègent des perturbations de nombreuses manières - en d'autres termes, elles accumulent ou ajoutent de la graisse. Par exemple, pour réduire le risque qu'un centre de travail soit inactif (en raison d'un inventaire arrivé en retard, d'un inventaire devant être mis au rebut, etc.), les gestionnaires peuvent choisir de stocker plus d'inventaire à chaque centre de travail. Cependant, ce tampon d'inventaire augmente également le travail en cours (WIP), ce qui entraîne toutes sortes de résultats indésirables, comme mentionné précédemment.

De même, pour réduire le risque qu'un centre de travail tombe en panne en raison d'une défaillance des machines, les gestionnaires peuvent augmenter la capacité en achetant plus d'équipements, en embauchant plus de personnel ou même en augmentant l'espace au sol. Toutes ces options augmentent les coûts.

En revanche, les performants de haut niveau obtiennent les mêmes résultats (ou mieux) en améliorant les opérations quotidiennes, en introduisant continuellement des tensions pour éléver la performance, ainsi qu'en intégrant plus de résilience dans leur système.

Considérez une expérience typique dans l'une des usines de matelas d'Aisin Seiki Global, l'un des principaux fournisseurs de Toyota. Supposons qu'ils aient deux lignes de production, chacune capable de produire cent unités par jour. Les jours de faible activité, ils enverraient toute la production sur une ligne, expérimentant des moyens d'augmenter la capacité et d'identifier les vulnérabilités dans leur processus, sachant que si la surcharge de la ligne causait une défaillance, ils pourraient envoyer toute la production sur la deuxième ligne.

Grâce à des expérimentations incessantes et constantes dans leur travail quotidien, ils ont pu augmenter continuellement la capacité, souvent sans ajouter de nouvel équipement ou embaucher plus de personnel. Le modèle émergent résultant de ces types de rituels d'amélioration non seulement améliore la performance, mais aussi la résilience, car l'organisation est toujours en état de tension et de changement. Ce processus d'application de stress pour augmenter la résilience a été nommé antifragilité par l'auteur et analyste des risques Nassim Nicholas Taleb.

Dans le flux de valeur technologique, nous pouvons introduire le même type de tension dans nos systèmes en cherchant toujours à réduire les délais de déploiement, augmenter la couverture des tests, diminuer les temps d'exécution des tests, et même en réarchitecturant si nécessaire pour augmenter la productivité des développeurs ou améliorer la fiabilité.

Nous pouvons également effectuer des exercices de simulation de panne, où nous répétons de grandes pannes, comme éteindre des centres de données entiers. Ou nous pouvons injecter des pannes de plus en plus grandes dans l'environnement de production (comme le célèbre "Chaos Monkey" de Netflix, qui tue aléatoirement des processus et des serveurs de calcul en production) pour nous assurer que nous sommes aussi résilients que nous le souhaitons.

Les leaders renforcent une culture d'apprentissage

Traditionnellement, les leaders étaient censés être responsables de la définition des objectifs, de l'allocation des ressources pour atteindre ces objectifs, et de l'établissement de la bonne combinaison d'incitations. Les leaders établissent également le ton émotionnel des

organisations qu'ils dirigent. En d'autres termes, les leaders dirigent en « prenant toutes les bonnes décisions ».

Cependant, il existe des preuves significatives montrant que la grandeur n'est pas atteinte par les leaders prenant toutes les bonnes décisions - au lieu de cela, le rôle du leader est de créer les conditions permettant à leur équipe de découvrir la grandeur dans leur travail quotidien. En d'autres termes, créer la grandeur nécessite à la fois des leaders et des travailleurs, chacun étant mutuellement dépendant de l'autre.

Jim Womack, auteur de Gemba Walks, a décrit la relation de travail complémentaire et le respect mutuel qui doivent exister entre les leaders et les travailleurs de première ligne. Selon Womack, cette relation est nécessaire car aucun des deux ne peut résoudre les problèmes seul - les leaders ne sont pas suffisamment proches du travail, ce qui est nécessaire pour résoudre tout problème, et les travailleurs de première ligne n'ont pas le contexte organisationnel plus large ni l'autorité pour apporter des changements en dehors de leur domaine de travail.

Les leaders doivent éléver la valeur de l'apprentissage et de la résolution disciplinée des problèmes. Mike Rother a formalisé ces méthodes dans ce qu'il appelle le kata de coaching. Le résultat est un miroir de la méthode scientifique, où nous déclarons explicitement nos objectifs True North, tels que « maintenir zéro accident » dans le cas d'Alcoa, ou « doubler le débit en un an » dans le cas d'Aisin.

Ces objectifs stratégiques informent ensuite la création d'objectifs à court terme, qui sont déclinés et ensuite exécutés en établissant des conditions cibles au niveau du flux de valeur ou du centre de travail (par exemple, « réduire le délai de 10 % dans les deux prochaines semaines »).

Ces conditions cibles encadrent l'expérience scientifique : nous déclarons explicitement le problème que nous cherchons à résoudre, notre hypothèse sur la manière dont notre contre-mesure proposée le résoudra, nos méthodes pour tester cette hypothèse, notre interprétation des résultats, et notre utilisation des apprentissages pour informer la prochaine itération.

Le leader aide à coacher la personne menant l'expérience avec des questions qui peuvent inclure :

- Quelle a été votre dernière étape et que s'est-il passé ?
- Qu'avez-vous appris ?
- Quelle est votre condition actuelle ?
- Quelle est votre prochaine condition cible ?
- Quel obstacle travaillez-vous actuellement ?
- Quelle est votre prochaine étape ?
- Quel est le résultat attendu ?

- Quand pouvons-nous vérifier ?

Cette approche de résolution de problèmes dans laquelle les leaders aident les travailleurs à voir et à résoudre les problèmes dans leur travail quotidien est au cœur du Toyota Production System, des organisations apprenantes, du kata d'amélioration et des organisations à haute fiabilité. Mike Rother observe qu'il voit Toyota « comme une organisation définie principalement par les routines de comportement uniques qu'elle enseigne continuellement à tous ses membres. »

Dans le flux de valeur technologique, cette approche scientifique et cette méthode itérative guident tous nos processus d'amélioration interne, mais aussi la manière dont nous réalisons des expériences pour nous assurer que les produits que nous construisons aident réellement nos clients internes et externes à atteindre leurs objectifs.

Conclusion

Les principes de la Troisième Voie abordent la nécessité de valoriser l'apprentissage organisationnel, de permettre une haute confiance et des échanges inter-fonctionnels, d'accepter que des échecs se produiront toujours dans des systèmes complexes, et de rendre acceptable de parler des problèmes pour que nous puissions créer un système de travail sûr. Cela nécessite également d'institutionnaliser l'amélioration du travail quotidien, de convertir les apprentissages locaux en apprentissages globaux utilisables par toute l'organisation, ainsi que d'injecter continuellement de la tension dans notre travail quotidien.

Bien que favoriser une culture d'apprentissage et d'expérimentation continu soit le principe de la Troisième Voie, il est également entrelacé dans les Première et Deuxième Voies. En d'autres termes, améliorer le flux et les retours nécessite une approche itérative et scientifique qui comprend la définition d'une condition cible, la déclaration d'une hypothèse sur ce qui nous aidera à y parvenir, la conception et la réalisation d'expériences, et l'évaluation des résultats.

Les résultats ne sont pas seulement une meilleure performance, mais aussi une résilience accrue, une plus grande satisfaction au travail et une adaptabilité organisationnelle améliorée.

Conclusion de la Partie I

Dans la première partie de The DevOps Handbook, nous avons examiné plusieurs mouvements historiques qui ont contribué au développement de DevOps. Nous avons également analysé les trois principes fondamentaux qui forment la base des organisations DevOps réussies : les principes de Flux, de Retour d'Information et d'Apprentissage et d'Expérimentation Continus. Dans la deuxième partie, nous commencerons à examiner comment démarrer un mouvement DevOps dans votre organisation.

Partie II - Par où commencer

Introduction

Comment décider où commencer une transformation DevOps dans notre organisation ? Qui doit être impliqué ? Comment organiser nos équipes, protéger leur capacité de travail et maximiser leurs chances de succès ? Ce sont les questions auxquelles nous visons à répondre dans la Partie II du "DevOps Handbook".

Dans les chapitres suivants, nous allons parcourir le processus d'initiation d'une transformation DevOps. Nous commencerons par évaluer les flux de valeur dans notre organisation, identifier un bon point de départ, et former une stratégie pour créer une équipe dédiée à la transformation avec des objectifs d'amélioration spécifiques et une expansion éventuelle. Pour chaque flux de valeur en cours de transformation, nous identifierons le travail effectué et examinerons ensuite les stratégies de conception organisationnelle et les archétypes organisationnels qui soutiennent le mieux les objectifs de transformation.

Les principaux points d'attention dans ces chapitres incluent :

- Sélectionner les flux de valeur par lesquels commencer
- Comprendre le travail effectué dans nos flux de valeur candidats
- Concevoir notre organisation et notre architecture en tenant compte de la loi de Conway
- Permettre des résultats orientés vers le marché grâce à une collaboration plus efficace entre les fonctions tout au long du flux de valeur
- Protéger et habiliter nos équipes

Commencer toute transformation est plein d'incertitude - nous traçons un chemin vers un état final idéal, mais où pratiquement toutes les étapes intermédiaires sont inconnues. Ces prochains chapitres sont destinés à fournir un processus de réflexion pour guider nos décisions, proposer des actions concrètes à entreprendre et illustrer des études de cas à titre d'exemples.

Sélectionner le Flux de Valeur par lequel Commencer

Choisir un flux de valeur pour la transformation DevOps mérite une réflexion approfondie. Non seulement le flux de valeur que nous choisissons dicte la difficulté de notre transformation, mais il détermine également qui sera impliqué dans la transformation. Cela influencera comment nous devrons nous organiser en équipes et comment nous pourrons au mieux habiliter les équipes et les individus qui les composent.

Michael Rembetsky, qui a aidé à diriger la transformation DevOps en tant que Directeur des Opérations chez Etsy en 2009, a observé : "Nous devons choisir nos projets de transformation avec soin - quand nous avons des difficultés, nous n'avons pas beaucoup d'essais. Par conséquent, nous devons choisir avec soin et ensuite protéger ces projets d'amélioration qui amélioreront le plus l'état de notre organisation."

Examinons comment l'équipe de Nordstrom a commencé son initiative de transformation DevOps en 2013, que Courtney Kissler, leur Vice-Présidente des Technologies de l'E-Commerce et des Magasins, a décrite au DevOps Enterprise Summit en 2014 et 2015. Fondée en 1901, Nordstrom est un détaillant de mode de premier plan, axé sur l'offre de la meilleure expérience de shopping possible à ses clients. En 2015, Nordstrom a réalisé un chiffre d'affaires annuel de 13,5 milliards de dollars.

La scène pour le parcours DevOps de Nordstrom a probablement été plantée en 2011 lors de l'une de leurs réunions annuelles du conseil d'administration. Cette année-là, l'un des sujets stratégiques discutés était la nécessité de la croissance des revenus en ligne. Ils ont étudié les difficultés rencontrées par Blockbuster, Borders et Barnes & Noble, qui ont démontré les conséquences désastreuses lorsque les détaillants traditionnels tardaient à créer des capacités d'e-commerce compétitives - ces organisations risquaient clairement de perdre leur position sur le marché ou même de disparaître complètement.

À cette époque, Courtney Kissler était directrice principale de la Livraison des Systèmes et des Technologies de Vente, responsable d'une partie significative de l'organisation technologique, y compris leurs systèmes en magasin et leur site d'e-commerce. Comme l'a décrit Kissler, "En 2011, l'organisation technologique de Nordstrom était très optimisée pour les coûts - nous avions externalisé de nombreuses fonctions technologiques, nous avions un cycle de planification annuel avec de grandes séries de sorties logicielles en 'waterfall'. Même si nous atteignions un taux de succès de 97 % pour respecter nos objectifs de calendrier, de budget et de périmètre, nous n'étions pas équipés pour atteindre ce que la stratégie commerciale de cinq ans exigeait de nous, car Nordstrom commençait à s'optimiser pour la vitesse au lieu de simplement s'optimiser pour les coûts."

Kissler et l'équipe de gestion de la technologie de Nordstrom devaient décider par où commencer leurs premiers efforts de transformation. Ils ne voulaient pas provoquer de bouleversements dans tout le système. Au lieu de cela, ils voulaient se concentrer sur des zones très spécifiques de l'entreprise afin de pouvoir expérimenter et apprendre. Leur objectif était de démontrer des réussites précoces, ce qui donnerait à tout le monde la confiance que ces améliorations pourraient être reproduites dans d'autres domaines de l'organisation. Comment exactement cela serait réalisé était encore inconnu.

Ils se sont concentrés sur trois domaines : l'application mobile pour les clients, leurs systèmes de restaurants en magasin, et leurs propriétés numériques. Chacun de ces domaines avait des objectifs commerciaux qui n'étaient pas atteints ; ainsi, ils étaient plus réceptifs à envisager une manière différente de travailler. Les histoires des deux premiers sont décrites ci-dessous.

L'application mobile de Nordstrom avait connu un début peu prometteur. Comme l'a dit Kissler, "Nos clients étaient extrêmement frustrés par le produit, et nous avions des avis uniformément négatifs lorsque nous l'avons lancé dans l'App Store. Pire, la structure et les processus existants (alias 'le système') avaient conçu leurs processus de manière à ne pouvoir publier des mises à jour que deux fois par an." En d'autres termes, tout correctif de l'application devait attendre des mois avant d'atteindre le client.

Leur premier objectif était de permettre des sorties plus rapides ou à la demande, fournissant une itération plus rapide et la capacité de répondre aux commentaires des clients. Ils ont créé une équipe produit dédiée uniquement au support de l'application mobile, avec l'objectif de permettre à cette équipe d'implémenter, de tester et de livrer de la valeur au client de manière indépendante. En faisant cela, ils n'auraient plus à dépendre et à se coordonner avec des dizaines d'autres équipes chez Nordstrom. De plus, ils sont passés d'une planification annuelle à un processus de planification continue. Le résultat fut un backlog de travail priorisé unique pour l'application mobile basé sur les besoins des clients - toutes les priorités conflictuelles ont disparu lorsque l'équipe devait soutenir plusieurs produits.

Au cours de l'année suivante, ils ont éliminé les tests comme une phase de travail séparée, en l'intégrant dans le travail quotidien de chacun. Ils ont doublé le nombre de fonctionnalités livrées par mois et réduit de moitié le nombre de défauts - créant un résultat réussi.

Leur deuxième domaine de concentration était les systèmes supportant leurs restaurants Café Bistro en magasin. Contrairement au flux de valeur de l'application mobile où le besoin commercial était de réduire le délai de mise sur le marché et d'augmenter le débit des fonctionnalités, le besoin commercial ici était de diminuer les coûts et d'augmenter la qualité. En 2013, Nordstrom avait complété onze "reconceptualisations de restaurants" nécessitant des changements dans les applications en magasin, causant un certain nombre d'incidents ayant un impact sur les clients. De manière préoccupante, ils avaient prévu quarante-quatre autres reconceptualisations pour 2014 - quatre fois plus que l'année précédente.

Comme l'a déclaré Kissler, "Un de nos leaders commerciaux a suggéré que nous triplions la taille de notre équipe pour gérer ces nouvelles demandes, mais j'ai proposé que nous devions arrêter de lancer plus de ressources sur le problème et plutôt améliorer notre manière de travailler."

Ils ont pu identifier des domaines problématiques, comme dans leurs processus d'entrée de travail et de déploiement, sur lesquels ils ont concentré leurs efforts d'amélioration. Ils ont pu réduire de 60 % les délais de déploiement de code et diminuer de 60 % à 90 % le nombre d'incidents en production.

Ces succès ont donné aux équipes la confiance que les principes et pratiques DevOps étaient applicables à une grande variété de flux de valeur. Kissler a été promue Vice-Présidente des Technologies de l'E-Commerce et des Magasins en 2014.

En 2015, Kissler a déclaré que pour que l'organisation technologique de vente ou axée sur le client permette à l'entreprise d'atteindre ses objectifs, "...nous devions augmenter la productivité dans tous nos flux de valeur technologiques, pas seulement dans quelques-uns. À un niveau de gestion, nous avons créé un mandat transversal pour réduire les délais de cycle de 20 % pour tous les services orientés client."

Elle a poursuivi, "C'est un défi audacieux. Nous avons de nombreux problèmes dans notre état actuel - les processus et les délais de cycle ne sont pas mesurés de manière cohérente entre les équipes, ni visibles. Notre première condition cible exige que nous aidions toutes nos équipes à mesurer, rendre visible et réaliser des expériences pour commencer à réduire leurs temps de processus, itération par itération."

Kissler a conclu, "D'un point de vue global, nous croyons que des techniques telles que la cartographie des flux de valeur, la réduction de nos tailles de lots vers un flux à pièce unique, ainsi que l'utilisation de la livraison continue et des microservices nous mèneront à notre état souhaité. Cependant, bien que nous apprenions encore, nous sommes confiants que nous nous dirigeons dans la bonne direction, et tout le monde sait que cet effort est soutenu par les plus hauts niveaux de la direction."

Sélection du Flux de Valeur par lequel Commencer

Dans ce chapitre, divers modèles sont présentés pour nous permettre de reproduire les processus de réflexion utilisés par l'équipe de Nordstrom pour décider quels flux de valeur choisir en premier. Nous évaluerons nos flux de valeur candidats de plusieurs manières, y compris s'ils sont des services greenfield ou brownfield, des systèmes d'engagement ou des systèmes d'enregistrement. Nous estimerons également l'équilibre risque/récompense de la

transformation et évaluerons le niveau probable de résistance que nous pourrions rencontrer de la part des équipes avec lesquelles nous travaillerions.

Services Greenfield vs. Brownfield

Nous catégorisons souvent nos services ou produits logiciels comme greenfield ou brownfield. Ces termes proviennent initialement de l'urbanisme et des projets de construction. Un projet de développement greenfield est lorsque nous construisons sur un terrain non développé. Un projet brownfield, quant à lui, est lorsque nous construisons sur un terrain précédemment utilisé à des fins industrielles, potentiellement contaminé par des déchets dangereux ou de la pollution.

En développement urbain, de nombreux facteurs peuvent rendre les projets greenfield plus simples que les projets brownfield : il n'y a pas de structures existantes à démolir, ni de matériaux toxiques à éliminer.

Dans la technologie, un projet greenfield est un nouveau projet ou initiative logicielle, probablement aux premiers stades de la planification ou de la mise en œuvre, où nous construisons nos applications et infrastructures à partir de zéro, avec peu de contraintes. Commencer avec un projet logiciel greenfield peut être plus facile, surtout si le projet est déjà financé et qu'une équipe est en cours de création ou déjà en place. De plus, comme nous partons de zéro, nous pouvons nous soucier moins des bases de code existantes, des processus et des équipes.

Les projets DevOps greenfield sont souvent des pilotes pour démontrer la faisabilité des clouds publics ou privés, du déploiement automatisé, et d'outils similaires. Un exemple de projet DevOps greenfield est le produit Hosted LabVIEW en 2009 chez National Instruments, une organisation vieille de trente ans avec cinq mille employés et un milliard de dollars de chiffre d'affaires annuel. Pour mettre ce produit sur le marché rapidement, une nouvelle équipe a été créée et autorisée à opérer en dehors des processus informatiques existants et à explorer l'utilisation des clouds publics. L'équipe initiale comprenait un architecte d'applications, un architecte de systèmes, deux développeurs, un développeur d'automatisation des systèmes, un responsable des opérations et deux employés d'opérations offshore. En utilisant les pratiques DevOps, ils ont pu livrer Hosted LabVIEW sur le marché en moitié moins de temps que leurs introductions de produits normales.

À l'autre extrémité du spectre se trouvent les projets DevOps brownfield, qui sont des produits ou services existants déjà en service depuis des années, voire des décennies. Les projets brownfield viennent souvent avec une dette technique importante, comme l'absence d'automatisation des tests ou l'exécution sur des plateformes non prises en charge. Dans l'exemple de Nordstrom présenté plus tôt dans ce chapitre, les systèmes de restaurants en magasin et les systèmes d'e-commerce étaient des projets brownfield.

Bien que beaucoup croient que DevOps est principalement pour les projets greenfield, DevOps a été utilisé pour transformer avec succès des projets brownfield de toutes sortes. En fait, plus de 60 % des histoires de transformation partagées au DevOps Enterprise Summit en 2014 concernaient des projets brownfield. Dans ces cas, il y avait un écart de performance important entre ce dont le client avait besoin et ce que l'organisation livrait actuellement, et les transformations DevOps ont créé des bénéfices commerciaux considérables.

En effet, l'un des résultats du rapport 2015 State of DevOps a validé que l'âge de l'application n'était pas un prédicteur significatif de performance ; au contraire, ce qui prédisait la performance était de savoir si l'application était architecturée (ou pouvait être réarchitecturée) pour la testabilité et la déployabilité.

Les équipes supportant les projets brownfield peuvent être très réceptives à l'expérimentation avec DevOps, particulièrement lorsqu'il y a une croyance répandue que les méthodes traditionnelles sont insuffisantes pour atteindre leurs objectifs, et surtout s'il y a un fort sentiment d'urgence autour de la nécessité d'amélioration.

Lors de la transformation des projets brownfield, nous pouvons faire face à des obstacles et problèmes significatifs, surtout lorsqu'il n'existe pas de tests automatisés ou lorsqu'il y a une architecture étroitement couplée qui empêche les petites équipes de développer, tester et déployer du code de manière indépendante. Comment nous surmontons ces problèmes est discuté tout au long de ce livre.

Exemples de Transformations Brownfield Réussies

CSG (2013) : En 2013, CSG International avait un chiffre d'affaires de 747 millions de dollars et plus de 3 500 employés, permettant à plus de quatre-vingt-dix mille agents de service client de fournir des opérations de facturation et de l'assistance client à plus de cinquante millions de clients vidéo, voix et données, exécutant plus de six milliards de transactions et imprimant et envoyant par la poste plus de soixante-dix millions de relevés de factures papier chaque mois. Leur portée initiale d'amélioration était l'impression des factures, l'un de leurs principaux secteurs d'activité, et impliquait une application COBOL mainframe et les vingt plateformes technologiques environnantes. Dans le cadre de leur transformation, ils ont commencé à effectuer des déploiements quotidiens dans un environnement de type production et ont doublé la fréquence des versions clients de deux fois par an à quatre fois par an. En conséquence, ils ont considérablement augmenté la fiabilité de l'application et réduit les délais de déploiement de code de deux semaines à moins d'un jour.

Etsy (2009) : En 2009, Etsy comptait trente-cinq employés et générait 87 millions de dollars de chiffre d'affaires, mais après avoir "à peine survécu à la saison des fêtes", ils ont commencé à transformer pratiquement tous les aspects du fonctionnement de l'organisation, transformant

finalement l'entreprise en l'une des organisations DevOps les plus admirées et posant les bases d'une introduction en bourse réussie en 2015.

Considérer les Systèmes d'Enregistrement et les Systèmes d'Engagement

Le cabinet de recherche Gartner a récemment popularisé la notion de l'informatique bimodale, faisant référence au large spectre de services que les entreprises typiques prennent en charge. Dans l'informatique bimodale, il existe des systèmes d'enregistrement, les systèmes de type ERP qui gèrent nos activités (par exemple, MRP, systèmes de reporting RH, financiers), où l'exactitude des transactions et des données est primordiale ; et des systèmes d'engagement, qui sont des systèmes orientés vers les clients ou les employés, comme les systèmes d'e-commerce et les applications de productivité.

Les systèmes d'enregistrement ont généralement un rythme de changement plus lent et ont souvent des exigences réglementaires et de conformité (par exemple, SOX). Gartner appelle ces types de systèmes "Type 1", où l'organisation se concentre sur "bien faire les choses".

Les systèmes d'engagement ont généralement un rythme de changement beaucoup plus rapide pour soutenir des boucles de rétroaction rapides, permettant de mener des expérimentations afin de découvrir comment répondre au mieux aux besoins des clients. Gartner appelle ces types de systèmes « Type 2 », où l'organisation se concentre sur « faire vite ».

Il peut être pratique de diviser nos systèmes en ces catégories ; cependant, nous savons que le conflit chronique entre « bien faire les choses » et « faire vite » peut être résolu avec DevOps. Les données des rapports State of DevOps de Puppet Labs - suivant les leçons de la fabrication Lean - montrent que les organisations à haute performance sont capables de délivrer simultanément des niveaux élevés de throughput et de fiabilité.

De plus, en raison de l'interdépendance de nos systèmes, notre capacité à apporter des modifications à l'un de ces systèmes est limitée par le système le plus difficile à changer en toute sécurité, qui est presque toujours un système d'enregistrement.

Scott Prugh, VP of Product Development chez CSG, a observé : « Nous avons adopté une philosophie qui rejette l'informatique bimodale, car chacun de nos clients mérite la rapidité et la qualité. Cela signifie que nous avons besoin d'excellence technique, que l'équipe supporte une application mainframe vieille de 30 ans, une application Java ou une application mobile. »

Par conséquent, lorsque nous améliorons les systèmes brownfield, nous ne devrions pas seulement chercher à réduire leur complexité et à améliorer leur fiabilité et leur stabilité, mais aussi à les rendre plus rapides, plus sûrs et plus faciles à changer. Même lorsque de nouvelles fonctionnalités sont ajoutées uniquement aux systèmes d'engagement greenfield, elles causent

souvent des problèmes de fiabilité dans les systèmes d'enregistrement brownfield dont elles dépendent. En rendant ces systèmes en aval plus sûrs à changer, nous aidons toute l'organisation à atteindre plus rapidement et en toute sécurité ses objectifs.

Commencer avec les Groupes les Plus Sympathiques et Innovants

Au sein de chaque organisation, il y aura des équipes et des individus avec une large gamme d'attitudes envers l'adoption de nouvelles idées. Geoffrey A. Moore a d'abord représenté ce spectre sous la forme du cycle de vie d'adoption de la technologie dans *Crossing The Chasm*, où le gouffre représente la difficulté classique de toucher des groupes au-delà des innovateurs et des premiers adoptants.

En d'autres termes, les nouvelles idées sont souvent rapidement adoptées par les innovateurs et les premiers adoptants, tandis que d'autres avec des attitudes plus conservatrices les résistent (la majorité précoce, la majorité tardive et les retardataires). Notre objectif est de trouver ces équipes qui croient déjà en la nécessité des principes et pratiques DevOps, et qui possèdent le désir et la capacité démontrée d'innover et d'améliorer leurs propres processus. Idéalement, ces groupes seront des supporters enthousiastes du voyage DevOps.

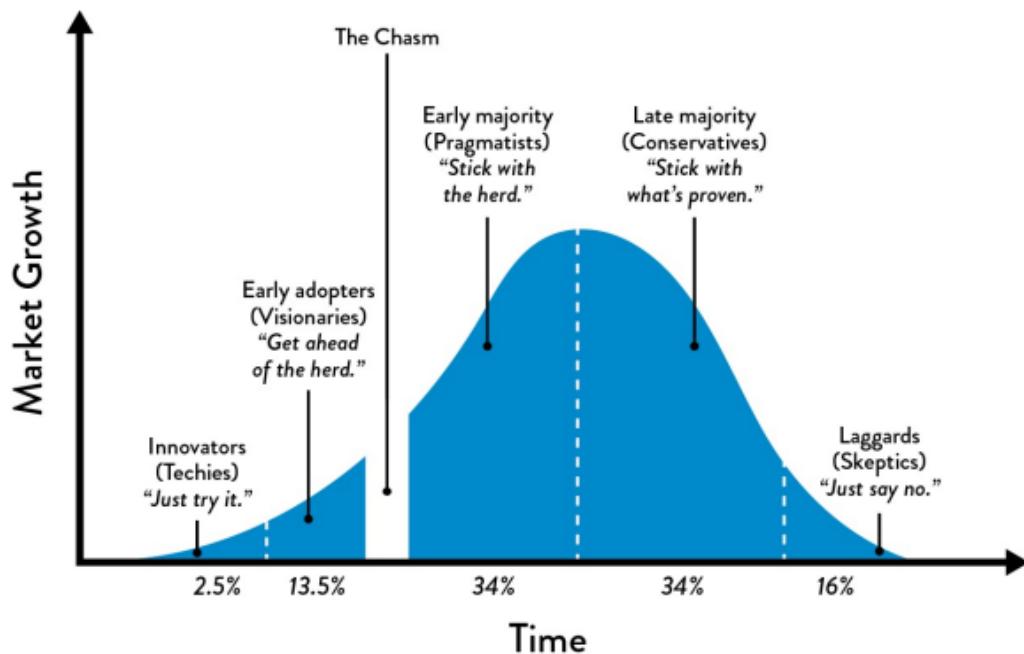


Figure 9: The Technology Adoption Curve (Source: Moore and McKenna, Crossing The Chasm, 15.)

Dans les premières étapes, nous ne passerons pas beaucoup de temps à essayer de convertir les groupes plus conservateurs. Au lieu de cela, nous concentrerons notre énergie à créer des succès avec les groupes moins réticents au risque et à partir de là, nous élargirons notre base

(un processus qui sera discuté plus en détail dans la section suivante). Même si nous avons les plus hauts niveaux de parrainage exécutif, nous éviterons l'approche de grande envergure (c'est-à-dire commencer partout en même temps), en choisissant plutôt de concentrer nos efforts sur quelques domaines de l'organisation, en veillant à ce que ces initiatives soient réussies, puis en nous étendant à partir de là.

Étendre le DevOps dans toute notre organisation

Quelle que soit la portée de notre effort initial, nous devons démontrer des victoires précoces et diffuser nos succès. Nous faisons cela en divisant nos objectifs d'amélioration plus larges en petites étapes incrémentielles. Cela non seulement crée nos améliorations plus rapidement, mais nous permet également de découvrir quand nous avons fait le mauvais choix de chaîne de valeur - en détectant nos erreurs tôt, nous pouvons rapidement revenir en arrière et essayer à nouveau, en prenant des décisions différentes avec nos nouvelles connaissances.

Au fur et à mesure que nous générerons des succès, nous gagnons le droit d'étendre la portée de notre initiative DevOps. Nous voulons suivre une séquence sûre qui augmente méthodiquement nos niveaux de crédibilité, d'influence et de soutien. La liste suivante, adaptée d'un cours enseigné par le Dr Roberto Fernandez, professeur de gestion William F. Pounds au MIT, décrit les phases idéales utilisées par les agents de changement pour construire et étendre leur coalition et leur base de soutien :

- Trouver les innovateurs et les premiers adoptants : Au début, nous concentrons nos efforts sur les équipes qui veulent vraiment aider - ce sont nos esprits apparentés et compagnons de voyage qui sont les premiers à se porter volontaires pour commencer le voyage DevOps. Idéalement, ce sont aussi des personnes respectées et exerçant une grande influence sur le reste de l'organisation, ce qui donne plus de crédibilité à notre initiative.
- Construire une masse critique et une majorité silencieuse : Dans la phase suivante, nous cherchons à étendre les pratiques DevOps à plus d'équipes et de chaînes de valeur avec pour objectif de créer une base de soutien stable. En travaillant avec des équipes réceptives à nos idées, même si elles ne sont pas les groupes les plus visibles ou influents, nous élargissons notre coalition qui génère plus de succès, créant un « effet de bande » qui augmente encore notre influence. Nous contournons spécifiquement les batailles politiques dangereuses qui pourraient compromettre notre initiative.
- Identifier les réfractaires : Les « réfractaires » sont les détracteurs influents et très en vue qui sont les plus susceptibles de résister (et peut-être même de saboter) nos efforts. En général, nous nous attaquons à ce groupe seulement après avoir atteint une majorité silencieuse, lorsque nous avons établi suffisamment de succès pour protéger efficacement notre initiative.

Élargir DevOps dans une organisation n'est pas une tâche facile. Cela peut créer des risques pour les individus, les départements et l'organisation dans son ensemble. Mais comme l'a dit Ron van Kemenade, CIO d'ING, qui a contribué à transformer l'organisation en l'une des organisations technologiques les plus admirées : « Diriger le changement nécessite du courage, surtout dans les environnements corporatifs où les gens ont peur et vous combattent. Mais si vous commencez petit, vous n'avez vraiment rien à craindre. Tout leader doit être suffisamment courageux pour affecter des équipes à prendre des risques calculés. »

Conclusion

Peter Drucker, un leader dans le développement de l'éducation en gestion, a observé que « les petits poissons apprennent à devenir de gros poissons dans de petits étangs. » En choisissant soigneusement où et comment commencer, nous sommes capables d'expérimenter et d'apprendre dans des domaines de notre organisation qui créent de la valeur sans mettre en danger le reste de l'organisation. En faisant cela, nous construisons notre base de soutien, gagnons le droit d'étendre l'utilisation de DevOps dans notre organisation et obtenons la reconnaissance et la gratitude d'une base de plus en plus large.

Comprendre le travail dans notre flux de valeur, le rendre visible et l'étendre à l'ensemble de l'organisation

Une fois que nous avons identifié un flux de valeur auquel nous voulons appliquer les principes et les modèles DevOps, notre prochaine étape consiste à acquérir une compréhension suffisante de la manière dont la valeur est livrée au client : quel travail est effectué, par qui, et quelles mesures pouvons-nous prendre pour améliorer le flux.

Dans le chapitre précédent, nous avons appris la transformation DevOps menée par Courtney Kissler et l'équipe de Nordstrom. Au fil des années, ils ont découvert que l'une des manières les plus efficaces de commencer à améliorer n'importe quel flux de valeur est d'organiser un atelier avec toutes les parties prenantes majeures et de réaliser un exercice de cartographie de flux de valeur - un processus (décrit plus tard dans ce chapitre) conçu pour aider à capturer toutes les étapes nécessaires à la création de valeur.

L'exemple préféré de Kissler des informations précieuses et inattendues pouvant découler de la cartographie des flux de valeur est lorsqu'ils ont essayé d'améliorer les longs délais associés aux demandes passant par l'application du Cosmetics Business Office, une application mainframe COBOL qui supportait tous les responsables de rayon et de département de leurs départements beauté et cosmétiques en magasin.

Cette application permettait aux responsables de département d'enregistrer de nouveaux vendeurs pour diverses lignes de produits présentes dans leurs magasins, afin qu'ils puissent suivre les commissions de vente, activer les remises des fournisseurs, etc.

Kissler a expliqué :

« Je connaissais bien cette application mainframe particulière — au début de ma carrière, je soutenais cette équipe technologique, donc je sais de première main que pendant près d'une décennie, lors de chaque cycle de planification annuelle, nous débattions de la nécessité de retirer cette application du mainframe. Bien sûr, comme dans la plupart des organisations, même avec un soutien total de la direction, nous n'avons jamais semblé trouver le temps de la migrer.

Mon équipe voulait mener un exercice de cartographie de flux de valeur pour déterminer si l'application COBOL était réellement le problème, ou s'il y avait un problème plus important que nous devions aborder. Ils ont organisé un atelier qui a réuni tous ceux ayant une responsabilité dans la livraison de valeur à nos clients internes, y compris nos partenaires commerciaux, l'équipe mainframe, les équipes de services partagés, etc.

Ce qu'ils ont découvert, c'est que lorsque les responsables de département soumettaient le formulaire de demande de « répartition de la ligne de produits », nous leur demandions un numéro d'employé, qu'ils n'avaient pas - ils laissaient donc le champ vide ou mettaient quelque chose comme « je ne sais pas ». Pire encore, pour remplir le formulaire, les responsables de département devaient quitter l'étage du magasin pour utiliser un PC dans l'arrière-boutique. Le résultat final était tout ce temps perdu, avec du travail qui faisait des allers-retours dans le processus. »

Pendant l'atelier, les participants ont mené plusieurs expériences, notamment la suppression du champ numéro d'employé dans le formulaire et la transmission de cette information à une autre étape en aval. Ces expériences, menées avec l'aide des responsables de département, ont montré une réduction de quatre jours du temps de traitement. L'équipe a ensuite remplacé l'application PC par une application iPad, permettant aux responsables de soumettre les informations nécessaires sans quitter l'étage du magasin, et le temps de traitement a été encore réduit à quelques secondes.

Elle a dit fièrement : « Avec ces améliorations incroyables, toutes les demandes de retrait de cette application du mainframe ont disparu. De plus, d'autres dirigeants d'entreprise ont pris note et ont commencé à venir nous voir avec toute une liste d'autres expériences qu'ils voulaient mener avec nous dans leurs propres organisations. Tout le monde dans les équipes business et technologiques était enthousiasmé par le résultat parce qu'ils ont résolu un véritable problème commercial et, surtout, ils ont appris quelque chose dans le processus. »

Dans le reste de ce chapitre, nous passerons en revue les étapes suivantes : identifier toutes les équipes nécessaires pour créer de la valeur client, créer une carte de flux de valeur pour rendre visible tout le travail nécessaire, et l'utiliser pour guider les équipes sur la manière de mieux et plus rapidement créer de la valeur. En faisant cela, nous pouvons reproduire les résultats incroyables décrits dans cet exemple de Nordstrom.

Identification des équipes soutenant notre flux de valeur

Comme le montre l'exemple de Nordstrom, dans les flux de valeur de toute complexité, personne ne connaît tous les travaux qui doivent être effectués pour créer de la valeur pour le client. Cela est d'autant plus vrai que le travail requis doit être réalisé par de nombreuses équipes différentes, souvent très éloignées les unes des autres, que ce soit dans les organigrammes, géographiquement ou par les incitations.

Par conséquent, après avoir sélectionné une application ou un service candidat pour notre initiative DevOps, nous devons identifier tous les membres du flux de valeur responsables de travailler ensemble pour créer de la valeur pour les clients servis. En général, cela inclut :

Propriétaire du produit : la voix interne de l'entreprise qui définit le prochain ensemble de fonctionnalités dans le service.

Développement : l'équipe responsable du développement des fonctionnalités de l'application dans le service.

Assurance qualité (QA) : l'équipe responsable de garantir que des boucles de rétroaction existent pour assurer que le service fonctionne comme prévu.

Opérations : l'équipe souvent responsable de la maintenance de l'environnement de production et de l'aide pour assurer que les niveaux de service requis sont atteints.

Sécurité de l'information (Infosec) : l'équipe responsable de la sécurisation des systèmes et des données.

Gestionnaires de version : les personnes responsables de la gestion et de la coordination des processus de déploiement et de livraison en production.

Cadres technologiques ou gestionnaires de flux de valeur : dans la littérature Lean, quelqu'un responsable de "garantir que le flux de valeur répond ou dépasse les exigences du client [et de l'organisation] pour l'ensemble du flux de valeur, du début à la fin".

Créer une carte de flux de valeur pour voir le travail

Après avoir identifié les membres de notre flux de valeur, notre prochaine étape est de comprendre concrètement comment le travail est effectué, documenté sous la forme d'une carte de flux de valeur. Dans notre flux de valeur, le travail commence probablement avec le propriétaire du produit, sous la forme d'une demande client ou de la formulation d'une hypothèse commerciale. Un peu plus tard, ce travail est accepté par l'équipe de développement, où les fonctionnalités sont implémentées dans le code et enregistrées dans notre dépôt de contrôle de version. Les builds sont ensuite intégrés, testés dans un environnement similaire à la production, et enfin déployés en production, où ils (idéalement) créent de la valeur pour notre client.

Dans de nombreuses organisations traditionnelles, ce flux de valeur comprendra des centaines, voire des milliers d'étapes, nécessitant le travail de centaines de personnes. Comme documenter une carte de flux de valeur aussi complexe prend probablement plusieurs jours, nous pouvons organiser un atelier de plusieurs jours, où nous rassemblons tous les acteurs clés et les éloignons des distractions de leur travail quotidien.

Notre objectif n'est pas de documenter chaque étape et les minuties associées, mais de comprendre suffisamment les domaines de notre flux de valeur qui mettent en péril nos objectifs de flux rapide, de délais courts et de résultats fiables pour les clients. Idéalement, nous avons rassemblé les personnes ayant l'autorité de changer leur portion du flux de valeur.

Damon Edwards, co-animateur du podcast DevOps Café, a observé : "D'après mon expérience, ce type d'exercice de cartographie des flux de valeur est toujours une révélation. Souvent, c'est la première fois que les gens voient combien de travail et de prouesses sont nécessaires pour livrer de la valeur au client. Pour les opérations, c'est peut-être la première fois qu'ils voient les conséquences qui résultent lorsque les développeurs n'ont pas accès à des environnements correctement configurés, ce qui contribue à encore plus de travail fou lors des déploiements de code. Pour les développeurs, c'est peut-être la première fois qu'ils voient toutes les prouesses nécessaires par les équipes de test et des opérations pour déployer leur code en production, longtemps après qu'ils ont marqué une fonctionnalité comme 'terminée'!"

En utilisant l'étendue complète des connaissances apportées par les équipes engagées dans le flux de valeur, nous devons concentrer notre investigation et notre examen sur les domaines suivants :

- Les endroits où le travail doit attendre des semaines ou même des mois, comme l'obtention d'environnements similaires à la production, les processus d'approbation des changements ou les processus de révision de la sécurité.
- Les endroits où des reprises significatives sont générées ou reçues.

Notre premier passage de la documentation de notre flux de valeur doit seulement consister en des blocs de processus de haut niveau. Typiquement, même pour des flux de valeur complexes, les groupes peuvent créer un diagramme avec cinq à quinze blocs de processus en quelques heures. Chaque bloc de processus doit inclure le temps de traitement et le temps de processus pour qu'un élément de travail soit traité, ainsi que le %C/A tel que mesuré par les consommateurs en aval de la sortie.

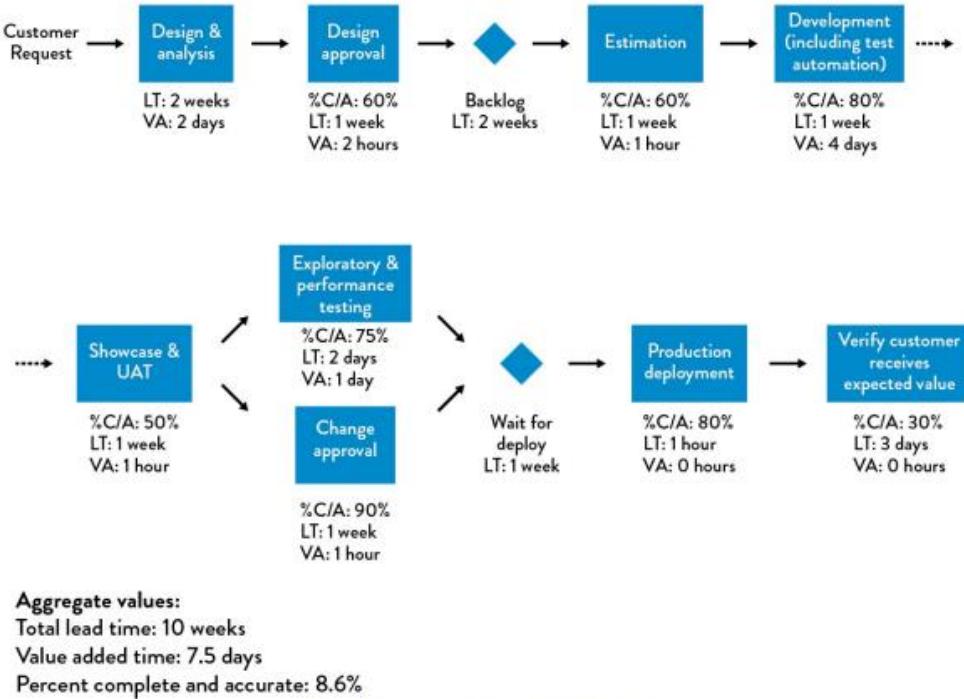


Figure 10: An example of a value stream map
 (Source: Humble, Molesky, and O'Reilly, Lean Enterprise, 139.)

Nous utilisons les métriques de notre carte de flux de valeur pour guider nos efforts d'amélioration

Dans l'exemple de Nordstrom, ils se sont concentrés sur les faibles taux de %C/A (pourcentage de complétiion/acceptation) des formulaires de demande soumis par les responsables de département en raison de l'absence de numéros d'employé. Dans d'autres cas, il peut s'agir de longs délais de traitement ou de faibles taux de %C/A lors de la livraison d'environnements de test correctement configurés aux équipes de développement, ou encore des longs délais nécessaires pour exécuter et réussir les tests de régression avant chaque version logicielle.

Une fois que nous avons identifié la métrique que nous souhaitons améliorer, nous devons effectuer le prochain niveau d'observations et de mesures pour mieux comprendre le problème, puis construire une carte de flux de valeur future idéalisée, qui sert de condition cible à atteindre d'ici une certaine date (par exemple, généralement entre trois et douze mois). La direction aide à définir cet état futur, puis guide et permet à l'équipe de proposer des hypothèses et des contre-mesures pour atteindre l'amélioration souhaitée de cet état, de réaliser des expériences pour tester ces hypothèses et d'interpréter les résultats pour déterminer si les hypothèses étaient correctes. Les équipes continuent de répéter et d'itérer, en utilisant toutes les nouvelles connaissances pour informer les prochaines expériences.

Créer une équipe de transformation dédiée

Un des défis inhérents aux initiatives telles que les transformations DevOps est qu'elles sont inévitablement en conflit avec les opérations commerciales en cours. Cela découle naturellement de l'évolution des entreprises prospères. Une organisation qui a réussi pendant

une période prolongée (années, décennies ou même siècles) a créé des mécanismes pour perpétuer les pratiques qui les ont rendues prospères, telles que le développement de produits, l'administration des commandes et les opérations de la chaîne d'approvisionnement.

De nombreuses techniques sont utilisées pour perpétuer et protéger le fonctionnement des processus actuels, comme la spécialisation, la focalisation sur l'efficacité et la répétabilité, les bureaucraties qui imposent des processus d'approbation et des contrôles pour se protéger contre les variations. En particulier, les bureaucraties sont incroyablement résilientes et conçues pour survivre à des conditions défavorables - on peut enlever la moitié des bureaucrates, et le processus survivra encore.

Bien que cela soit bénéfique pour préserver le statu quo, nous devons souvent changer notre façon de travailler pour nous adapter aux conditions changeantes du marché. Pour ce faire, il faut de la disruption et de l'innovation, ce qui nous met en conflit avec les groupes responsables des opérations quotidiennes et les bureaucraties internes, qui gagneront presque toujours.

Dans leur livre « The Other Side of Innovation: Solving the Execution Challenge », le Dr Vijay Govindarajan et le Dr Chris Trimble, tous deux membres du corps professoral de la Tuck School of Business de Dartmouth College, ont décrit leurs études sur la manière dont l'innovation disruptive est réalisée malgré ces puissantes forces des opérations quotidiennes. Ils ont documenté comment les produits d'assurance automobile orientés client ont été développés et commercialisés avec succès chez Allstate, comment l'activité de publication numérique rentable a été créée au Wall Street Journal, le développement de la chaussure de trail révolutionnaire chez Timberland, et le développement de la première voiture électrique chez BMW.

Selon leurs recherches, le Dr Govindarajan et le Dr Trimble affirment que les organisations doivent créer une équipe de transformation dédiée capable de fonctionner en dehors du reste de l'organisation responsable des opérations quotidiennes (qu'ils appellent respectivement l'"équipe dédiée" et le "moteur de performance").

Tout d'abord, nous tiendrons cette équipe dédiée responsable de l'obtention d'un résultat de niveau système clairement défini et mesurable (par exemple, réduire de 50 % le délai de déploiement depuis "le code validé dans le contrôle de version jusqu'à son exécution réussie en production"). Pour exécuter une telle initiative, nous faisons ce qui suit :

- Assigner aux membres de l'équipe dédiée la responsabilité exclusive des efforts de transformation DevOps (au lieu de "maintenir toutes vos responsabilités actuelles, mais consacrer 20 % de votre temps à cette nouvelle chose DevOps").
- Sélectionner des membres de l'équipe qui sont des généralistes, ayant des compétences dans une grande variété de domaines.
- Sélectionner des membres de l'équipe qui ont des relations de longue date et mutuellement respectueuses avec le reste de l'organisation.

- Créer un espace physique séparé pour l'équipe dédiée, si possible, pour maximiser le flux de communication au sein de l'équipe et créer une certaine isolation par rapport au reste de l'organisation.

Si possible, libérer l'équipe de transformation de nombreuses règles et politiques qui restreignent le reste de l'organisation, comme l'a fait National Instruments, décrit dans le chapitre précédent. Après tout, les processus établis sont une forme de mémoire institutionnelle - nous avons besoin que l'équipe dédiée crée les nouveaux processus et apprentissages nécessaires pour générer les résultats souhaités, créant ainsi une nouvelle mémoire institutionnelle.

Créer une équipe dédiée est non seulement bénéfique pour l'équipe, mais aussi pour le moteur de performance. En créant une équipe distincte, nous créons l'espace nécessaire pour qu'ils expérimentent de nouvelles pratiques, protégeant le reste de l'organisation des perturbations et distractions potentielles associées.

S'accorder sur un objectif commun

L'une des parties les plus importantes de toute initiative d'amélioration est de définir un objectif mesurable avec une date limite clairement définie, entre six mois et deux ans dans le futur. Cet objectif doit nécessiter un effort considérable, mais rester atteignable. La réalisation de cet objectif doit apporter une valeur évidente à l'organisation dans son ensemble ainsi qu'à nos clients.

Ces objectifs et le calendrier doivent être approuvés par les dirigeants et connus de tous dans l'organisation. Nous devons également limiter le nombre de ces types d'initiatives en cours simultanément pour éviter de surcharger la capacité de gestion du changement organisationnel des dirigeants et de l'organisation. Voici quelques exemples d'objectifs d'amélioration :

- Réduire de 50 % la part du budget consacrée au support produit et au travail imprévu.
- Assurer un délai de mise en production de moins d'une semaine pour 95 % des modifications.
- Garantir que les mises en production peuvent toujours être effectuées pendant les heures ouvrables normales sans temps d'arrêt.
- Intégrer tous les contrôles de sécurité de l'information nécessaires dans le pipeline de déploiement pour répondre à toutes les exigences de conformité requises.

Une fois l'objectif de haut niveau clarifié, les équipes doivent décider d'une cadence régulière pour mener à bien le travail d'amélioration. Tout comme le travail de développement produit, nous voulons que le travail de transformation soit effectué de manière itérative et incrémentale. Une itération typique dure de deux à quatre semaines. Pour chaque itération, les équipes doivent s'accorder sur un petit ensemble d'objectifs qui génèrent de la valeur et font progresser vers l'objectif à long terme. À la fin de chaque itération, les équipes doivent examiner leurs progrès et fixer de nouveaux objectifs pour la prochaine itération.

Garder des horizons de planification courts

Dans tout projet de transformation DevOps, nous devons garder des horizons de planification courts, comme si nous étions une startup réalisant un développement produit ou client. Notre initiative doit s'efforcer de générer des améliorations mesurables ou des données exploitables en quelques semaines (ou, dans le pire des cas, en quelques mois).

En gardant nos horizons de planification et nos intervalles d'itération courts, nous atteignons les objectifs suivants :

- Flexibilité et capacité de reprioriser et de replanifier rapidement
- Réduction du délai entre le travail accompli et l'amélioration réalisée, ce qui renforce notre boucle de rétroaction, rendant plus probable le renforcement des comportements souhaités - lorsque les initiatives d'amélioration réussissent, cela encourage davantage d'investissements
- Apprentissage plus rapide généré dès la première itération, ce qui signifie une intégration plus rapide de nos apprentissages dans la prochaine itération
- Réduction de l'énergie d'activation nécessaire pour obtenir des améliorations
- Réalisation plus rapide des améliorations qui font une différence significative dans notre travail quotidien
- Moins de risque que notre projet soit annulé avant que nous puissions générer des résultats démontrables

Réserver 20 % des cycles aux exigences non fonctionnelles et à la réduction de la dette technique

Un problème commun à tout effort d'amélioration des processus est de savoir comment le prioriser correctement - après tout, les organisations qui en ont le plus besoin sont celles qui ont le moins de temps à consacrer à l'amélioration. Cela est particulièrement vrai dans les organisations technologiques en raison de la dette technique.

Les organisations qui luttent contre la dette financière ne font que payer les intérêts sans jamais réduire le principal du prêt, et peuvent éventuellement se retrouver dans des situations où elles ne peuvent plus payer les intérêts. De même, les organisations qui ne réduisent pas leur dette technique peuvent se retrouver si accablées par des solutions de contournement quotidiennes pour des problèmes non résolus qu'elles ne peuvent plus accomplir de nouveaux travaux. En d'autres termes, elles ne font maintenant que payer les intérêts de leur dette technique.

Nous allons gérer activement cette dette technique en veillant à investir au moins 20 % de tous les cycles de développement et d'exploitation dans le refactoring, l'investissement dans le travail d'automatisation et l'architecture, et les exigences non fonctionnelles (NFR, parfois appelées les "ilities"), telles que la maintenabilité, la gérabilité, l'évolutivité, la fiabilité, la testabilité, la déployabilité et la sécurité.

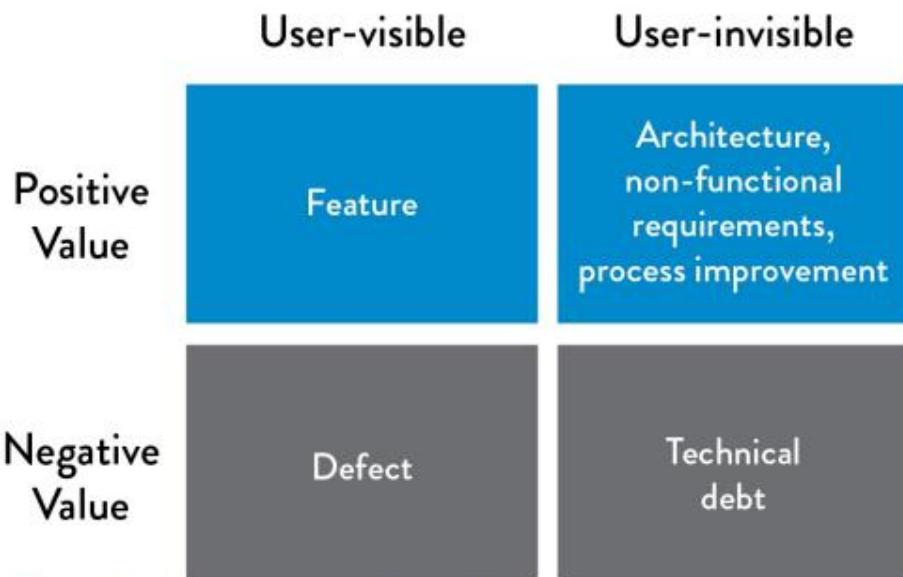


Figure 11: Invest 20% of cycles on those that create positive, user-invisible value

(Source: "Machine Learning and Technical Debt with D. Sculley," Software

Engineering Daily podcast, November 17, 2015,

<http://softwareengineeringdaily.com/2015/11/17/machine-learning-and-technical-debt-with-d-sculley/.>)

Après l'expérience proche de la mort d'eBay à la fin des années 1990, Marty Cagan, auteur de « Inspired: How To Create Products Customers Love », un livre de référence sur la conception et la gestion de produits, a codifié la leçon suivante :

L'accord entre les propriétaires de produits et l'ingénierie est le suivant : la gestion des produits prélève 20 % de la capacité de l'équipe dès le départ et la donne à l'ingénierie pour qu'ils l'utilisent comme ils le jugent nécessaire. Ils peuvent l'utiliser pour réécrire, réarchitecturer ou refactoriser les parties problématiques de la base de code... tout ce qu'ils estiment nécessaire pour éviter d'avoir à dire un jour à l'équipe : "nous devons arrêter et réécrire tout notre code." Si vous êtes dans une situation vraiment critique aujourd'hui, vous pourriez avoir besoin de consacrer 30 % ou plus des ressources. Cependant, je m'inquiète lorsque je trouve des équipes qui pensent pouvoir se contenter de beaucoup moins que 20 %.

Cagan note que lorsque les organisations ne paient pas leur "taxe de 20 %", la dette technique augmente au point où une organisation finit par consacrer tous ses cycles à rembourser cette dette technique. À un certain moment, les services deviennent si fragiles que la livraison de nouvelles fonctionnalités s'arrête parce que tous les ingénieurs travaillent sur des problèmes de fiabilité ou contournent des problèmes existants.

En consacrant 20 % de nos cycles pour que les équipes de développement et d'exploitation puissent créer des contre-mesures durables aux problèmes rencontrés dans notre travail

quotidien, nous nous assurons que la dette technique n'entrave pas notre capacité à développer et à exploiter rapidement et en toute sécurité nos services en production. Réduire la pression ajoutée par la dette technique peut également diminuer les niveaux de burn-out des travailleurs.

Étude de Cas - L'opération InVersion chez LinkedIn (2011)

L'opération InVersion de LinkedIn présente une étude de cas intéressante qui illustre la nécessité de rembourser la dette technique dans le cadre du travail quotidien. Six mois après leur introduction en bourse réussie en 2011, LinkedIn continuait à lutter avec des déploiements problématiques qui devenaient si pénibles qu'ils ont lancé l'opération InVersion, où ils ont arrêté tout développement de nouvelles fonctionnalités pendant deux mois afin de réviser leurs environnements informatiques, déploiements et architectures.

LinkedIn a été créé en 2003 pour aider les utilisateurs à "se connecter à leur réseau pour de meilleures opportunités d'emploi." À la fin de leur première semaine d'exploitation, ils avaient 2 700 membres. Un an plus tard, ils avaient plus d'un million de membres, et ont depuis lors connu une croissance exponentielle. En novembre 2015, LinkedIn comptait plus de 350 millions de membres, générant des dizaines de milliers de requêtes par seconde, ce qui se traduit par des millions de requêtes par seconde sur les systèmes backend de LinkedIn.

Depuis le début, LinkedIn fonctionnait principalement sur leur application maison Leo, une application Java monolithique qui servait chaque page via des servlets et gérait les connexions JDBC à diverses bases de données Oracle backend. Cependant, pour faire face à l'augmentation du trafic dans leurs premières années, deux services critiques ont été découplés de Leo : le premier traitait des requêtes concernant le graphe de connexion des membres entièrement en mémoire, et le second était la recherche de membres, qui se superposait au premier.

En 2010, la plupart des nouveaux développements se faisaient dans de nouveaux services, avec près d'une centaine de services fonctionnant en dehors de Leo. Le problème était que Leo n'était déployé qu'une fois toutes les deux semaines.

Josh Clemm, un senior engineering manager chez LinkedIn, a expliqué qu'en 2010, l'entreprise rencontrait des problèmes importants avec Leo. Malgré une mise à l'échelle verticale de Leo en ajoutant de la mémoire et des CPU, "Leo tombait souvent en panne en production, il était difficile à dépanner et à récupérer, et difficile de déployer du nouveau code.... Il était clair que nous devions 'tuer Leo' et le diviser en de nombreux petits services fonctionnels et sans état."

En 2013, la journaliste Ashlee Vance de Bloomberg a décrit comment "quand LinkedIn essayait d'ajouter un tas de nouvelles choses en même temps, le site s'effondrait dans un désordre total, obligeant les ingénieurs à travailler tard dans la nuit pour résoudre les problèmes." À l'automne 2011, les nuits tardives n'étaient plus un rite de passage ou une activité de cohésion, car les problèmes étaient devenus intolérables. Certains des meilleurs ingénieurs de LinkedIn, y compris Kevin Scott, qui avait rejoint LinkedIn en tant que VP de l'ingénierie trois mois avant leur introduction en bourse, ont décidé d'arrêter complètement le travail d'ingénierie sur de nouvelles fonctionnalités et de consacrer tout le département à la réparation de l'infrastructure principale du site. Ils ont appelé cet effort l'opération InVersion.

Scott a lancé l'opération InVersion comme un moyen « d'injecter les débuts d'un manifeste culturel dans la culture d'ingénierie de son équipe. Il n'y aurait pas de développement de nouvelles fonctionnalités jusqu'à ce que l'architecture informatique de LinkedIn soit révisée - c'est ce dont l'entreprise et son équipe avaient besoin. »

Scott a décrit un inconvénient : « Vous entrez en bourse, avez tout le monde qui vous observe, et ensuite nous disons à la direction que nous n'allons rien livrer de nouveau pendant que toute l'ingénierie travaille sur ce projet [InVersion] pendant les deux prochains mois. C'était effrayant. »

Cependant, Vance a décrit les résultats massivement positifs de l'opération InVersion. « LinkedIn a créé une suite complète de logiciels et d'outils pour l'aider à développer du code pour le site. Au lieu d'attendre des semaines pour que leurs nouvelles fonctionnalités atteignent le site principal de LinkedIn, les ingénieurs pouvaient développer un nouveau service, faire examiner le code par une série de systèmes automatisés pour détecter les bogues et les problèmes que le service pourrait avoir en interagissant avec les fonctionnalités existantes, et le lancer directement sur le site LinkedIn en direct... Le corps d'ingénierie de LinkedIn [désormais] effectue des mises à niveau majeures du site trois fois par jour. » En créant un système de travail plus sûr, la valeur créée incluait moins de sessions de travail nocturnes, laissant plus de temps pour développer de nouvelles fonctionnalités innovantes.

Comme l'a décrit Josh Clemm dans son article sur la mise à l'échelle chez LinkedIn, « La mise à l'échelle peut être mesurée sur de nombreuses dimensions, y compris l'organisationnelle.... [L'opération InVersion] a permis à l'ensemble de l'organisation d'ingénierie de se concentrer sur l'amélioration des outils et du déploiement, de l'infrastructure et de la productivité des développeurs. Elle a réussi à permettre l'agilité d'ingénierie dont nous avons besoin pour construire les nouveaux produits évolutifs que nous avons aujourd'hui.... [En] 2010, nous avions déjà plus de 150 services distincts. Aujourd'hui, nous avons plus de 750 services. »

Kevin Scott a déclaré : "Votre travail en tant qu'ingénieur et votre objectif en tant qu'équipe technologique est d'aider votre entreprise à gagner. Si vous dirigez une équipe d'ingénieurs, il est préférable d'adopter la perspective d'un PDG. Votre travail consiste à déterminer ce dont votre

entreprise, votre marché, votre environnement concurrentiel ont besoin. Appliquez cela à votre équipe d'ingénierie pour que votre entreprise gagne. »

En permettant à LinkedIn de rembourser près d'une décennie de dette technique, l'opération InVersion a permis de garantir stabilité et sécurité, tout en préparant la prochaine phase de croissance de l'entreprise. Cependant, cela a nécessité deux mois de focalisation totale sur les exigences non fonctionnelles, au détriment de toutes les fonctionnalités promises aux marchés publics lors d'une introduction en bourse.

En trouvant et en résolvant les problèmes dans le cadre de notre travail quotidien, nous gérons notre dette technique de manière à éviter ces expériences de "mort imminente".

Augmenter la visibilité du travail

Afin de pouvoir savoir si nous progressons vers notre objectif, il est essentiel que chaque membre de l'organisation connaisse l'état actuel du travail. Il existe de nombreuses façons de rendre l'état actuel visible, mais ce qui importe le plus, c'est que les informations que nous affichons soient à jour, et que nous révisons constamment ce que nous mesurons pour nous assurer que cela nous aide à comprendre les progrès vers nos conditions cibles actuelles. La section suivante discute des modèles qui peuvent aider à créer de la visibilité et de l'alignement entre les équipes et les fonctions.

Utiliser les outils pour renforcer les comportements souhaités

Comme l'a observé Christopher Little, un cadre du secteur des logiciels et l'un des premiers chroniqueurs du DevOps, "Les anthropologues décrivent les outils comme un artefact culturel. Toute discussion sur la culture après l'invention du feu doit également porter sur les outils." De même, dans le flux de valeur DevOps, nous utilisons des outils pour renforcer notre culture et accélérer les changements de comportement souhaités.

Un objectif est que nos outils renforcent l'idée que le Développement et les Opérations ont non seulement des objectifs communs, mais disposent également d'un backlog de travail commun, idéalement stocké dans un système de travail commun et utilisant un vocabulaire partagé, de sorte que le travail puisse être priorisé au niveau mondial.

En faisant cela, le Développement et les Opérations peuvent finir par créer une file d'attente de travail partagée, au lieu de chaque silo utilisant une file d'attente différente (par exemple, le Développement utilise JIRA tandis que les Opérations utilisent ServiceNow). Un avantage significatif de cela est que lorsque les incidents de production sont affichés dans les mêmes systèmes de travail que le travail de développement, il sera évident quand les incidents en cours devraient interrompre d'autres travaux, surtout lorsque nous avons un tableau kanban.

Un autre avantage de faire en sorte que le Développement et les Opérations utilisent un outil partagé est un backlog uniifié, où tout le monde priorise les projets d'amélioration d'un point de vue global, sélectionnant le travail ayant la plus grande valeur pour l'organisation ou réduisant le plus la dette technique. Au fur et à mesure que nous identifions la dette technique, nous l'ajoutons à notre backlog priorisé si nous ne pouvons pas y répondre immédiatement. Pour les problèmes non résolus, nous pouvons utiliser notre "temps de 20% pour les exigences non fonctionnelles" pour résoudre les principaux éléments de notre backlog.

D'autres technologies qui renforcent les objectifs partagés sont les salles de discussion, telles que les canaux IRC, HipChat, Campfire, Slack, Flowdock et OpenFire. Les salles de discussion permettent le partage rapide d'informations (par opposition au remplissage de formulaires traités par des workflows prédéfinis), la possibilité d'inviter d'autres personnes si nécessaire, et des journaux d'historique qui sont automatiquement enregistrés pour la postérité et peuvent être analysés lors de séances de post-mortem.

Une dynamique incroyable est créée lorsque nous disposons d'un mécanisme qui permet à n'importe quel membre de l'équipe d'aider rapidement les autres membres de l'équipe, voire les personnes extérieures à leur équipe - le temps nécessaire pour obtenir des informations ou un travail nécessaire peut passer de jours à minutes. De plus, comme tout est enregistré, nous n'avons peut-être pas besoin de demander de l'aide à quelqu'un d'autre à l'avenir - nous cherchons simplement.

Cependant, l'environnement de communication rapide facilité par les salles de discussion peut également être un inconvénient. Comme le fait remarquer Ryan Martens, le fondateur et CTO de Rally Software, "Dans une salle de discussion, si quelqu'un n'obtient pas de réponse en quelques minutes, il est totalement accepté et attendu que vous puissiez les relancer jusqu'à ce qu'ils obtiennent ce dont ils ont besoin."

Les attentes de réponse immédiate peuvent bien sûr entraîner des résultats indésirables. Un flot constant d'interruptions et de questions peut empêcher les gens d'accomplir le travail nécessaire. Par conséquent, les équipes peuvent décider que certains types de demandes doivent passer par des outils plus structurés et asynchrones.

Conclusion

Dans ce chapitre, nous avons identifié toutes les équipes soutenant notre flux de valeur et capturé dans une carte de flux de valeur quel travail est requis afin de fournir de la valeur au client. La carte de flux de valeur fournit la base pour comprendre notre état actuel, y compris notre délai d'exécution et nos métriques %C/A pour les zones problématiques, et informe la manière dont nous établissons un état futur. Cela permet aux équipes de transformation dédiées d'itérer rapidement et d'expérimenter pour améliorer les performances. Nous nous assurons également d'allouer une quantité suffisante de temps pour l'amélioration, la résolution des problèmes connus et des problèmes architecturaux, y compris nos exigences non fonctionnelles. Les études de cas de Nordstrom et de LinkedIn démontrent à quel point des améliorations spectaculaires peuvent être apportées en termes de délais d'exécution et de qualité lorsque nous identifions des problèmes dans notre flux de valeur et remboursions la dette technique.

Comment concevoir notre organisation et notre architecture en tenant compte de la loi de Conway

Dans les chapitres précédents, nous avons identifié un flux de valeur pour commencer notre transformation DevOps et établi des objectifs et des pratiques partagés pour permettre à une équipe dédiée à la transformation d'améliorer la manière dont nous apportons de la valeur au client.

Dans ce chapitre, nous allons commencer à réfléchir à la façon de nous organiser pour atteindre au mieux nos objectifs de flux de valeur. Après tout, la façon dont nous organisons nos équipes affecte notre manière de travailler. Le Dr. Melvin Conway a mené une expérience célèbre en 1968 avec une organisation de recherche sous contrat composée de huit personnes, chargées de produire un compilateur COBOL et un compilateur ALGOL. Il a observé : « Après quelques estimations initiales de la difficulté et du temps, cinq personnes ont été affectées au projet COBOL et trois au projet ALGOL. Le compilateur COBOL résultant fonctionnait en cinq phases, tandis que le compilateur ALGOL fonctionnait en trois. »

Ces observations ont conduit à ce qui est maintenant connu sous le nom de loi de Conway, qui stipule que « **les organisations qui conçoivent des systèmes... sont contraintes de produire des conceptions qui sont des copies des structures de communication de ces organisations... Plus une organisation est grande, moins elle est flexible et plus le phénomène est prononcé.** » Eric S. Raymond, auteur du livre The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary, a formulé une version simplifiée (et maintenant plus célèbre) de la loi de Conway dans son Jargon File : « **L'organisation du logiciel et l'organisation de l'équipe de développement logiciel seront congruentes ; souvent énoncé ainsi : 'si vous avez quatre groupes travaillant sur un compilateur, vous obtiendrez un compilateur à quatre passes.'** »

En d'autres termes, la façon dont nous organisons nos équipes a un effet puissant sur les logiciels que nous produisons, ainsi que sur nos résultats architecturaux et de production. Pour obtenir un flux rapide de travail entre le développement et les opérations, avec une haute qualité et de bons résultats pour les clients, nous devons organiser nos équipes et notre travail de manière que la loi de Conway joue en notre faveur. Mal exécutée, la loi de Conway empêchera les équipes de travailler de manière sûre et indépendante ; au lieu de cela, elles seront étroitement liées, attendant toutes les unes des autres pour que le travail soit effectué, avec même de petits changements créant potentiellement des conséquences globales catastrophiques.

Un exemple de la façon dont la loi de Conway peut soit entraver, soit renforcer nos objectifs peut être vu dans une technologie développée chez Etsy appelée Sprouter. Le parcours DevOps d'Etsy a commencé en 2009 et est l'une des organisations DevOps les plus admirées, avec un

chiffre d'affaires de près de 200 millions de dollars en 2014 et une introduction en bourse réussie en 2015.

Développé à l'origine en 2007, Sprouter connectait les personnes, les processus et la technologie d'une manière qui créait de nombreux résultats indésirables. Sprouter, abréviation de "stored procedure router" (routeur de procédure stockée), a été initialement conçu pour faciliter la vie des développeurs et des équipes de base de données. Comme l'a dit Ross Snyder, ingénieur principal chez Etsy, lors de sa présentation à Surge 2011, « Sprouter a été conçu pour permettre aux équipes de développement d'écrire du code PHP dans l'application, aux DBA d'écrire du SQL dans Postgres, avec Sprouter les aidant à se rencontrer au milieu. »

Sprouter résidait entre leur application PHP frontale et la base de données Postgres, centralisant l'accès à la base de données et masquant la mise en œuvre de la base de données de la couche application. Le problème était que l'ajout de toute modification à la logique métier entraînait une friction significative entre les développeurs et les équipes de base de données. Comme l'a observé Snyder, « Pour presque toute nouvelle fonctionnalité du site, Sprouter nécessitait que les DBA écrivent une nouvelle procédure stockée. En conséquence, chaque fois que les développeurs voulaient ajouter de nouvelles fonctionnalités, ils avaient besoin de quelque chose des DBA, ce qui les obligeait souvent à se frayer un chemin à travers une tonne de bureaucratie. » En d'autres termes, les développeurs créant de nouvelles fonctionnalités avaient une dépendance envers l'équipe DBA, ce qui nécessitait d'être priorisé, communiqué et coordonné, entraînant du travail en attente, des réunions, des délais de livraison plus longs, etc. Cela s'explique par le fait que Sprouter créait un couplage étroit entre les équipes de développement et de base de données, empêchant les développeurs de développer, tester et déployer leur code de manière indépendante en production.

De plus, les procédures stockées de la base de données étaient étroitement couplées à Sprouter - chaque fois qu'une procédure stockée était modifiée, cela nécessitait également des modifications de Sprouter. Le résultat était que Sprouter devenait un point de défaillance unique de plus en plus grand. Snyder a expliqué que tout était tellement étroitement couplé et nécessitait un niveau de synchronisation si élevé que presque chaque déploiement provoquait une mini-panne.

Les problèmes associés à Sprouter et leur solution éventuelle peuvent être expliqués par la loi de Conway. Etsy avait initialement deux équipes, les développeurs et les DBA, qui étaient chacune responsables de deux couches du service, la couche de logique applicative et la couche de procédures stockées. Deux équipes travaillant sur deux couches, comme le prédit la loi de Conway. Sprouter était censé faciliter la vie des deux équipes, mais cela n'a pas fonctionné comme prévu - lorsque les règles métier changeaient, au lieu de changer uniquement deux couches, elles devaient maintenant apporter des modifications à trois couches (dans l'application, dans les procédures stockées et maintenant dans Sprouter). Les défis résultant de la coordination et de la priorisation du travail entre trois équipes augmentaient considérablement les délais de livraison et causaient des problèmes de fiabilité.

Au printemps 2009, dans le cadre de ce que Snyder a appelé « la grande transformation culturelle d'Etsy », Chad Dickerson a rejoint l'entreprise en tant que nouveau CTO. Dickerson a mis en œuvre de nombreuses initiatives, y compris un investissement massif dans la stabilité du site, permettant aux développeurs de réaliser leurs propres déploiements en production, ainsi qu'une démarche de deux ans pour éliminer Sprouter.

Pour ce faire, l'équipe a décidé de déplacer toute la logique métier de la couche base de données vers la couche applicative, supprimant ainsi le besoin de Sprouter. Ils ont créé une petite équipe qui a écrit une couche de mapping objet-relationnel (ORM) en PHP, permettant aux développeurs frontend de faire des appels directement à la base de données et réduisant le nombre d'équipes nécessaires pour changer la logique métier de trois équipes à une seule.

Comme l'a décrit Snyder, « Nous avons commencé à utiliser l'ORM pour toutes les nouvelles zones du site et avons migré de petites parties de notre site de Sprouter vers l'ORM au fil du temps. Il nous a fallu deux ans pour migrer l'ensemble du site de Sprouter. Et même si nous nous sommes tous plaints de Sprouter tout le temps, il est resté en production tout au long du processus. » En éliminant Sprouter, ils ont également éliminé les problèmes associés au besoin de coordination entre plusieurs équipes pour les changements de logique métier, réduit le nombre de transferts et augmenté de manière significative la vitesse et le succès des déploiements en production, améliorant ainsi la stabilité du site. De plus, parce que les petites équipes pouvaient développer et déployer leur code de manière indépendante sans nécessiter qu'une autre équipe apporte des modifications dans d'autres parties du système, la productivité des développeurs a augmenté.

Sprouter a finalement été retiré de la production et des dépôts de contrôle de version d'Etsy au début de 2001. Comme l'a dit Snyder, « Wow, ça faisait du bien. »

Comme l'ont vécu Snyder et Etsy, la façon dont nous concevons notre organisation dicte comment le travail est effectué et, par conséquent, les résultats que nous obtenons. Tout au long du reste de ce chapitre, nous explorerons comment la loi de Conway peut avoir un impact négatif sur la performance de notre flux de valeur et, plus important encore, comment nous organisons nos équipes pour utiliser la loi de Conway à notre avantage.

Archétypes organisationnels

Dans le domaine des sciences de la décision, il existe trois principaux types de structures organisationnelles qui influencent la manière dont nous concevons nos flux de valeur DevOps en tenant compte de la loi de Conway : fonctionnelle, matricielle et orientée marché. Ils sont définis par le Dr. Roberto Fernandez comme suit :

Les organisations orientées fonctionnellement optimisent pour l'expertise, la division du travail ou la réduction des coûts. Ces organisations centralisent l'expertise, ce qui aide à favoriser la croissance professionnelle et le développement des compétences, et ont souvent des structures organisationnelles hiérarchiques importantes. C'est la méthode d'organisation prédominante pour les opérations (c'est-à-dire que les administrateurs de serveurs, les administrateurs réseau, les administrateurs de bases de données, etc., sont tous organisés en groupes séparés).

Les organisations orientées matriciellement tentent de combiner l'orientation fonctionnelle et celle du marché. Cependant, comme beaucoup de ceux qui travaillent dans ou dirigent des organisations matricielles l'observent, ces organisations aboutissent souvent à des structures organisationnelles compliquées, comme des contributeurs individuels rendant compte à deux managers ou plus, et atteignent parfois ni les objectifs de l'orientation fonctionnelle ni ceux de l'orientation marché.

Les organisations orientées marché optimisent pour répondre rapidement aux besoins des clients. Ces organisations tendent à être plates, composées de multiples disciplines transversales (par exemple, marketing, ingénierie, etc.), ce qui entraîne souvent des redondances potentielles à travers l'organisation. C'est ainsi que fonctionnent de nombreuses organisations adoptant DevOps - dans des exemples extrêmes, comme chez Amazon ou Netflix, chaque équipe de service est simultanément responsable de la livraison des fonctionnalités et du support des services.

Avec ces trois catégories d'organisations en tête, explorons plus en détail comment une orientation fonctionnelle excessive, en particulier dans les opérations, peut causer des résultats indésirables dans le flux de valeur technologique, comme le prédit la loi de Conway.

Problèmes souvent causés par une orientation fonctionnelle excessive (optimisation des coûts)

Dans les organisations traditionnelles des opérations informatiques, nous utilisons souvent l'orientation fonctionnelle pour organiser nos équipes par spécialités. Nous mettons les administrateurs de bases de données dans un groupe, les administrateurs réseau dans un autre, les administrateurs de serveurs dans un troisième, et ainsi de suite. Une des conséquences les plus visibles de cela est les longs délais d'attente, surtout pour les activités complexes comme les déploiements importants où nous devons ouvrir des tickets avec plusieurs groupes et coordonner les transferts de travail, ce qui entraîne un travail en attente dans de longues files d'attente à chaque étape.

Pour aggraver le problème, la personne qui exécute le travail a souvent peu de visibilité ou de compréhension de la manière dont son travail est lié à des objectifs de flux de valeur (par exemple, « Je configure juste des serveurs parce que quelqu'un me l'a dit. »). Cela place les travailleurs dans un vide de créativité et de motivation.

Le problème est exacerbé lorsque chaque domaine fonctionnel des opérations doit servir plusieurs flux de valeur (c'est-à-dire, plusieurs équipes de développement) qui rivalisent tous pour leurs cycles rares. Pour que les équipes de développement terminent leur travail en temps opportun, nous devons souvent escalader les problèmes à un manager ou un directeur, et finalement à quelqu'un (généralement un cadre) qui peut enfin prioriser le travail par rapport aux objectifs globaux de l'organisation plutôt qu'aux objectifs des silos fonctionnels. Cette décision doit ensuite être transmise dans chacune des zones fonctionnelles pour changer les priorités locales, ce qui ralentit les autres équipes. Lorsque chaque équipe accélère son travail, le résultat net est que chaque projet avance au même rythme lent.

En plus des longues files d'attente et des longs délais, cette situation entraîne de mauvais transferts, de grandes quantités de travail supplémentaire, des problèmes de qualité, des goulots d'étranglement et des retards. Ce blocage empêche d'atteindre des objectifs organisationnels importants, qui dépassent souvent de loin le désir de réduire les coûts.

De même, l'orientation fonctionnelle peut également se retrouver dans les fonctions centralisées de QA et d'Infosec, qui peuvent avoir bien fonctionné (ou du moins suffisamment bien) lors de la réalisation de déploiements logiciels moins fréquents. Cependant, à mesure que nous augmentons le nombre d'équipes de développement et leurs fréquences de déploiement et de sortie, la plupart des organisations orientées fonctionnellement auront du mal à suivre et à fournir des résultats satisfaisants, surtout lorsque leur travail est effectué manuellement. Maintenant, nous allons étudier comment fonctionnent les organisations orientées marché.

Permettre aux équipes orientées marché (optimisation de la vitesse)

De manière générale, pour atteindre les résultats DevOps, nous devons réduire les effets de l'orientation fonctionnelle (« optimisation des coûts ») et permettre l'orientation marché (« optimisation de la vitesse ») afin que nous puissions avoir de nombreuses petites équipes travaillant en toute sécurité et indépendamment, livrant rapidement de la valeur au client.

Portées à l'extrême, les équipes orientées marché sont responsables non seulement du développement des fonctionnalités, mais aussi des tests, de la sécurisation, du déploiement et du support de leur service en production, de la conception de l'idée à son retrait. Ces équipes sont conçues pour être transversales et indépendantes - capables de concevoir et de mener des expériences utilisateur, de créer et de livrer de nouvelles fonctionnalités, de déployer et de gérer leur service en production, et de corriger tout défaut sans dépendances manuelles sur d'autres équipes, leur permettant ainsi de se déplacer plus rapidement. Ce modèle a été adopté par Amazon et Netflix et est présenté par Amazon comme l'une des principales raisons de leur capacité à se déplacer rapidement même en grandissant.

Pour atteindre une orientation marché, nous ne ferons pas une grande réorganisation descendante, qui crée souvent de grandes quantités de perturbations, de peurs et de paralysie. Au lieu de cela, nous intégrerons les ingénieurs et les compétences fonctionnels (par exemple, Ops, QA, Infosec) dans chaque équipe de service, ou fournirons leurs capacités aux équipes par le biais de plateformes automatisées en libre-service qui offrent des environnements semblables à ceux de la production, lancent des tests automatisés ou effectuent des déploiements.

Cela permet à chaque équipe de service de livrer indépendamment de la valeur au client sans avoir à ouvrir des tickets avec d'autres groupes, tels que les opérations informatiques, la QA ou l'Infosec.

Faire fonctionner l'orientation fonctionnelle

Après avoir recommandé des équipes orientées vers le marché, il est important de souligner qu'il est possible de créer des organisations efficaces et à haute vitesse avec une orientation fonctionnelle. Les équipes transversales et orientées vers le marché sont un moyen d'atteindre un flux rapide et une fiabilité, mais elles ne sont pas la seule voie. Nous pouvons également atteindre nos résultats DevOps souhaités grâce à l'orientation fonctionnelle, tant que tout le monde dans le flux de valeur considère les résultats pour le client et l'organisation comme un objectif commun, peu importe où ils se trouvent dans l'organisation.

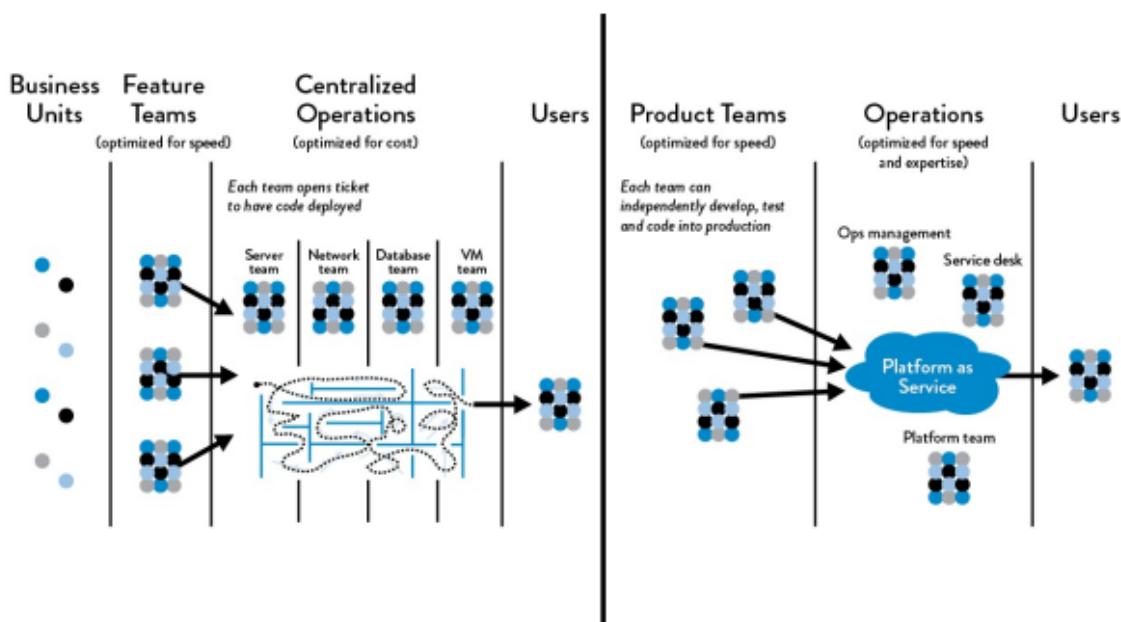


Figure 12: Functional vs. market orientation

Left: Functional orientation: all work flows through centralized IT Operations; Right: Market orientation: all product teams can deploy their loosely-coupled components self-service into production. (Source: Humble, Molesky, and O'Reilly, Lean Enterprise, Kindle edition, 4523 & 4592.)

Par exemple, des performances élevées avec un groupe Operations fonctionnel et centralisé sont possibles, tant que les équipes de service obtiennent de manière fiable et rapide ce dont elles ont besoin de la part des Operations (idéalement à la demande) et vice-versa. De nombreuses organisations DevOps les plus admirées conservent une orientation fonctionnelle des Operations, y compris Etsy, Google et GitHub.

Ce que ces organisations ont en commun, c'est une culture de haute confiance qui permet à tous les départements de travailler ensemble efficacement, où tout le travail est priorisé de manière transparente et où il y a suffisamment de marge dans le système pour permettre aux travaux prioritaires d'être réalisés rapidement. Cela est en partie rendu possible par des plateformes automatisées en libre-service qui intègrent la qualité dans les produits que tout le monde construit.

Dans le mouvement Lean manufacturing des années 1980, de nombreux chercheurs étaient perplexes face à l'orientation fonctionnelle de Toyota, qui était en contradiction avec la meilleure pratique consistant à avoir des équipes transversales et orientées vers le marché. Ils étaient si perplexes qu'ils ont appelé cela le "deuxième paradoxe de Toyota".

Comme l'a écrit Mike Rother dans Toyota Kata, "Aussi tentant que cela puisse paraître, on ne peut pas réorganiser son chemin vers l'amélioration continue et l'adaptabilité. Ce qui est décisif, ce n'est pas la forme de l'organisation, mais la manière dont les gens agissent et réagissent. Les racines du succès de Toyota ne résident pas dans ses structures organisationnelles, mais dans le développement des capacités et des habitudes de ses employés. Cela surprend beaucoup de gens, en fait, de découvrir que Toyota est largement organisé selon un style traditionnel de département fonctionnel." C'est ce développement des habitudes et des capacités des personnes et de la main-d'œuvre qui est au centre de nos prochaines sections.

Tester, opérations et sécurité comme le travail de tous, tous les jours

Dans les organisations à haute performance, tout le monde au sein de l'équipe partage un objectif commun : la qualité, la disponibilité et la sécurité ne sont pas la responsabilité de départements individuels, mais font partie du travail de chacun, chaque jour.

Cela signifie que le problème le plus urgent de la journée peut être de travailler sur ou de déployer une fonctionnalité pour les clients ou de corriger un incident de production de gravité 1. Alternativement, la journée peut nécessiter la révision du changement d'un collègue ingénieur, l'application de correctifs de sécurité d'urgence aux serveurs de production ou la réalisation d'améliorations pour que les collègues ingénieurs soient plus productifs.

En réfléchissant aux objectifs partagés entre le développement et les opérations, Jody Mulkey, CTO de Ticketmaster, a déclaré : "Pendant près de 25 ans, j'ai utilisé une métaphore du football américain pour décrire Dev et Ops. Vous savez, Ops est la défense, qui empêche l'autre équipe

de marquer, et Dev est l'attaque, essayant de marquer des buts. Et un jour, j'ai réalisé à quel point cette métaphore était erronée, car ils ne jouent jamais tous sur le terrain en même temps. Ils ne sont pas réellement dans la même équipe !"

Il a continué : "L'analogie que j'utilise maintenant est que les Ops sont les bloqueurs offensifs, et Dev sont les 'positions de compétence' (comme le quarterback et les receveurs larges) dont le travail est de faire avancer le ballon sur le terrain - le travail des Ops est de s'assurer que Dev a suffisamment de temps pour exécuter correctement les jeux."

Un exemple frappant de la manière dont la douleur partagée peut renforcer les objectifs partagés est lorsque Facebook connaissait une croissance énorme en 2009. Ils rencontraient des problèmes significatifs liés aux déploiements de code - bien que tous les problèmes ne causent pas des problèmes impactant les clients, il y avait des interventions d'urgence chroniques et de longues heures de travail. Pedro Canahuati, leur directeur de l'ingénierie de production, a décrit une réunion pleine d'ingénieurs Ops où quelqu'un a demandé à toutes les personnes ne travaillant pas sur un incident de fermer leurs ordinateurs portables, et personne ne pouvait le faire.

L'une des choses les plus significatives qu'ils ont faites pour aider à changer les résultats des déploiements a été de faire en sorte que tous les ingénieurs de Facebook, les responsables de l'ingénierie et les architectes tournent pour les astreintes des services qu'ils construisaient. En faisant cela, tout le monde travaillant sur le service a ressenti un retour d'expérience concret sur les décisions architecturales et de codage qu'ils prenaient en amont, ce qui a eu un impact positif énorme sur les résultats en aval.

Permettre à chaque membre de l'équipe d'être un généraliste

Dans des cas extrêmes d'une organisation Operations fonctionnellement orientée, nous avons des départements de spécialistes, comme les administrateurs réseau, les administrateurs de stockage, etc. Lorsque les départements se spécialisent excessivement, cela cause la silotisation, que le Dr Spear décrit comme lorsque les départements "opèrent plus comme des états souverains". Toute activité opérationnelle complexe nécessite alors de multiples transferts et files d'attente entre les différentes zones de l'infrastructure, entraînant des délais plus longs (par exemple, parce que chaque changement de réseau doit être effectué par quelqu'un du département réseau).

Parce que nous dépendons d'un nombre toujours croissant de technologies, nous devons avoir des ingénieurs qui se sont spécialisés et ont atteint une maîtrise dans les domaines technologiques dont nous avons besoin. Cependant, nous ne voulons pas créer de spécialistes qui sont "figés dans le temps", ne comprenant et n'étant capables de contribuer qu'à une seule zone du flux de valeur.

Une contre-mesure consiste à permettre et encourager chaque membre de l'équipe à être un généraliste. Nous faisons cela en offrant des opportunités aux ingénieurs d'apprendre toutes les compétences nécessaires pour construire et faire fonctionner les systèmes dont ils sont responsables, et en faisant régulièrement tourner les personnes à travers différents rôles. Le terme "ingénieur full stack" est maintenant couramment utilisé (parfois comme une source riche de parodie) pour décrire des généralistes qui sont familiers - au moins avoir un niveau de compréhension général - avec l'ensemble de la pile applicative (par exemple, le code de l'application, les bases de données, les systèmes d'exploitation, le réseau, le cloud).

"I-shaped" (Specialists)	"T-shaped" (Generalists)	"E-shaped"
Deep expertise in one area	Deep expertise in one area	Deep expertise in a few areas
Very few skills or experience in other areas	Broad skills across many areas	Experience across many areas
		Proven execution skills Always innovating
Creates bottlenecks quickly	Can step up to remove bottlenecks	Almost limitless potential
Insensitive to downstream waste and impact	Sensitive to downstream waste and impact	
Prevents planning flexibility or absorption of variability	Helps make planning flexible and absorbs variability	

(Source: Scott Prugh, "Continuous Delivery," [ScaledAgileFramework.com, February 14, 2013, <http://scaledagileframework.com/continuous-delivery/>.](http://scaledagileframework.com/continuous-delivery/)

Scott Prugh écrit que CSG International a entrepris une transformation qui rassemble la plupart des ressources nécessaires pour construire et gérer le produit au sein d'une seule équipe, incluant l'analyse, l'architecture, le développement, les tests et les opérations. "En formant de manière croisée et en développant les compétences en ingénierie, les généralistes peuvent accomplir des ordres de grandeur plus de travail que leurs homologues spécialistes, et cela améliore également notre flux de travail global en supprimant les files d'attente et les temps d'attente." Cette approche va à l'encontre des pratiques traditionnelles de recrutement, mais, comme l'explique Prugh, cela en vaut la peine.

"Les managers traditionnels s'opposent souvent à l'embauche d'ingénieurs avec des compétences de généralistes, arguant qu'ils sont plus coûteux et que 'je peux embaucher deux administrateurs de serveurs pour chaque ingénieur des opérations aux compétences

"multiples!" Cependant, les avantages commerciaux de permettre un flux de travail plus rapide sont écrasants. De plus, comme le note Prugh, "[I]nvestir dans la formation croisée est la bonne chose à faire pour la croissance de carrière [des employés], et rend le travail de tout le monde plus amusant."

Lorsque nous valorisons les gens uniquement pour leurs compétences existantes ou leur performance dans leur rôle actuel plutôt que pour leur capacité à acquérir et à déployer de nouvelles compétences, nous renforçons (souvent involontairement) ce que le Dr Carol Dweck décrit comme l'état d'esprit fixe, où les gens considèrent leur intelligence et leurs capacités comme des "données" statiques qui ne peuvent pas être changées de manière significative.

Au lieu de cela, nous voulons encourager l'apprentissage, aider les gens à surmonter l'anxiété liée à l'apprentissage, veiller à ce que les personnes aient des compétences pertinentes et une feuille de route de carrière définie, etc. En faisant cela, nous aidons à favoriser un état d'esprit de croissance chez nos ingénieurs - après tout, une organisation apprenante nécessite des personnes prêtes à apprendre. En encourageant tout le monde à apprendre, ainsi qu'en fournissant une formation et un soutien, nous créons la manière la plus durable et la moins coûteuse de créer l'excellence dans nos équipes - en investissant dans le développement des personnes que nous avons déjà.

Comme l'a décrit Jason Cox, directeur de l'ingénierie des systèmes chez Disney, "Au sein des Opérations, nous avons dû changer nos pratiques de recrutement. Nous avons cherché des personnes ayant de la '**curiosité, du courage et de la franchise**', qui étaient non seulement capables d'être des généralistes, mais aussi des renégats... Nous voulons promouvoir la perturbation positive afin que notre entreprise ne reste pas bloquée et puisse avancer vers l'avenir." Comme nous le verrons dans la section suivante, la manière dont nous finançons nos équipes affecte également nos résultats.

Financer non pas des projets, mais des services et des produits

Une autre manière de permettre des résultats performants est de créer des équipes de service stables avec un financement continu pour exécuter leur propre stratégie et feuille de route d'initiatives. Ces équipes disposent des ingénieurs dédiés nécessaires pour honorer des engagements concrets pris envers les clients internes et externes, tels que des fonctionnalités, des récits et des tâches.

Cela contraste avec le modèle plus traditionnel où les équipes de développement et de test sont affectées à un "projet" puis réaffectées à un autre projet dès que le projet est terminé et que le financement est épuisé. Cela conduit à toutes sortes de résultats indésirables, notamment les développeurs étant incapables de voir les conséquences à long terme des décisions qu'ils prennent (une forme de retour d'information) et un modèle de financement qui

ne valorise et ne paie que pour les premières étapes du cycle de vie du logiciel - ce qui, tragiquement, est également la partie la moins coûteuse pour les produits ou services réussis.

Notre objectif avec un modèle de financement basé sur les produits est de valoriser la réalisation des résultats organisationnels et des résultats clients, tels que les revenus, la valeur à vie des clients ou le taux d'adoption des clients, idéalement avec un minimum de production (par exemple, quantité d'effort ou de temps, lignes de code). Cela contraste avec la manière dont les projets sont généralement mesurés, comme s'ils ont été achevés dans les limites du budget, du temps et de la portée promis.

Concevoir les limites des équipes selon la loi de Conway

À mesure que les organisations grandissent, l'un des plus grands défis est de maintenir une communication et une coordination efficaces entre les personnes et les équipes. Trop souvent, lorsque les personnes et les équipes se trouvent à un étage différent, dans un bâtiment différent ou dans un fuseau horaire différent, créer et maintenir une compréhension partagée et une confiance mutuelle devient plus difficile, entravant une collaboration efficace. La collaboration est également entravée lorsque les principaux mécanismes de communication sont des tickets de travail et des demandes de changement, ou pire, lorsque les équipes sont séparées par des frontières contractuelles, comme lorsque le travail est effectué par une équipe externalisée.

Comme nous l'avons vu dans l'exemple de Etsy Sprouter au début de ce chapitre, la façon dont nous organisons les équipes peut créer de mauvais résultats, un effet secondaire de la loi de Conway. Cela inclut le fractionnement des équipes par fonction (par exemple, en mettant les développeurs et les testeurs dans des emplacements différents ou en externalisant complètement les testeurs) ou par couche architecturale (par exemple, application, base de données).

Ces configurations nécessitent une communication et une coordination importantes entre les équipes, mais entraînent toujours une grande quantité de retouches, de désaccords sur les spécifications, de mauvaises transmissions et des personnes attendant sans rien faire que quelqu'un d'autre intervienne. Idéalement, notre architecture logicielle devrait permettre à de petites équipes d'être indépendamment productives, suffisamment découpées les unes des autres pour que le travail puisse être effectué sans communication et coordination excessives ou inutiles.

Créer des architectures découpées pour favoriser la productivité et la sécurité des développeurs

Quand nous avons une architecture fortement couplée, de petits changements peuvent entraîner des défaillances à grande échelle. En conséquence, toute personne travaillant sur une

partie du système doit constamment se coordonner avec les autres personnes travaillant sur d'autres parties du système qu'elles peuvent affecter, incluant la navigation dans des processus de gestion des changements complexes et bureaucratiques.

De plus, tester que l'ensemble du système fonctionne ensemble nécessite l'intégration des changements avec ceux de centaines, voire de milliers d'autres développeurs, qui peuvent à leur tour avoir des dépendances sur des dizaines, des centaines ou des milliers de systèmes interconnectés. Les tests sont effectués dans des environnements de test d'intégration rares, qui nécessitent souvent des semaines pour être obtenus et configurés. Le résultat est non seulement des délais longs pour les changements (généralement mesurés en semaines ou en mois) mais aussi une faible productivité des développeurs et de mauvais résultats de déploiement.

En revanche, lorsque nous avons une architecture qui permet à de petites équipes de développeurs de mettre en œuvre, tester et déployer du code en production de manière sûre et rapide, nous pouvons augmenter et maintenir la productivité des développeurs et améliorer les résultats des déploiements. Ces caractéristiques peuvent être trouvées dans les architectures orientées services (SOA) décrites pour la première fois dans les années 1990, dans lesquelles les services sont testables et déployables indépendamment. Une caractéristique clé des SOA est qu'ils sont composés de services faiblement couplés avec des contextes délimités.

Avoir une architecture faiblement couplée signifie que les services peuvent être mis à jour en production indépendamment, sans avoir à mettre à jour d'autres services. Les services doivent être découplés des autres services et, tout aussi important, des bases de données partagées (bien qu'ils puissent partager un service de base de données, à condition de ne pas avoir de schémas communs).

Les contextes délimités sont décrits dans le livre Domain Driven Design d'Eric J. Evans. L'idée est que les développeurs devraient être capables de comprendre et de mettre à jour le code d'un service sans connaître quoi que ce soit sur les internes de ses services pairs. Les services interagissent avec leurs pairs strictement par le biais d'API et ne partagent donc pas de structures de données, de schémas de base de données ou d'autres représentations internes d'objets. Les contextes délimités garantissent que les services sont compartimentés et ont des interfaces bien définies, ce qui facilite également les tests.

Randy Shoup, ancien directeur de l'ingénierie pour Google App Engine, a observé que "les organisations avec ces types d'architectures orientées services, telles que Google et Amazon, ont une flexibilité et une évolutivité incroyables. Ces organisations comptent des dizaines de milliers de développeurs où de petites équipes peuvent encore être incroyablement productives."

Maintenir la taille des équipes petites (la règle de l'équipe de deux pizzas)

La loi de Conway nous aide à concevoir les limites de nos équipes dans le contexte des schémas de communication souhaités, mais elle nous encourage également à maintenir la taille de nos équipes petite, réduisant la quantité de communication inter-équipes et nous encourageant à garder le périmètre de chaque domaine d'équipe petit et délimité.

Dans le cadre de son initiative de transformation pour s'éloigner d'une base de code monolithique en 2002, Amazon a utilisé la règle des deux pizzas pour maintenir la taille des équipes petite - une équipe seulement aussi grande que peut être nourrie avec deux pizzas - généralement environ cinq à dix personnes.

Cette limite de taille a quatre effets importants :

- Elle garantit que l'équipe a une compréhension claire et partagée du système sur lequel elle travaille. À mesure que les équipes grandissent, la quantité de communication requise pour que tout le monde sache ce qui se passe augmente de manière combinatoire.
- Elle limite le taux de croissance du produit ou du service sur lequel on travaille. En limitant la taille de l'équipe, nous limitons le taux auquel leur système peut évoluer. Cela aide également à garantir que l'équipe maintient une compréhension partagée du système.
- Elle décentralise le pouvoir et permet l'autonomie. Chaque équipe de deux pizzas (2PT) est aussi autonome que possible. Le responsable de l'équipe, travaillant avec l'équipe exécutive, décide de la métrique commerciale clé dont l'équipe est responsable, connue sous le nom de fonction de fitness, qui devient le critère d'évaluation global pour les expériences de l'équipe. L'équipe est alors capable d'agir de manière autonome pour maximiser cette métrique.
- Diriger une 2PT est un moyen pour les employés d'acquérir de l'expérience en leadership dans un environnement où l'échec n'a pas de conséquences catastrophiques. Un élément essentiel de la stratégie d'Amazon était le lien entre la structure organisationnelle d'une 2PT et l'approche architecturale d'une architecture orientée services.

Le CTO d'Amazon, Werner Vogels, a expliqué les avantages de cette structure à Larry Dignan de Baseline en 2005. Dignan écrit :

"Les petites équipes sont rapides... et ne se laissent pas submerger par ce qu'on appelle l'administrivia... Chaque groupe affecté à un domaine d'activité particulier en est entièrement responsable... L'équipe évalue la solution, la conçoit, la construit, la met en œuvre et surveille son utilisation continue. De cette manière, les programmeurs et architectes technologiques obtiennent des retours directs des personnes d'affaires qui utilisent leur code ou leurs applications - lors de réunions régulières et de conversations informelles."

Un autre exemple de la manière dont l'architecture peut améliorer profondément la productivité est le programme d'activation des API chez Target, Inc.

Étude de cas : La mise en place d'API chez Target (2015)

Target, sixième plus grand détaillant aux États-Unis, dépense plus d'un milliard de dollars en technologie chaque année. Heather Mickman, directrice du développement pour Target, décrit les débuts de leur parcours DevOps : « À l'époque, il fallait dix équipes différentes pour provisionner un serveur chez Target, et lorsque des problèmes survenaient, nous arrêtons de faire des changements pour éviter d'aggraver les choses, ce qui empirait tout. »

Les difficultés pour obtenir des environnements et effectuer des déploiements compliquaient le travail des équipes de développement, tout comme l'accès aux données nécessaires. Mickman explique que les données essentielles, comme les informations sur les stocks et les prix, étaient enfermées dans des systèmes hérités et des mainframes avec des sources multiples de vérité, notamment entre le commerce électronique et les magasins physiques.

Pour résoudre ce problème de données, Mickman a dirigé l'équipe d'habilitation des API en 2012 pour permettre aux équipes de développement de « livrer de nouvelles capacités en quelques jours au lieu de mois. » Ils voulaient que toute équipe d'ingénierie de Target puisse obtenir et stocker les données nécessaires.

Mickman explique que pour réussir, ils avaient besoin d'une équipe capable d'effectuer le travail, sans le confier à des sous-traitants. Ils ont pris en charge les exigences opérationnelles, introduit de nouveaux outils pour l'intégration et la livraison continues, et utilisé des technologies comme la base de données Cassandra et le courtier de messages Kafka.

En deux ans, l'équipe d'habilitation des API a permis cinquante-trois nouvelles capacités commerciales, y compris Ship to Store et Gift Registry, ainsi que leurs intégrations avec Instacart et Pinterest. En 2014, ils géraient plus de 1,5 milliard d'appels API par mois, chiffre qui a atteint 17 milliards par mois en 2015.

Ces changements ont généré d'importants bénéfices pour Target, augmentant les ventes numériques de 42 % pendant les fêtes de 2014 et de 32 % au deuxième trimestre. En 2015, leur objectif était de permettre à 450 de leurs 1 800 magasins de traiter les commandes de commerce électronique, contre cent auparavant.

Mickman conclut : « L'équipe d'habilitation des API montre ce qu'une équipe de passionnés du changement peut accomplir. » Cela les prépare à la prochaine étape : étendre DevOps à toute l'organisation technologique.

Conclusion

Les études de cas d'Etsy et Target montrent comment l'architecture et la conception organisationnelle peuvent améliorer nos résultats. Mal appliquée, la loi de Conway entraînera de mauvais résultats, empêchant la sécurité et l'agilité. Bien appliquée, elle permettra aux développeurs de développer, tester et déployer en toute sécurité et de manière indépendante, apportant de la valeur au client.

Comment obtenir d'excellents résultats en intégrant les opérations dans le travail quotidien du développement

Notre objectif est de permettre des résultats orientés vers le marché où de nombreuses petites équipes peuvent rapidement et indépendamment apporter de la valeur aux clients. Cela peut être difficile à réaliser lorsque les opérations sont centralisées et orientées fonctionnellement, devant répondre aux besoins de nombreuses équipes de développement aux besoins potentiellement très différents. Le résultat peut souvent être des délais de réalisation longs pour les travaux nécessaires aux Ops, des priorisations et des escalades constantes, et de mauvais résultats de déploiement.

Nous pouvons créer des résultats plus orientés vers le marché en intégrant mieux les capacités des Ops dans les équipes de développement, rendant les deux plus efficaces et productives. Dans ce chapitre, nous explorerons de nombreuses façons d'y parvenir, à la fois au niveau organisationnel et à travers des rituels quotidiens. En faisant cela, les Ops peuvent améliorer significativement la productivité des équipes de développement dans toute l'organisation, ainsi que permettre une meilleure collaboration et de meilleurs résultats organisationnels.

Chez Big Fish Games, qui développe et soutient des centaines de jeux mobiles et des milliers de jeux PC et a réalisé plus de 266 millions de dollars de revenus en 2013, Paul Farrall, vice-président des opérations informatiques, avait la charge de l'organisation centralisée des opérations. Il était responsable du soutien de nombreuses unités commerciales différentes ayant une grande autonomie.

Chacune de ces unités commerciales avait des équipes de développement dédiées qui choisissaient souvent des technologies très différentes. Lorsque ces groupes voulaient déployer de nouvelles fonctionnalités, ils devaient se disputer une réserve commune de ressources Ops rares. De plus, tout le monde avait du mal avec des environnements de test et d'intégration peu fiables, ainsi que des processus de libération extrêmement lourds.

Farrall pensait que la meilleure façon de résoudre ce problème était d'intégrer l'expertise des Ops dans les équipes de développement. Il a observé : « Lorsque les équipes de développement avaient des problèmes de test ou de déploiement, elles avaient besoin de plus que de la technologie ou des environnements. Elles avaient également besoin d'aide et de coaching. Au début, nous avons intégré des ingénieurs et des architectes Ops dans chacune des équipes de développement, mais il n'y avait tout simplement pas assez d'ingénieurs Ops pour couvrir autant d'équipes. Nous avons pu aider davantage d'équipes avec ce que nous appelions un modèle de liaison Ops et avec moins de personnes. »

Farrall a défini deux types de liaisons Ops : le gestionnaire de relations commerciales et l'ingénieur de libération dédié. Les gestionnaires de relations commerciales travaillaient avec la

gestion des produits, les propriétaires d'unités commerciales, la gestion de projet, la gestion du développement et les développeurs. Ils devenaient intimement familiers avec les moteurs commerciaux des groupes de produits et les feuilles de route des produits, agissaient en tant qu'avocats des propriétaires de produits au sein des opérations, etaidaient leurs équipes de produits à naviguer dans le paysage des opérations pour prioriser et rationaliser les demandes de travail.

De même, l'ingénieur de libération dédié devenait intimement familier avec les problèmes de développement et de QA du produit, et les aidait à obtenir ce dont ils avaient besoin de l'organisation Ops pour atteindre leurs objectifs. Ils étaient familiers avec les demandes typiques de développement et de QA pour les Ops, et exécutaient souvent eux-mêmes le travail nécessaire. Au besoin, ils faisaient également appel à des ingénieurs Ops techniques dédiés (par exemple, DBA, Infosec, ingénieurs de stockage, ingénieurs réseau), et aidaient à déterminer quels outils en libre-service l'ensemble du groupe des opérations devrait prioriser.

En faisant cela, Farrall a pu aider les équipes de développement de toute l'organisation à devenir plus productives et à atteindre leurs objectifs d'équipe. De plus, il a aidé les équipes à prioriser autour de ses contraintes globales des Ops, réduisant le nombre de surprises découvertes en cours de projet et augmentant finalement le débit global du projet.

Farrall note que les relations de travail avec les Ops et la vitesse de libération du code se sont nettement améliorées à la suite des changements. Il conclut : « Le modèle de liaison Ops nous a permis d'intégrer l'expertise des opérations informatiques dans les équipes de développement et de produit sans ajouter de nouveaux effectifs. »

La transformation DevOps chez Big Fish Games montre comment une équipe d'opérations centralisée a pu atteindre les résultats généralement associés aux équipes orientées vers le marché. Nous pouvons employer les trois stratégies larges suivantes :

- Créer des capacités en libre-service pour permettre aux développeurs des équipes de service d'être productifs.
- Intégrer des ingénieurs Ops dans les équipes de service.
- Assigner des liaisons Ops aux équipes de service lorsque l'intégration des Ops n'est pas possible.

Enfin, nous décrivons comment les ingénieurs Ops peuvent s'intégrer aux rituels des équipes de développement utilisés dans leur travail quotidien, y compris les stand-ups quotidiens, la planification et les rétrospectives.

Créer des services partagés pour améliorer la productivité des équipes de développement

Une façon de permettre des résultats orientés vers le marché consiste pour les opérations à créer un ensemble de plateformes centralisées et de services d'outils que toute équipe de développement peut utiliser pour devenir plus productive, tels que l'obtention d'environnements similaires à la production, des pipelines de déploiement, des outils de test automatisés, des tableaux de bord de télémétrie de production, etc.

En faisant cela, nous permettons aux équipes de développement de passer plus de temps à construire des fonctionnalités pour leurs clients, plutôt qu'à obtenir toute l'infrastructure nécessaire pour livrer et supporter ces fonctionnalités en production.

Toutes les plateformes et services que nous fournissons devraient idéalement être automatisés et disponibles à la demande, sans nécessiter qu'un développeur ouvre un ticket et attende que quelqu'un effectue le travail manuellement. Cela garantit que les opérations ne deviennent pas un goulet d'étranglement pour leurs clients (par exemple, "Nous avons reçu votre demande de travail, et cela prendra six semaines pour configurer manuellement ces environnements de test.").

En procédant ainsi, nous permettons aux équipes produits d'obtenir ce dont elles ont besoin, quand elles en ont besoin, et réduisons la nécessité de communications et de coordination. Comme l'a observé Damon Edwards, "Sans ces plateformes d'opérations en libre-service, le cloud n'est que l'hébergement coûteux 2.0."

Dans presque tous les cas, nous ne rendrons pas obligatoire pour les équipes internes d'utiliser ces plateformes et services - ces équipes de plateformes devront conquérir et satisfaire leurs clients internes, parfois même en compétition avec des fournisseurs externes. En créant ce marché interne efficace de capacités, nous aidons à garantir que les plateformes et services que nous créons soient le choix le plus facile et le plus attrayant disponible (le chemin de la moindre résistance).

Par exemple, nous pouvons créer une plateforme qui fournit un référentiel de contrôle de version partagé avec des bibliothèques de sécurité pré-validées, un pipeline de déploiement qui exécute automatiquement des outils de contrôle de qualité du code et de sécurité, et qui déploie nos applications dans des environnements connus et bons qui ont déjà des outils de surveillance de production installés. Idéalement, nous rendons la vie tellement plus facile pour les équipes de développement qu'elles décideront massivement que l'utilisation de notre plateforme est le moyen le plus simple, le plus sûr et le plus sécurisé de mettre leurs applications en production.

Nous intégrons dans ces plateformes l'expérience cumulative et collective de tous les membres de l'organisation, y compris la QA, les opérations et la sécurité de l'information, ce qui aide à créer un système de travail toujours plus sûr. Cela augmente la productivité des développeurs et facilite l'utilisation par les équipes produits de processus communs, tels que l'exécution de tests automatisés et la satisfaction des exigences de sécurité et de conformité.

Créer et maintenir ces plateformes et outils est un véritable développement de produit - les clients de notre plateforme ne sont pas nos clients externes mais nos équipes de développement internes. Comme pour la création de tout excellent produit, la création de plateformes géniales que tout le monde aime ne se fait pas par accident. Une équipe de plateforme interne avec un mauvais sens du client créera probablement des outils que tout le monde détestera et abandonnera rapidement pour d'autres alternatives, que ce soit pour une autre équipe de plateforme interne ou un fournisseur externe.

Dianne Marsh, Directrice des outils d'ingénierie chez Netflix, déclare que la mission de son équipe est de "soutenir l'innovation et la vitesse de nos équipes d'ingénierie. Nous ne construisons ou ne déployons rien pour ces équipes, ni ne gérons leurs configurations. Au lieu de cela, nous construisons des outils pour permettre l'auto-service. Il est acceptable que les gens dépendent de nos outils, mais il est important qu'ils ne deviennent pas dépendants de nous."

Souvent, ces équipes de plateformes fournissent d'autres services pour aider leurs clients à apprendre leur technologie, migrer d'autres technologies, et même offrir du coaching et du conseil pour aider à éléver l'état des pratiques au sein de l'organisation. Ces services partagés facilitent également la standardisation, permettant aux ingénieurs de devenir rapidement productifs, même s'ils changent d'équipe. Par exemple, si chaque équipe produit choisit une chaîne d'outils différente, les ingénieurs peuvent devoir apprendre un ensemble de technologies entièrement nouveau pour effectuer leur travail, mettant les objectifs de l'équipe avant les objectifs globaux.

Dans les organisations où les équipes ne peuvent utiliser que des outils approuvés, nous pouvons commencer par supprimer cette exigence pour quelques équipes, comme l'équipe de transformation, afin que nous puissions expérimenter et découvrir quelles capacités rendent ces équipes plus productives.

Les équipes de services partagés internes doivent continuellement rechercher des chaînes d'outils internes qui sont largement adoptées dans l'organisation, décidant lesquelles il est logique de soutenir de manière centralisée et de mettre à la disposition de tous. En général, prendre quelque chose qui fonctionne déjà quelque part et étendre son utilisation est beaucoup plus susceptible de réussir que de construire ces capacités à partir de zéro.

Intégrer des ingénieurs Ops dans nos équipes de service

Une autre façon de permettre des résultats plus orientés vers le marché consiste à permettre aux équipes produits de devenir plus autonomes en intégrant des ingénieurs Ops en leur sein, réduisant ainsi leur dépendance à l'égard des opérations centralisées. Ces équipes produits peuvent également être complètement responsables de la livraison et du support des services.

En intégrant des ingénieurs Ops dans les équipes de développement, leurs priorités sont presque entièrement dictées par les objectifs des équipes produits dans lesquelles ils sont intégrés - contrairement aux Ops qui se concentrent sur la résolution de leurs propres problèmes internes. En conséquence, les ingénieurs Ops deviennent plus étroitement liés à leurs clients internes et externes. De plus, les équipes produits ont souvent le budget pour financer l'embauche de ces ingénieurs Ops, bien que les décisions d'entretien et d'embauche soient probablement toujours prises par le groupe centralisé des opérations, pour garantir la cohérence et la qualité du personnel.

Jason Cox a déclaré, "Dans de nombreuses parties de Disney, nous avons intégré des ingénieurs Ops (ingénieurs système) dans les équipes produits de nos unités commerciales, ainsi que dans le développement, les tests et même la sécurité de l'information. Cela a totalement changé la dynamique de notre travail. En tant qu'ingénieurs des opérations, nous créons les outils et les capacités qui transforment la façon dont les gens travaillent et même leur façon de penser. Dans les Ops traditionnels, nous nous contentions de conduire le train que quelqu'un d'autre avait construit. Mais dans l'ingénierie des opérations moderne, nous aidons non seulement à construire le train, mais aussi les ponts sur lesquels les trains roulent."

Pour les nouveaux projets de développement de grande envergure, nous pouvons initialement intégrer des ingénieurs Ops dans ces équipes. Leur travail peut inclure l'aide à la décision de ce qu'il faut construire et comment le construire, influencer l'architecture des produits, aider à influencer les choix technologiques internes et externes, aider à créer de nouvelles capacités dans nos plateformes internes et peut-être même générer de nouvelles capacités opérationnelles. Après que le produit est mis en production, les ingénieurs Ops intégrés peuvent aider aux responsabilités de production de l'équipe de développement.

Ils participeront à tous les rituels de l'équipe de développement, tels que les réunions de planification, les stand-ups quotidiens et les démonstrations où l'équipe présente de nouvelles fonctionnalités et décide lesquelles expédier. À mesure que le besoin de connaissances et de capacités Ops diminue, les ingénieurs Ops peuvent passer à différents projets ou engagements, suivant le schéma général selon lequel la composition au sein des équipes produits change tout au long de son cycle de vie.

Ce paradigme a un autre avantage important : le jumelage des ingénieurs Dev et Ops est un moyen extrêmement efficace de former de manière croisée les connaissances et l'expertise des opérations dans une équipe de service. Cela peut également avoir le puissant avantage de

transformer les connaissances des opérations en code automatisé qui peut être beaucoup plus fiable et largement réutilisé.

Assigner un Liaison Ops à Chaque Équipe de Service

Pour diverses raisons, comme les coûts et la rareté des ressources, nous ne pouvons pas toujours intégrer des ingénieurs Ops dans chaque équipe produit. Cependant, nous pouvons obtenir de nombreux avantages similaires en assignant un liaison désigné à chaque équipe produit.

Chez Etsy, ce modèle est appelé « Ops désigné ». Leur groupe centralisé des opérations continue de gérer tous les environnements - pas seulement les environnements de production mais aussi les environnements de préproduction - pour aider à garantir leur cohérence. L'ingénieur Ops désigné est responsable de comprendre :

- Quelle est la nouvelle fonctionnalité du produit et pourquoi nous la construisons.
- Comment elle fonctionne en termes d'opérabilité, de scalabilité et de visibilité (la création de diagrammes est fortement encouragée).
- Comment surveiller et collecter des métriques pour garantir le progrès, le succès ou l'échec de la fonctionnalité.
- Toute déviation par rapport aux architectures et modèles précédents, et les justifications pour celles-ci.
- Tous les besoins supplémentaires en infrastructure et comment l'utilisation affectera la capacité de l'infrastructure.
- Les plans de lancement des fonctionnalités.

De plus, comme dans le modèle des Ops intégrés, cette liaison assiste aux réunions debout de l'équipe, intégrant leurs besoins dans la feuille de route des opérations et exécutant toute tâche nécessaire. Nous comptons sur ces liaisons pour escalader tout conflit de ressources ou de priorités. En faisant cela, nous identifions tous les conflits de ressources ou de temps qui devraient être évalués et priorisés dans le contexte des objectifs organisationnels plus larges.

Assigner des liaisons Ops nous permet de soutenir plus d'équipes produit que le modèle des Ops intégrés. Notre objectif est de garantir que les Ops ne soient pas une contrainte pour les équipes produit. Si nous constatons que les liaisons Ops sont trop étirées, empêchant les équipes produit d'atteindre leurs objectifs, nous devrons probablement soit réduire le nombre d'équipes soutenues par chaque liaison, soit intégrer temporairement un ingénieur Ops dans certaines équipes.

Intégrer les Ops dans les rituels Dev

Lorsque les ingénieurs Ops sont intégrés ou assignés en tant que liaisons dans nos équipes produit, nous pouvons les intégrer dans nos rituels d'équipe Dev. Dans cette section, notre

objectif est d'aider les ingénieurs Ops et autres non-développeurs à mieux comprendre la culture de développement existante et à s'intégrer de manière proactive dans tous les aspects de la planification et du travail quotidien. En conséquence, les Ops sont mieux en mesure de planifier et de diffuser toute connaissance nécessaire dans les équipes produit, influençant le travail bien avant qu'il n'atteigne la production. Les sections suivantes décrivent certains des rituels standard utilisés par les équipes de développement utilisant des méthodes agiles et comment nous intégrerions les ingénieurs Ops en leur sein. Les pratiques agiles ne sont en aucun cas une condition préalable à cette étape - en tant qu'ingénieurs Ops, notre objectif est de découvrir quels rituels suivent les équipes produit, de nous y intégrer et d'y ajouter de la valeur.

Inviter les Ops à nos Réunions Debout

Un des rituels Dev popularisés par Scrum est la réunion debout quotidienne, une réunion rapide où chaque membre de l'équipe présente trois choses : ce qui a été fait hier, ce qui sera fait aujourd'hui et ce qui empêche l'avancement du travail.

Le but de cette cérémonie est de diffuser l'information à travers l'équipe et de comprendre le travail en cours et à venir. En ayant les membres de l'équipe qui présentent cette information, nous apprenons quels sont les blocages et découvrons des moyens de nous entraider pour avancer dans notre travail. De plus, avec la présence des managers, nous pouvons rapidement résoudre les conflits de priorités et de ressources.

Un problème courant est que cette information est compartimentée au sein de l'équipe de développement. En ayant les ingénieurs Ops présents, les opérations peuvent mieux comprendre les activités de l'équipe de développement, permettant une meilleure planification et préparation. Par exemple, si nous découvrons que l'équipe produit planifie un déploiement de grande envergure dans deux semaines, nous pouvons nous assurer que les bonnes personnes et ressources sont disponibles pour le soutenir. Alternativement, nous pouvons mettre en évidence les domaines nécessitant une interaction plus étroite ou une préparation supplémentaire (par exemple, création de plus de contrôles de surveillance ou de scripts d'automatisation). En faisant cela, nous créons les conditions où les opérations peuvent aider à résoudre nos problèmes d'équipe actuels (par exemple, améliorer les performances en optimisant la base de données au lieu du code) ou prévenir les problèmes futurs avant qu'ils ne deviennent une crise (par exemple, création de plus d'environnements de test d'intégration pour permettre des tests de performance).

Inviter les Ops à nos Rétrospectives Dev

Un autre rituel agile répandu est la rétrospective. À la fin de chaque intervalle de développement, l'équipe discute de ce qui a été réussi, de ce qui pourrait être amélioré et comment incorporer les succès et améliorations dans les futures itérations ou projets. L'équipe propose des idées pour améliorer les choses et passe en revue les expériences de l'itération précédente. C'est un des mécanismes principaux où l'apprentissage organisationnel et le

développement de contre-mesures se produisent, avec du travail résultant mis en œuvre immédiatement ou ajouté au backlog de l'équipe.

Faire participer les ingénieurs Ops aux rétrospectives de l'équipe projet signifie qu'ils peuvent également bénéficier des nouvelles connaissances. De plus, lorsqu'il y a un déploiement ou une sortie durant cet intervalle, les Ops devraient présenter les résultats et toute nouvelle leçon apprise, créant ainsi un retour d'information dans l'équipe produit. En faisant cela, nous pouvons améliorer la planification et l'exécution du travail futur, améliorant ainsi nos résultats.

Exemples de retours d'information que les Ops peuvent apporter à une rétrospective :

- « Il y a deux semaines, nous avons trouvé un angle mort dans la surveillance et avons convenu de la façon de le corriger. Cela a fonctionné. Nous avons eu un incident mardi dernier, et nous avons pu le détecter et le corriger rapidement avant que des clients ne soient impactés. »
- « Le déploiement de la semaine dernière a été l'un des plus difficiles et longs que nous ayons eu depuis plus d'un an. Voici quelques idées sur comment cela peut être amélioré. »
- « La campagne promotionnelle que nous avons menée la semaine dernière a été bien plus difficile que nous ne le pensions, et nous ne devrions probablement pas refaire une offre comme celle-ci. Voici quelques idées d'autres offres que nous pouvons faire pour atteindre nos objectifs. »
- « Lors du dernier déploiement, le plus gros problème que nous avons eu était que nos règles de pare-feu sont maintenant des milliers de lignes de long, rendant extrêmement difficile et risqué de les changer. Nous devons ré-architecturer comment nous empêchons le trafic réseau non autorisé. »

Les retours d'information des Ops aident nos équipes produit à mieux voir et comprendre l'impact en aval des décisions qu'elles prennent. Lorsqu'il y a des résultats négatifs, nous pouvons apporter les modifications nécessaires pour les prévenir à l'avenir. Les retours des Ops identifieront probablement plus de problèmes et de défauts à corriger - ils peuvent même révéler des problèmes architecturaux plus importants qui doivent être abordés.

Le travail supplémentaire identifié durant les rétrospectives des équipes projet entre dans la catégorie générale du travail d'amélioration, comme la correction de défauts, le refactoring et l'automatisation du travail manuel. Les responsables produits et chefs de projet peuvent vouloir différer ou déprioriser le travail d'amélioration en faveur des fonctionnalités client. Cependant, nous devons rappeler à tout le monde que l'amélioration du travail quotidien est plus importante que le travail quotidien lui-même, et que toutes les équipes doivent avoir une capacité dédiée pour cela (par exemple, réservé 20 % de tous les cycles pour le travail d'amélioration, programmer un jour par semaine ou une semaine par mois, etc.). Sans cela, la productivité de l'équipe risque presque certainement de s'arrêter sous le poids de sa propre dette technique et de processus.

Rendre le Travail des Ops Visible sur des Tableaux Kanban Partagés

Souvent, les équipes de développement rendent leur travail visible sur un tableau de projet ou un tableau kanban. Cependant, il est beaucoup moins courant que ces tableaux montrent le travail des opérations nécessaire pour que l'application fonctionne correctement en production, où la valeur client est réellement créée. En conséquence, nous ne sommes pas conscients du travail des opérations nécessaire jusqu'à ce qu'il devienne une crise urgente, mettant en péril les délais ou provoquant une panne de production.

Étant donné que les opérations font partie de la chaîne de valeur du produit, nous devrions afficher le travail des opérations pertinent pour la livraison du produit sur le tableau kanban partagé. Cela nous permet de voir plus clairement tout le travail nécessaire pour déplacer notre code en production, ainsi que de suivre tout le travail des opérations nécessaire pour soutenir le produit. De plus, cela nous permet de voir où le travail des opérations est bloqué et où le travail nécessite une escalade, mettant en évidence les domaines nécessitant des améliorations.

Les tableaux kanban sont un outil idéal pour créer de la visibilité, et la visibilité est un composant clé pour bien reconnaître et intégrer le travail des opérations dans toutes les chaînes de valeur pertinentes. Lorsque nous faisons cela correctement, nous obtenons des résultats orientés vers le marché, indépendamment de la manière dont nous avons dessiné nos organigrammes.

Conclusion

Tout au long de ce chapitre, nous avons exploré des moyens d'intégrer les opérations dans le travail quotidien du développement et examiné comment rendre notre travail plus visible pour les opérations. Pour y parvenir, nous avons exploré trois grandes stratégies, notamment la création de capacités en libre-service pour permettre aux développeurs des équipes de service d'être productifs, l'intégration des ingénieurs Ops dans les équipes de service, et l'assignation de liaisons Ops aux équipes de service lorsque l'intégration des ingénieurs Ops n'était pas possible. Enfin, nous avons décrit comment les ingénieurs Ops peuvent s'intégrer à l'équipe Dev par leur inclusion dans le travail quotidien, y compris les réunions debout quotidiennes, la planification et les rétrospectives.

Conclusion de la Partie 2

Dans la Partie II : Par Où Commencer, nous avons exploré diverses façons de penser aux transformations DevOps, y compris comment choisir où commencer, les aspects pertinents de l'architecture et de la conception organisationnelle, ainsi que comment organiser nos équipes. Nous avons également étudié comment intégrer les opérations dans tous les aspects de la planification et du travail quotidien du développement.

Dans la Partie III : La Première Voie, les Pratiques Techniques du Flux, nous allons maintenant commencer à explorer comment mettre en œuvre les pratiques techniques spécifiques pour réaliser les principes de flux, permettant un flux rapide de travail du développement aux opérations sans causer de chaos et de perturbations en aval.

Partie III - La Première Voie, les Pratiques Techniques du Flux

Dans la Partie III, notre objectif est de créer les pratiques techniques et l'architecture nécessaires pour permettre et soutenir le flux rapide de travail du développement aux opérations sans causer de chaos et de perturbations dans l'environnement de production ou pour nos clients. Cela signifie que nous devons réduire le risque associé au déploiement et à la mise en production des modifications. Nous y parviendrons en mettant en œuvre un ensemble de pratiques techniques connues sous le nom de livraison continue.

La livraison continue comprend la création des fondations de notre pipeline de déploiement automatisé, en veillant à ce que nous ayons des tests automatisés qui valident constamment que nous sommes dans un état déployable, en intégrant le code des développeurs dans la branche principale quotidiennement, et en architecturant nos environnements et notre code pour permettre des mises en production à faible risque. Les points principaux abordés dans ces chapitres incluent :

- Créer la fondation de notre pipeline de déploiement
- Permettre des tests automatisés rapides et fiables
- Activer et pratiquer l'intégration et les tests continus
- Automatiser, activer et architecturer pour des mises en production à faible risque

La mise en œuvre de ces pratiques réduit le temps nécessaire pour obtenir des environnements semblables à la production, permet des tests continus qui fournissent à tous des retours rapides sur leur travail, permet à de petites équipes de développer, tester et déployer leur code en toute sécurité et indépendamment dans la production, et fait des déploiements et des mises en production des routines du travail quotidien.

De plus, intégrer les objectifs de la QA et des opérations dans le travail quotidien de chacun réduit les interventions d'urgence, les difficultés et la pénibilité, tout en rendant les personnes plus productives et en augmentant la satisfaction dans le travail que nous faisons. Nous n'améliorons pas seulement les résultats, mais notre organisation est également mieux placée pour gagner sur le marché.

Créer les Fondations de Notre Pipeline de Déploiement

Pour créer un flux rapide et fiable entre le développement et les opérations, nous devons nous assurer d'utiliser des environnements similaires à la production à chaque étape de la chaîne de valeur. De plus, ces environnements doivent être créés de manière automatisée, idéalement à la demande via des scripts et des informations de configuration stockées dans le contrôle de version, et entièrement en libre-service, sans travail manuel requis de la part des opérations. Notre objectif est de pouvoir recréer l'ensemble de l'environnement de production en se basant sur ce qui est dans le contrôle de version.

Bien trop souvent, la seule fois où nous découvrons comment nos applications se comportent dans un environnement semblable à la production, c'est lors du déploiement en production - trop tard pour corriger les problèmes sans que le client ne soit impacté négativement. Un exemple illustratif des problèmes que peuvent causer des applications et des environnements construits de manière incohérente est le programme Enterprise Data Warehouse dirigé par Em Campbell-Pretty dans une grande entreprise de télécommunications australienne en 2009. Campbell-Pretty est devenue la directrice générale et la sponsor commerciale de ce programme de 200 millions de dollars, héritant de la responsabilité de tous les objectifs stratégiques reposant sur cette plateforme.

Dans sa présentation au DevOps Enterprise Summit de 2014, Campbell-Pretty a expliqué : « À l'époque, il y avait dix flux de travail en cours, tous utilisant des processus en cascade, et les dix flux étaient significativement en retard. Un seul des dix flux avait réussi à atteindre les tests d'acceptation utilisateur [UAT] dans les délais, et il a fallu six mois supplémentaires pour que ce flux termine l'UAT, avec des résultats bien en deçà des attentes commerciales. Cette sous-performance a été le principal catalyseur de la transformation Agile du département. »

Cependant, après avoir utilisé Agile pendant près d'un an, ils n'ont constaté que de petites améliorations, restant toujours en deçà des résultats commerciaux nécessaires. Campbell-Pretty a organisé une rétrospective à l'échelle du programme et a demandé : « Après avoir réfléchi à toutes nos expériences au cours de la dernière version, quelles sont les choses que nous pourrions faire pour doubler notre productivité ? »

Tout au long du projet, il y avait des plaintes concernant le « manque d'engagement commercial ». Cependant, lors de la rétrospective, « améliorer la disponibilité des environnements » était en tête de liste. Rétrospectivement, c'était évident - les équipes de développement avaient besoin d'environnements provisionnés pour commencer à travailler et attendaient souvent jusqu'à huit semaines.

Ils ont créé une nouvelle équipe d'intégration et de construction responsable de « construire la qualité dans nos processus, plutôt que d'essayer d'inspecter la qualité après coup ». Elle était initialement composée d'administrateurs de bases de données (DBA) et de spécialistes de l'automatisation chargés d'automatiser leur processus de création d'environnements. L'équipe

a rapidement fait une découverte surprenante : seulement 50 % du code source dans leurs environnements de développement et de test correspondaient à ce qui fonctionnait en production.

Campbell-Pretty a observé : « Soudain, nous avons compris pourquoi nous rencontrions tant de défauts chaque fois que nous déployions notre code dans de nouveaux environnements. Dans chaque environnement, nous continuons à corriger des problèmes, mais les changements que nous faisions n'étaient pas remis dans le contrôle de version. »

L'équipe a soigneusement rétro-conçu tous les changements qui avaient été apportés aux différents environnements et les a tous mis dans le contrôle de version. Ils ont également automatisé leur processus de création d'environnements afin de pouvoir les lancer de manière répétée et correcte.

Campbell-Pretty a décrit les résultats, notant que « le temps nécessaire pour obtenir un environnement correct est passé de huit semaines à un jour. C'était l'un des ajustements clés qui nous a permis d'atteindre nos objectifs concernant notre délai de livraison, le coût de livraison, et le nombre de défauts échappés qui arrivaient en production. »

L'histoire de Campbell-Pretty montre la variété des problèmes qui peuvent être attribués à des environnements construits de manière incohérente et à des changements qui ne sont pas systématiquement remis dans le contrôle de version.

Tout au long du reste de ce chapitre, nous discuterons de la manière de construire les mécanismes qui nous permettront de créer des environnements à la demande, d'étendre l'utilisation du contrôle de version à tout le monde dans la chaîne de valeur, de rendre l'infrastructure plus facile à reconstruire qu'à réparer, et de s'assurer que les développeurs exécutent leur code dans des environnements semblables à la production à chaque étape du cycle de vie du développement logiciel.

Activer la création à la demande d'environnements de développement, de test et de production

Comme on l'a vu dans l'exemple de l'entrepôt de données d'entreprise ci-dessus, l'une des principales causes contributives des versions logicielles chaotiques, perturbatrices, et parfois même catastrophiques, est que la première fois que nous voyons comment notre application se comporte dans un environnement semblable à la production avec une charge réaliste et des ensembles de données de production, c'est lors de la version.

Dans de nombreux cas, les équipes de développement peuvent avoir demandé des environnements de test dès les premières étapes du projet. Cependant, lorsque de longs délais sont nécessaires pour que les opérations livrent des environnements de test, les équipes peuvent ne pas les recevoir assez tôt pour effectuer des tests adéquats. Pire, les environnements de test sont souvent mal configurés ou tellement différents de nos environnements de production que nous nous retrouvons toujours avec de gros problèmes de production malgré avoir effectué des tests avant le déploiement.

Dans cette étape, nous voulons que les développeurs exécutent des environnements semblables à la production sur leurs propres postes de travail, créés à la demande et en libre-service. En faisant cela, les développeurs peuvent exécuter et tester leur code dans des environnements semblables à la production dans le cadre de leur travail quotidien, fournissant un retour d'information précoce et constant sur la qualité de leur travail.

Au lieu de simplement documenter les spécifications de l'environnement de production dans un document ou sur une page wiki, nous créons un mécanisme de construction commun qui crée tous nos environnements, tels que pour le développement, le test et la production. En faisant cela, n'importe qui peut obtenir des environnements semblables à la production en quelques minutes, sans ouvrir de ticket, sans parler d'attendre des semaines.

Pour ce faire, il est nécessaire de définir et d'automatiser la création de nos environnements connus et fiables, qui sont stables, sécurisés et dans un état de risque réduit, incarnant les connaissances collectives de l'organisation. Toutes nos exigences sont intégrées, non pas dans des documents ou comme des connaissances dans la tête de quelqu'un, mais codifiées dans notre processus de construction d'environnements automatisé.

Au lieu que les opérations construisent et configurent manuellement l'environnement, nous pouvons utiliser l'automatisation pour tout ou partie des éléments suivants :

- Copier un environnement virtualisé (par exemple, une image VMware, exécuter un script Vagrant, démarrer un fichier Amazon Machine Image dans EC2)
- Construire un processus de création d'environnement automatisé qui commence à partir de « métal nu » (par exemple, installation PXE à partir d'une image de base)
- Utiliser des outils de gestion de configuration en tant que code (par exemple, Puppet, Chef, Ansible, Salt, CFEngine, etc.)
- Utiliser des outils de configuration automatisée du système d'exploitation (par exemple, Solaris Jumpstart, Red Hat Kickstart, Debian preseed)
- Assembler un environnement à partir d'un ensemble d'images virtuelles ou de conteneurs (par exemple, Vagrant, Docker)
- Lancer un nouvel environnement dans un cloud public (par exemple, Amazon Web Services, Google App Engine, Microsoft Azure), un cloud privé ou un autre PaaS (plateforme en tant que service, tel que OpenStack ou Cloud Foundry, etc.).

Parce que nous avons soigneusement défini tous les aspects de l'environnement à l'avance, nous sommes non seulement capables de créer de nouveaux environnements rapidement, mais aussi de nous assurer que ces environnements seront stables, fiables, cohérents et sécurisés. Cela profite à tout le monde.

Les opérations bénéficient de cette capacité à créer de nouveaux environnements rapidement, car l'automatisation du processus de création d'environnements renforce la cohérence et réduit le travail manuel fastidieux et sujet aux erreurs. De plus, le développement bénéficie de la possibilité de reproduire toutes les parties nécessaires de l'environnement de production pour construire, exécuter et tester leur code sur leurs postes de travail. En faisant cela, nous permettons aux développeurs de trouver et de corriger de nombreux problèmes, même aux premières étapes du projet, plutôt que lors des tests d'intégration ou pire, en production.

En fournissant aux développeurs un environnement qu'ils contrôlent entièrement, nous leur permettons de reproduire, diagnostiquer et corriger rapidement les défauts, en toute sécurité, isolés des services de production et des autres ressources partagées. Ils peuvent également expérimenter des changements dans les environnements, ainsi que dans le code d'infrastructure qui les crée (par exemple, les scripts de gestion de configuration), créant ainsi davantage de connaissances partagées entre le développement et les opérations.

Créer notre répertoire unique de vérité pour l'ensemble du système

À l'étape précédente, nous avons activé la création à la demande des environnements de développement, de test et de production. Maintenant, nous devons nous assurer que toutes les parties de notre système logiciel.

Depuis des décennies, l'utilisation complète du contrôle de version est devenue de plus en plus une pratique obligatoire pour les développeurs individuels et les équipes de développement.

Un système de contrôle de version enregistre les modifications apportées à des fichiers ou ensembles de fichiers stockés dans le système. Il peut s'agir de code source, d'actifs ou d'autres documents faisant partie d'un projet de développement logiciel. Nous apportons des modifications par groupes appelés validations ou révisions. Chaque révision, avec des métadonnées telles que l'auteur de la modification et quand, est stockée dans le système d'une manière ou d'une autre, nous permettant de valider, comparer, fusionner et restaurer des révisions antérieures aux objets dans le dépôt. Cela réduit également les risques en établissant un moyen de revenir en arrière dans les objets en production vers des versions antérieures. (Dans ce livre, les termes suivants seront utilisés de manière interchangeable : validés dans le contrôle de version, validés dans le contrôle de version, validation du code, modification de la validation, validation.)

Lorsque les développeurs mettent tous leurs fichiers sources d'application et configurations dans le contrôle de version, cela devient le répertoire unique de vérité qui contient l'état précisément voulu du système. Cependant, parce que fournir de la valeur au client nécessite à la fois notre code et les environnements dans lesquels ils s'exécutent, nous avons également

besoin de nos environnements dans le contrôle de version. En d'autres termes, le contrôle de version est pour tout le monde dans notre chaîne de valeur, y compris QA, Operations, Infosec, ainsi que les développeurs. En mettant tous les artefacts de production dans le contrôle de version, notre référentiel de contrôle de version nous permet de reproduire de manière répétée et fiable tous les composants de notre système logiciel en cours de fonctionnement - cela inclut nos applications et l'environnement de production, ainsi que tous nos environnements de préproduction.

Pour assurer que nous pouvons restaurer le service de production de manière répétée et prévisible (et idéalement rapidement) même en cas d'événements catastrophiques, nous devons enregistrer les actifs suivants dans notre référentiel de contrôle de version partagé :

- Tout le code d'application et les dépendances (par exemple, bibliothèques, contenu statique, etc.)
- Tout script utilisé pour créer des schémas de base de données, des données de référence d'application, etc.
- Tous les outils de création d'environnement et les artefacts décrits à l'étape précédente (par exemple, images VMware ou AMI, recettes Puppet ou Chef, etc.)
- Tout fichier utilisé pour créer des conteneurs (par exemple, fichiers de définition ou de composition Docker ou Rocket)
- Tous les tests automatisés de support et tous les scripts de test manuels
- Tout script qui prend en charge l'emballage de code, le déploiement, la migration de base de données et la provision d'environnement
- Tous les artefacts de projet (par exemple, documentation des exigences, procédures de déploiement, notes de version, etc.)
- Tous les fichiers de configuration cloud (par exemple, modèles AWS Cloudformation, fichiers DSC Microsoft Azure Stack, OpenStack HEAT)
- Tout autre script ou information de configuration nécessaire pour créer une infrastructure qui prend en charge plusieurs services (par exemple, bus de services d'entreprise, systèmes de gestion de base de données, fichiers de zone DNS, règles de configuration pour les pare-feu et autres périphériques réseau).

Nous pouvons avoir plusieurs référentiels pour différents types d'objets et de services, où ils sont étiquetés et marqués aux côtés de notre code source. Par exemple, nous pouvons stocker de grandes images de machine virtuelle, des fichiers ISO, des binaires compilés, etc., dans des référentiels d'artefacts (par exemple, Nexus, Artifactory). Alternativement, nous pouvons les mettre dans des magasins de blobs (par exemple, des compartiments S3 Amazon) ou mettre des images Docker dans des registres Docker, etc.

Il ne suffit pas de simplement pouvoir recréer n'importe quel état antérieur de l'environnement de production; nous devons également être en mesure de recréer l'ensemble des processus de préproduction et de construction. Par conséquent, nous devons mettre sous contrôle de version tout ce sur quoi reposent nos processus de construction, y compris nos outils (par exemple, compilateurs, outils de test) et les environnements dont ils dépendent.

Dans le rapport 2014 de Puppet Labs sur l'état de DevOps, l'utilisation du contrôle de version par Ops était le prédicteur le plus élevé à la fois des performances IT et des performances organisationnelles. En fait, que les Ops utilisent le contrôle de version était un prédicteur plus

élevé à la fois des performances IT et des performances organisationnelles que si Dev utilisait le contrôle de version.

Les conclusions du rapport 2014 de Puppet Labs sur l'état de DevOps soulignent le rôle critique joué par le contrôle de version dans le processus de développement logiciel. Nous savons maintenant que lorsque tous les changements d'application et d'environnement sont enregistrés dans le contrôle de version, cela nous permet non seulement de voir rapidement tous les changements qui pourraient avoir contribué à un problème, mais fournit également les moyens de revenir à un état précédent connu, fonctionnant, nous permettant de récupérer plus rapidement des échecs.

Mais pourquoi l'utilisation du contrôle de version pour nos environnements prédit-elle mieux les performances IT et organisationnelles que l'utilisation du contrôle de version pour notre code? Parce que dans presque tous les cas, il y a des ordres de grandeur plus de paramètres configurables dans notre environnement que dans notre code. Par conséquent, c'est l'environnement qui doit être le plus contrôlé par version.

Le contrôle de version fournit également un moyen de communication pour tous ceux qui travaillent dans le flux de valeur - avoir le développement, QA, Infosec et Operations capables de voir les changements des autres aide à réduire les surprises, crée une visibilité sur le travail des autres et aide à construire et renforcer la confiance.

Faire de l'infrastructure plus facile à reconstruire qu'à réparer

Lorsque nous pouvons reconstruire et recréer rapidement nos applications et environnements à la demande, nous pouvons également les reconstruire rapidement au lieu de les réparer lorsque les choses tournent mal. Bien que cela soit une pratique courante dans les opérations web à grande échelle (c'est-à-dire plus de mille serveurs), nous devrions adopter cette pratique même si nous n'avons qu'un seul serveur en production.

Bill Baker, un ingénieur distingué chez Microsoft, a plaisanté en disant que nous avions l'habitude de traiter les serveurs comme des animaux domestiques : "On leur donne un nom et quand ils tombent malades, on les soigne pour les ramener à la santé. [Maintenant] les serveurs sont [traités] comme du bétail. On les numérote et quand ils tombent malades, on les abat."

Grâce à des systèmes de création d'environnement reproductibles, nous pouvons facilement augmenter la capacité en ajoutant plus de serveurs en rotation (c'est-à-dire un dimensionnement horizontal). Nous évitons également la catastrophe qui résulte inévitablement lorsque nous devons restaurer le service après une défaillance catastrophique d'une infrastructure non reproductible, créée au fil des années par des modifications de production non documentées et manuelles.

Pour assurer la cohérence de nos environnements, chaque fois que nous apportons des modifications de production (changements de configuration, patchs, mises à niveau, etc.), ces

changements doivent être reproduits partout dans nos environnements de production et de préproduction, ainsi que dans tout nouvel environnement créé.

Plutôt que de se connecter manuellement aux serveurs et d'apporter des modifications, nous devons apporter des modifications de manière à garantir que toutes les modifications sont reproduites automatiquement partout et que toutes nos modifications sont intégrées dans le contrôle de version.

Nous pouvons compter sur nos systèmes de configuration automatisés pour assurer la cohérence (par exemple, Puppet, Chef, Ansible, Salt, Bosh, etc.), ou nous pouvons créer de nouvelles machines virtuelles ou conteneurs à partir de notre mécanisme de construction automatisé et les déployer en production, en détruisant les anciens ou en les retirant de la rotation.

Le dernier modèle est ce qui est devenu connu sous le nom d'infrastructure immuable, où les modifications manuelles de l'environnement de production ne sont plus autorisées ; la seule façon d'apporter des modifications de production est de les mettre dans le contrôle de version et de recréer le code et les environnements à partir de zéro. En faisant cela, aucune variation ne peut s'introduire en production.

Pour éviter les variations de configuration non contrôlées, nous pouvons désactiver les connexions distantes aux serveurs de production ou remplacer régulièrement et tuer les instances de production, en veillant à ce que les modifications de production appliquées manuellement soient supprimées. Cette action incite tout le monde à apporter leurs modifications de la bonne manière à travers le contrôle de version. En appliquant de telles mesures, nous réduisons systématiquement les moyens par lesquels notre infrastructure peut s'écartier de nos états connus et valides (par exemple, dérive de configuration, artefacts fragiles, œuvres d'art, flocons de neige, etc.).

De plus, nous devons maintenir nos environnements de préproduction à jour, en particulier en veillant à ce que les développeurs utilisent notre environnement le plus récent. Les développeurs voudront souvent continuer à utiliser des environnements plus anciens par peur que les mises à jour d'environnement ne cassent des fonctionnalités existantes. Cependant, nous voulons les mettre à jour fréquemment afin de pouvoir identifier les problèmes dès le début du cycle de vie.

Modifier notre définition du « DONE » de développement pour inclure l'exécution dans des environnements de type production

Maintenant que nos environnements peuvent être créés à la demande et que tout est vérifié dans le contrôle de version, notre objectif est de nous assurer que ces environnements sont

utilisés quotidiennement dans le travail de développement. Nous devons vérifier que notre application s'exécute comme prévu dans un environnement similaire à la production bien avant la fin du projet ou avant notre première mise en production.

La plupart des méthodologies modernes de développement logiciel préconisent des intervalles de développement courts et itératifs, contrairement à l'approche "big bang" (par exemple, le modèle en cascade). En général, plus l'intervalle entre les déploiements est long, plus les résultats sont mauvais. Par exemple, dans la méthodologie Scrum, un sprint est un intervalle de développement limité dans le temps (généralement un mois ou moins) au cours duquel nous devons être "terminés", largement défini comme ayant du "code fonctionnel et potentiellement livrable".

Notre objectif est de nous assurer que le développement et l'assurance qualité intègrent régulièrement le code dans des environnements similaires à la production à des intervalles de plus en plus fréquents tout au long du projet.

Nous faisons cela en élargissant la définition du « DONE » au-delà de la simple fonctionnalité du code (ajout en texte gras) : à la fin de chaque intervalle de développement, nous avons un code intégré, testé, fonctionnel et potentiellement livrable, démontré dans un environnement similaire à la production.

En d'autres termes, nous n'accepterons le travail de développement comme terminé que s'il peut être construit, déployé et confirmé qu'il s'exécute comme prévu dans un environnement similaire à la production, plutôt que simplement parce qu'un développeur croit que c'est terminé - idéalement, il s'exécute sous une charge similaire à la production avec un jeu de données similaire bien avant la fin d'un sprint. Cela évite les situations où une fonctionnalité est considérée comme terminée simplement parce qu'un développeur peut l'exécuter avec succès sur son ordinateur portable mais nulle part ailleurs.

En faisant écrire, tester et exécuter leur propre code aux développeurs dans un environnement similaire à la production, la majorité du travail pour intégrer avec succès notre code et nos environnements se fait pendant notre travail quotidien, plutôt qu'à la fin de la version.

À la fin de notre premier intervalle, notre application peut être démontrée pour s'exécuter correctement dans un environnement similaire à la production, le code et l'environnement ayant été intégrés à plusieurs reprises, idéalement avec toutes les étapes automatisées (aucune bricolage manuelle requise).

Mieux encore, à la fin du projet, nous aurons déployé et fait fonctionner notre code dans des environnements similaires à la production des centaines, voire des milliers de fois, nous

donnant confiance que la plupart de nos problèmes de déploiement en production ont été identifiés et résolus.

Idéalement, nous utilisons les mêmes outils, tels que la surveillance, le journalisation et le déploiement, dans nos environnements de préproduction que dans la production. En faisant cela, nous avons une familiarité et une expérience qui nous aideront à déployer et à exécuter en douceur, ainsi qu'à diagnostiquer et à résoudre nos services lorsqu'ils sont en production.

En permettant aux développements et aux opérations de développer une maîtrise partagée de l'interaction entre le code et l'environnement, et en pratiquant les déploiements tôt et souvent, nous réduisons considérablement les risques de déploiement associés aux mises en production du code. Cela nous permet également d'éliminer toute une classe de défauts opérationnels, de sécurité et de problèmes architecturaux qui sont généralement découverts trop tard dans le projet pour être corrigés.

Conclusion

Le flux rapide du travail du développement vers les opérations nécessite que n'importe qui puisse obtenir des environnements similaires à la production à la demande. En permettant aux développeurs d'utiliser des environnements similaires à la production même aux premiers stades d'un projet logiciel, nous réduisons considérablement le risque de problèmes de production ultérieurs. C'est l'une des nombreuses pratiques qui montrent comment les opérations peuvent rendre les développeurs beaucoup plus productifs. Nous imposons la pratique aux développeurs d'exécuter leur code dans des environnements similaires à la production en l'incorporant dans la définition du « DONE ».

De plus, en mettant tous les artefacts de production dans le contrôle de version, nous avons une « source unique de vérité » qui nous permet de recréer tout l'environnement de production de manière rapide, reproductible et documentée, en utilisant les mêmes pratiques de développement pour le travail des opérations que celles que nous utilisons pour le développement.

Et en rendant l'infrastructure de production plus facile à reconstruire qu'à réparer, nous facilitons la résolution des problèmes et nous accélérons, ainsi que facilitons l'extension de la capacité.

La mise en place de ces pratiques prépare le terrain pour permettre l'automatisation des tests complets, qui est explorée dans le prochain chapitre.

Activer des tests automatisés rapides et fiables

À ce stade, les équipes de développement et de QA utilisent des environnements similaires à la production dans leur travail quotidien, et nous intégrons et exécutons avec succès notre code dans un environnement similaire à la production pour chaque fonctionnalité acceptée, avec tous les changements enregistrés dans le contrôle de version. Cependant, nous risquons d'obtenir des résultats indésirables si nous trouvons et corrigons les erreurs dans une phase de test séparée, exécutée par un département QA séparé uniquement après que tout le développement a été terminé. De plus, si les tests ne sont effectués que quelques fois par an, les développeurs découvrent leurs erreurs des mois après avoir introduit la modification qui a causé l'erreur. À ce moment-là, le lien entre cause et effet a probablement disparu, résoudre le problème nécessite des interventions d'urgence et de l'archéologie, et, pire encore, notre capacité à apprendre de l'erreur et à l'intégrer dans notre travail futur est considérablement diminuée.

Les tests automatisés répondent à un autre problème important et inquiétant. Gary Gruver observe que « sans tests automatisés, plus nous écrivons de code, plus il faut de temps et d'argent pour tester notre code - dans la plupart des cas, c'est un modèle économique totalement non évolutif pour toute organisation technologique ».

Bien que Google incarne aujourd'hui indéniablement une culture qui valorise les tests automatisés à grande échelle, cela n'a pas toujours été le cas. En 2005, lorsque Mike Bland a rejoint l'organisation, le déploiement sur Google.com était souvent extrêmement problématique, en particulier pour l'équipe Google Web Server (GWS).

Comme Bland l'explique, « L'équipe GWS s'était retrouvée dans une position au milieu des années 2000 où il était extrêmement difficile d'apporter des modifications au serveur web, une application C++ qui gérait toutes les requêtes vers la page d'accueil de Google et de nombreuses autres pages web de Google. Aussi importante et emblématique que soit Google.com, être dans l'équipe GWS n'était pas une affectation prestigieuse - c'était souvent le dépotoir pour toutes les différentes équipes qui créaient diverses fonctionnalités de recherche, toutes développant du code indépendamment les unes des autres. Elles avaient des problèmes tels que les builds et les tests prenaient trop de temps, le code était mis en production sans être testé, et les équipes enregistraient des modifications importantes et peu fréquentes qui entraient en conflit avec celles d'autres équipes. »

Les conséquences de cela étaient importantes - les résultats de recherche pouvaient contenir des erreurs ou devenir inacceptablement lents, affectant des milliers de requêtes de recherche sur Google.com. Le résultat potentiel était non seulement une perte de revenus, mais aussi de la confiance des clients.

Bland décrit comment cela affectait les développeurs déployant des modifications : « La peur devenait l'ennemie de l'esprit. La peur empêchait les nouveaux membres de l'équipe de modifier les choses parce qu'ils ne comprenaient pas le système. Mais la peur empêchait également les personnes expérimentées de modifier les choses parce qu'elles comprenaient trop bien le système. »

Bland faisait partie du groupe déterminé à résoudre ce problème. Le chef de l'équipe GWS, Bharat Mediratta, croyait que les tests automatisés aideraient. Comme Bland le décrit, « Ils ont créé une ligne dure : aucune modification ne serait acceptée dans GWS sans tests automatisés accompagnants. Ils ont mis en place une build continue et l'ont maintenue religieusement. Ils ont mis en place un suivi de la couverture des tests et ont veillé à ce que leur niveau de couverture des tests augmente avec le temps. Ils ont rédigé des politiques et des guides de test, et ont insisté pour que les contributeurs tant à l'intérieur qu'à l'extérieur de l'équipe les suivent. »

Les résultats étaient étonnantes. Comme Bland le note, « GWS est rapidement devenue l'une des équipes les plus productives de l'entreprise, intégrant un grand nombre de changements provenant de différentes équipes chaque semaine tout en maintenant un calendrier de publication rapide. Les nouveaux membres de l'équipe ont pu apporter des contributions productives à ce système complexe rapidement, grâce à une bonne couverture des tests et à la santé du code. Finalement, leur politique radicale a permis à la page d'accueil de Google.com d'étendre rapidement ses capacités et de prospérer dans un paysage technologique incroyablement dynamique et compétitif. »

Mais GWS était encore une équipe relativement petite dans une entreprise grande et en croissance. L'équipe voulait étendre ces pratiques à l'ensemble de l'organisation. Ainsi, le Testing Grouplet est né, un groupe informel d'ingénieurs qui voulait éléver les pratiques de tests automatisés à travers toute l'organisation. Au cours des cinq années suivantes, ils ont aidé à reproduire cette culture de tests automatisés dans tout Google.

Maintenant, lorsqu'un développeur de Google soumet du code, il est automatiquement exécuté contre une suite de centaines de milliers de tests automatisés. Si le code passe, il est automatiquement fusionné dans la branche principale, prêt à être déployé en production. De nombreuses propriétés Google font des builds horaires ou quotidiens, puis choisissent quelles builds publier ; d'autres adoptent une philosophie de livraison continue « Push on Green ».

Les enjeux sont plus élevés que jamais - une seule erreur de déploiement de code chez Google peut mettre hors service toutes les propriétés, en même temps (comme un changement d'infrastructure global ou lorsqu'un défaut est introduit dans une bibliothèque de base dont dépendent toutes les propriétés).

Eran Messeri, un ingénieur du groupe Google Developer Infrastructure, note : « Les grandes pannes se produisent occasionnellement. Vous recevez une tonne de messages instantanés et d'ingénieurs frappant à votre porte. [Lorsque la pipeline de déploiement est brisée,] nous devons la réparer immédiatement, car les développeurs ne peuvent plus soumettre de code. Par conséquent, nous voulons rendre très facile le retour en arrière. »

Ce qui permet à ce système de fonctionner chez Google, c'est le professionnalisme des ingénieurs et une culture de haute confiance qui suppose que tout le monde veut faire du bon travail, ainsi que la capacité de détecter et de corriger rapidement les problèmes. Messeri explique : « Il n'y a pas de politiques strictes chez Google, telles que : 'Si vous cassez la production pour plus de dix projets, vous avez un SLA pour résoudre le problème en dix minutes.' Au lieu de cela, il y a un respect mutuel entre les équipes et un accord implicite selon lequel tout le monde fait tout ce qui est nécessaire pour maintenir la pipeline de déploiement en marche. Nous savons tous qu'un jour, je casserais votre projet par accident ; le lendemain, vous pourriez casser le mien. »

Ce que Mike Bland et l'équipe du Testing Grouplet ont accompli a fait de Google l'une des organisations technologiques les plus productives au monde. En 2013, les tests automatisés et l'intégration continue chez Google ont permis à plus de quatre mille petites équipes de travailler ensemble et de rester productives, tout en développant, intégrant, testant et déployant leur code en production simultanément. Tout leur code est dans un seul référentiel partagé, composé de milliards de fichiers, tous construits et intégrés en continu, avec 50 % de leur code étant modifié chaque mois. Parmi d'autres statistiques impressionnantes sur leurs performances, on trouve :

- 40 000 soumissions de code par jour
- 50 000 builds par jour (en semaine, cela peut dépasser 90 000)
- 120 000 suites de tests automatisés
- 75 millions de cas de test exécutés quotidiennement
- Plus de 100 ingénieurs travaillant sur les outils d'ingénierie de tests, d'intégration continue et de gestion des versions pour accroître la productivité des développeurs (représentant 0,5 % de la main-d'œuvre en R&D)

Dans le reste de ce chapitre, nous passerons en revue les pratiques d'intégration continue nécessaires pour reproduire ces résultats.

Construire, tester et intégrer continuellement notre code et nos environnements

Notre objectif est d'incorporer la qualité dans notre produit dès les premières étapes en faisant en sorte que les développeurs construisent des tests automatisés dans le cadre de leur travail quotidien. Cela crée une boucle de rétroaction rapide qui aide les développeurs à trouver rapidement les problèmes et à les résoudre rapidement, lorsqu'il y a le moins de contraintes (par exemple, le temps, les ressources).

Dans cette étape, nous créons des suites de tests automatisés qui augmentent la fréquence d'intégration et de test de notre code et de nos environnements, passant de périodique à continue. Nous faisons cela en construisant notre pipeline de déploiement, qui effectuera l'intégration de notre code et de nos environnements et déclenchera une série de tests chaque fois qu'un nouveau changement est intégré dans le contrôle de version.

Le pipeline de déploiement, défini pour la première fois par Jez Humble et David Farley dans leur livre "Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation", garantit que tout code intégré dans le contrôle de version est automatiquement construit et testé dans un environnement similaire à la production. En faisant cela, nous détectons toute erreur de construction, de test ou d'intégration dès qu'un changement est introduit, ce qui nous permet de les corriger immédiatement. Correctement fait, cela nous permet d'être toujours assurés d'être dans un état déployable et livrable.

Pour y parvenir, nous devons créer des processus de construction et de test automatisés qui s'exécutent dans des environnements dédiés. Cela est crucial pour les raisons suivantes :

- Notre processus de construction et de test peut s'exécuter en continu, indépendamment des habitudes de travail des ingénieurs individuels.
- Un processus de construction et de test séparé nous assure de comprendre toutes les dépendances nécessaires pour construire, empaqueter, exécuter et tester notre code (c'est-à-dire en éliminant le problème "ça fonctionnait sur l'ordinateur du développeur, mais ça a échoué en production").
- Nous pouvons empaqueter notre application pour permettre l'installation répétable du code et des configurations dans un environnement (par exemple, sur Linux RPM, yum, npm ; sur Windows, OneGet ; alternativement, des systèmes de packaging spécifiques au framework peuvent être utilisés, tels que les fichiers EAR et WAR pour Java, les gems pour Ruby, etc.).
- Au lieu de mettre notre code dans des paquets, nous pouvons choisir d'empaqueter nos applications dans des conteneurs déployables (par exemple, Docker, Rkt, LXD, AMIs).
- Les environnements peuvent être rendus plus semblables à la production de manière cohérente et répétable (par exemple, les compilateurs sont retirés de l'environnement, les drapeaux de débogage sont désactivés, etc.).

Notre pipeline de déploiement valide après chaque changement que notre code s'intègre avec succès dans un environnement similaire à la production. Il devient la plateforme par laquelle les testeurs demandent et certifient les builds lors des tests d'acceptation et des tests de convivialité, et il exécutera des validations automatisées de performance et de sécurité.

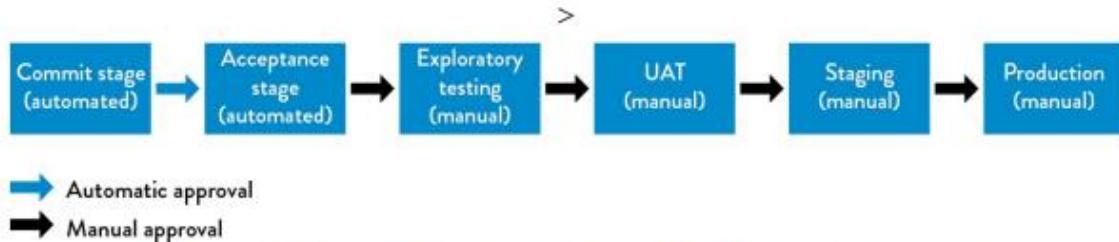


Figure 13: The deployment pipeline (Source: Humble and Farley, Continuous Delivery, 3.)

De plus, il sera utilisé pour les builds en libre-service pour les environnements de tests d'acceptation utilisateur (UAT), de tests d'intégration et de tests de sécurité. Dans les étapes futures, au fur et à mesure que nous faisons évoluer le pipeline de déploiement, il sera également utilisé pour gérer toutes les activités requises pour faire passer nos changements du contrôle de version au déploiement.

Une variété d'outils a été conçus pour fournir des fonctionnalités de pipeline de déploiement, dont beaucoup sont open source (par exemple, Jenkins, ThoughtWorks Go, Concourse, Bamboo, Microsoft Team Foundation Server, TeamCity, Gitlab CI, ainsi que des solutions basées sur le cloud telles que Travis CI et Snap).

Nous commençons le pipeline de déploiement en exécutant l'étape de commit, qui construit et empaquette le logiciel, exécute des tests unitaires automatisés, et effectue des validations supplémentaires telles que l'analyse de code statique, l'analyse de duplication et de couverture de test, et la vérification du style.

Si cette étape est réussie, elle déclenche l'étape d'acceptation, qui déploie automatiquement les packages créés lors de l'étape de commit dans un environnement similaire à la production et exécute les tests d'acceptation automatisés.

Une fois les changements acceptés dans le contrôle de version, nous souhaitons empaqueter notre code une seule fois, afin que les mêmes packages soient utilisés pour déployer le code tout au long de notre pipeline de déploiement. En faisant cela, le code sera déployé dans nos environnements de test intégrés et de préproduction de la même manière qu'il est déployé en production. Cela réduit les variations qui peuvent éviter des erreurs en aval difficiles à diagnostiquer (par exemple, l'utilisation de compilateurs, de drapeaux de compilation, de versions de bibliothèques ou de configurations différentes).

L'objectif du pipeline de déploiement est de fournir à tous les intervenants de la chaîne de valeur, en particulier aux développeurs, le retour d'information le plus rapide possible indiquant qu'un changement nous a fait sortir d'un état déployable. Cela pourrait être un changement de

notre code, de l'un de nos environnements, de nos tests automatisés, ou même de l'infrastructure du pipeline de déploiement (par exemple, un paramètre de configuration de Jenkins).

En conséquence, notre infrastructure de pipeline de déploiement devient aussi fondamentale pour nos processus de développement que notre infrastructure de contrôle de version. Notre pipeline de déploiement stocke également l'historique de chaque build de code, y compris des informations sur les tests effectués sur chaque build, les builds déployés dans chaque environnement, et les résultats des tests. En combinaison avec les informations de notre historique de contrôle de version, nous pouvons rapidement déterminer ce qui a causé la rupture de notre pipeline de déploiement et, probablement, comment corriger l'erreur.

Ces informations nous aident également à satisfaire les exigences de preuve pour les audits et la conformité, les preuves étant automatiquement générées dans le cadre du travail quotidien.

Maintenant que nous avons une infrastructure de pipeline de déploiement fonctionnelle, nous devons créer nos pratiques d'intégration continue, qui nécessitent trois capacités :

- Un ensemble complet et fiable de tests automatisés qui valident que nous sommes dans un état déployable.
- Une culture qui "arrête toute la chaîne de production" lorsque nos tests de validation échouent.
- Des développeurs travaillant en petits lots sur la branche principale plutôt que sur des branches de fonctionnalités de longue durée.

Dans la section suivante, nous décrivons pourquoi des tests automatisés rapides et fiables sont nécessaires et comment les construire.

Construire une suite de tests de validation automatisée rapide et fiable

Dans l'étape précédente, nous avons commencé à créer l'infrastructure de tests automatisés qui valide que nous avons un build réussi (c'est-à-dire que ce qui est dans le contrôle de version est dans un état construisible et déployable). Pour souligner pourquoi nous devons effectuer cette étape d'intégration et de test de manière continue, considérons ce qui se passe lorsque nous effectuons cette opération uniquement de manière périodique, comme lors d'un processus de build nocturne.

Supposons que nous ayons une équipe de dix développeurs, chacun intégrant son code dans le contrôle de version quotidiennement, et qu'un développeur introduise un changement qui casse notre build et notre tâche de test nocturnes. Dans ce scénario, lorsque nous découvrons le lendemain que nous n'avons plus un build réussi, il faudra des minutes, voire plus probablement des heures, à notre équipe de développement pour déterminer quel changement a causé le problème, qui l'a introduit et comment le corriger.

Pire encore, supposons que le problème n'ait pas été causé par un changement de code, mais par un problème d'environnement de test (par exemple, un paramètre de configuration incorrect quelque part). L'équipe de développement peut croire qu'elle a corrigé le problème parce que tous les tests unitaires passent, pour découvrir que les tests échoueront encore plus tard dans la nuit.

Pour compliquer encore la situation, dix autres changements auront été intégrés dans le contrôle de version par l'équipe ce jour-là. Chacun de ces changements a le potentiel d'introduire plus d'erreurs qui pourraient casser nos tests automatisés, augmentant encore la difficulté de diagnostiquer et de corriger le problème avec succès.

En bref, le retour d'information lent et périodique tue. Surtout pour les grandes équipes de développement. Le problème devient encore plus redoutable lorsque nous avons des dizaines, des centaines, voire des milliers d'autres développeurs intégrant leurs changements dans le contrôle de version chaque jour. Le résultat est que nos builds et nos tests automatisés sont fréquemment cassés, et les développeurs arrêtent même d'intégrer leurs changements dans le contrôle de version ("Pourquoi déranger, puisque les builds et les tests sont toujours cassés ?"). Au lieu de cela, ils attendent d'intégrer leur code à la fin du projet, ce qui entraîne tous les résultats indésirables des grandes tailles de lot, des intégrations en grande explosion et des déploiements en production.

Pour éviter ce scénario, nous avons besoin de tests automatisés rapides qui s'exécutent dans nos environnements de build et de test chaque fois qu'un nouveau changement est intégré dans le contrôle de version. De cette manière, nous pouvons trouver et corriger tout problème immédiatement, comme l'a démontré l'exemple du Google Web Server. En faisant cela, nous nous assurons que nos lots restent petits et, à tout moment, nous restons dans un état déployable.

En général, les tests automatisés se répartissent dans l'une des catégories suivantes, de la plus rapide à la plus lente :

- **Tests unitaires** : Ils testent généralement une seule méthode, classe ou fonction isolément, assurant au développeur que son code fonctionne comme prévu. Pour de nombreuses raisons, y compris la nécessité de garder nos tests rapides et sans état, les tests unitaires "écrasent" souvent les bases de données et autres dépendances externes (par exemple, les fonctions sont modifiées pour retourner des valeurs statiques prédéfinies, au lieu d'appeler la véritable base de données).
- **Tests d'acceptation** : Ils testent généralement l'application dans son ensemble pour assurer qu'un niveau de fonctionnalité plus élevé fonctionne comme prévu (par exemple, les critères d'acceptation métier pour une user story, la correction d'une API), et que les erreurs de régression n'ont pas été introduites (c'est-à-dire que nous n'avons

pas cassé une fonctionnalité qui fonctionnait correctement auparavant). Humble et Farley définissent la différence entre les tests unitaires et les tests d'acceptation ainsi : "Le but d'un test unitaire est de montrer qu'une partie unique de l'application fait ce que le programmeur entendait... L'objectif des tests d'acceptation est de prouver que notre application fait ce que le client voulait, non qu'elle fonctionne comme les programmeurs pensent qu'elle devrait." Après qu'un build passe nos tests unitaires, notre pipeline de déploiement l'exécute contre nos tests d'acceptation. Tout build qui passe nos tests d'acceptation est ensuite généralement mis à disposition pour des tests manuels (par exemple, des tests exploratoires, des tests UI, etc.), ainsi que pour des tests d'intégration.

- **Tests d'Intégration :** Les tests d'intégration visent à garantir que notre application interagit correctement avec d'autres applications et services en production, contrairement à l'utilisation d'interfaces simulées. Comme l'observent Humble et Farley, "beaucoup de travail dans l'environnement SIT (System Integration Testing) consiste à déployer de nouvelles versions de chaque application jusqu'à ce qu'elles coopèrent toutes. Dans cette situation, le test de fumée est généralement un ensemble complet de tests d'acceptation qui s'exécutent contre toute l'application." Les tests d'intégration sont effectués sur des builds qui ont passé nos tests unitaires et d'acceptation. Comme les tests d'intégration sont souvent fragiles, nous voulons minimiser leur nombre et trouver autant de nos défauts que possible lors des tests unitaires et d'acceptation. La capacité d'utiliser des versions virtuelles ou simulées des services distants lors de l'exécution des tests d'acceptation devient une exigence architecturale essentielle.

Face aux pressions des délais, les développeurs peuvent arrêter de créer des tests unitaires dans le cadre de leur travail quotidien, indépendamment de la manière dont nous avons défini le 'DONE. Pour détecter cela, nous pouvons choisir de mesurer et de rendre visible notre couverture de test (en fonction du nombre de classes, de lignes de code, de permutations, etc.), et peut-être même échouer notre suite de tests de validation lorsque cela tombe en dessous d'un certain niveau (par exemple, lorsque moins de 80 % de nos classes ont des tests unitaires).

Martin Fowler observe qu'en général, "un build [et un processus de test] de dix minutes est parfaitement raisonnable... [Nous commençons] par la compilation et les tests plus localisés avec la base de données complètement simulée. Ces tests peuvent s'exécuter très rapidement, respectant la ligne directrice des dix minutes. Cependant, les bugs impliquant des interactions plus larges, notamment ceux impliquant la base de données réelle, ne seront pas trouvés. Le build de la seconde étape exécute une suite de tests différente [tests d'acceptation] qui touche la base de données réelle et implique plus de comportements de bout en bout. Cette suite peut prendre quelques heures à s'exécuter."

Attraper les Erreurs le Plus Tôt Possible dans Nos Tests Automatisés

Un objectif de conception spécifique de notre suite de tests automatisés est de trouver les erreurs le plus tôt possible dans les tests. C'est pourquoi nous exécutons des tests automatisés plus rapides (par exemple, des tests unitaires) avant les tests automatisés plus lents (par

exemple, des tests d'acceptation et d'intégration), qui sont tous exécutés avant tout test manuel.

Un autre corollaire de ce principe est que toutes les erreurs doivent être trouvées avec la catégorie de tests la plus rapide possible. Si la plupart de nos erreurs sont trouvées dans nos tests d'acceptation et d'intégration, le retour d'information que nous fournissons aux développeurs est d'un ordre de grandeur plus lent qu'avec les tests unitaires, et les tests d'intégration nécessitent l'utilisation d'environnements de test d'intégration rares et complexes, qui ne peuvent être utilisés que par une équipe à la fois, retardant encore davantage les retours.

De plus, non seulement les erreurs détectées lors des tests d'intégration sont difficiles et chronophages à reproduire pour les développeurs, mais même valider qu'elles ont été corrigées est difficile (c'est-à-dire qu'un développeur crée une correction mais doit ensuite attendre quatre heures pour savoir si les tests d'intégration passent désormais).

Par conséquent, chaque fois que nous trouvons une erreur avec un test d'acceptation ou d'intégration, nous devrions créer un test unitaire qui pourrait trouver l'erreur plus rapidement, plus tôt et à moindre coût. Martin Fowler a décrit la notion de "pyramide de tests idéale", où nous sommes capables de détecter la plupart de nos erreurs en utilisant nos tests unitaires. (Voir figure 14.) En revanche, dans de nombreux programmes de test, c'est l'inverse qui est vrai, où la plupart des investissements sont dans les tests manuels et d'intégration.

Ideal vs. Non-Ideal Testing Pyramids

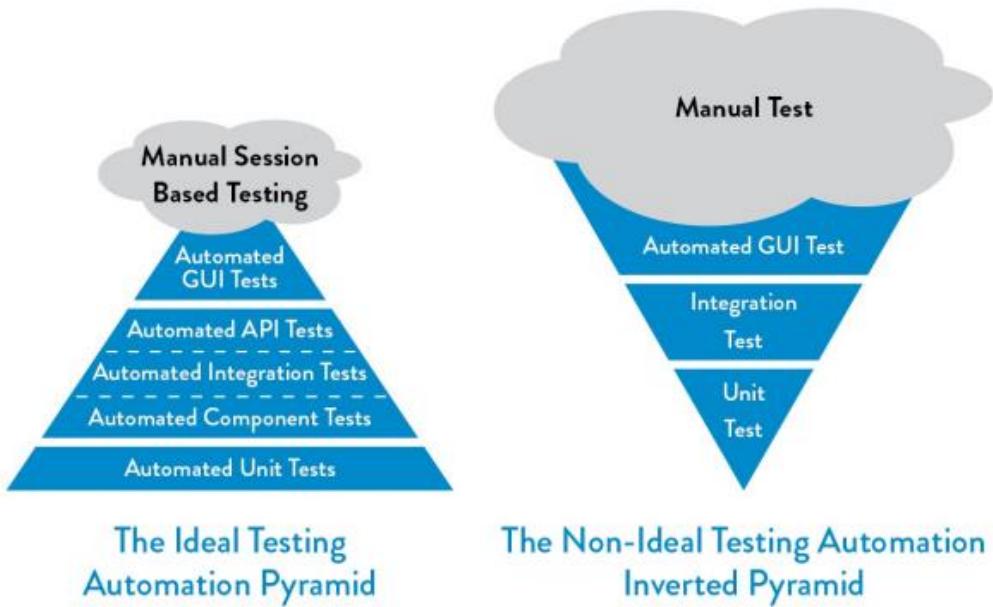


Figure 14: The ideal and non-ideal automated testing pyramids (Source: Martin Fowler, "TestPyramid.")

Si nous constatons que les tests unitaires ou d'acceptation sont trop difficiles et coûteux à écrire et à maintenir, il est probable que notre architecture soit trop fortement couplée, où la séparation stricte entre nos frontières de modules n'existe plus (ou n'a peut-être jamais existé). Dans ce cas, nous devrons créer un système plus faiblement couplé afin que les modules puissent être testés indépendamment sans environnements d'intégration. Des suites de tests d'acceptation pour même les applications les plus complexes, qui s'exécutent en quelques minutes, sont possibles.

Assurer que les tests s'exécutent rapidement (en parallèle, si nécessaire)

Comme nous voulons que nos tests s'exécutent rapidement, nous devons concevoir nos tests pour qu'ils s'exécutent en parallèle, potentiellement sur de nombreux serveurs différents. Nous pouvons également vouloir exécuter différentes catégories de tests en parallèle. Par exemple, lorsqu'un build passe nos tests d'acceptation, nous pouvons exécuter nos tests de performance en parallèle avec nos tests de sécurité, comme le montre la figure 15. Nous pouvons ou non autoriser les tests exploratoires manuels jusqu'à ce que le build ait passé tous nos tests automatisés - ce qui permet un retour plus rapide, mais peut également permettre des tests manuels sur des builds qui échoueront éventuellement.

Nous rendons disponible tout build qui passe tous nos tests automatisés pour les tests exploratoires, ainsi que pour d'autres formes de tests manuels ou intensifs en ressources

(comme les tests de performance). Nous voulons effectuer tous ces tests aussi fréquemment que possible et pratique, soit en continu, soit selon un calendrier.

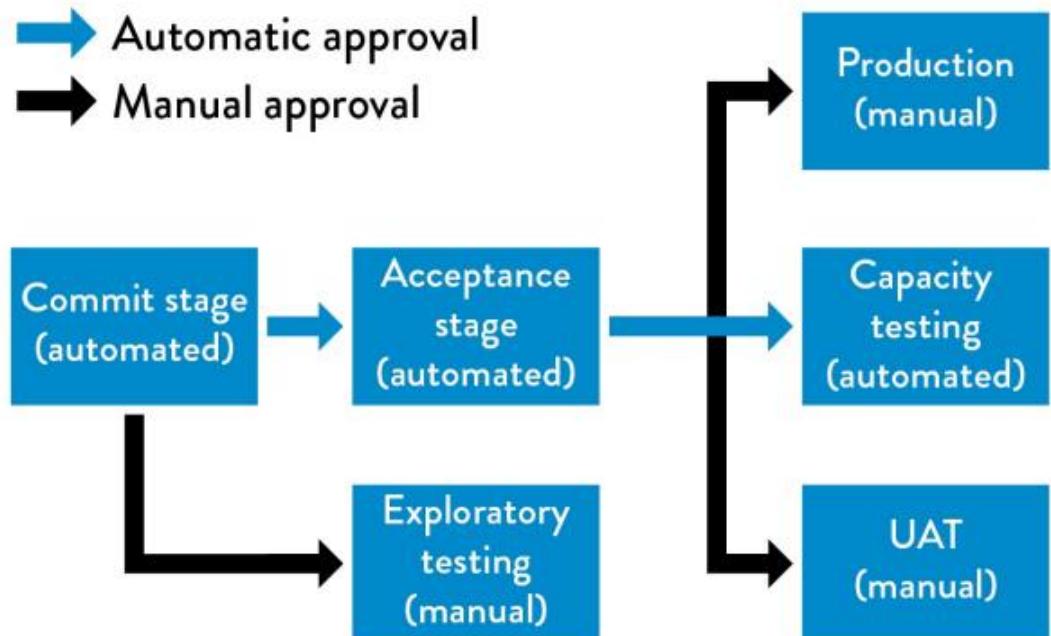


Figure 15: Running automated and manual tests in parallel

(Source: Humble and Farley, Continuous Delivery, Kindle edition, location 3868.)

Tout testeur (ce qui inclut tous nos développeurs) doit utiliser le dernier build ayant passé tous les tests automatisés, au lieu d'attendre que les développeurs signalent un build spécifique comme prêt à être testé. En faisant cela, nous nous assurons que les tests sont effectués le plus tôt possible dans le processus.

Écrire nos tests automatisés avant d'écrire le code (Développement piloté par les tests)

L'une des manières les plus efficaces de garantir que nous disposons de tests automatisés fiables est d'écrire ces tests dans le cadre de notre travail quotidien, en utilisant des techniques telles que le développement piloté par les tests (TDD) et le développement piloté par les tests d'acceptation (ATDD). Cela signifie que nous commençons chaque changement du système en écrivant d'abord un test automatisé qui valide que le comportement attendu échoue, puis nous écrivons le code pour faire passer les tests.

Cette technique a été développée par Kent Beck à la fin des années 1990 dans le cadre de l'Extreme Programming et comporte les trois étapes suivantes :

- Assurez-vous que les tests échouent. « Écrivez un test pour la prochaine fonctionnalité que vous souhaitez ajouter. » Validez dans le dépôt.

- Assurez-vous que les tests passent. « Écrivez le code fonctionnel jusqu'à ce que le test passe. » Validez dans le dépôt.
- « Refactorisez le code nouveau et ancien pour qu'il soit bien structuré. » Assurez-vous que les tests passent. Validez de nouveau.

Ces suites de tests automatisés sont enregistrées dans le contrôle de version aux côtés de notre code, ce qui constitue une spécification vivante et à jour du système. Les développeurs souhaitant comprendre comment utiliser le système peuvent consulter cette suite de tests pour trouver des exemples pratiques de l'utilisation de l'API du système.

Automatiser le plus possible nos tests manuels

Notre objectif est de trouver autant d'erreurs de code que possible grâce à nos suites de tests automatisés, réduisant ainsi notre dépendance aux tests manuels. Dans sa présentation de 2013 à Flowcon intitulée « On the Care and Feeding of Feedback Cycles », Elisabeth Hendrickson a observé : « Bien que les tests puissent être automatisés, la création de la qualité ne le peut pas. Faire exécuter par des humains des tests qui devraient être automatisés est un gaspillage de potentiel humain. »

En faisant cela, nous permettons à tous nos testeurs (qui, bien sûr, incluent les développeurs) de se concentrer sur des activités à haute valeur ajoutée qui ne peuvent pas être automatisées, comme les tests exploratoires ou l'amélioration du processus de test lui-même.

Cependant, l'automatisation de tous nos tests manuels peut entraîner des résultats indésirables - nous ne voulons pas de tests automatisés peu fiables ou générant de faux positifs (c'est-à-dire des tests qui auraient dû réussir parce que le code est fonctionnellement correct, mais qui ont échoué en raison de problèmes tels que des performances lentes, des délais d'attente, un état initial non contrôlé ou un état involontaire dû à l'utilisation de stubs de base de données ou d'environnements de test partagés).

Les tests peu fiables qui génèrent des faux positifs créent des problèmes importants - ils gaspillent du temps précieux (par exemple, en obligeant les développeurs à relancer le test pour déterminer s'il y a réellement un problème), augmentent l'effort global de l'exécution et de l'interprétation de nos résultats de tests, et souvent, les développeurs stressés ignorent complètement les résultats des tests ou désactivent les tests automatisés pour se concentrer sur la création de code.

Le résultat est toujours le même : nous détectons les problèmes plus tard, ils sont plus difficiles à résoudre, et nos clients ont de moins bons résultats, ce qui crée du stress tout au long de la chaîne de valeur.

Pour atténuer cela, un petit nombre de tests automatisés fiables est presque toujours préférable à un grand nombre de tests manuels ou de tests automatisés peu fiables. Par conséquent, nous nous concentrons sur l'automatisation uniquement des tests qui valident réellement les objectifs commerciaux que nous essayons d'atteindre. Si l'abandon d'un test entraîne des défauts en production, nous devons le réintégrer à notre suite de tests manuels, avec l'idéal de l'automatiser éventuellement.

Comme l'a décrit Gary Gruver, ancien VP de l'ingénierie qualité, de l'ingénierie des versions et des opérations pour Macys.com, « Pour un grand site de commerce électronique de détaillant, nous sommes passés de l'exécution de 1 300 tests manuels tous les dix jours à l'exécution de seulement dix tests automatisés à chaque validation de code - il est bien préférable d'exécuter quelques tests que nous faisons confiance plutôt que d'exécuter des tests qui ne sont pas fiables. Au fil du temps, nous avons fait croître cette suite de tests pour inclure des centaines de milliers de tests automatisés. »

En d'autres termes, nous commençons par un petit nombre de tests automatisés fiables et nous les ajoutons au fil du temps, créant ainsi un niveau d'assurance croissant que nous détecterons rapidement toute modification du système qui nous ferait sortir d'un état déployable.

Intégrer les tests de performance dans notre suite de tests

Trop souvent, nous découvrons que notre application fonctionne mal lors des tests d'intégration ou après son déploiement en production. Les problèmes de performance sont souvent difficiles à détecter, comme lorsque les performances se dégradent avec le temps, passant inaperçues jusqu'à ce qu'il soit trop tard (par exemple, des requêtes de base de données sans index). De nombreux problèmes sont également difficiles à résoudre, surtout lorsqu'ils sont causés par des décisions architecturales que nous avons prises ou par des limitations imprévues de notre réseau, de notre base de données, de notre stockage ou d'autres systèmes.

Notre objectif est d'écrire et d'exécuter des tests de performance automatisés qui valident nos performances sur l'ensemble de la pile d'application (code, base de données, stockage, réseau, virtualisation, etc.) en tant que partie du pipeline de déploiement, afin de détecter les problèmes tôt, lorsque les corrections sont les moins coûteuses et les plus rapides.

En comprenant le comportement de notre application et de nos environnements sous une charge similaire à celle de la production, nous pouvons mieux planifier la capacité, ainsi que détecter des conditions telles que :

- Lorsque les temps de requête de base de données augmentent de manière non linéaire (par exemple, lorsque nous oublions d'activer l'indexation de la base de données et que le temps de chargement de la page passe de cent minutes à trente secondes).
- Lorsqu'un changement de code entraîne une multiplication par dix du nombre d'appels à la base de données, de l'utilisation du stockage ou du trafic réseau.
- Lorsque nous disposons de tests d'acceptation pouvant être exécutés en parallèle, nous pouvons les utiliser comme base de nos tests de performance. Par exemple, supposons que nous exploitons un site de commerce électronique et que nous avons identifié « recherche » et « paiement » comme deux opérations à haute valeur ajoutée devant bien fonctionner sous charge. Pour tester cela, nous pouvons exécuter des milliers de tests d'acceptation de recherche en parallèle avec des milliers de tests d'acceptation de paiement.

En raison de la grande quantité de calcul et d'E/S requise pour exécuter des tests de performance, la création d'un environnement de test de performance peut être plus complexe que la création de l'environnement de production pour l'application elle-même. Pour cette raison, nous pouvons construire notre environnement de test de performance au début de tout projet et nous assurer de dédier les ressources nécessaires pour le construire tôt et correctement.

Pour détecter les problèmes de performance tôt, nous devrions consigner les résultats de performance et évaluer chaque exécution de performance par rapport aux résultats précédents. Par exemple, nous pourrions faire échouer les tests de performance si les performances diffèrent de plus de 2 % par rapport à l'exécution précédente.

Intégrer les tests des exigences non fonctionnelles dans notre suite de tests

En plus de tester que notre code fonctionne comme prévu et qu'il performe sous des charges similaires à la production, nous voulons également valider tous les autres attributs du système qui nous importent. Ceux-ci sont souvent appelés les exigences non fonctionnelles, qui incluent la disponibilité, la scalabilité, la capacité, la sécurité, et ainsi de suite.

Beaucoup de ces exigences sont remplies par la configuration correcte de nos environnements, donc nous devons également créer des tests automatisés pour valider que nos environnements sont configurés de manière sécurisée et correcte (par exemple, server-spec).

En outre, tout comme nous exécutons des outils d'analyse sur notre application dans notre pipeline de déploiement (par exemple, analyse statique du code, analyse de la couverture des tests), nous devrions exécuter des outils qui analysent le code qui construit nos environnements (par exemple, Foodcritic pour Chef, puppet-lint pour Puppet).

Lorsque nous utilisons des outils de gestion de configuration d'infrastructure en tant que code (comme Puppet, Chef, Ansible, Salt), nous pouvons utiliser les mêmes frameworks de test que ceux utilisés pour tester notre code afin de tester également la configuration et le déploiement corrects de nos environnements.

En résumé, en intégrant les tests de performance et les tests des exigences non fonctionnelles dans notre suite de tests automatisés, nous augmentons notre capacité à détecter et corriger rapidement les problèmes potentiels avant qu'ils n'affectent nos utilisateurs finaux. Cela contribue à maintenir notre pipeline de déploiement en état de fonctionnement optimal et à garantir que nous livrons continuellement de la valeur à nos clients.

Actionner notre corde Andon lorsque le pipeline de déploiement est interrompu

Lorsque nous avons un build vert dans notre pipeline de déploiement, nous avons un degré élevé de confiance que notre code et notre environnement fonctionneront comme prévu lorsque nous déployons nos changements en production.

Afin de maintenir notre pipeline de déploiement en état vert, nous allons créer un "Cordon Andon" virtuel, similaire à celui utilisé dans le Système de Production Toyota. Chaque fois que quelqu'un introduit un changement qui provoque un échec dans notre build ou nos tests automatisés, aucun nouveau travail n'est autorisé à entrer dans le système tant que le problème n'est pas résolu. Si quelqu'un a besoin d'aide pour résoudre le problème, il peut faire appel à toute l'aide nécessaire, comme dans l'exemple de Google au début de ce chapitre.

Lorsque notre pipeline de déploiement est cassé, nous notifions au moins toute l'équipe de l'échec, afin que n'importe qui puisse soit corriger le problème, soit annuler le commit. Nous pouvons même configurer le système de contrôle de version pour empêcher de nouveaux commits de code tant que la première étape (c'est-à-dire les builds et les tests unitaires) du pipeline de déploiement n'est pas de retour en état vert. Si le problème était dû à un test automatisé générant une erreur fausse positive, le test incriminé devrait être réécrit ou supprimé.

Chaque membre de l'équipe doit être habilité à annuler le commit pour retrouver un état vert. Randy Shoup, ancien directeur de l'ingénierie pour Google App Engine, a souligné l'importance de ramener le déploiement à un état vert : "Nous priorisons les objectifs de l'équipe par-dessus les objectifs individuels - chaque fois que nous aidons quelqu'un à faire avancer son travail, nous aidons toute l'équipe. Cela s'applique que nous aidions à corriger le build, un test automatisé, ou même à effectuer une revue de code pour eux. Et bien sûr, nous savons qu'ils feront de même pour nous quand nous aurons besoin d'aide. Ce système fonctionnait sans beaucoup de formalisme ou de politique - tout le monde savait que notre travail n'était pas seulement 'écrire du code', mais 'faire fonctionner un service'. C'est pourquoi nous donnions la priorité à toutes les questions de qualité, en particulier celles liées à la fiabilité et à l'évolutivité,

au plus haut niveau, les traitant comme des problèmes de priorité 0 'bloquant'. D'un point de vue systémique, ces pratiques nous empêchent de régresser."

Lorsque les étapes ultérieures du pipeline de déploiement échouent, comme les tests d'acceptation ou de performance, au lieu d'arrêter tout nouveau travail, nous avons des développeurs et des testeurs en astreinte chargés de résoudre ces problèmes immédiatement. Ils doivent également créer de nouveaux tests qui s'exécutent à une étape antérieure dans le pipeline de déploiement pour détecter toute régression future. Par exemple, si nous découvrons un défaut dans nos tests d'acceptation, nous devrions écrire un test unitaire pour le détecter. De même, si nous découvrons un défaut lors de tests exploratoires, nous devrions écrire un test unitaire ou d'acceptation.

Pour accroître la visibilité des échecs des tests automatisés, nous devrions créer des indicateurs très visibles afin que toute l'équipe puisse voir quand nos builds ou nos tests automatisés échouent. De nombreuses équipes ont créé des lumières de build hautement visibles montées sur un mur, indiquant l'état actuel du build, ou d'autres moyens ludiques pour informer l'équipe que le build est cassé, comme des lampes à lave, la lecture d'un échantillon vocal ou d'une chanson, des klaxons, des feux de circulation, etc.

À bien des égards, cette étape est plus difficile que la création de nos builds et serveurs de tests - ceux-ci étaient des activités purement techniques, tandis que cette étape nécessite un changement de comportement humain et des incitations. Cependant, l'intégration continue et la livraison continue nécessitent ces changements, comme nous l'explorons dans la prochaine section.

Pourquoi nous devons tirer le cordon Andon

Ne pas tirer le cordon Andon et ne pas résoudre immédiatement les problèmes du pipeline de déploiement entraîne le problème trop familier où il devient de plus en plus difficile de ramener nos applications et notre environnement dans un état déployable. Considérons la situation suivante :

- Quelqu'un commet du code qui casse le build ou nos tests automatisés, mais personne ne le corrige.
- Quelqu'un d'autre commet un autre changement sur le build cassé, qui ne passe pas non plus nos tests automatisés - mais personne ne voit les résultats de test en échec qui nous auraient permis de voir le nouveau défaut, encore moins de le corriger.
- Nos tests existants ne s'exécutent pas de manière fiable, donc il est très peu probable que nous développions de nouveaux tests. (Pourquoi s'embêter ? Nous ne parvenons même pas à faire fonctionner les tests actuels.)

Lorsque cela se produit, nos déploiements vers n'importe quel environnement deviennent aussi peu fiables que lorsque nous n'avions pas de tests automatisés ou que nous utilisions une méthode en cascade, où la majorité de nos problèmes sont découverts en production. Le résultat inévitable de ce cycle vicieux est que nous finissons là où nous avons commencé, avec

une "phase de stabilisation" imprévisible qui prend des semaines ou des mois, où toute notre équipe est plongée dans la crise, essayant de faire passer tous nos tests, prenant des raccourcis en raison des pressions de délais et ajoutant à notre dette technique.

Conclusion

Dans ce chapitre, nous avons créé un ensemble complet de tests automatisés pour confirmer que nous disposons d'un build vert qui est toujours dans un état passant et déployable. Nous avons organisé nos suites de tests et nos activités de test dans un pipeline de déploiement. Nous avons également créé la norme culturelle de faire tout ce qui est nécessaire pour retrouver un état de build vert si quelqu'un introduit un changement qui casse l'un de nos tests automatisés.

En agissant ainsi, nous préparons le terrain pour la mise en œuvre de l'intégration continue, qui permet à de nombreuses petites équipes de développer, tester et déployer indépendamment et en toute sécurité du code en production, en fournissant de la valeur aux clients.

Activer et pratiquer l'intégration continue

Dans le chapitre précédent, nous avons créé des pratiques de test automatisé pour garantir que les développeurs reçoivent rapidement des retours sur la qualité de leur travail. Cela devient encore plus important à mesure que nous augmentons le nombre de développeurs et le nombre de branches sur lesquelles ils travaillent dans le système de contrôle de version.

La capacité à « brancher » dans les systèmes de contrôle de version a été créée principalement pour permettre aux développeurs de travailler sur différentes parties du système logiciel en parallèle, sans risquer que des développeurs individuels valident des modifications pouvant déstabiliser ou introduire des erreurs dans le tronc (parfois aussi appelé master ou mainline).

Cependant, plus les développeurs sont autorisés à travailler dans leurs branches en isolation, plus il devient difficile d'intégrer et de fusionner les modifications de chacun dans le tronc. En fait, l'intégration de ces modifications devient exponentiellement plus difficile à mesure que nous augmentons le nombre de branches et le nombre de modifications dans chaque branche de code.

Les problèmes d'intégration entraînent une quantité importante de réajustements pour revenir à un état déployable, y compris des modifications conflictuelles qui doivent être fusionnées manuellement ou des fusions qui cassent nos tests automatisés ou manuels, nécessitant généralement l'intervention de plusieurs développeurs pour être résolues avec succès. Et comme l'intégration a traditionnellement été réalisée à la fin du projet, lorsqu'elle prend beaucoup plus de temps que prévu, nous sommes souvent obligés de faire des compromis pour respecter la date de livraison.

Cela provoque une autre spirale descendante : lorsque la fusion du code est pénible, nous avons tendance à le faire moins souvent, rendant les fusions futures encore pires. L'intégration continue a été conçue pour résoudre ce problème en intégrant la fusion dans le tronc dans le travail quotidien de chacun.

La surprenante étendue des problèmes que l'intégration continue résout, ainsi que les solutions elles-mêmes, sont illustrées par l'expérience de Gary Gruver en tant que directeur de l'ingénierie pour la division des micrologiciels LaserJet de HP, qui développe les micrologiciels exécutant tous leurs scanners, imprimantes et appareils multifonctions.

L'équipe comptait quatre cents développeurs répartis aux États-Unis, au Brésil et en Inde. Malgré la taille de leur équipe, ils progressaient beaucoup trop lentement. Pendant des années, ils n'ont pas pu livrer de nouvelles fonctionnalités aussi rapidement que l'entreprise en avait besoin.

Gruver a décrit le problème ainsi : « Le marketing venait nous voir avec un million d'idées pour éblouir nos clients, et nous leur disions simplement : "Parmi votre liste, choisissez les deux choses que vous aimerez obtenir dans les six à douze prochains mois." »

Ils ne réalisaient que deux versions de micrologiciels par an, la majeure partie de leur temps étant consacrée au portage de code pour supporter de nouveaux produits. Gruver estimait que seulement 5 % de leur temps était consacré à la création de nouvelles fonctionnalités - le reste du temps était passé à des travaux non productifs liés à leur dette technique, comme la gestion de multiples branches de code et les tests manuels, comme indiqué ci-dessous :

- 20 % en planification détaillée (leur faible débit et leurs longs délais étaient attribués à tort à de mauvaises estimations, et donc, dans l'espoir d'obtenir une meilleure réponse, ils étaient invités à estimer le travail plus en détail)
- 25 % consacrés au portage de code, tous maintenus sur des branches de code séparées
- 10 % consacrés à l'intégration de leur code entre les branches de développeurs
- 15 % consacrés à réaliser des tests manuels

Gruver et son équipe se sont fixé pour objectif d'augmenter le temps passé sur l'innovation et les nouvelles fonctionnalités par un facteur de dix. L'équipe espérait atteindre cet objectif grâce à :

- L'intégration continue et le développement basé sur le tronc
- Un investissement significatif dans l'automatisation des tests
- La création d'un simulateur matériel permettant d'exécuter les tests sur une plateforme virtuelle
- La reproduction des échecs de test sur les postes de travail des développeurs
- Une nouvelle architecture pour supporter l'exécution de toutes les imprimantes à partir d'une construction et d'une version communes

Auparavant, chaque gamme de produits nécessitait une nouvelle branche de code, chaque modèle ayant une construction de micrologiciel unique avec des capacités définies au moment de la compilation. La nouvelle architecture aurait tous les développeurs travaillant dans une base de code commune, avec une seule version de micrologiciel supportant tous les modèles LaserJet construits à partir du tronc, les capacités des imprimantes étant établies à l'exécution dans un fichier de configuration XML.

Quatre ans plus tard, ils avaient une base de code supportant toutes les vingt-quatre gammes de produits HP LaserJet développées sur le tronc. Gruver admet que le développement basé sur le tronc nécessite un grand changement de mentalité. Les ingénieurs pensaient que le développement basé sur le tronc ne fonctionnerait jamais, mais une fois qu'ils ont commencé, ils ne pouvaient plus imaginer revenir en arrière. Au fil des ans, plusieurs ingénieurs ont quitté HP, et ils m'appelaient pour me parler du développement rétrograde dans leurs nouvelles entreprises, soulignant à quel point il est difficile d'être efficace et de livrer du bon code sans le retour d'information que l'intégration continue leur donnait.

Cependant, le développement basé sur le tronc leur a demandé de construire des tests automatisés plus efficaces. Gruver a observé : « Sans tests automatisés, l'intégration continue est le moyen le plus rapide d'obtenir un gros tas de déchets qui ne se compile jamais ou ne fonctionne jamais correctement. » Au début, un cycle complet de tests manuels nécessitait six semaines.

Pour que toutes les constructions de micrologiciels soient testées automatiquement, ils ont beaucoup investi dans leurs simulateurs d'imprimantes et ont créé une ferme de tests en six semaines - en quelques années, deux mille simulateurs d'imprimantes fonctionnaient sur six racks de serveurs qui chargeaient les constructions de micrologiciels à partir de leur pipeline de déploiement. Leur système d'intégration continue (CI) exécutait l'ensemble de leurs tests unitaires automatisés, tests d'acceptation et tests d'intégration sur les constructions à partir du tronc, comme décrit dans le chapitre précédent.

De plus, ils ont créé une culture qui arrêtait tout travail chaque fois qu'un développeur cassait le pipeline de déploiement, garantissant que les développeurs ramenaient rapidement le système à un état stable.

Les tests automatisés ont créé des retours rapides permettant aux développeurs de confirmer rapidement que leur code validé fonctionnait réellement. Les tests unitaires s'exécutaient sur leurs postes de travail en quelques minutes, trois niveaux de tests automatisés s'exécutaient à chaque validation ainsi que toutes les deux et quatre heures. Le test de régression complet final s'exécutait toutes les vingt-quatre heures. Pendant ce processus, ils ont :

- Réduit la construction à une par jour, pour finalement faire dix à quinze constructions par jour
- Passé d'environ vingt validations par jour effectuées par un « boss des builds » à plus de cent validations par jour effectuées par des développeurs individuels
- Permis aux développeurs de changer ou d'ajouter 75k-100k lignes de code chaque jour
- Réduit les temps de tests de régression de six semaines à un jour

Ce niveau de productivité n'aurait jamais pu être soutenu avant d'adopter l'intégration continue, lorsque créer une construction stable nécessitait des jours d'efforts héroïques. Les avantages commerciaux résultants étaient stupéfiants :

- Le temps passé à conduire l'innovation et à écrire de nouvelles fonctionnalités est passé de 5 % du temps des développeurs à 40 %.
- Les coûts de développement globaux ont été réduits d'environ 40 %.
- Les programmes en cours de développement ont augmenté d'environ 140 %.
- Les coûts de développement par programme ont diminué de 78 %.

L'expérience de Gruver montre que, après l'utilisation complète du contrôle de version, l'intégration continue est l'une des pratiques les plus critiques permettant un flux rapide de travail dans notre flux de valeur, permettant à de nombreuses équipes de développement de développer, tester et livrer de la valeur de manière indépendante. Néanmoins, l'intégration continue reste une pratique controversée. Le reste de ce chapitre décrit les pratiques nécessaires pour mettre en œuvre l'intégration continue, ainsi que comment surmonter les objections courantes.

Développement en petits lots et ce qui se passe lorsque nous validons du code dans le tronc peu fréquemment

Comme décrit dans les chapitres précédents, chaque fois que des changements sont introduits dans le contrôle de version qui font échouer notre pipeline de déploiement, nous nous rassemblons rapidement pour résoudre le problème et ramener notre pipeline de déploiement à un état stable. Cependant, des problèmes significatifs surviennent lorsque les développeurs travaillent dans des branches privées de longue durée (également appelées "branches de fonctionnalités"), ne fusionnant dans le tronc que sporadiquement, ce qui entraîne une grande taille de lot de modifications. Comme décrit dans l'exemple HP LaserJet, cela entraîne un chaos important et des réajustements pour mettre leur code dans un état publiable.

Jeff Atwood, fondateur du site Stack Overflow et auteur du blog Coding Horror, observe que bien qu'il existe de nombreuses stratégies de branchement, elles peuvent toutes être placées sur le spectre suivant :

- **Optimiser pour la productivité individuelle** : Chaque personne du projet travaille dans sa propre branche privée. Tout le monde travaille indépendamment, et personne ne peut perturber le travail de quiconque ; cependant, la fusion devient un cauchemar. La collaboration devient presque comiquement difficile - le travail de chaque personne doit être minutieusement fusionné avec celui de tous les autres pour voir même la plus petite partie du système complet.
- **Optimiser pour la productivité de l'équipe** : Tout le monde travaille dans la même zone commune. Il n'y a pas de branches, juste une ligne droite et ininterrompue de développement. Il n'y a rien à comprendre, donc les validations sont simples, mais chaque validation peut casser l'ensemble du projet et arrêter tout progrès.

L'observation d'Atwood est absolument correcte - dit plus précisément, l'effort nécessaire pour fusionner avec succès les branches augmente exponentiellement à mesure que le nombre de branches augmente. Le problème réside non seulement dans le réajustement que crée cet "enfer de fusion", mais aussi dans le retour d'information retardé que nous recevons de notre pipeline de déploiement. Par exemple, au lieu de tester les performances d'un système entièrement intégré de manière continue, cela se produira probablement uniquement à la fin de notre processus.

De plus, à mesure que nous augmentons le rythme de production de code en ajoutant plus de développeurs, nous augmentons la probabilité qu'une modification donnée impacte quelqu'un d'autre et augmente le nombre de développeurs qui seront impactés lorsqu'une personne casse le pipeline de déploiement.

Voici un dernier effet secondaire troublant des fusions de grande taille de lot : lorsque la fusion est difficile, nous sommes moins capables et motivés pour améliorer et refactoriser notre code, car les refactorisations sont plus susceptibles de provoquer des réajustements pour tout le monde. Lorsque cela se produit, nous sommes plus réticents à modifier du code qui a des dépendances dans toute la base de code, ce qui est (tragiquement) là où nous pourrions avoir les rendements les plus élevés.

C'est ainsi que Ward Cunningham, développeur du premier wiki, a décrit pour la première fois la dette technique : lorsque nous ne refactorisons pas agressivement notre base de code, il devient plus difficile d'apporter des modifications et de maintenir au fil du temps, ralentissant le rythme auquel nous pouvons ajouter de nouvelles fonctionnalités. Résoudre ce problème était l'une des principales raisons derrière la création des pratiques d'intégration continue et de développement basé sur le tronc, pour optimiser la productivité de l'équipe par rapport à la productivité individuelle.

Adopter les pratiques de développement basées sur le tronc

Notre contre-mesure aux fusions de grande taille de lot consiste à instituer des pratiques d'intégration continue et de développement basé sur le tronc, où tous les développeurs valident leur code dans le tronc au moins une fois par jour. Valider le code aussi fréquemment réduit notre taille de lot au travail effectué par l'ensemble de notre équipe de développeurs en une seule journée. Plus les développeurs valident leur code fréquemment dans le tronc, plus la taille des lots est petite et plus nous nous rapprochons de l'idéal théorique du flux pièce unique.

Des validations de code fréquentes dans le tronc signifient que nous pouvons exécuter tous les tests automatisés sur notre système logiciel dans son ensemble et recevoir des alertes lorsqu'un changement casse une autre partie de l'application ou interfère avec le travail d'un autre développeur. Et parce que nous pouvons détecter les problèmes de fusion lorsqu'ils sont petits, nous pouvons les corriger plus rapidement.

Nous pouvons même configurer notre pipeline de déploiement pour rejeter toute validation (par exemple, modifications de code ou d'environnement) qui nous retire d'un état déployable. Cette méthode est appelée validations verrouillées, où le pipeline de déploiement confirme d'abord que la modification soumise sera fusionnée avec succès, se construira comme prévu et passera tous les tests automatisés avant d'être effectivement fusionnée dans le tronc. Sinon, le développeur sera notifié, permettant des corrections sans impacter quiconque dans le flux de valeur.

Et La discipline des validations de code quotidiennes nous oblige également à décomposer notre travail en morceaux plus petits tout en gardant le tronc dans un état fonctionnel et publiable. Et le contrôle de version devient un mécanisme intégral de la manière dont l'équipe communique entre elle - tout le monde a une meilleure compréhension partagée du système, est au courant de l'état du pipeline de déploiement et peut s'entraider lorsqu'il est cassé. En conséquence, nous obtenons une meilleure qualité et des délais de déploiement plus rapides.

Avec ces pratiques en place, nous pouvons maintenant modifier à nouveau notre définition de "DONE" : "À la fin de chaque intervalle de développement, nous devons avoir du code intégré, testé, fonctionnel et potentiellement livrable, démontré dans un environnement de type production, créé à partir du tronc à l'aide d'un processus en un clic, et validé par des tests automatisés."

Respecter cette définition révisée de "DONE" nous aide à garantir davantage la testabilité et la déployabilité continues du code que nous produisons. En gardant notre code dans un état déployable, nous sommes en mesure d'éliminer la pratique courante d'avoir une phase distincte de test et de stabilisation à la fin du projet.

Étude de cas - Intégration continue chez Bazaarvoice (2012)

Ernest Mueller, qui a contribué à la transformation DevOps chez National Instruments, a ensuite aidé à transformer les processus de développement et de publication chez Bazaarvoice en 2012. Bazaarvoice fournit du contenu généré par les clients (par exemple, avis, évaluations) pour des milliers de détaillants, tels que Best Buy, Nike et Walmart.

À cette époque, Bazaarvoice avait un chiffre d'affaires de 120 millions de dollars et se préparait à une introduction en bourse.

L'activité était principalement soutenue par l'application Bazaarvoice Conversations, une application monolithique en Java composée de près de cinq millions de lignes de code datant de 2006, couvrant quinze mille fichiers. Le service fonctionnait sur 1 200 serveurs répartis sur quatre centres de données et plusieurs fournisseurs de services cloud.

En partie à cause du passage à un processus de développement Agile et à des intervalles de développement de deux semaines, il y avait un désir énorme d'augmenter la fréquence des publications par rapport à leur calendrier de publication en production de dix semaines. Ils avaient également commencé à découpler des parties de leur application monolithique, en la décomposant en microservices.

Leur première tentative d'un calendrier de publication de deux semaines a eu lieu en janvier 2012. Mueller a observé : « Cela ne s'est pas bien passé. Cela a causé un chaos massif, avec quarante-quatre incidents de production signalés par nos clients. La réaction principale de la direction a été essentiellement "Ne faisons plus jamais cela". »

Mueller a pris en charge les processus de publication peu de temps après, avec pour objectif de faire des publications bihebdomadaires sans causer d'interruption pour les clients. Les objectifs commerciaux pour des publications plus fréquentes comprenaient la possibilité de faire des tests A/B plus rapidement (déscrits dans les chapitres suivants) et d'augmenter le flux de fonctionnalités en production.

Mueller a identifié trois problèmes principaux :

- Le manque d'automatisation des tests rendait tout niveau de test pendant les intervalles de deux semaines insuffisant pour prévenir des échecs à grande échelle.
- La stratégie de branchement du contrôle de version permettait aux développeurs de valider du nouveau code jusqu'à la publication en production.
- Les équipes gérant les microservices effectuaient également des publications indépendantes, ce qui causait souvent des problèmes lors de la publication du monolith ou vice versa.

Mueller a conclu que le processus de déploiement de l'application monolithique Conversations devait être stabilisé, ce qui nécessitait une intégration continue. Au cours des six semaines suivantes, les développeurs ont cessé de travailler sur les fonctionnalités pour se concentrer sur l'écriture de suites de tests automatisés, y compris des tests unitaires en JUnit, des tests de régression en Selenium, et la mise en place d'un pipeline de déploiement dans TeamCity.

« En exécutant ces tests tout le temps, nous sentions que nous pouvions apporter des modifications avec un certain niveau de sécurité. Et surtout, nous pouvions immédiatement savoir quand quelqu'un cassait quelque chose, au lieu de le découvrir seulement après que ce soit en production. »

Ils ont également changé pour un modèle de publication tronc/branche, où toutes les deux semaines, ils créaient une nouvelle branche de publication dédiée, sans nouvelles validations autorisées sur cette branche sauf en cas d'urgence - toutes les modifications passaient par un processus d'approbation, soit par ticket, soit par équipe via leur wiki interne. Cette branche passait ensuite par un processus de QA, puis était promue en production.

Les améliorations en termes de prévisibilité et de qualité des publications étaient stupéfiantes :

- Publication de janvier 2012 : quarante-quatre incidents clients (début de l'effort d'intégration continue)
- Publication du 6 mars 2012 : cinq jours de retard, cinq incidents clients
- Publication du 22 mars 2012 : à l'heure, un incident client
- Publication du 5 avril 2012 : à l'heure, zéro incident client

Mueller a décrit plus en détail le succès de cet effort :

Nous avons eu tellement de succès avec les publications toutes les deux semaines que nous sommes passés à des publications hebdomadaires, ce qui n'a presque pas nécessité de changements de la part des équipes d'ingénierie. Parce que les publications sont devenues si routinières, il suffisait de doubler le nombre de publications sur le calendrier et de publier lorsque le calendrier nous le disait. Sérieusement, c'était presque un non-événement. La majorité des changements nécessaires étaient dans nos équipes de service client et de marketing, qui ont dû changer leurs processus, comme modifier le calendrier de leurs courriels

hebdomadaires aux clients pour s'assurer que les clients savaient que des modifications de fonctionnalités étaient à venir. Après cela, nous avons commencé à travailler vers nos prochains objectifs, ce qui nous a finalement amenés à accélérer nos temps de test de plus de trois heures à moins d'une heure, à réduire le nombre d'environnements de quatre à trois (Dev, Test, Production, en éliminant Staging), et à passer à un modèle de livraison continue complet où nous permettons des déploiements rapides en un clic.

Conclusion

Le développement basé sur le tronc est probablement la pratique la plus controversée discutée dans ce livre. De nombreux ingénieurs ne croiront pas que c'est possible, même ceux qui préfèrent travailler sans interruption sur une branche privée sans avoir à gérer les autres développeurs. Cependant, les données du rapport State of DevOps 2015 de Puppet Labs sont claires : le développement basé sur le tronc prédit une productivité plus élevée et une meilleure stabilité, ainsi qu'une plus grande satisfaction au travail et des taux de burn-out plus faibles.

Bien qu'il puisse être difficile de convaincre les développeurs au début, une fois qu'ils verront les avantages extraordinaires, ils deviendront probablement des convertis à vie, comme l'illustrent les exemples de HP LaserJet et Bazaarvoice. Les pratiques d'intégration continue préparent le terrain pour la prochaine étape, qui consiste à automatiser le processus de déploiement et à permettre des publications à faible risque.

Automatiser et permettre des déploiements à faible risque

Chuck Rossi est le directeur de l'ingénierie des déploiements chez Facebook. Une de ses responsabilités est de superviser la mise en production quotidienne du code. En 2012, Rossi a décrit leur processus comme suit : "À partir de 13 heures environ, je passe en 'mode opérations' et je travaille avec mon équipe pour préparer le lancement des changements qui seront déployés sur Facebook.com ce jour-là. C'est la partie la plus stressante du travail et cela repose vraiment beaucoup sur le jugement et l'expérience passée de mon équipe. Nous travaillons pour nous assurer que tous ceux qui ont des changements à déployer soient présents et testent activement et soutiennent leurs modifications."

Juste avant la mise en production, tous les développeurs ayant des changements à déployer doivent être présents et se connecter sur leur canal de chat IRC - tout développeur non présent voit ses changements automatiquement retirés du paquet de déploiement. Rossi a poursuivi : "Si tout semble bon et que nos tableaux de bord de test et nos tests canaris sont verts, nous appuyons sur le gros bouton rouge et toute la flotte de serveurs de Facebook.com reçoit le nouveau code. En vingt minutes, des milliers et des milliers de machines sont mises à jour avec le nouveau code sans impact visible pour les utilisateurs du site."

Plus tard cette année-là, Rossi a doublé la fréquence de leurs déploiements de logiciels à deux fois par jour. Il a expliqué que la deuxième mise en production permettait aux ingénieurs ne se trouvant pas sur la côte ouest des États-Unis de "travailler et déployer aussi rapidement que n'importe quel autre ingénieur de l'entreprise," et donnait également à chacun une deuxième opportunité chaque jour de déployer du code et de lancer des fonctionnalités.

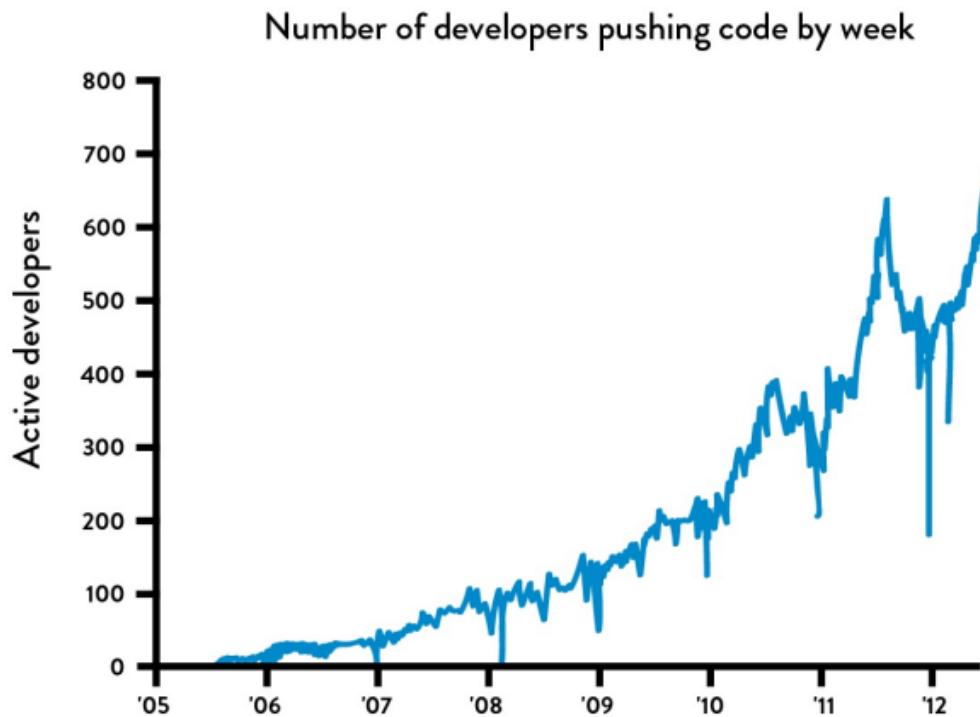


Figure 16: Number of developers deploying per week at Facebook
 (Source: Chuck Rossi, "Ship early and ship twice as often.")

Kent Beck, le créateur de la méthodologie Extreme Programming, l'un des principaux défenseurs du développement dirigé par les tests, et coach technique chez Facebook, commente également leur stratégie de déploiement de code dans un article publié sur sa page Facebook : "Chuck Rossi a observé qu'il semble y avoir un nombre fixe de changements que Facebook peut gérer lors d'un déploiement. Si nous voulons plus de changements, nous avons besoin de plus de déploiements. Cela a conduit à une augmentation constante du rythme des déploiements au cours des cinq dernières années, passant de déploiements hebdomadaires à quotidiens puis à trois fois par jour pour notre code PHP et de cycles de déploiement de six à quatre à deux semaines pour nos applications mobiles. Cette amélioration a été principalement pilotée par l'équipe d'ingénierie des déploiements."

En utilisant l'intégration continue et en rendant le déploiement de code un processus à faible risque, Facebook a permis que le déploiement de code fasse partie du travail quotidien de chacun et maintienne la productivité des développeurs. Cela nécessite que le déploiement de code soit automatisé, répétable et prévisible. Dans les pratiques décrites jusqu'à présent dans le livre, bien que notre code et nos environnements aient été testés ensemble, il est probable que nous ne déployions pas très souvent en production car les déploiements sont manuels, chronophages, pénibles, fastidieux et sujets aux erreurs, et impliquent souvent un transfert fastidieux et peu fiable entre le développement et les opérations.

Et parce que c'est pénible, nous avons tendance à le faire de moins en moins fréquemment, ce qui entraîne une autre spirale descendante auto-renforçante. En reportant les déploiements en production, nous accumulons des différences de plus en plus grandes entre le code à déployer et ce qui est en production, augmentant la taille des lots de déploiement. À mesure que la taille des lots de déploiement augmente, le risque d'événements inattendus associés au changement augmente également, ainsi que la difficulté à les corriger.

Dans ce chapitre, nous réduisons les frictions associées aux déploiements en production, en veillant à ce qu'ils puissent être effectués fréquemment et facilement, soit par les opérations, soit par le développement. Nous y parvenons en étendant notre pipeline de déploiement.

Au lieu de simplement intégrer continuellement notre code dans un environnement semblable à la production, nous permettrons la promotion en production de toute version qui passe notre processus de test et de validation automatisé, soit sur demande (c'est-à-dire en appuyant sur un bouton) soit automatiquement (c'est-à-dire toute version qui passe tous les tests est automatiquement déployée).

En raison du nombre de pratiques présentées, des notes de bas de page étendues sont fournies avec de nombreux exemples et informations supplémentaires, sans interrompre la présentation des concepts dans le chapitre.

AUTOMATISER NOTRE PROCESSUS DE DÉPLOIEMENT

Pour obtenir des résultats similaires à ceux de Facebook, il faut disposer d'un mécanisme automatisé qui déploie notre code en production. Surtout si nous avons un processus de déploiement en place depuis des années, nous devons documenter entièrement les étapes du processus de déploiement, comme dans un exercice de cartographie de flux de valeur, que nous pouvons assembler lors d'un atelier ou documenter de manière incrémentielle (par exemple, dans un wiki).

Une fois que nous avons documenté le processus, notre objectif est de simplifier et d'automatiser autant que possible les étapes manuelles, telles que :

- Emballer le code de manière appropriée pour le déploiement
- Créer des images de machine virtuelle préconfigurées ou des conteneurs
- Automatiser le déploiement et la configuration des intergiciels
- Copier des paquets ou des fichiers sur les serveurs de production
- Redémarrer les serveurs, les applications ou les services
- Générer des fichiers de configuration à partir de modèles
- Exécuter des tests automatisés pour s'assurer que le système fonctionne et est correctement configuré
- Exécuter des procédures de test
- Script et automatisation des migrations de base de données

Dans la mesure du possible, nous réorganiserons pour supprimer les étapes, en particulier celles qui prennent beaucoup de temps à réaliser. Nous voulons également réduire non seulement nos délais de réalisation mais aussi le nombre de transferts autant que possible afin de réduire les erreurs et la perte de connaissances.

Faire en sorte que les développeurs se concentrent sur l'automatisation et l'optimisation du processus de déploiement peut conduire à des améliorations significatives dans le flux de déploiement, comme s'assurer que les petits changements de configuration d'application ne nécessitent plus de nouveaux déploiements ou de nouveaux environnements.

Cependant, cela nécessite que le développement travaille en étroite collaboration avec les opérations pour s'assurer que tous les outils et processus que nous cocréons peuvent être utilisés en aval, au lieu d'aliéner les opérations ou de réinventer la roue.

De nombreux outils qui fournissent une intégration et des tests continus prennent également en charge la possibilité d'étendre le pipeline de déploiement afin que les versions validées puissent être promues en production, généralement après l'exécution des tests d'acceptation de la production (par exemple, le plugin Jenkins Build Pipeline, ThoughtWorks Go.cd et Snap CI, Microsoft Visual Studio Team Services et Pivotal Concourse).

Les exigences pour notre pipeline de déploiement incluent :

- **Déployer de la même manière dans tous les environnements** : en utilisant le même mécanisme de déploiement pour chaque environnement (par exemple, développement, test et production), nos déploiements en production seront probablement beaucoup plus réussis, car nous savons qu'ils ont déjà été réalisés avec succès de nombreuses fois plus tôt dans le pipeline.
- **Tester nos déploiements** : lors du processus de déploiement, nous devons tester notre capacité à nous connecter à tous les systèmes de support (par exemple, bases de données, bus de messages, services externes) et exécuter une seule transaction de test dans le système pour nous assurer que notre système fonctionne comme prévu. Si l'un de ces tests échoue, nous devons échouer le déploiement.
- **Assurer la cohérence des environnements** : dans les étapes précédentes, nous avons créé un processus de construction d'environnement en une seule étape afin que les environnements de développement, de test et de production aient un mécanisme de construction commun. Nous devons continuellement nous assurer que ces environnements restent synchronisés.

Bien sûr, lorsque des problèmes surviennent lors du déploiement, nous tirons le cordon Andon et nous nous regroupons pour résoudre le problème jusqu'à ce qu'il soit résolu, tout comme nous le faisons lorsque notre pipeline de déploiement échoue à l'une des étapes précédentes.

Étude de Cas : Déploiements quotidiens chez CSG International (2013)

CSG International gère l'une des plus grandes opérations d'impression de factures aux États-Unis. Scott Prugh, leur architecte en chef et vice-président du développement, dans le but d'améliorer la prévisibilité et la fiabilité de leurs livraisons de logiciels, a doublé la fréquence de leurs livraisons, passant de deux par an à quatre par an (réduisant ainsi leur intervalle de déploiement de vingt-huit semaines à quatorze semaines).

Bien que les équipes de développement utilisaient l'intégration continue pour déployer leur code dans les environnements de test quotidiennement, les mises en production étaient effectuées par l'équipe des opérations. Prugh a observé : "C'était comme si nous avions une 'équipe d'entraînement' qui s'entraînait quotidiennement (ou même plus fréquemment) dans des environnements de test à faible risque, perfectionnant leurs processus et outils. Mais notre 'équipe de match' en production avait très peu d'occasions de s'entraîner, seulement deux fois par an. Pire encore, ils s'entraînaient dans des environnements de production à haut risque, qui étaient souvent très différents des environnements préproduction avec des contraintes différentes - les environnements de développement manquaient de nombreux actifs de production tels que la sécurité, les pare-feux, les équilibrEURS de charge, et un SAN."

Pour résoudre ce problème, ils ont créé une équipe des opérations partagée (SOT) responsable de la gestion de tous les environnements (développement, test, production) effectuant des déploiements quotidiens dans ces environnements de développement et de test, ainsi que des déploiements et livraisons en production toutes les quatorze semaines.

Parce que la SOT effectuait des déploiements chaque jour, tous les problèmes qu'ils rencontraient et qui n'étaient pas résolus se reproduiraient simplement le lendemain. Cela créait une énorme motivation pour automatiser les étapes manuelles fastidieuses ou sujettes aux erreurs et pour corriger tous les problèmes susceptibles de se reproduire. Parce que les déploiements étaient effectués presque cent fois avant la livraison en production, la plupart des problèmes étaient découverts et corrigés bien avant cela.

Cela a révélé des problèmes qui étaient auparavant uniquement rencontrés par l'équipe des opérations, qui devenaient alors des problèmes pour toute la chaîne de valeur à résoudre. Les déploiements quotidiens permettaient un retour d'information quotidien sur les pratiques qui fonctionnaient et celles qui ne fonctionnaient pas.

Ils se sont également concentrés sur la ressemblance maximale de tous leurs environnements, y compris les droits d'accès restreints à la sécurité et les équilibrEURS de charge. Prugh écrit : "Nous avons rendu les environnements de non-production aussi similaires que possible à la production, et nous avons cherché à émuler les contraintes de production de toutes les manières possibles. Une exposition précoce à des environnements de classe production a modifié les conceptions de l'architecture pour les rendre plus conviviales dans ces

environnements contraints ou différents. Tout le monde devient plus intelligent grâce à cette approche."

Prugh observe également :

"Nous avons connu de nombreux cas où les modifications de schémas de bases de données étaient soit 1) remises à une équipe de DBA pour qu'ils 'trouvent une solution', soit 2) des tests automatisés exécutés sur des ensembles de données irréalistes (c'est-à-dire 'des centaines de Mo contre des centaines de Go'), ce qui conduisait à des échecs en production. Dans notre ancienne manière de travailler, cela devenait un jeu de reproches nocturne entre les équipes essayant de démêler le désordre. Nous avons créé un processus de développement et de déploiement qui supprimait le besoin de remises aux DBA en formant les développeurs, en automatisant les changements de schéma et en les exécutant quotidiennement. Nous avons créé des tests de charge réalistes contre des données client assainies, en exécutant idéalement les migrations chaque jour. En faisant cela, nous exécutons notre service des centaines de fois avec des scénarios réalistes avant de voir le trafic réel de production."

Leurs résultats étaient étonnantes. En effectuant des déploiements quotidiens et en doublant la fréquence des livraisons en production, le nombre d'incidents en production a diminué de 91 %, le MTTR a diminué de 80 %, et le délai de déploiement nécessaire pour que le service fonctionne en production dans un "état entièrement sans intervention" est passé de quatorze jours à un jour.

Prugh a rapporté que les déploiements devenaient si routiniers que l'équipe des opérations jouait à des jeux vidéo à la fin du premier jour. En plus de rendre les déploiements plus fluides pour le développement et les opérations, dans 50 % des cas, le client recevait la valeur en moitié moins de temps, soulignant comment des déploiements plus fréquents peuvent être bénéfiques pour le développement, l'assurance qualité, les opérations et le client.

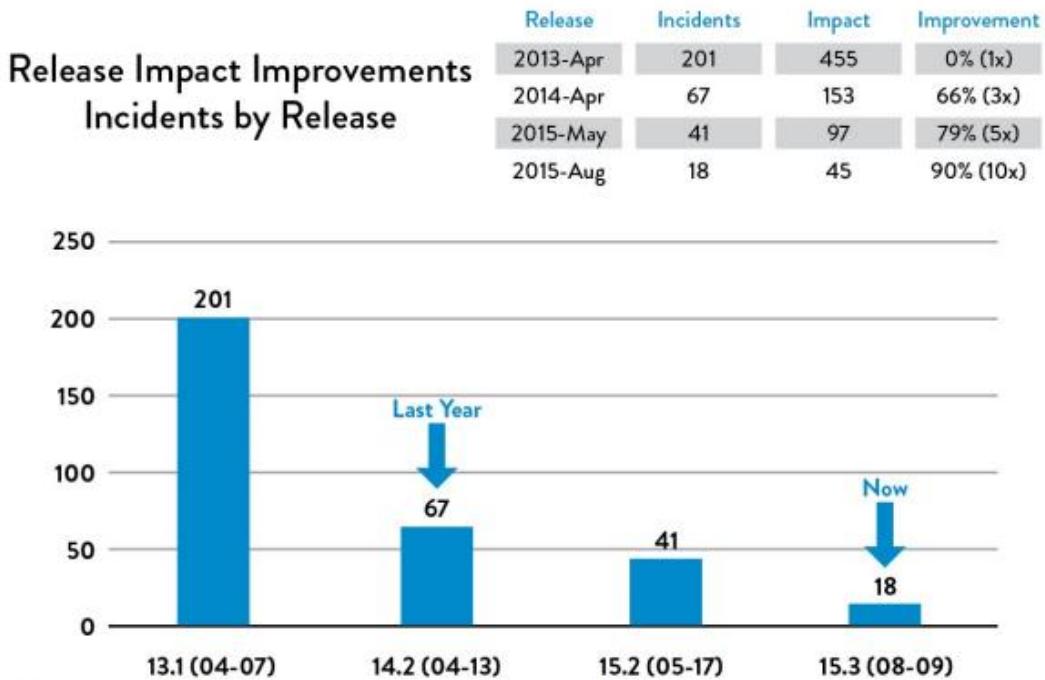


Figure 17: Daily deployments and increasing release frequency resulted in decrease in # of production incidents and MTTR (Source: “DOES15 - Scott Prugh & Erica Morrison - Conway & Taylor Meet the Strangler (v2.0),” YouTube video, 29:39, posted by DevOps Enterprise Summit, November 5, 2015, <https://www.youtube.com/watch?v=tKdIHCL0DUg.>)

Automatisé les déploiements en libre-service

Considérons la citation suivante de Tim Tischler, Directeur de l'Automatisation des Opérations chez Nike, Inc., qui décrit l'expérience commune d'une génération de développeurs : « En tant que développeur, il n'y a jamais eu de moment plus satisfaisant dans ma carrière que lorsque j'écrivais le code, lorsque j'appuyais sur le bouton pour le déployer, lorsque je pouvais voir les métriques de production confirmer qu'il fonctionnait réellement en production, et lorsque je pouvais le réparer moi-même s'il ne fonctionnait pas. »

La capacité des développeurs à déployer eux-mêmes du code en production, à voir rapidement des clients satisfaits lorsque leur fonctionnalité fonctionne, et à corriger rapidement tout problème sans avoir à ouvrir un ticket avec les opérations a diminué au cours de la dernière décennie, en partie à cause d'un besoin de contrôle et de supervision, peut-être motivé par les exigences de sécurité et de conformité.

La pratique courante qui en résulte est que les opérations effectuent les déploiements de code, car la séparation des tâches est une pratique largement acceptée pour réduire le risque de pannes en production et de fraudes. Cependant, pour atteindre les résultats de DevOps, notre

objectif est de déplacer notre dépendance vers d'autres mécanismes de contrôle qui peuvent atténuer ces risques de manière égale ou même plus efficace, comme les tests automatisés, le déploiement automatisé et la revue par les pairs des modifications.

Le rapport de Puppet Labs sur l'état de DevOps de 2013, qui a interrogé plus de quatre mille professionnels de la technologie, a révélé qu'il n'y avait pas de différence statistiquement significative dans les taux de réussite des changements entre les organisations où le développement déployait le code et celles où les opérations déPLOYaient le code.

En d'autres termes, lorsque des objectifs communs englobent le développement et les opérations, et qu'il y a transparence, responsabilité et responsabilité pour les résultats des déploiements, il importe peu de savoir qui effectue le déploiement. En fait, nous pouvons même avoir d'autres rôles, comme les testeurs ou les chefs de projet, capables de déployer dans certains environnements pour qu'ils puissent réaliser leur propre travail rapidement, comme configurer des démonstrations de fonctionnalités spécifiques dans des environnements de test ou de recette utilisateur.

Pour permettre un flux rapide, nous voulons un processus de promotion de code qui peut être effectué par le développement ou les opérations, idéalement sans aucune étape manuelle ou transfert de responsabilité. Cela affecte les étapes suivantes :

- **Build** : Notre pipeline de déploiement doit créer des packages à partir du contrôle de version qui peuvent être déployés dans n'importe quel environnement, y compris la production.
- **Test** : N'importe qui doit pouvoir exécuter tout ou partie de notre suite de tests automatisés sur son poste de travail ou sur nos systèmes de test.
- **Déploiement** : N'importe qui doit pouvoir déployer ces packages dans n'importe quel environnement où il a accès, exécuté en lançant des scripts également vérifiés dans le contrôle de version.

Ce sont les pratiques qui permettent de réaliser des déploiements réussis, peu importe qui effectue le déploiement.

Intégrer le déploiement de code dans le pipeline de déploiement

Une fois le processus de déploiement de code automatisé, nous pouvons l'intégrer au pipeline de déploiement. Par conséquent, notre automatisation de déploiement doit fournir les capacités suivantes :

- S'assurer que les packages créés pendant le processus d'intégration continue sont adaptés pour être déployés en production
- Montrer l'état de préparation des environnements de production en un coup d'œil
- Fournir une méthode en libre-service par simple pression d'un bouton pour que n'importe quelle version adaptée du code empaqueté soit déployée en production

- Enregistrer automatiquement, à des fins d'audit et de conformité, quelles commandes ont été exécutées sur quelles machines et quand, qui l'a autorisé et quel en était le résultat
- Exécuter un test de validation pour s'assurer que le système fonctionne correctement et que les paramètres de configuration, y compris des éléments tels que les chaînes de connexion à la base de données, sont corrects
- Fournir un retour d'information rapide au déployeur afin qu'il puisse rapidement déterminer si son déploiement a réussi (par exemple, le déploiement a-t-il réussi, l'application fonctionne-t-elle comme prévu en production, etc.)

Notre objectif est de garantir que les déploiements soient rapides - nous ne voulons pas avoir à attendre des heures pour déterminer si notre déploiement de code a réussi ou échoué, puis avoir besoin de plusieurs heures pour déployer les correctifs nécessaires. Maintenant que nous disposons de technologies telles que les conteneurs, il est possible de réaliser même les déploiements les plus complexes en quelques secondes ou minutes. Dans le rapport de Puppet Labs sur l'état de DevOps de 2014, les données ont montré que les meilleurs performants avaient des délais de déploiement mesurés en minutes ou en heures, tandis que les moins performants avaient des délais de déploiement mesurés en mois.

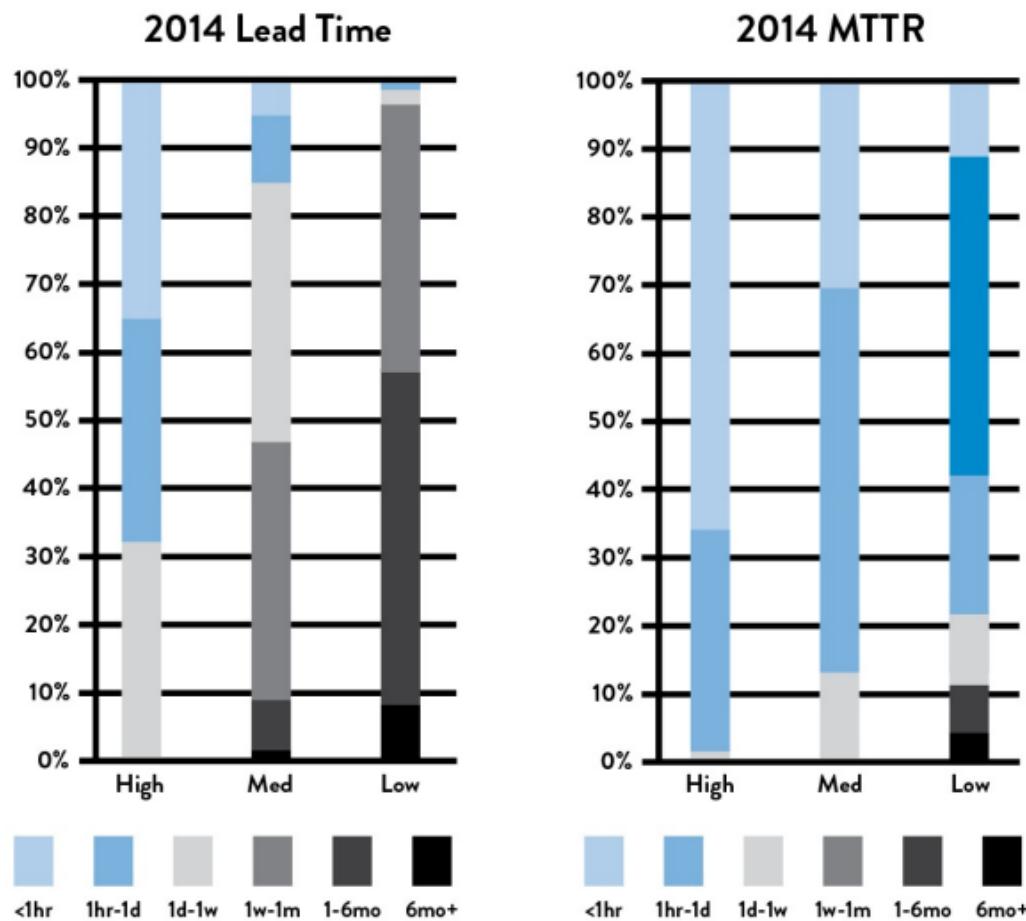


Figure 18: High performers had much faster deployment lead times and much faster time to restore production service after incidents (Source: Puppet Labs, 2014 State of DevOps Report.)

En construisant cette capacité, nous avons maintenant un bouton "déployer le code" qui nous permet de promouvoir en toute sécurité et rapidement les changements de notre code et de nos environnements en production à travers notre pipeline de déploiement.

Étude de cas – Etsy - Déploiement par les développeurs en libre-service, un exemple de déploiement continu (2014)

Contrairement à Facebook où les déploiements sont gérés par des ingénieurs de release, chez Etsy, les déploiements sont effectués par toute personne souhaitant effectuer un déploiement, qu'il s'agisse de Développement, d'Opérations ou de l'Infosec. Le processus de déploiement chez Etsy est devenu si sûr et routinier que les nouveaux ingénieurs effectuent un déploiement en production dès leur premier jour de travail - tout comme les membres du conseil d'administration d'Etsy et même les chiens !

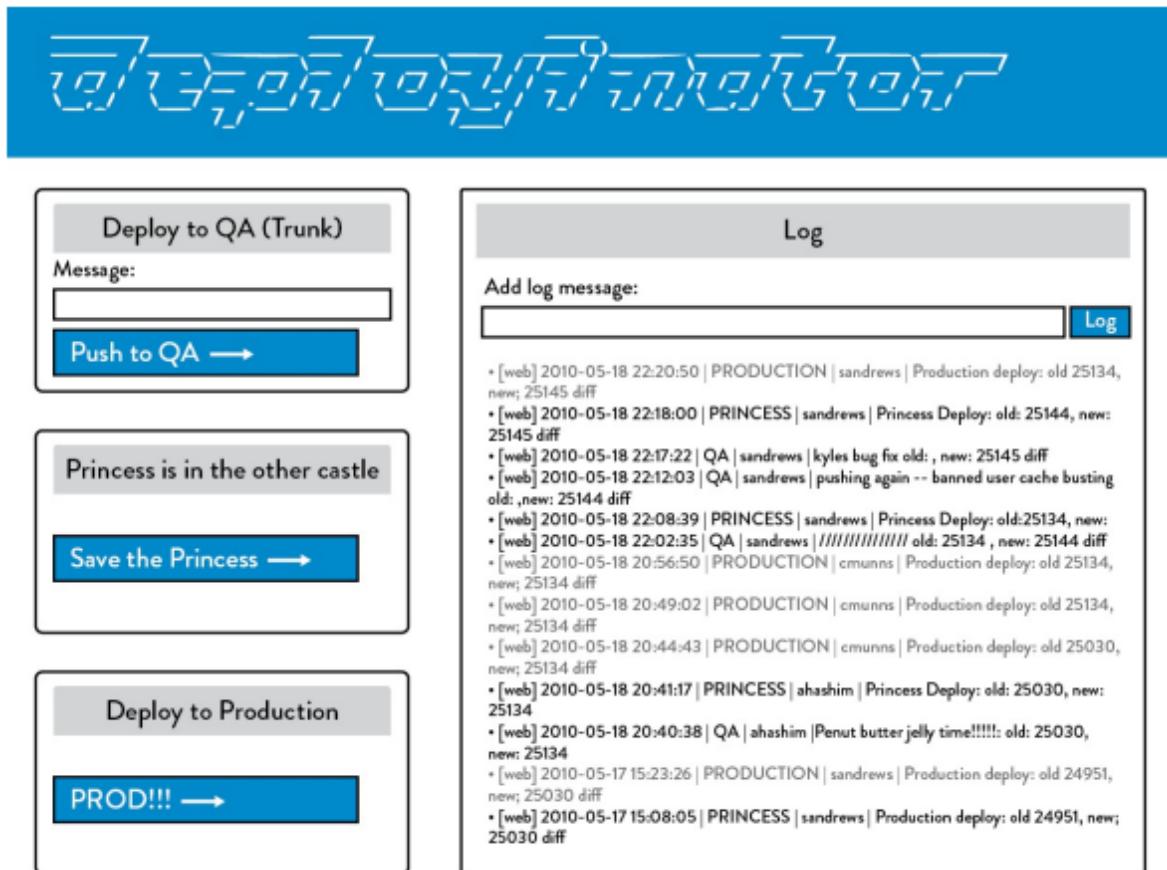
Comme l'a écrit Noah Sussman, architecte de test chez Etsy, « À 8h du matin un jour ouvrable normal, environ 15 personnes et des chiens commencent à faire la queue, tous s'attendant à déployer collectivement jusqu'à 25 ensembles de modifications avant la fin de la journée. »

Les ingénieurs qui souhaitent déployer leur code vont d'abord dans une salle de discussion, où les ingénieurs s'ajoutent à la file d'attente de déploiement, voient l'activité de déploiement en cours, voient qui d'autre est dans la file d'attente, diffusent leurs activités et obtiennent de l'aide d'autres ingénieurs lorsqu'ils en ont besoin. Quand vient le tour d'un ingénieur de déployer, il est notifié dans la salle de discussion.

L'objectif chez Etsy a été de rendre facile et sûr le déploiement en production avec le moins d'étapes possibles et le moins de cérémonial. Probablement avant même que le développeur ne valide son code, il exécutera sur son poste de travail les 4 500 tests unitaires, ce qui prend moins d'une minute. Tous les appels à des systèmes externes, comme les bases de données, ont été simulés.

Après avoir enregistré leurs modifications dans la branche principale du contrôle de version, plus de sept mille tests automatisés de la branche principale sont instantanément exécutés sur leurs serveurs d'intégration continue (CI). Sussman écrit, « Par essais et erreurs, nous avons déterminé qu'environ 11 minutes est le temps maximum pour l'exécution des tests automatisés pendant une poussée. Cela laisse le temps de relancer les tests une fois pendant un déploiement [si quelqu'un casse quelque chose et doit le réparer], sans dépasser trop les 20 minutes. »

Si tous les tests étaient exécutés de manière séquentielle, Sussman indique que « les 7 000 tests de la branche principale prendraient environ une demi-heure à s'exécuter. Nous avons donc divisé ces tests en sous-ensembles et les avons répartis sur les 10 machines de notre cluster Jenkins [CI]... Diviser notre suite de tests et exécuter de nombreux tests en parallèle nous donne le temps d'exécution désiré de 11 minutes. »



Les prochains tests à exécuter sont les tests de fumée, qui sont des tests de niveau système utilisant cURL pour exécuter des cas de test PHPUnit. Après ces tests, les tests fonctionnels sont exécutés, réalisant des tests GUI de bout en bout sur un serveur en direct—ce serveur est soit leur environnement QA, soit un environnement de staging (surnommé "Princess"), qui est en fait un serveur de production retiré de la rotation, garantissant qu'il correspond exactement à l'environnement de production.

Une fois que c'est le tour d'un ingénieur de déployer, Erik Kastner écrit : « vous allez sur Deployinator [un outil développé en interne, voir figure 19] et appuyez sur le bouton pour l'envoyer sur QA. De là, il visite Princess... Ensuite, lorsqu'il est prêt à passer en direct, vous appuyez sur le bouton "Prod" et bientôt votre code est en direct, et tout le monde dans IRC [canal de discussion] sait qui a poussé quel code, avec un lien vers le diff. Pour ceux qui ne sont pas sur IRC, il y a l'email que tout le monde reçoit avec les mêmes informations. »

En 2009, le processus de déploiement chez Etsy était une source de stress et de peur. En 2011, c'était devenu une opération de routine, se produisant vingt-cinq à cinquante fois par jour, aidant les ingénieurs à mettre rapidement leur code en production, apportant de la valeur à leurs clients.

Découpler les déploiements des lancements

Dans le lancement traditionnel d'un projet logiciel, les sorties sont déterminées par notre date de lancement marketing. La veille, nous déployons notre logiciel complet (ou aussi complet que

possible) en production. Le lendemain matin, nous annonçons nos nouvelles capacités au monde, commençons à prendre des commandes, livrons la nouvelle fonctionnalité aux clients, etc.

Cependant, souvent les choses ne se passent pas comme prévu. Nous pouvons rencontrer des charges de production que nous n'avons jamais testées ou pour lesquelles nous n'avons pas conçu, provoquant l'échec spectaculaire de notre service, tant pour nos clients que pour notre organisation. Pire encore, restaurer le service peut nécessiter un processus de retour en arrière douloureux ou une opération de correction en avant tout aussi risquée, où nous effectuons des changements directement en production, ce qui peut être une expérience vraiment misérable pour les travailleurs. Lorsque tout fonctionne enfin, tout le monde pousse un soupir de soulagement, reconnaissant que les déploiements en production et les lancements ne se produisent pas plus souvent.

Bien sûr, nous savons que nous devons déployer plus fréquemment pour atteindre notre objectif de flux fluide et rapide, et non moins fréquemment. Pour permettre cela, nous devons découpler nos déploiements en production de nos lancements de fonctionnalités. En pratique, les termes déploiement et lancement sont souvent utilisés de manière interchangeable. Cependant, ce sont deux actions distinctes qui servent deux objectifs très différents :

- Le déploiement est l'installation d'une version spécifiée de logiciel dans un environnement donné (par exemple, déployer du code dans un environnement de test d'intégration ou déployer du code en production). Spécifiquement, un déploiement peut ou non être associé à un lancement de fonctionnalité aux clients.
- Le lancement est lorsque nous rendons une fonctionnalité (ou un ensemble de fonctionnalités) disponible pour tous nos clients ou un segment de clients (par exemple, nous activons la fonctionnalité pour 5% de notre base de clients). Notre code et nos environnements doivent être conçus de manière que le lancement de fonctionnalités ne nécessite pas de modifier notre code d'application.

En d'autres termes, lorsque nous confondons déploiement et lancement, il devient difficile de créer une responsabilité pour les résultats réussis - découpler ces deux activités nous permet de responsabiliser le Développement et les Opérations pour la réussite des déploiements rapides et fréquents, tout en permettant aux propriétaires de produits d'être responsables des résultats commerciaux réussis du lancement (c'est-à-dire, la construction et le lancement de la fonctionnalité en valaient-ils la peine).

Les pratiques décrites jusqu'ici dans ce livre garantissent que nous effectuons des déploiements en production rapides et fréquents tout au long du développement des fonctionnalités, dans le but de réduire le risque et l'impact des erreurs de déploiement. Le risque restant est le risque de lancement, c'est-à-dire si les fonctionnalités que nous mettons en production atteignent les résultats clients et commerciaux souhaités.

Si nous avons des délais de déploiement extrêmement longs, cela dicte la fréquence à laquelle nous pouvons lancer de nouvelles fonctionnalités sur le marché. Cependant, à mesure que nous devenons capables de déployer à la demande, la rapidité avec laquelle nous exposons de nouvelles fonctionnalités aux clients devient une décision commerciale et marketing, et non une décision technique. Il existe deux grandes catégories de modèles de lancement que nous pouvons utiliser :

- **Modèles de lancement basés sur l'environnement** : c'est là où nous avons deux environnements ou plus dans lesquels nous déployons, mais un seul environnement reçoit le trafic des clients en direct (par exemple, en configurant nos répartiteurs de charge). Le nouveau code est déployé dans un environnement non actif, et le lancement est effectué en déplaçant le trafic vers cet environnement. Ces modèles sont extrêmement puissants, car ils nécessitent généralement peu ou pas de modifications à nos applications. Ces modèles incluent les déploiements blue-green, les lancements canari et les systèmes immunitaires de cluster, qui seront tous discutés prochainement.
- **Modèles de lancement basés sur l'application** : c'est là où nous modifions notre application afin que nous puissions sélectivement lancer et exposer des fonctionnalités spécifiques de l'application par de petits changements de configuration. Par exemple, nous pouvons implémenter des drapeaux de fonctionnalité qui exposent progressivement de nouvelles fonctionnalités en production à l'équipe de développement, à tous les employés internes, à 1% de nos clients ou, lorsque nous sommes confiants que le lancement fonctionnera comme prévu, à l'ensemble de notre base de clients. Comme mentionné précédemment, cela permet une technique appelée lancement sombre, où nous préparons toutes les fonctionnalités à lancer en production et les testons avec du trafic de production avant notre lancement. Par exemple, nous pouvons tester invisiblement notre nouvelle fonctionnalité avec du trafic de production pendant des semaines avant notre lancement afin d'exposer les problèmes pour qu'ils puissent être corrigés avant notre lancement réel.

Modèles de lancement basés sur l'environnement

Découpler les déploiements de nos lancements change radicalement notre manière de travailler. Nous n'avons plus besoin de réaliser des déploiements en pleine nuit ou les weekends pour réduire le risque d'impact négatif sur les clients. Au lieu de cela, nous pouvons effectuer des déploiements pendant les heures ouvrables habituelles, permettant aux Ops d'avoir enfin des heures de travail normales, comme tout le monde.

Cette section se concentre sur les modèles de lancement basés sur l'environnement, qui ne nécessitent aucune modification du code de l'application. Nous le faisons en ayant plusieurs environnements dans lesquels déployer, mais un seul d'entre eux reçoit le trafic des clients en direct. En faisant cela, nous pouvons réduire considérablement le risque associé aux lancements en production et réduire le délai de déploiement.

Le modèle de déploiement Blue-Green

Le plus simple des trois modèles est appelé déploiement blue-green. Dans ce modèle, nous avons deux environnements de production : blue et green. À tout moment, un seul de ces environnements sert le trafic des clients - dans la figure 20, l'environnement green est en direct.

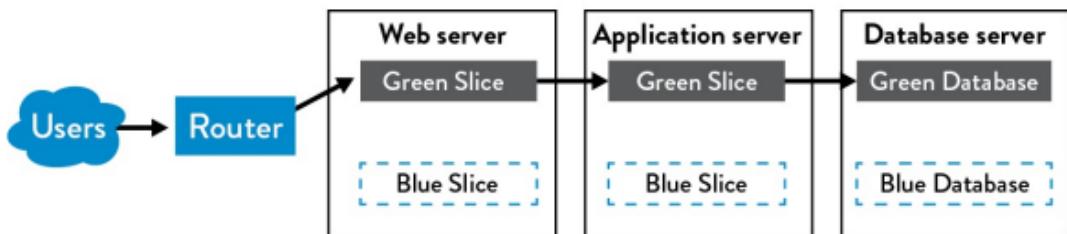


Figure 20: Blue-green deployment patterns (Source: Humble and North, Continuous Delivery, 261.)

Pour publier une nouvelle version de notre service, nous déployons dans l'environnement inactif où nous pouvons effectuer nos tests sans interrompre l'expérience utilisateur. Lorsque nous sommes convaincus que tout fonctionne comme prévu, nous procédons à notre publication en redirigeant le trafic vers l'environnement bleu. Ainsi, le bleu devient actif et le vert devient de mise en scène. Le retour en arrière est effectué en redirigeant le trafic des clients vers l'environnement vert.

Le modèle de déploiement blue-green est simple et peut être facilement adapté aux systèmes existants. Il présente également d'incroyables avantages, comme permettre à l'équipe d'effectuer des déploiements pendant les heures normales de travail et de réaliser des changements simples (par exemple, modifier un paramètre de routeur, changer un lien symbolique) pendant les périodes creuses. Cela seul peut considérablement améliorer les conditions de travail pour l'équipe réalisant le déploiement.

Traitement des changements de base de données

Avoir deux versions de notre application en production pose un problème lorsqu'elles dépendent d'une base de données commune. Lorsque le déploiement nécessite des changements de schéma de base de données ou l'ajout, la modification ou la suppression de tables ou de colonnes, la base de données ne peut pas prendre en charge les deux versions de notre application. Il existe deux approches générales pour résoudre ce problème :

- **Créer deux bases de données** (c'est-à-dire une base de données bleue et une base de données verte) : Chaque version (bleue pour l'ancienne et verte pour la nouvelle) de l'application a sa propre base de données. Lors du déploiement, nous mettons la base de données bleue en mode lecture seule, effectuons une sauvegarde, la restaurons dans la base de données verte, puis redirigeons enfin le trafic vers l'environnement vert. Le problème avec ce modèle est que si nous devons revenir à la version bleue, nous risquons potentiellement de perdre des transactions si nous ne les migrons pas manuellement à partir de la version verte d'abord.
- **Découpler les changements de base de données des changements d'application** : Au lieu de supporter deux bases de données, nous découplons la publication des

changements de base de données de la publication des changements d'application en faisant deux choses : d'abord, nous n'apportons que des modifications additives à notre base de données, nous ne mutons jamais les objets de base de données existants, et deuxièmement, nous ne faisons aucune hypothèse dans notre application sur la version de base de données qui sera en production. Cela est très différent de ce à quoi nous avons été traditionnellement formés à penser sur les bases de données, où nous évitons de dupliquer les données. Le processus de découplage des changements de base de données des changements d'application a été utilisé par IMVU (entre autres) vers 2009, leur permettant de réaliser cinquante déploiements par jour, dont certains nécessitaient des changements de base de données.

Étude de cas - Dixons Retail - Déploiement blue-green pour le système de point de vente (2008)

Dan North et Dave Farley, co-auteurs de Continuous Delivery, travaillaient sur un projet pour Dixons Retail, un grand détaillant britannique impliquant des milliers de systèmes de point de vente (POS) installés dans des centaines de magasins de détail et opérant sous plusieurs marques clients différentes.

Bien que les déploiements blue-green soient principalement associés aux services web en ligne, North et Farley ont utilisé ce modèle pour réduire considérablement les risques et les temps de changement pour les mises à jour des POS.

Traditionnellement, la mise à niveau des systèmes POS est un projet de type big bang, cascade : les clients POS et le serveur centralisé sont mis à niveau en même temps, ce qui nécessite une longue période d'arrêt (souvent un week-end entier), ainsi qu'une bande passante réseau significative pour déployer le nouveau logiciel client dans tous les magasins de détail. Lorsque les choses ne se passent pas tout à fait comme prévu, cela peut perturber considérablement les opérations du magasin.

Pour cette mise à niveau, il n'y avait pas assez de bande passante réseau pour mettre à niveau tous les systèmes POS simultanément, ce qui rendait la stratégie traditionnelle impossible. Pour résoudre ce problème, ils ont utilisé la stratégie blue-green et créé deux versions de production du logiciel serveur centralisé, leur permettant de prendre en charge simultanément les anciennes et nouvelles versions des clients POS.

Avant même la planification de la mise à niveau des POS, ils ont commencé à envoyer les nouveaux installateurs de logiciels clients POS vers les magasins de détail via les liens réseau lents, déployant le nouveau logiciel sur les systèmes POS de manière inactive. Pendant ce temps, l'ancienne version continuait de fonctionner normalement.

Lorsque tous les clients POS avaient tout en place pour la mise à niveau (le client et le serveur mis à niveau avaient été testés ensemble avec succès, et le nouveau logiciel client avait été déployé sur tous les clients), les responsables de magasin étaient autorisés à décider quand libérer la nouvelle version.

Selon leurs besoins commerciaux, certains gestionnaires ont voulu utiliser immédiatement les nouvelles fonctionnalités et les ont libérées tout de suite, tandis que d'autres ont préféré attendre. Dans les deux cas, qu'il s'agisse de libérer immédiatement des fonctionnalités ou d'attendre, c'était beaucoup mieux pour les gestionnaires que d'avoir le département informatique centralisé choisir pour eux le moment de la publication.

Le résultat a été une publication beaucoup plus fluide et rapide, une satisfaction plus élevée des gestionnaires de magasin, et beaucoup moins de perturbation des opérations de magasin. De plus, cette application des déploiements blue-green aux applications PC client lourdes démontre comment les modèles DevOps peuvent être universellement appliqués à différentes technologies, souvent de manière très surprenante mais avec les mêmes résultats fantastiques.

Les modèles de libération canari et de système immunitaire de grappe

Le modèle de déploiement blue-green est facile à mettre en œuvre et peut considérablement augmenter la sécurité des publications logicielles. Il existe des variantes de ce modèle qui peuvent encore améliorer la sécurité et les délais de déploiement en utilisant l'automatisation, mais avec le risque de complexité supplémentaire.

Le modèle de libération canari automatise le processus de libération en promouvant vers des environnements de plus en plus grands et plus critiques à mesure que nous confirmons que le code fonctionne comme prévu.

Le terme de libération canari vient de la tradition des mineurs de charbon qui apportaient des canaris en cage dans les mines pour détecter précocement les niveaux toxiques de monoxyde de carbone. S'il y avait trop de gaz dans la grotte, cela tuait les canaris avant de tuer les mineurs, les alertant pour évacuer.

Dans ce modèle, lorsque nous effectuons une libération, nous surveillons le fonctionnement du logiciel dans chaque environnement. Lorsque quelque chose semble mal se passer, nous revenons en arrière ; sinon, nous passons au déploiement dans l'environnement suivant.

Figure 21 montre les groupes d'environnements que Facebook a créés pour prendre en charge ce modèle de publication :

- Groupe A1 : Serveurs de production qui ne servent que les employés internes

- Groupe A2 : Serveurs de production qui ne servent qu'un petit pourcentage de clients et sont déployés lorsque certains critères d'acceptation ont été remplis (automatiquement ou manuellement)
- Groupe A3 : Le reste des serveurs de production, qui sont déployés après que le logiciel fonctionnant dans le cluster A2 a rempli certains critères d'acceptation

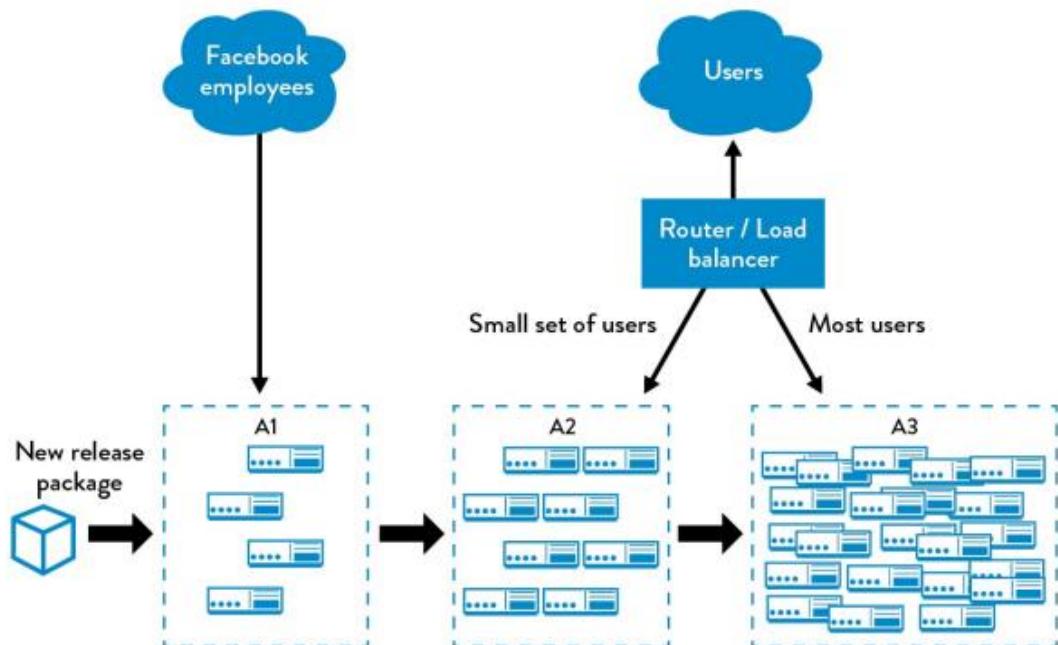


Figure 21: The canary release pattern (Source: Humble and Farley, Continuous Delivery, 263.)

Le système immunitaire de cluster étend le modèle de déploiement canari en reliant notre système de surveillance de production à notre processus de déploiement et en automatisant le retour en arrière du code lorsque les performances visibles par l'utilisateur du système de production s'écartent d'une plage attendue prédéfinie, comme lorsque les taux de conversion des nouveaux utilisateurs chutent en dessous de nos normes historiques de 15 à 20 %.

Il existe deux avantages significatifs à ce type de dispositif de sécurité. Premièrement, nous nous protégeons contre les défauts difficiles à détecter à travers des tests automatisés, comme un changement de page web qui rend invisible un élément critique de la page (par exemple, un changement CSS). Deuxièmement, nous réduisons le temps nécessaire pour détecter et répondre aux performances dégradées causées par notre changement.

Modèles basés sur l'application pour permettre des déploiements plus sûrs

Dans la section précédente, nous avons créé des modèles basés sur l'environnement qui nous ont permis de découpler nos déploiements de nos mises en production en utilisant plusieurs environnements et en basculant entre ceux qui sont en ligne, ce qui peut être entièrement mis en œuvre au niveau de l'infrastructure.

Dans cette section, nous décrivons des modèles de déploiement basés sur l'application que nous pouvons implémenter dans notre code, permettant une flexibilité encore accrue dans la manière dont nous lançons en toute sécurité de nouvelles fonctionnalités à nos clients, souvent sur une base par fonctionnalité. Étant donné que les modèles de déploiement basés sur l'application sont implantés dans l'application, ils nécessitent la participation du Développement.

Implémentation des bascules de fonctionnalités

La principale façon dont nous activons les modèles de déploiement basés sur l'application est en implantant des bascules de fonctionnalités, qui nous fournissent le mécanisme pour activer et désactiver sélectivement des fonctionnalités sans nécessiter de déploiement de code en production. Les bascules de fonctionnalités peuvent également contrôler les fonctionnalités visibles et disponibles pour des segments d'utilisateurs spécifiques (par exemple, employés internes, segments de clients).

Les bascules de fonctionnalités sont généralement mises en œuvre en enveloppant la logique d'application ou les éléments d'interface utilisateur avec une instruction conditionnelle, où la fonctionnalité est activée ou désactivée en fonction d'un paramètre de configuration stocké quelque part. Cela peut être aussi simple qu'un fichier de configuration d'application (par exemple, des fichiers de configuration en JSON, XML), ou cela peut passer par un service de répertoire ou même un service web spécialement conçu pour gérer les bascules de fonctionnalités.

Les bascules de fonctionnalités nous permettent également de faire ce qui suit :

- **Revenir facilement en arrière** : Les fonctionnalités qui posent un problème ou qui perturbent la production peuvent être rapidement et en toute sécurité désactivées en changeant simplement le paramètre de bascule de fonctionnalités. C'est particulièrement précieux lorsque les déploiements sont peu fréquents - désactiver les fonctionnalités d'un intervenant particulier est généralement beaucoup plus facile que de revenir en arrière sur tout un déploiement.
- **Degrader la performance de manière élégante** : Lorsque notre service rencontre des charges extrêmement élevées qui nécessiteraient normalement d'augmenter la capacité ou, pire, risquer de faire échouer notre service en production, nous pouvons utiliser les bascules de fonctionnalités pour réduire la qualité de service. En d'autres termes, nous pouvons augmenter le nombre d'utilisateurs que nous servons en réduisant le niveau de fonctionnalité offert (par exemple, réduire le nombre de clients pouvant accéder à certaines fonctionnalités, désactiver des fonctionnalités intensives en CPU telles que les recommandations, etc.).

- **Augmenter notre résilience grâce à une architecture orientée service** : Si nous avons une fonctionnalité qui dépend d'un autre service qui n'est pas encore complet, nous pouvons toujours déployer notre fonctionnalité en production mais la masquer derrière une bascule de fonctionnalités. Lorsque ce service devient enfin disponible, nous pouvons activer la bascule de fonctionnalités. De même, lorsque le service sur lequel nous comptons échoue, nous pouvons désactiver la fonctionnalité pour éviter les appels au service en aval tout en maintenant le reste de l'application en cours d'exécution.

Pour nous assurer que nous trouvons des erreurs dans les fonctionnalités enveloppées dans des bascules de fonctionnalités, nos tests d'acceptation automatisés doivent s'exécuter avec toutes les bascules de fonctionnalités activées. (Nous devrions également tester que notre fonctionnalité de bascule de fonctionnalités fonctionne correctement !)

Les bascules de fonctionnalités permettent le découplage des déploiements de code et des mises en production de fonctionnalités ; plus tard dans le livre, nous utilisons les bascules de fonctionnalités pour permettre le développement piloté par hypothèse et les tests A/B, ce qui renforce notre capacité à atteindre les résultats commerciaux souhaités.

Réalisation de lancements sombres

Les bascules de fonctionnalités nous permettent de déployer des fonctionnalités en production sans les rendre accessibles aux utilisateurs, en activant une technique connue sous le nom de lancement sombre. C'est là que nous déployons toute la fonctionnalité en production, puis testons cette fonctionnalité tout en la rendant invisible pour les clients. Pour les changements importants ou risqués, nous faisons souvent cela pendant des semaines avant le lancement en production, ce qui nous permet de tester en toute sécurité avec des charges similaires à celles de la production anticipée.

Par exemple, supposons que nous effectuons un lancement sombre d'une nouvelle fonctionnalité qui présente un risque important de lancement, comme de nouvelles fonctionnalités de recherche, des processus de création de compte ou de nouvelles requêtes de base de données. Une fois que tout le code est en production, en maintenant la nouvelle fonctionnalité désactivée, nous pouvons modifier le code de session utilisateur pour effectuer des appels aux nouvelles fonctions - au lieu d'afficher les résultats à l'utilisateur, nous enregistrons simplement ou ignorons les résultats.

Par exemple, nous pouvons faire en sorte que 1 % de nos utilisateurs en ligne effectuent des appels invisibles à une nouvelle fonctionnalité planifiée pour voir comment notre nouvelle fonctionnalité se comporte sous charge. Après avoir identifié et corrigé les problèmes éventuels, nous augmentons progressivement la charge simulée en augmentant la fréquence et le nombre d'utilisateurs utilisant la nouvelle fonctionnalité. De cette manière, nous sommes en

mesure de simuler en toute sécurité des charges similaires à celles de la production, ce qui nous donne la confiance que notre service fonctionnera comme il se doit.

De plus, lors du lancement d'une fonctionnalité, nous pouvons la déployer progressivement auprès de petits segments de clients, en interrompant le déploiement si des problèmes sont détectés. De cette manière, nous minimisons le nombre de clients à qui une fonctionnalité est offerte pour ensuite la leur retirer parce que nous avons trouvé un défaut ou que nous ne parvenons pas à maintenir les performances requises.

En 2009, lorsque John Allspaw était vice-président des opérations chez Flickr, il a écrit à l'équipe de direction exécutive de Yahoo! au sujet de leur processus de lancement sombre, déclarant que cela "augmente la confiance de tous presque à l'apathie, en ce qui concerne la peur des problèmes liés à la charge. Je n'ai aucune idée du nombre de déploiements de code qui ont été réalisés en production chaque jour au cours des 5 dernières années... parce que, pour la plupart, cela m'importe peu, étant donné que ces changements apportés en production ont une si faible probabilité de causer des problèmes. Lorsqu'ils ont causé des problèmes, tous les membres de l'équipe Flickr peuvent trouver sur une page Web quand le changement a été effectué, qui a fait le changement, et exactement (ligne par ligne) ce que le changement était."

Plus tard, lorsque nous aurons mis en place une télémétrie de production adéquate dans notre application et nos environnements, nous pourrons également activer des cycles de rétroaction plus rapides pour valider immédiatement nos hypothèses commerciales et nos résultats après le déploiement de la fonctionnalité en production.

En faisant cela, nous n'attendons plus une grande mise en production pour tester si les clients veulent utiliser les fonctionnalités que nous développons. Au lieu de cela, au moment où nous annonçons et déployons notre grande fonctionnalité, nous avons déjà testé nos hypothèses commerciales et mené de nombreuses expériences pour affiner continuellement notre produit avec de vrais clients, ce qui nous aide à valider que les fonctionnalités atteindront les résultats souhaités pour les clients.

Étude de cas- Lancement en mode sombre de Facebook Chat (2008)

Pendant près d'une décennie, Facebook a été l'un des sites Internet les plus visités, mesuré par le nombre de pages vues et d'utilisateurs uniques. En 2008, il comptait plus de soixante-dix millions d'utilisateurs actifs par jour, ce qui a posé un défi à l'équipe développant la nouvelle fonctionnalité Facebook Chat.

Eugene Letuchy, ingénieur dans l'équipe Chat, a écrit sur le défi majeur en ingénierie logicielle que représentait le nombre d'utilisateurs simultanés : "L'opération la plus intensive en

ressources dans un système de chat n'est pas l'envoi de messages. Il s'agit plutôt de maintenir chaque utilisateur en ligne informé des états en ligne, inactif ou hors ligne de leurs amis, afin que les conversations puissent commencer."

L'implémentation de cette fonctionnalité intensive en calcul fut l'une des plus grandes entreprises techniques jamais entreprises chez Facebook et a pris près d'un an pour être achevée.

Une partie de la complexité du projet venait de la variété étendue de technologies nécessaires pour atteindre les performances souhaitées, comprenant C++, JavaScript, PHP, ainsi que leur première utilisation d'Erlang dans leur infrastructure backend.

Tout au long de cette année, l'équipe Chat a vérifié leur code dans le contrôle de version, où il serait déployé en production au moins une fois par jour. Au début, la fonctionnalité Chat n'était visible que pour l'équipe Chat. Plus tard, elle a été rendue visible pour tous les employés internes, mais complètement cachée aux utilisateurs externes de Facebook grâce à Gatekeeper, le service de basculement de fonctionnalités de Facebook.

Dans le cadre de leur processus de lancement en mode sombre, chaque session utilisateur Facebook, exécutant JavaScript dans le navigateur de l'utilisateur, avait un harnais de test chargé dedans : les éléments de l'interface utilisateur du chat étaient cachés, mais le client du navigateur envoyait des messages de chat de test invisibles au service de chat en production, leur permettant de simuler des charges similaires à la production tout au long du projet, leur permettant de trouver et de résoudre les problèmes de performance bien avant le déploiement auprès des clients.

En faisant cela, chaque utilisateur Facebook faisait partie d'un programme massif de tests de charge, ce qui a permis à l'équipe de gagner en confiance quant à la capacité de leurs systèmes à gérer des charges similaires à celles de la production. Le déploiement et le lancement du Chat ont nécessité seulement deux étapes : la modification de la configuration de Gatekeeper pour rendre la fonctionnalité Chat visible à une partie des utilisateurs externes, et le chargement par les utilisateurs Facebook d'un nouveau code JavaScript qui rendait l'interface utilisateur du Chat visible et désactivait le harnais de test invisible. Si quelque chose se passait mal, les deux étapes pouvaient être inversées.

Lorsque le jour du lancement du Facebook Chat est arrivé, il a été étonnamment réussi et sans incident, semblant scaler sans effort de zéro à soixante-dix millions d'utilisateurs du jour au lendemain. Pendant le lancement, ils ont progressivement activé la fonctionnalité de chat pour des segments de plus en plus grands de la population cliente – d'abord pour tous les employés internes de Facebook, puis pour 1 % de la population cliente, puis pour 5 %, et ainsi de suite.

Comme l'a écrit Letuchy : "Le secret pour passer de zéro à soixante-dix millions d'utilisateurs du jour au lendemain est d'éviter de le faire tout d'un coup."

Enquête sur la livraison continue et le déploiement continu en pratique

Dans La Livraison Continue, Jez Humble et David Farley définissent le terme livraison continue. Le terme déploiement continu a été mentionné pour la première fois par Tim Fitz dans son billet de blog "Déploiement Continu chez IMVU : Faire l'impossible cinquante fois par jour". Cependant, en 2015, lors de la construction de The DevOps Handbook, Jez Humble a commenté : "Au cours des cinq dernières années, il y a eu une confusion autour des termes livraison continue par rapport au déploiement continu - et, en effet, ma propre réflexion et mes définitions ont changé depuis que nous avons écrit le livre. Chaque organisation devrait créer ses variantes, basées sur ce dont elle a besoin. La chose principale à laquelle nous devrions nous intéresser n'est pas la forme, mais les résultats : les déploiements devraient être des événements à faible risque et pouvant être effectués sur demande d'un simple clic."

Ses définitions mises à jour de la livraison continue et du déploiement continu sont les suivantes :

- Lorsque tous les développeurs travaillent en petites séries sur le tronc commun, ou lorsque tout le monde travaille sur des branches de fonctionnalités de courte durée qui sont régulièrement fusionnées avec le tronc, et lorsque le tronc est toujours maintenu dans un état livrable, et lorsque nous pouvons déployer sur demande d'un simple clic pendant les heures de bureau normales, nous faisons de la livraison continue. Les développeurs obtiennent un retour rapide lorsqu'ils introduisent des erreurs de régression, y compris des défauts, des problèmes de performance, des problèmes de sécurité, des problèmes d'utilisabilité, etc. Lorsque ces problèmes sont identifiés, ils sont corrigés immédiatement afin que le tronc soit toujours déployable.
- En plus de ce qui précède, lorsque nous déployons régulièrement de bons builds en production par un service en libre-service (déployé par Dev ou par Ops) – ce qui signifie généralement que nous déployons en production au moins une fois par jour par développeur, voire que nous déployons automatiquement chaque modification qu'un développeur valide – c'est à ce moment-là que nous engageons le déploiement continu.

Défini de cette manière, la livraison continue est la condition préalable au déploiement continu – tout comme l'intégration continue est une condition préalable à la livraison continue. Le déploiement continu est probablement applicable dans le contexte de services web livrés en ligne. Cependant, la livraison continue est applicable dans presque tous les contextes où nous désirons des déploiements et des mises en production de haute qualité, à délais rapides et ayant des résultats prévisibles et à faible risque, y compris pour les systèmes embarqués, les produits COTS et les applications mobiles.

Chez Amazon et Google, la plupart des équipes pratiquent la livraison continue, bien que certaines effectuent le déploiement continu – ainsi, il existe une variation considérable entre les équipes quant à la fréquence à laquelle elles déploient du code et à la manière dont elles effectuent les déploiements. Les équipes sont habilitées à choisir comment déployer en fonction des risques qu'elles gèrent. Par exemple, l'équipe Google App Engine déploie souvent une fois par jour, tandis que la propriété Google Search déploie plusieurs fois par semaine.

De même, la plupart des études de cas présentées dans ce livre relèvent également de la livraison continue, comme le logiciel embarqué sur les imprimantes HP LaserJet, les opérations d'impression de factures CSG sur vingt plates-formes technologiques différentes, y compris une application COBOL sur mainframe, Facebook et Etsy. Ces mêmes modèles peuvent être utilisés pour les logiciels fonctionnant sur des téléphones mobiles, les stations de contrôle au sol qui contrôlent les satellites, et ainsi de suite.

Conclusion

Comme l'ont montré les exemples de Facebook, Etsy et CSG, les mises en production et les déploiements ne doivent pas nécessairement être des événements à haut risque et dramatiques nécessitant des dizaines ou des centaines d'ingénieurs travaillant jour et nuit pour être achevés. Au contraire, ils peuvent être rendus entièrement routiniers et intégrés au travail quotidien de chacun.

En faisant cela, nous pouvons réduire nos délais de mise en production de plusieurs mois à quelques minutes, permettant à nos organisations de livrer rapidement de la valeur à nos clients sans causer de chaos ni de perturbations. De plus, en faisant collaborer Dev et Ops, nous pouvons enfin rendre le travail opérationnel plus humain.

Architecturer pour des mises en production à faible risque

Presque tous les exemples bien connus de DevOps ont connu des expériences quasi fatales en raison de problèmes architecturaux, comme dans les histoires présentées à propos de LinkedIn, Google, eBay, Amazon et Etsy. Dans chaque cas, ils ont réussi à migrer vers une architecture plus adaptée qui répondait à leurs problèmes actuels et aux besoins organisationnels.

C'est le principe de l'architecture évolutive - Jez Humble observe que l'architecture de « tout produit ou organisation à succès évoluera nécessairement au cours de son cycle de vie ». Avant son mandat chez Google, Randy Shoup a été ingénieur en chef et architecte distingué chez eBay de 2004 à 2011. Il observe que « eBay et Google en sont chacun à leur cinquième réécriture complète de leur architecture de haut en bas ».

Il réfléchit : « Avec le recul et une vision claire, certaines [choix technologiques et architecturaux] semblent visionnaires et d'autres paraissent myopes. Chaque décision a probablement le mieux servi les objectifs organisationnels de l'époque. Si nous avions essayé de mettre en œuvre l'équivalent de microservices de 1995 dès le départ, nous aurions probablement échoué, nous effondrant sous notre propre poids et probablement emportant toute l'entreprise avec nous. »

Le défi est de continuer à migrer de l'architecture que nous avons à l'architecture dont nous avons besoin. Dans le cas d'eBay, lorsqu'ils avaient besoin de réarchitecturer, ils commençaient par un petit projet pilote pour prouver qu'ils comprenaient suffisamment bien le problème pour entreprendre cet effort. Par exemple, lorsque l'équipe de Shoup prévoyait de déplacer certaines parties du site vers une pile Java complète en 2006, ils cherchaient la zone qui leur apporterait le meilleur rendement en triant les pages du site par revenu généré. Ils choisissaient les zones à plus haut revenu, s'arrêtant lorsque le retour sur investissement commercial ne justifiait plus l'effort.

Ce que l'équipe de Shoup a fait chez eBay est un exemple classique de conception évolutive, utilisant une technique appelée le modèle de l'application strangler - au lieu de « tout arracher et remplacer » les anciens services par des architectures qui ne soutiennent plus nos objectifs organisationnels, nous plaçons la fonctionnalité existante derrière une API et évitons d'y apporter d'autres modifications. Toute nouvelle fonctionnalité est alors mise en œuvre dans les nouveaux services qui utilisent la nouvelle architecture souhaitée, en appelant l'ancien système si nécessaire.

Le modèle de l'application strangler est particulièrement utile pour aider à migrer des portions d'une application monolithique ou de services étroitement couplés vers une architecture plus faiblement couplée. Trop souvent, nous nous retrouvons à travailler dans une architecture devenue trop étroitement couplée et trop interconnectée, souvent créée il y a des années (ou des décennies).

Les conséquences des architectures trop étroites sont faciles à repérer : chaque fois que nous tentons d'intégrer du code dans le tronc ou de mettre du code en production, nous risquons de créer des échecs globaux (par exemple, nous cassons les tests et la fonctionnalité des autres, ou tout le site tombe en panne). Pour éviter cela, chaque petit changement nécessite d'énormes quantités de communication et de coordination sur des jours ou des semaines, ainsi que des approbations de tout groupe potentiellement affecté. Les déploiements deviennent également problématiques - le nombre de modifications regroupées pour chaque déploiement augmente, compliquant davantage l'intégration et l'effort de test, et augmentant la probabilité déjà élevée que quelque chose tourne mal.

Déployer de petits changements peut nécessiter de se coordonner avec des centaines (voire des milliers) d'autres développeurs, chacun d'entre eux pouvant provoquer une défaillance catastrophique, nécessitant potentiellement des semaines pour trouver et résoudre le problème. (Cela entraîne un autre symptôme : « Mes développeurs passent seulement 15 % de leur temps à coder - le reste de leur temps est passé en réunions. »)

Tous ces éléments contribuent à un système de travail extrêmement dangereux, où de petits changements entraînent des conséquences apparemment imprévisibles et catastrophiques. Cela contribue également souvent à la peur d'intégrer et de déployer notre code, et à la spirale descendante auto-renforcée de déploiements de moins en moins fréquents.

D'un point de vue de l'architecture d'entreprise, cette spirale descendante est la conséquence de la deuxième loi de la thermodynamique architecturale, surtout dans les grandes organisations complexes. Charles Betz, auteur de "Architecture and Patterns for IT Service Management, Resource Planning, and Governance: Making Shoes for the Cobbler's Children", observe que « [les responsables des projets informatiques] ne sont pas tenus responsables de leurs contributions à l'entropie globale du système ». En d'autres termes, réduire notre complexité globale et augmenter la productivité de toutes nos équipes de développement est rarement l'objectif d'un projet individuel.

Dans ce chapitre, nous décrirons les étapes que nous pouvons suivre pour inverser la spirale descendante, examiner les principaux archétypes architecturaux, analyser les attributs des architectures qui favorisent la productivité des développeurs, la testabilité, la déployabilité et la sécurité, ainsi qu'évaluer les stratégies qui nous permettent de migrer en toute sécurité de notre architecture actuelle vers une architecture qui favorise mieux la réalisation de nos objectifs organisationnels.

Une architecture qui favorise la productivité, la testabilité et la sécurité

Contrairement à une architecture étroitement couplée qui peut entraver la productivité de tout le monde et la capacité à apporter des changements en toute sécurité, une architecture faiblement couplée avec des interfaces bien définies qui imposent la manière dont les modules se connectent les uns aux autres favorise la productivité et la sécurité. Elle permet à de petites équipes productives de deux pizzas de réaliser de petits changements qui peuvent être déployés en toute sécurité et indépendamment. Et parce que chaque service a également une API bien définie, cela facilite les tests de services et la création de contrats et de SLA entre les équipes.

Google Cloud Datastore

- **Cloud Datastore: NoSQL service**
 - Highly scalable and resilient
 - Strong transactional consistency
 - SQL-like rich query capabilities
- **Megastore: geo-scale structured database**
 - Multi-row transactions
 - Synchronous cross-datacenter replication
- **Bigtable: cluster-level structured storage**
 - (row, column, timestamp) -> cell contents
- **Colossus: next-generation clustered file system**
 - Block distribution and replication
- **Cluster management infrastructure**
 - Task scheduling, machine assignment

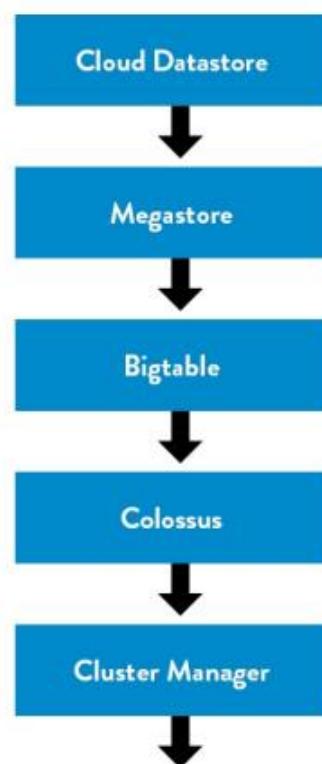


Figure 22: Google cloud datastore (Source: Shoup, “From the Monolith to Microservices.”)

Comme le décrit Randy Shoup, « Ce type d'architecture a très bien servi Google - pour un service comme Gmail, il y a cinq ou six autres couches de services en dessous, chacune très concentrée sur une fonction très spécifique. Chaque service est soutenu par une petite équipe, qui construit et gère sa fonctionnalité, chaque groupe pouvant potentiellement faire des choix technologiques différents. Un autre exemple est le service Google Cloud Datastore, qui est l'un des plus grands services NoSQL au monde - et pourtant il est soutenu par une équipe d'environ huit personnes seulement, en grande partie parce qu'il est basé sur des couches de services fiables construits les uns sur les autres. »

Ce type d'architecture orientée services permet à de petites équipes de travailler sur des unités de développement plus petites et plus simples que chaque équipe peut déployer indépendamment, rapidement et en toute sécurité. Shoup note, « Les organisations avec ce type d'architectures, telles que Google et Amazon, montrent comment cela peut impacter les structures organisationnelles, créant flexibilité et évolutivité. Ce sont toutes deux des organisations avec des dizaines de milliers de développeurs, où de petites équipes peuvent encore être incroyablement productives. »

Archétypes architecturaux : Monolithes vs. Microservices

À un moment donné de leur histoire, la plupart des organisations DevOps ont été entravées par des architectures monolithiques étroitement couplées qui - bien qu'extrêmement réussies pour les aider à atteindre une adéquation produit/marché - les mettaient en risque d'échec organisationnel une fois qu'elles devaient fonctionner à grande échelle (par exemple, l'application monolithique C++ d'eBay en 2001, l'application monolithique OBIDOS d'Amazon en 2001, le front-end monolithique Rails de Twitter en 2009 et l'application monolithique Leo de LinkedIn en 2011). Dans chacun de ces cas, ils ont pu réarchitecturer leurs systèmes et préparer le terrain non seulement pour survivre, mais aussi pour prospérer et gagner sur le marché.

Les architectures monolithiques ne sont pas intrinsèquement mauvaises - en fait, elles sont souvent le meilleur choix pour une organisation au début du cycle de vie d'un produit. Comme le remarque Randy Shoup, « Il n'existe pas une architecture parfaite pour tous les produits et toutes les échelles. Toute architecture répond à un ensemble particulier d'objectifs ou de gammes de besoins et de contraintes, comme le temps de mise sur le marché, la facilité de développement des fonctionnalités, l'évolutivité, etc. La fonctionnalité de tout produit ou service évoluera presque certainement au fil du temps - il ne faut pas être surpris que nos besoins architecturaux changent également. Ce qui fonctionne à l'échelle 1x fonctionne rarement à l'échelle 10x ou 100x. »

Les principaux archétypes architecturaux sont présentés dans le tableau 3, chaque ligne indiquant un besoin évolutif différent pour une organisation, chaque colonne donnant les avantages et les inconvénients de chacun des différents archétypes. Comme le montre le tableau, une architecture monolithique qui soutient une startup (par exemple, prototypage rapide de nouvelles fonctionnalités et pivots ou grands changements de stratégies potentiels) est très différente d'une architecture qui nécessite des centaines d'équipes de développeurs, chacune devant être capable de livrer indépendamment de la valeur au client. En soutenant des architectures évolutives, nous pouvons nous assurer que notre architecture répond toujours aux besoins actuels de l'organisation.

Table 3: Architectural archetypes

	Pros	Cons
Monolithic v1 (All functionality in one application)	<ul style="list-style-type: none"> • Simple at first • Low inter-process latencies • Single codebase, one deployment unit • Resource-efficient at small scales 	<ul style="list-style-type: none"> • Coordination overhead increases as team grows • Poor enforcement of modularity • Poor scaling • All-or-nothing deploy (downtime, failures) • Long build times
Monolithic v2 (Sets of monolithic tiers: “front end presentation,” “application server,” “database layer”)	<ul style="list-style-type: none"> • Simple at first • Join queries are easy • Single schema, deployment • Resource-efficient at small scales 	<ul style="list-style-type: none"> • Tendency for increased coupling over time • Poor scaling and redundancy (all or nothing, vertical only) • Difficult to tune properly • All-or-nothing schema management
Microservice (Modular, independent, graph relationship vs. tiers, isolated persistence)	<ul style="list-style-type: none"> • Each unit is simple • Independent scaling and performance • Independent testing and deployment • Can optimally tune performance (caching, replication, etc.) 	<ul style="list-style-type: none"> • Many cooperating units • Many small repos • Requires more sophisticated tooling and dependency management • Network latencies

Étude de cas - Architecture évolutive chez Amazon (2002)

L'une des transformations architecturales les plus étudiées s'est produite chez Amazon. Dans une interview avec Jim Gray, lauréat du prix ACM Turing et membre technique de Microsoft, le CTO d'Amazon Werner Vogels explique qu'Amazon.com a démarré en 1996 comme une « application monolithique, fonctionnant sur un serveur web, parlant à une base de données en arrière-plan. Cette application, surnommée Obidos, a évolué pour contenir toute la logique métier, toute la logique d'affichage et toutes les fonctionnalités pour lesquelles Amazon est finalement devenu célèbre : similitudes, recommandations, Listmania, critiques, etc. »

Avec le temps, Obidos est devenu trop enchevêtré, avec des relations de partage complexes signifiant que des pièces individuelles ne pouvaient pas être mises à l'échelle selon les besoins. Vogels raconte à Gray que cela signifiait que « de nombreuses choses que vous aimeriez voir dans un bon environnement logiciel ne pouvaient plus être faites ; il y avait de nombreuses pièces de logiciel complexes combinées en un seul système. Il ne pouvait plus évoluer. »

Décrivant le processus de réflexion derrière la nouvelle architecture souhaitée, il dit à Gray, « Nous avons traversé une période d'introspection sérieuse et conclu qu'une architecture orientée services nous donnerait le niveau d'isolation qui nous permettrait de construire de nombreux composants logiciels rapidement et indépendamment. »

Vogels note, « Le grand changement architectural qu'Amazon a traversé au cours des cinq dernières années [de 2001 à 2005] a été de passer d'un monolithe à deux niveaux à une plateforme de services entièrement distribuée et décentralisée, servant de nombreuses applications différentes. Beaucoup d'innovations ont été nécessaires pour que cela se produise, car nous avons été l'un des premiers à adopter cette approche. » Les leçons tirées de l'expérience de Vogel chez Amazon qui sont importantes pour notre compréhension des changements architecturaux incluent les suivantes :

- Leçon 1 : Lorsqu'elle est appliquée rigoureusement, l'orientation stricte vers les services est une excellente technique pour obtenir l'isolation ; vous obtenez un niveau de propriété et de contrôle qui n'a jamais été vu auparavant.
- Leçon 2 : Interdire l'accès direct à la base de données par les clients rend possible l'amélioration de la mise à l'échelle et de la fiabilité de votre état de service sans impliquer vos clients.
- Leçon 3 : Le processus de développement et d'exploitation bénéficie grandement du passage à l'orientation vers les services. Le modèle de services a été un élément clé pour créer des équipes capables d'innover rapidement avec un fort accent sur le client. Chaque service a une équipe associée, et cette équipe est entièrement responsable du service - de la définition de la fonctionnalité à l'architecture, en passant par la construction et l'exploitation.

La mesure dans laquelle l'application de ces leçons améliore la productivité des développeurs et la fiabilité est impressionnante. En 2011, Amazon effectuait environ quinze mille déploiements par jour. En 2015, ils en effectuaient près de 136 000 par jour.

Utiliser le modèle de l'application Strangler pour faire évoluer notre architecture d'entreprise en toute sécurité

Le terme application strangler a été inventé par Martin Fowler en 2004 après avoir été inspiré par la vue de lianes étranglantes massives lors d'un voyage en Australie, écrivant, « Elles se sèment dans les branches supérieures d'un figuier et descendent progressivement jusqu'à ce qu'elles s'enracinent dans le sol. Au fil des ans, elles prennent des formes fantastiques et magnifiques, tout en étranglant et tuant l'arbre qui les hébergeait. »

Si nous avons déterminé que notre architecture actuelle est trop étroitement couplée, nous pouvons commencer à découpler en toute sécurité des parties de la fonctionnalité de notre architecture existante. En faisant cela, nous permettons aux équipes supportant la fonctionnalité découpée de développer, tester et déployer leur code en production de manière autonome et sécurisée, et de réduire l'entropie architecturale.

Comme décrit précédemment, le modèle de l'application strangler implique de placer la fonctionnalité existante derrière une API, où elle reste inchangée, et de mettre en œuvre de nouvelles fonctionnalités en utilisant notre architecture souhaitée, en faisant des appels au système ancien si nécessaire. Lorsque nous mettons en œuvre des applications strangler, nous cherchons à accéder à tous les services via des API versionnées, également appelées services versionnés ou services immuables.

Les API versionnées nous permettent de modifier le service sans impacter les appelants, ce qui permet au système d'être plus faiblement couplé - si nous devons modifier les arguments, nous créons une nouvelle version d'API et migrons les équipes qui dépendent de notre service vers la nouvelle version. Après tout, nous n'atteignons pas nos objectifs de réarchitecture si nous permettons à notre nouvelle application strangler d'être étroitement couplée à d'autres services (par exemple, en se connectant directement à la base de données d'un autre service).

Si les services que nous appelons n'ont pas d'API clairement définies, nous devrions les construire ou au moins masquer la complexité de la communication avec ces systèmes dans une bibliothèque cliente qui dispose d'une API clairement définie.

En découplant de manière répétée la fonctionnalité de notre système étroitement couplé existant, nous déplaçons notre travail vers un écosystème sûr et dynamique où les développeurs peuvent être beaucoup plus productifs, ce qui entraîne la réduction de la

fonctionnalité de l'application legacy. Elle pourrait même disparaître entièrement à mesure que toute la fonctionnalité nécessaire migre vers notre nouvelle architecture.

En créant des applications strangler, nous évitons simplement de reproduire la fonctionnalité existante dans une nouvelle architecture ou technologie - souvent, nos processus métier sont beaucoup plus complexes que nécessaire en raison des particularités des systèmes existants, que nous finirons par reproduire. (En recherchant l'utilisateur, nous pouvons souvent réorganiser le processus afin que nous puissions concevoir un moyen beaucoup plus simple et rationalisé d'atteindre l'objectif commercial.)

Une observation de Martin Fowler souligne ce risque : « Une grande partie de ma carrière a impliqué des réécritures de systèmes critiques. Vous penseriez qu'une telle chose est facile - il suffit de faire en sorte que le nouveau fasse ce que l'ancien faisait. Pourtant, ils sont toujours beaucoup plus complexes qu'ils ne le paraissent, et débordant de risques. La grande date de basculement approche, et la pression monte. Bien que les nouvelles fonctionnalités (il y a toujours de nouvelles fonctionnalités) soient appréciées, les anciennes doivent rester. Même les anciens bugs doivent souvent être ajoutés au système réécrit. »

Comme pour toute transformation, nous cherchons à créer des victoires rapides et à fournir une valeur incrémentale tôt avant de continuer à itérer. Une analyse préalable nous aide à identifier la plus petite pièce de travail possible qui atteindra utilement un résultat commercial en utilisant la nouvelle architecture.

Étude de cas - Modèle Strangler chez Blackboard Learn (2011)

Blackboard Inc. est l'un des pionniers dans la fourniture de technologies pour les établissements d'enseignement, avec un chiffre d'affaires annuel d'environ 650 millions de dollars en 2011. À cette époque, l'équipe de développement de leur produit phare Learn, un logiciel packagé installé et exécuté sur site chez leurs clients, vivait avec les conséquences quotidiennes d'une base de code héritée J2EE datant de 1997.

Comme le remarque David Ashman, leur architecte en chef, "nous avons encore des fragments de code Perl intégrés dans toute notre base de code."

En 2010, Ashman était concentré sur la complexité et les délais croissants associés à l'ancien système, observant que "nos processus de build, d'intégration et de test devenaient de plus en plus complexes et sujets aux erreurs. Et plus le produit grandissait, plus nos délais augmentaient et plus les résultats pour nos clients se détérioraient. Obtenir un retour de notre processus d'intégration nécessitait entre vingt-quatre et trente-six heures."

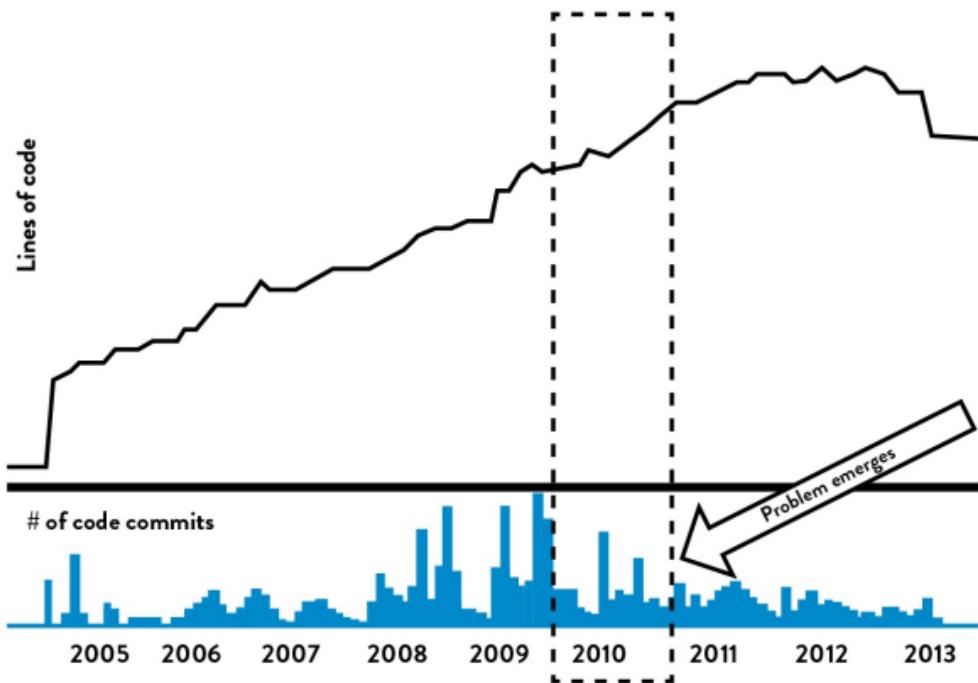


Figure 23: Blackboard Learn code repository: before Building Blocks (Source: "DOES14 - David Ashman - Blackboard Learn - Keep Your Head in the Clouds," YouTube video, 30:43, posted by DevOps Enterprise Summit 2014, October 28, 2014, <https://www.youtube.com/watch?v=SSmixnMpsI4>.)

Comme cela a commencé à avoir un impact sur la productivité des développeurs, c'est devenu visible pour Ashman à travers des graphiques générés à partir de leur référentiel de code source remontant jusqu'en 2005.

Sur la figure 24, le graphique supérieur représente le nombre de lignes de code dans le référentiel de code monolithique de Blackboard Learn ; le graphique inférieur représente le nombre de commits de code. Le problème devenu évident pour Ashman était que le nombre de commits de code commençait à diminuer, montrant objectivement la difficulté croissante d'introduire des changements de code, tandis que le nombre de lignes de code continuait d'augmenter. Ashman a noté : "Pour moi, cela indiquait que nous devions faire quelque chose, sinon les problèmes continueraient de s'aggraver, sans fin en vue."

En conséquence, en 2012, Ashman s'est concentré sur la mise en œuvre d'un projet de réarchitecture du code utilisant le modèle strangler. L'équipe a accompli cela en créant ce qu'ils appelaient en interne les "Building Blocks", qui permettaient aux développeurs de travailler dans des modules séparés, découplés de la base de code monolithique et accessibles via des APIs fixes. Cela leur a permis de travailler avec beaucoup plus d'autonomie, sans avoir à communiquer et coordonner constamment avec d'autres équipes de développement.

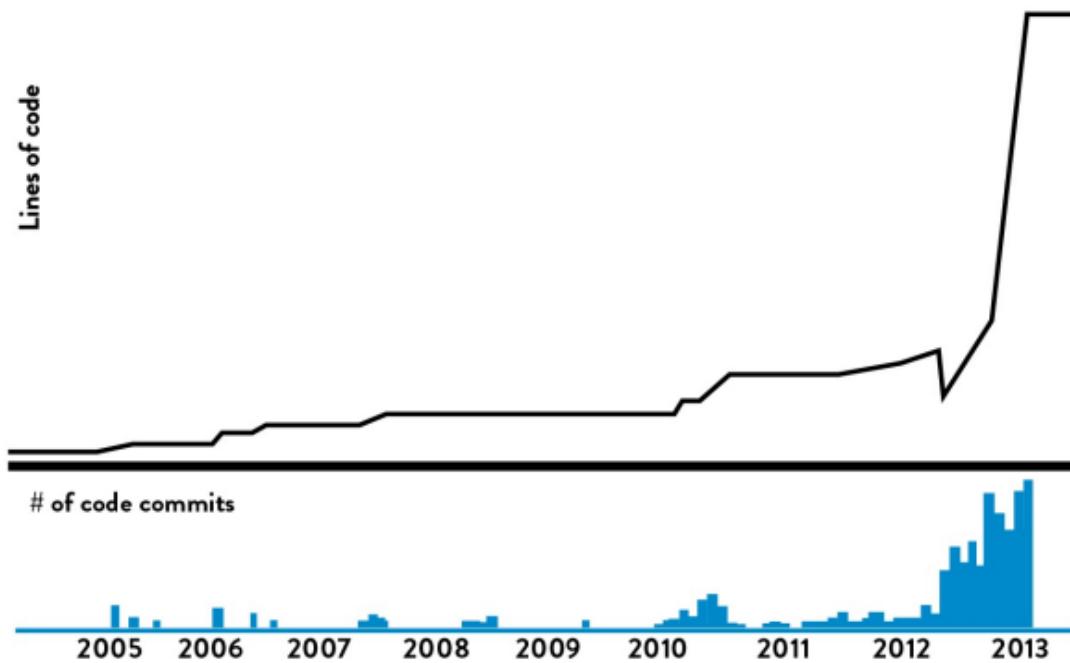


Figure 24: Blackboard Learn code repository: after Building Blocks (Source: “DOES14 - David Ashman - Blackboard Learn - Keep Your Head in the Clouds.” YouTube video, 30:43, posted by DevOps Enterprise Summit 2014, October 28, 2014, <https://www.youtube.com/watch?v=SSmixnMpsI4.>)

Lorsque les Building Blocks ont été mis à la disposition des développeurs, la taille du référentiel de code source du monolithe a commencé à diminuer (mesurée par le nombre de lignes de code). Ashman a expliqué que cela était dû au fait que les développeurs déplaçaient leur code vers le référentiel de code source des modules Building Blocks. "En fait," a rapporté Ashman, "chaque développeur ayant le choix préférerait travailler dans le référentiel de code des Building Blocks, où ils pouvaient travailler avec plus d'autonomie, de liberté et de sécurité."

Le graphique ci-dessus montre la corrélation entre la croissance exponentielle du nombre de lignes de code et la croissance exponentielle du nombre de commits de code pour les référentiels de code des Building Blocks. Le nouveau référentiel de code des Building Blocks a permis aux développeurs d'être plus productifs et ils ont rendu le travail plus sûr car les erreurs entraînaient des échecs locaux mineurs plutôt que des catastrophes majeures affectant le système global.

Ashman a conclu : "Le fait d'avoir les développeurs travaillant dans l'architecture des Building Blocks a entraîné des améliorations impressionnantes en termes de modularité du code, leur permettant de travailler avec plus d'indépendance et de liberté. En combinaison avec les mises à jour de notre processus de build, ils ont également obtenu des retours plus rapides et de meilleure qualité sur leur travail."

Conclusion

Dans une large mesure, l'architecture dans laquelle nos services opèrent dicte la manière dont nous testons et déployons notre code. Cela a été validé dans le rapport 2015 sur l'état de DevOps de Puppet Labs, montrant que l'architecture est l'un des meilleurs prédicteurs de la productivité des ingénieurs qui y travaillent et de la rapidité et de la sécurité avec lesquelles les changements peuvent être réalisés.

Étant donné que nous sommes souvent confrontés à des architectures optimisées pour un ensemble différent d'objectifs organisationnels, ou pour une époque révolue, nous devons être capables de migrer en toute sécurité d'une architecture à une autre. Les études de cas présentées dans ce chapitre, ainsi que l'étude de cas précédemment présentée sur Amazon, décrivent des techniques telles que le modèle strangler qui peuvent nous aider à migrer de manière incrémentielle entre les architectures, nous permettant de nous adapter aux besoins de l'organisation.

Conclusion de la partie III

Dans les chapitres précédents de la Partie III, nous avons mis en œuvre l'architecture et les pratiques techniques qui permettent un flux rapide du travail de Dev à Ops, afin que la valeur puisse être livrée rapidement et en toute sécurité aux clients.

Dans la Partie IV : Le Deuxième Chemin, les Pratiques Techniques du Feedback, nous allons créer l'architecture et les mécanismes pour permettre un flux rapide et réciproque du feedback de droite à gauche, pour trouver et résoudre les problèmes plus rapidement, diffuser le feedback, et assurer de meilleurs résultats à partir de notre travail. Cela permet à notre organisation d'augmenter encore le rythme auquel elle peut s'adapter.

Partie IV - La deuxième voie - Les pratiques du feedback

Introduction

Dans la Partie III, nous avons décrit l'architecture et les pratiques techniques nécessaires pour créer un flux rapide du développement vers les opérations. Maintenant, dans la Partie IV, nous décrivons comment mettre en œuvre les pratiques techniques de la Deuxième Voie, qui sont nécessaires pour créer un retour d'information rapide et continu des opérations vers le développement.

En faisant cela, nous raccourcisons et amplifions les boucles de rétroaction afin de voir les problèmes au fur et à mesure qu'ils se produisent et de diffuser cette information à tout le monde dans la chaîne de valeur. Cela nous permet de trouver et de résoudre rapidement les problèmes plus tôt dans le cycle de vie du développement logiciel, idéalement bien avant qu'ils ne causent une défaillance catastrophique.

De plus, nous créerons un système de travail où les connaissances acquises en aval dans les opérations sont intégrées dans le travail en amont du développement et de la gestion des produits. Cela nous permet de créer rapidement des améliorations et des apprentissages, que ce soit à partir d'un problème de production, d'un problème de déploiement, d'indicateurs précoce de problèmes ou de nos modèles d'utilisation par les clients.

En outre, nous créerons un processus qui permet à chacun de recevoir des retours sur son travail, de rendre l'information visible pour favoriser l'apprentissage et de tester rapidement des hypothèses de produits, nous aidant à déterminer si les fonctionnalités que nous construisons nous aident à atteindre nos objectifs organisationnels.

Nous démontrerons également comment créer une télémétrie à partir de nos processus de construction, de test et de déploiement, ainsi que du comportement des utilisateurs, des problèmes de production et des pannes, des problèmes d'audit et des violations de sécurité. En amplifiant les signaux dans le cadre de notre travail quotidien, nous rendons possible la détection et la résolution des problèmes au fur et à mesure qu'ils se produisent, et nous développons des systèmes de travail sûrs qui nous permettent de faire des changements et des expériences produits en toute confiance, sachant que nous pouvons rapidement détecter et remédier aux échecs. Nous ferons tout cela en explorant les points suivants :

- Créer une télémétrie pour permettre de voir et de résoudre les problèmes
- Utiliser notre télémétrie pour mieux anticiper les problèmes et atteindre les objectifs
- Intégrer la recherche utilisateur et les retours dans le travail des équipes produits
- Permettre un retour d'information afin que Dev et Ops puissent effectuer des déploiements en toute sécurité
- Permettre un retour d'information pour augmenter la qualité de notre travail grâce aux revues par les pairs et à la programmation en binôme

Les modèles de ce chapitre aident à renforcer les objectifs communs de la gestion des produits, du développement, de l'assurance qualité, des opérations et de la sécurité de l'information, et les encouragent à partager la responsabilité d'assurer le bon fonctionnement des services en production et à collaborer à l'amélioration du système dans son ensemble. Dans la mesure du possible, nous voulons lier la cause à l'effet. Plus nous pouvons invalider d'hypothèses, plus nous pouvons découvrir et résoudre rapidement les problèmes, mais aussi plus nous pouvons apprendre et innover.

Tout au long des chapitres suivants, nous mettrons en œuvre des boucles de rétroaction, permettant à chacun de travailler ensemble vers des objectifs communs, de voir les problèmes au fur et à mesure qu'ils se produisent, de permettre une détection et une récupération rapides, et de garantir que les fonctionnalités non seulement fonctionnent comme prévu en production, mais atteignent également les objectifs organisationnels et soutiennent l'apprentissage organisationnel.

Créer de la télémétrie pour permettre de voir et de résoudre les problèmes

Un fait de la vie en opérations est que les choses tournent mal : de petits changements peuvent entraîner de nombreux résultats inattendus, y compris des pannes et des défaillances globales qui impactent tous nos clients. C'est la réalité d'opérer des systèmes complexes ; aucune personne seule ne peut voir l'ensemble du système et comprendre comment toutes les pièces s'emboîtent.

Lorsque des pannes de production et d'autres problèmes se produisent dans notre travail quotidien, nous n'avons souvent pas les informations nécessaires pour résoudre le problème. Par exemple, lors d'une panne, nous ne pouvons pas déterminer si le problème est dû à une défaillance de notre application (par exemple, un défaut dans le code), de notre environnement (par exemple, un problème de réseau, un problème de configuration du serveur), ou quelque chose d'entièrement externe à nous (par exemple, une attaque massive par déni de service).

En opérations, nous pouvons traiter ce problème avec la règle empirique suivante : quand quelque chose tourne mal en production, nous redémarrons simplement le serveur. Si cela ne fonctionne pas, redémarrez le serveur à côté. Si cela ne fonctionne pas, redémarrez tous les serveurs. Si cela ne fonctionne pas, blâmez les développeurs, ils causent toujours des pannes.

En revanche, l'étude du Microsoft Operations Framework (MOF) en 2001 a révélé que les organisations avec les niveaux de service les plus élevés redémarraient leurs serveurs vingt fois moins fréquemment que la moyenne et avaient cinq fois moins de « blue screens of death ». En d'autres termes, ils ont constaté que les organisations les plus performantes étaient beaucoup plus aptes à diagnostiquer et à résoudre les incidents de service, dans ce que Kevin Behr, Gene Kim et George Spafford ont appelé une « culture de la causalité » dans The Visible Ops Handbook. Les meilleurs performeurs utilisaient une approche disciplinée pour résoudre les problèmes, utilisant la télémétrie de production pour comprendre les facteurs contributifs possibles afin de concentrer leur résolution de problèmes, contrairement aux performeurs plus faibles qui redémarraient les serveurs aveuglément.

Pour permettre ce comportement de résolution de problèmes discipliné, nous devons concevoir nos systèmes de manière qu'ils créent continuellement de la télémétrie, largement définie comme « un processus de communication automatisé par lequel des mesures et d'autres données sont collectées à des points distants et sont ensuite transmises à des équipements de réception pour surveillance ». Notre objectif est de créer de la télémétrie au sein de nos applications et environnements, à la fois dans nos environnements de production et de pré-production ainsi que dans notre pipeline de déploiement.

Michael Rembetsy et Patrick McDonnell ont décrit comment la surveillance de la production était une partie essentielle de la transformation DevOps d'Etsy qui a commencé en 2009. C'était parce qu'ils standardisaient et faisaient la transition de leur pile technologique entière vers la pile LAMP (Linux, Apache, MySQL et PHP), abandonnant une myriade de technologies différentes utilisées en production qui devenaient de plus en plus difficiles à supporter.

Lors de la conférence Velocity en 2012, McDonnell a décrit à quel point cela créait du risque, « Nous changions une partie de notre infrastructure la plus critique, ce que, idéalement, les clients ne devraient jamais remarquer. Cependant, ils le remarqueraient certainement si nous commettions une erreur. Nous avions besoin de plus de métriques pour nous donner confiance que nous ne cassions pas réellement les choses pendant que nous faisions ces grands changements, à la fois pour nos équipes d'ingénierie et pour les membres de l'équipe dans les domaines non techniques, comme le marketing. »

McDonnell a expliqué plus loin, « Nous avons commencé à collecter toutes les informations de nos serveurs dans un outil appelé Ganglia, affichant toutes les informations dans Graphite, un outil open source dans lequel nous avons beaucoup investi. Nous avons commencé à agréger les métriques ensemble, de tout, des métriques commerciales aux déploiements. C'est à ce moment-là que nous avons modifié Graphite avec ce que nous avons appelé 'notre technologie de ligne verticale inégalée et inégalable' qui superposait sur chaque graphique de métrique lorsque les déploiements se produisaient. En faisant cela, nous pouvions plus rapidement voir les effets secondaires involontaires des déploiements. Nous avons même commencé à mettre des écrans de télévision tout autour du bureau afin que tout le monde puisse voir comment nos services se comportaient. »

En permettant aux développeurs d'ajouter de la télémétrie à leurs fonctionnalités dans le cadre de leur travail quotidien, ils ont créé suffisamment de télémétrie pour aider à rendre les déploiements sûrs. En 2011, Etsy suivait plus de deux cent mille métriques de production à chaque couche de la pile d'applications (par exemple, fonctionnalités de l'application, santé de l'application, base de données, système d'exploitation, stockage, réseau, sécurité, etc.) avec les trente métriques commerciales les plus importantes affichées en évidence sur leur « tableau de bord de déploiement ». En 2014, ils suivaient plus de huit cent mille métriques, montrant leur objectif implacable d'instrumenter tout et de rendre facile pour les ingénieurs de le faire.

Comme l'a plaisanté Ian Malpass, un ingénieur chez Etsy, « Si l'ingénierie chez Etsy avait une religion, ce serait l'Église des Graphiques. Si ça bouge, nous le suivons. Parfois, nous traçons un graphique de quelque chose qui ne bouge pas encore, juste au cas où cela déciderait de se déplacer... Suivre tout est la clé pour avancer rapidement, mais la seule façon de le faire est de rendre le suivi de n'importe quoi facile... Nous permettons aux ingénieurs de suivre ce dont ils ont besoin, en un clin d'œil, sans nécessiter des changements de configuration chronophages ou des processus compliqués. »

Une des conclusions du rapport State of DevOps 2015 était que les meilleurs performeurs pouvaient résoudre les incidents de production 168 fois plus rapidement que leurs pairs, avec un MTTR médian pour les meilleurs performeurs mesuré en minutes, tandis que le MTTR médian des moins bons performeurs était mesuré en jours. Les deux principales pratiques techniques qui permettaient un MTTR rapide étaient l'utilisation du contrôle de version par les opérations et la présence de télémétrie et de surveillance proactive dans l'environnement de production.

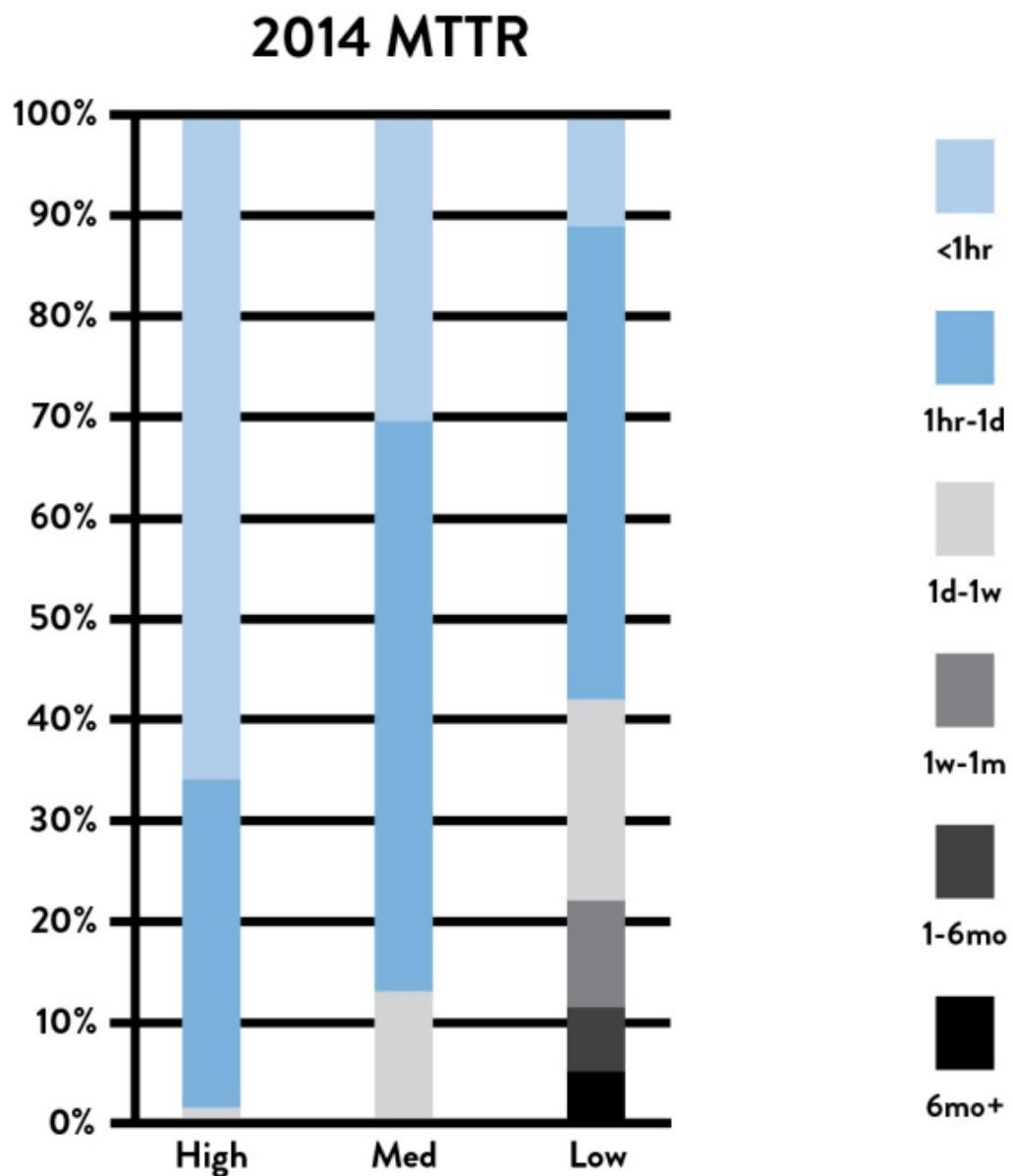


Figure 25: Incident resolution time for high, medium, and low performers
(Source: Puppet Labs, 2014 State of DevOps Report.)

Comme cela a été créé chez Etsy, notre objectif dans ce chapitre est de garantir que nous avons toujours suffisamment de télémétrie pour confirmer que nos services fonctionnent correctement en production. Et lorsque des problèmes surviennent, rendre possible la détermination rapide de ce qui ne va pas et prendre des décisions éclairées sur la meilleure

façon de les résoudre, idéalement bien avant que les clients ne soient impactés. De plus, la télémétrie est ce qui nous permet d'assembler notre meilleure compréhension de la réalité et de détecter quand notre compréhension de la réalité est incorrecte.

Créer notre infrastructure de télémétrie centralisée

La surveillance opérationnelle et la journalisation ne sont en aucun cas nouvelles : plusieurs générations d'ingénieurs en opérations ont utilisé et personnalisé des cadres de surveillance (par exemple, HP OpenView, IBM Tivoli, et BMC Patrol/BladeLogic) pour garantir la santé des systèmes de production. Les données étaient généralement collectées par des agents qui s'exécutaient sur des serveurs ou par une surveillance sans agent (par exemple, des pièges SNMP ou des moniteurs basés sur le sondage). Il y avait souvent une interface utilisateur graphique (GUI) frontale, et les rapports back-end étaient souvent complétés par des outils comme Crystal Reports.

De même, les pratiques de développement d'applications avec une journalisation efficace et de gestion de la télémétrie résultante ne sont pas nouvelles : une variété de bibliothèques de journalisation matures existe pour presque tous les langages de programmation.

Cependant, pendant des décennies, nous avons fini par avoir des silos d'information, où le développement ne crée que des événements de journalisation intéressants pour les développeurs, et les opérations ne surveillent que si les environnements sont en ligne ou hors ligne. En conséquence, lorsque des événements inopportunus se produisent, personne ne peut déterminer pourquoi l'ensemble du système ne fonctionne pas comme prévu ou quel composant spécifique échoue, entravant notre capacité à remettre notre système dans un état fonctionnel.

Pour que nous puissions voir tous les problèmes au fur et à mesure qu'ils se produisent, nous devons concevoir et développer nos applications et environnements de manière qu'ils génèrent suffisamment de télémétrie, nous permettant de comprendre comment notre système se comporte dans son ensemble. Lorsque tous les niveaux de notre pile d'applications disposent de surveillance et de journalisation, nous permettons d'autres capacités importantes, telles que le graphisme et la visualisation de nos métriques, la détection d'anomalies, l'alerte proactive et l'escalade, etc.

Dans *The Art of Monitoring*, James Turnbull décrit une architecture de surveillance moderne, qui a été développée et utilisée par les ingénieurs en opérations dans des entreprises de grande envergure (par exemple, Google, Amazon, Facebook). L'architecture consistait souvent en des outils open source, tels que Nagios et Zenoss, qui étaient personnalisés et déployés à une échelle difficile à réaliser avec des logiciels commerciaux sous licence à l'époque. Cette architecture comporte les composants suivants :

- **Collecte de données au niveau de la logique métier, des applications et des environnements** : À chacun de ces niveaux, nous créons de la télémétrie sous forme d'événements, de journaux et de métriques. Les journaux peuvent être stockés dans des fichiers spécifiques à l'application sur chaque serveur (par exemple, varlog/httpd-error.log), mais nous préférons envoyer tous nos journaux à un service commun qui permet une centralisation, une rotation et une suppression faciles. Cela est fourni par la plupart des systèmes d'exploitation, tels que syslog pour Linux, le journal des événements pour Windows, etc. De plus, nous recueillons des métriques à tous les niveaux de la pile d'applications pour mieux comprendre comment notre système se comporte. Au niveau du système d'exploitation, nous pouvons collecter des métriques telles que l'utilisation du CPU, de la mémoire, du disque ou du réseau au fil du temps en utilisant des outils comme collectd, Ganglia, etc. D'autres outils qui collectent des informations sur les performances incluent AppDynamics, New Relic et Pingdom.
- **Un routeur d'événements responsable du stockage de nos événements et métriques** : Cette capacité permet potentiellement la visualisation, la tendance, l'alerte, la détection d'anomalies, etc. En collectant, stockant et agrégant toute notre télémétrie, nous permettons mieux les analyses ultérieures et les vérifications de santé. C'est également là que nous stockons les configurations liées à nos services (et à leurs applications et environnements de support) et où nous effectuons probablement des alertes et des vérifications de santé basées sur des seuils.

Une fois que nous avons centralisé nos journaux, nous pouvons les transformer en métriques en les comptant dans le routeur d'événements - par exemple, un événement de journal tel que « child pid 14024 exit signal Segmentation fault » peut être compté et résumé en une seule métrique de segfault dans toute notre infrastructure de production.

En transformant les journaux en métriques, nous pouvons désormais effectuer des opérations statistiques sur eux, comme utiliser la détection d'anomalies pour trouver des valeurs aberrantes et des variances encore plus tôt dans le cycle de problème. Par exemple, nous pourrions configurer notre système d'alerte pour nous notifier si nous passons de « dix segfaults la semaine dernière » à « des milliers de segfaults au cours de la dernière heure », nous incitant à enquêter davantage.

En plus de collecter de la télémétrie de nos services et environnements de production, nous devons également collecter de la télémétrie de notre pipeline de déploiement lorsque des événements importants se produisent, comme lorsque nos tests automatisés réussissent ou échouent et lorsque nous effectuons des déploiements dans n'importe quel environnement. Nous devons également collecter de la télémétrie sur le temps qu'il nous faut pour exécuter nos builds et tests. En faisant cela, nous pouvons détecter des conditions qui pourraient indiquer des problèmes, par exemple si le test de performance ou notre build prend deux fois plus de temps que d'habitude, nous permettant de trouver et de corriger les erreurs avant qu'elles n'entrent en production.

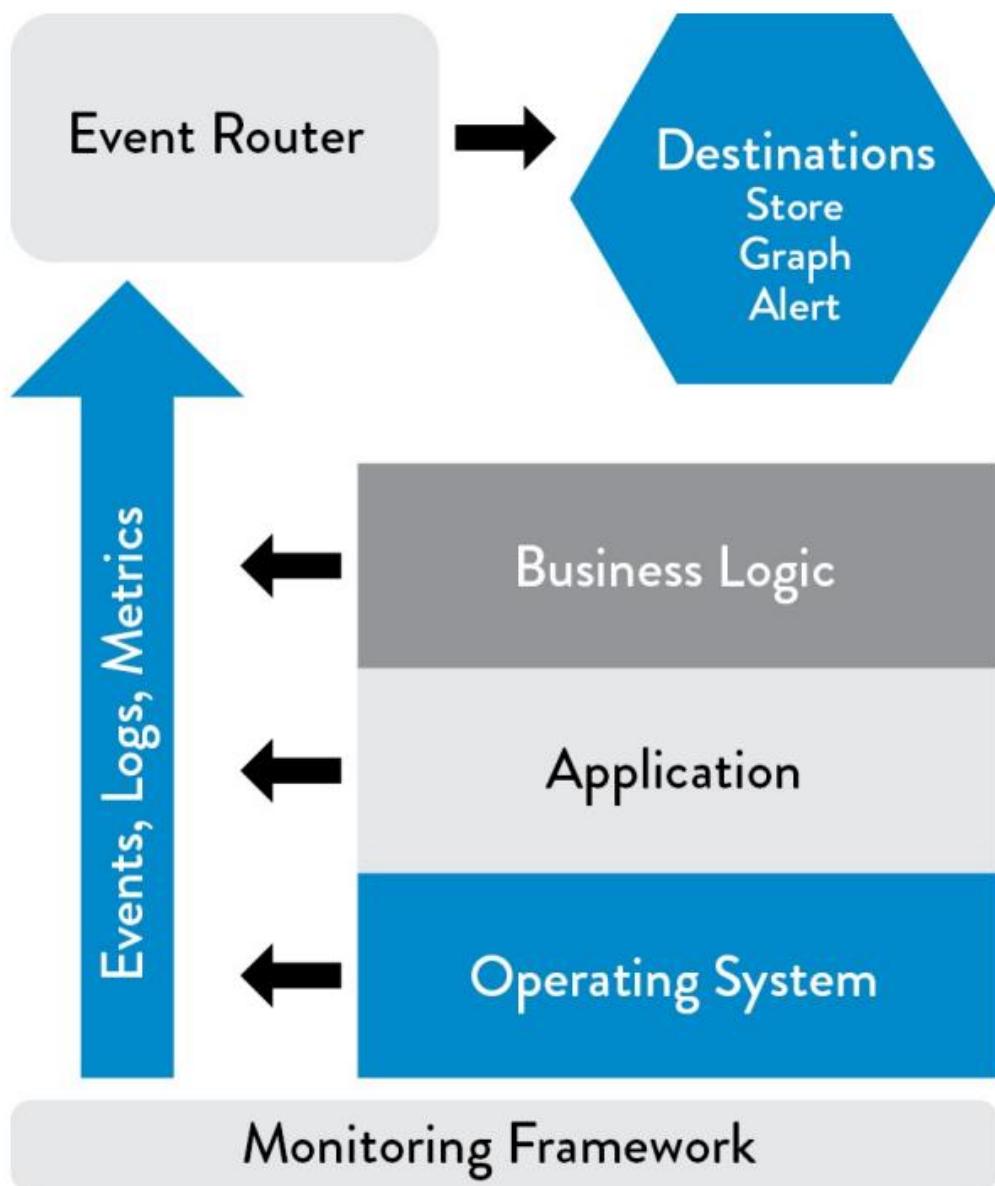


Figure 26: Monitoring framework (Source: Turnbull, *The Art of Monitoring*, Kindle edition, chap. 2.)

En outre, nous devrions nous assurer qu'il soit facile d'entrer et de récupérer des informations de notre infrastructure de télémétrie. De préférence, tout devrait être fait via des API en libre-service, plutôt que de demander aux gens de créer des tickets et d'attendre pour obtenir des rapports.

Idéalement, nous créerons une télémétrie qui nous indique exactement quand quelque chose d'intéressant se produit, ainsi que où et comment. Notre télémétrie devrait également être adaptée à l'analyse manuelle et automatisée et devrait pouvoir être analysée sans avoir l'application qui a produit les journaux à portée de main. Comme l'a souligné Adrian Cockcroft,

"La surveillance est tellement importante que nos systèmes de surveillance doivent être plus disponibles et évolutifs que les systèmes surveillés."

Désormais, le terme télémétrie sera utilisé de manière interchangeable avec les métriques, ce qui inclut tous les journaux d'événements et les métriques créés par nos services à tous les niveaux de notre pile d'applications et générés à partir de tous nos environnements de production et de préproduction, ainsi que de notre pipeline de déploiement.

Créer une télémétrie de journalisation des applications qui aide à la production

Maintenant que nous avons une infrastructure de télémétrie centralisée, nous devons nous assurer que les applications que nous construisons et exploitons créent une télémétrie suffisante. Nous le faisons en demandant aux ingénieurs Dev et Ops de créer une télémétrie de production dans le cadre de leur travail quotidien, tant pour les nouveaux services que pour les services existants.

Scott Prugh, architecte en chef et vice-président du développement chez CSG, a déclaré : "Chaque fois que la NASA lance une fusée, elle dispose de millions de capteurs automatisés qui rapportent l'état de chaque composant de cet actif précieux. Et pourtant, nous ne prenons souvent pas le même soin avec les logiciels – nous avons constaté que créer une télémétrie pour les applications et l'infrastructure était l'un des investissements les plus rentables que nous ayons réalisés. En 2014, nous avons créé plus d'un milliard d'événements de télémétrie par jour, avec plus de cent mille emplacements de code instrumentés."

Dans les applications que nous créons et exploitons, chaque fonctionnalité doit être instrumentée – si cela a été suffisamment important pour qu'un ingénieur le mette en œuvre, il est certainement suffisamment important de générer une télémétrie de production afin que nous puissions confirmer qu'elle fonctionne comme prévu et que les résultats souhaités sont atteints.

Chaque membre de notre chaîne de valeur utilisera la télémétrie de différentes manières. Par exemple, les développeurs peuvent temporairement créer plus de télémétrie dans leur application pour mieux diagnostiquer les problèmes sur leur poste de travail, tandis que les ingénieurs Ops peuvent utiliser la télémétrie pour diagnostiquer un problème de production. De plus, les équipes de sécurité de l'information et les auditeurs peuvent examiner la télémétrie pour confirmer l'efficacité d'un contrôle requis, et un chef de produit peut les utiliser pour suivre les résultats commerciaux, l'utilisation des fonctionnalités ou les taux de conversion.

Pour prendre en charge ces différents modèles d'utilisation, nous avons différents niveaux de journalisation, dont certains peuvent également déclencher des alertes, tels que les suivants :

- Niveau DEBUG : Les informations à ce niveau concernent tout ce qui se passe dans le programme, le plus souvent utilisé lors du débogage. Souvent, les journaux de débogage sont désactivés en production mais temporairement activés lors du dépannage.
- Niveau INFO : Les informations à ce niveau consistent en des actions qui sont déclenchées par l'utilisateur ou spécifiques au système (par exemple, "début de la transaction par carte de crédit").
- Niveau WARN : Les informations à ce niveau nous informent des conditions qui pourraient potentiellement devenir une erreur (par exemple, un appel de base de données prenant plus de temps qu'un certain délai prédéfini). Ces informations déclencheront probablement une alerte et un dépannage, tandis que d'autres messages de journalisation peuvent nous aider à mieux comprendre ce qui a conduit à cette condition.
- Niveau ERROR : Les informations à ce niveau se concentrent sur les conditions d'erreur (par exemple, les échecs d'appels d'API, les conditions d'erreur internes).
- Niveau FATAL : Les informations à ce niveau nous indiquent quand nous devons terminer (par exemple, un démon réseau ne peut pas lier un socket réseau).

Choisir le bon niveau de journalisation est important. Dan North, un ancien consultant de ThoughtWorks qui a été impliqué dans plusieurs projets où les concepts de livraison continue ont pris forme, observe : "Lorsqu'il s'agit de décider si un message doit être ERROR ou WARN, imaginez être réveillé à 4 heures du matin. Un niveau bas de toner d'imprimante n'est pas une ERREUR."

Pour nous assurer que nous avons des informations pertinentes pour le fonctionnement fiable et sécurisé de notre service, nous devons nous assurer que tous les événements d'application potentiellement significatifs génèrent des entrées de journalisation, y compris ceux fournis dans cette liste assemblée par Anton A. Chuvakin, vice-président de recherche au groupe GTP Security and Risk Management de Gartner :

- Décisions d'authentification/autorisation (y compris la déconnexion)
- Accès au système et aux données
- Modifications du système et des applications (en particulier les modifications privilégiées)
- Modifications des données, telles que l'ajout, la modification ou la suppression de données
- Entrée invalide (injection potentiellement malveillante, menaces, etc.)
- Ressources (RAM, disque, CPU, bande passante ou toute autre ressource ayant des limites dures ou molles)
- Santé et disponibilité
- Démarrages et arrêts
- Pannes et erreurs
- Déclenchements de disjoncteurs
- Retards
- Succès/échec des sauvegardes

Pour faciliter l'interprétation et donner un sens à toutes ces entrées de journal, nous devrions (idéalement) créer des catégories hiérarchiques de journalisme, telles que pour les attributs non fonctionnels (par exemple, performance, sécurité) et pour les attributs liés aux fonctionnalités (par exemple, recherche, classement).

Utiliser la télémétrie pour guider la résolution de problèmes

Comme décrit au début de ce chapitre, les performeurs de haut niveau utilisent une approche disciplinée pour résoudre les problèmes. Cela contraste avec la pratique plus courante d'utiliser des rumeurs et des ouï-dire, ce qui peut conduire à la métrique malheureuse du temps moyen jusqu'à la déclaration d'innocence : à quelle vitesse pouvons-nous convaincre tout le monde que nous ne sommes pas responsables de l'incident.

Lorsqu'il y a une culture de culpabilité autour des pannes et des problèmes, les groupes peuvent éviter de documenter les changements et d'afficher la télémétrie là où tout le monde peut la voir, pour éviter d'être blâmés pour les pannes.

D'autres résultats négatifs dus au manque de télémétrie publique incluent une atmosphère politique fortement chargée, le besoin de détourner les accusations, et, pire encore, l'incapacité de créer une connaissance institutionnelle sur la façon dont les incidents se sont produits et les apprentissages nécessaires pour prévenir ces erreurs à l'avenir.

En revanche, la télémétrie nous permet d'utiliser la méthode scientifique pour formuler des hypothèses sur ce qui cause un problème particulier et ce qui est nécessaire pour le résoudre. Des exemples de questions auxquelles nous pouvons répondre lors de la résolution de problèmes incluent :

- Quelles preuves avons-nous de notre surveillance indiquant qu'un problème se produit réellement ?
- Quels sont les événements pertinents et les changements dans nos applications et environnements qui pourraient avoir contribué au problème ?
- Quelles hypothèses pouvons-nous formuler pour confirmer le lien entre les causes proposées et les effets ?
- Comment pouvons-nous prouver quelles hypothèses sont correctes et mettre en œuvre une correction réussie ?

La valeur d'une résolution de problèmes basée sur les faits réside non seulement dans un MTTR (Mean Time To Repair) considérablement plus rapide (et de meilleurs résultats pour le client), mais aussi dans sa consolidation de la perception d'une relation gagnant-gagnant entre le Développement et les Opérations.

Activer la création de métriques de production comme partie du travail quotidien

Pour permettre à tout le monde de trouver et de résoudre des problèmes dans leur travail quotidien, nous devons permettre à chacun de créer des métriques dans leur travail quotidien qui peuvent être facilement créées, affichées et analysées. Pour ce faire, nous devons créer l'infrastructure et les bibliothèques nécessaires pour rendre aussi facile que possible à toute personne en Développement ou en Opérations de créer de la télémétrie pour toute fonctionnalité qu'ils construisent. Dans l'idéal, cela devrait être aussi simple que d'écrire une ligne de code pour créer une nouvelle métrique qui apparaît dans un tableau de bord commun où tout le monde dans la chaîne de valeur peut la voir.

C'était la philosophie qui a guidé le développement de l'une des bibliothèques de métriques les plus largement utilisées, appelée StatsD, qui a été créée et open-source chez Etsy. Comme l'a décrit John Allspaw, "Nous avons conçu StatsD pour empêcher tout développeur de dire : 'C'est trop compliqué d'instrumenter mon code!' Maintenant, ils peuvent le faire avec une seule ligne de code. Il était important pour nous que pour un développeur, ajouter de la télémétrie de production ne soit pas aussi difficile que de modifier un schéma de base de données."

StatsD peut générer des minuteries et des compteurs avec une seule ligne de code (en Ruby, Perl, Python, Java, et d'autres langages) et est souvent utilisé en conjonction avec Graphite ou Grafana, qui rendent les événements de métrique sous forme de graphiques et de tableaux de bord.

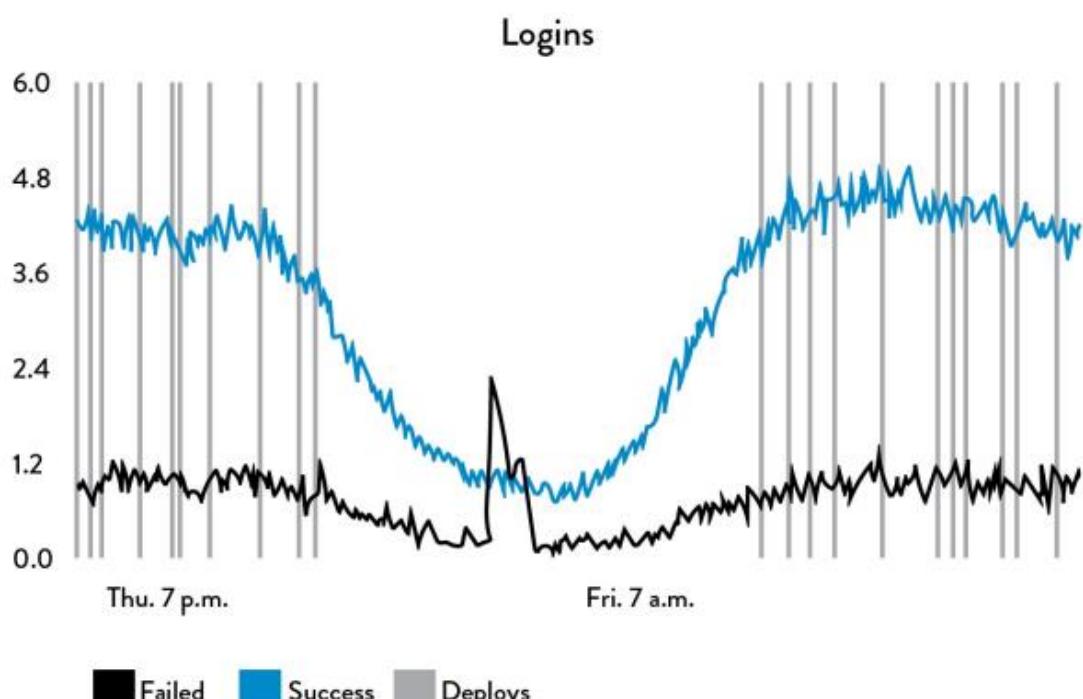


Figure 27: One line of code to generate telemetry using StatsD and Graphite at Etsy
(Source: Ian Malpass, "Measure Anything, Measure Everything.")

Figure 27 ci-dessus montre un exemple de la manière dont une seule ligne de code crée un événement de connexion utilisateur (dans ce cas, une ligne de code PHP : "StatsD::increment("login.successes")). Le graphique résultant montre le nombre de connexions réussies et échouées par minute, avec des lignes verticales représentant un déploiement en production superposées au graphique.

Lorsque nous générerons des graphiques de notre télémétrie, nous superposons également les moments où des modifications en production se produisent, car nous savons que la majorité des problèmes en production sont causés par des changements en production, y compris les déploiements de code. Cela fait partie de ce qui nous permet d'avoir un taux de changement élevé tout en maintenant un système de travail sûr.

Des bibliothèques alternatives à StatsD qui permettent aux développeurs de générer une télémétrie de production facilement agrégée et analysée incluent JMX et codahale metrics. D'autres outils qui créent des métriques inestimables pour la résolution de problèmes incluent New Relic, AppDynamics et Dynatrace. Des outils tels que munin et collectd peuvent être utilisés pour créer des fonctionnalités similaires.

En générant de la télémétrie de production dans le cadre de notre travail quotidien, nous créons une capacité en constante amélioration pour non seulement voir les problèmes au fur et à mesure qu'ils se produisent, mais aussi pour concevoir notre travail de manière à révéler des problèmes de conception et d'exploitation, permettant de suivre un nombre croissant de métriques, comme nous l'avons vu dans l'étude de cas Etsy.

Créer un accès en libre-service à la télémétrie et aux radiateurs d'information

Dans les étapes précédentes, nous avons permis à la Développement et aux Opérations de créer et d'améliorer la télémétrie de production dans le cadre de leur travail quotidien. Dans cette étape, notre objectif est de diffuser ces informations au reste de l'organisation, en veillant à ce que toute personne souhaitant des informations sur l'un des services que nous gérons puisse les obtenir sans avoir besoin d'accès aux systèmes de production ou de comptes privilégiés, ou sans devoir ouvrir un ticket et attendre des jours que quelqu'un configure le graphique pour eux.

En rendant la télémétrie rapide, facile à obtenir et suffisamment centralisée, tout le monde dans le flux de valeur peut partager une vue commune de la réalité. En général, cela signifie que les métriques de production seront diffusées sur des pages web générées par un serveur centralisé, tel que Graphite ou l'une des autres technologies décrites dans la section précédente.

Nous voulons que notre télémétrie de production soit très visible, ce qui signifie la placer dans des zones centrales où travaillent la Développement et les Opérations, permettant ainsi à tous ceux qui sont intéressés de voir comment nos services fonctionnent. Au minimum, cela inclut tout le monde dans notre flux de valeur, comme la Développement, les Opérations, la Gestion de Produit et l'Infosec.

Cela est souvent appelé un radiateur d'information, défini par l'Agile Alliance comme "le terme générique pour toute une série d'affichages manuscrits, dessinés, imprimés ou électroniques qu'une équipe place dans un endroit très visible, de sorte que tous les membres de l'équipe ainsi que les passants puissent voir les dernières informations en un coup d'œil : nombre de tests automatisés, vitesse, rapports d'incidents, état de l'intégration continue, etc. Cette idée a été initialement développée dans le cadre du système de production Toyota."

En plaçant des radiateurs d'information dans des endroits très visibles, nous promouvons la responsabilité parmi les membres de l'équipe, en démontrant activement les valeurs suivantes :

- L'équipe n'a rien à cacher à ses visiteurs (clients, parties prenantes, etc.)
- L'équipe n'a rien à cacher à elle-même : elle reconnaît et confronte les problèmes

Maintenant que nous possédons l'infrastructure pour créer et diffuser la télémétrie de production à toute l'organisation, nous pouvons également choisir de diffuser ces informations à nos clients internes et même à nos clients externes. Par exemple, nous pourrions le faire en créant des pages d'état de service accessibles au public afin que les clients puissent savoir comment les services dont ils dépendent fonctionnent.

Bien qu'il puisse y avoir une certaine résistance à fournir ce niveau de transparence, Ernest Mueller décrit la valeur de le faire :

"Une des premières actions que j'entreprends en intégrant une organisation est d'utiliser des radiateurs d'information pour communiquer les problèmes et détailler les changements que nous apportons – cela est généralement très bien accueilli par nos unités commerciales, qui étaient souvent laissées dans l'ombre auparavant. Et pour les groupes Développement et Opérations qui doivent travailler ensemble pour fournir un service aux autres, nous avons besoin de cette communication, information et retour constants."

Nous pouvons même étendre cette transparence davantage – au lieu d'essayer de garder les problèmes impactant les clients secrets, nous pouvons diffuser cette information à nos clients externes. Cela montre que nous valorisons la transparence, aidant ainsi à construire et à gagner la confiance des clients.

Étude de cas - Créer des métriques en libre-service chez LinkedIn (2011)

Comme décrit dans la Partie III, LinkedIn a été créé en 2003 pour aider les utilisateurs à se connecter "à votre réseau pour de meilleures opportunités professionnelles." En novembre 2015, LinkedIn comptait plus de 350 millions de membres générant des dizaines de milliers de requêtes par seconde, résultant en des millions de requêtes par seconde sur les systèmes backend de LinkedIn.

Prachi Gupta, directrice de l'ingénierie chez LinkedIn, a écrit en 2011 sur l'importance de la télémétrie de production : "Chez LinkedIn, nous insistons pour nous assurer que le site est en ligne et que nos membres ont accès à toutes les fonctionnalités du site à tout moment. Pour remplir cet engagement, nous devons détecter et répondre aux pannes et aux goulets d'étranglement dès qu'ils commencent à se produire. C'est pourquoi nous utilisons ces graphiques de séries temporelles pour la surveillance du site afin de détecter et réagir aux incidents en quelques minutes... Cette technique de surveillance s'est avérée être un excellent outil pour les ingénieurs. Elle nous permet d'agir rapidement et nous donne le temps de détecter, trier et résoudre les problèmes."

Cependant, en 2010, même s'il y avait un volume incroyablement large de télémétrie générée, il était extrêmement difficile pour les ingénieurs d'accéder aux données, sans parler de les analyser. Ainsi a commencé le projet de stage d'été d'Eric Wong chez LinkedIn, qui s'est transformé en l'initiative de télémétrie de production qui a créé InGraphs.

Wong a écrit : "Pour obtenir quelque chose d'aussi simple que l'utilisation du CPU de tous les hôtes exécutant un service particulier, il fallait déposer un ticket et quelqu'un passait 30 minutes à le [rapport] préparer." À l'époque, LinkedIn utilisait Zenoss pour collecter les métriques, mais comme l'explique Wong, "Obtenir des données de Zenoss nécessitait de fouiller dans une interface web lente, alors j'ai écrit quelques scripts python pour aider à rationaliser le processus. Bien qu'il y ait encore une intervention manuelle dans la mise en place de la collecte des métriques, j'ai pu réduire le temps passé à naviguer dans l'interface de Zenoss."

Au cours de l'été, il a continué à ajouter des fonctionnalités à InGraphs afin que les ingénieurs puissent voir exactement ce qu'ils voulaient voir, ajoutant la capacité de faire des calculs sur plusieurs ensembles de données, de visualiser les tendances semaine après semaine pour comparer les performances historiques, et même de définir des tableaux de bord personnalisés pour choisir exactement quelles métriques seraient affichées sur une seule page.

En écrivant sur les résultats de l'ajout de fonctionnalités à InGraphs et la valeur de cette capacité, Gupta note : "L'efficacité de notre système de surveillance a été mise en évidence lorsqu'une de nos fonctionnalités de surveillance InGraphs liée à un grand fournisseur de web-mail a commencé à montrer une tendance à la baisse et que le fournisseur s'est rendu compte qu'il avait un problème dans son système seulement après que nous les avons contactés !"

Ce qui a commencé comme un projet de stage d'été est maintenant l'une des parties les plus visibles des opérations de LinkedIn. InGraphs a été tellement réussi que les graphiques en temps réel sont présentés en évidence dans les bureaux d'ingénierie de la société où les visiteurs ne peuvent pas les manquer.

Trouver et combler les lacunes de télémétrie

Nous avons maintenant créé l'infrastructure nécessaire pour rapidement créer de la télémétrie de production à travers toute notre pile d'applications et la diffuser dans toute notre organisation.

Dans cette étape, nous allons identifier les lacunes dans notre télémétrie qui entravent notre capacité à détecter et résoudre rapidement les incidents – c'est particulièrement pertinent si Dev et Ops ont actuellement peu (ou pas) de télémétrie. Nous utiliserons ces données plus tard pour mieux anticiper les problèmes, ainsi que pour permettre à chacun de recueillir les informations dont il a besoin pour prendre de meilleures décisions afin d'atteindre les objectifs organisationnels.

Pour y parvenir, nous devons créer suffisamment de télémétrie à tous les niveaux de la pile d'applications pour tous nos environnements, ainsi que pour les pipelines de déploiement qui les soutiennent. Nous avons besoin de métriques des niveaux suivants :

- **Niveau commercial** : Exemples incluent le nombre de transactions de vente, le revenu des transactions de vente, les inscriptions d'utilisateurs, le taux de désabonnement, les résultats des tests A/B, etc.
- **Niveau applicatif** : Exemples incluent les temps de transaction, les temps de réponse des utilisateurs, les défauts d'application, etc.
- **Niveau infrastructure** (par exemple, base de données, système d'exploitation, réseau, stockage) : Exemples incluent le trafic du serveur web, la charge CPU, l'utilisation du disque, etc.
- **Niveau logiciel client** (par exemple, JavaScript sur le navigateur client, application mobile) : Exemples incluent les erreurs et plantages d'application, les temps de transaction mesurés par l'utilisateur, etc.
- **Niveau pipeline de déploiement** : Exemples incluent l'état du pipeline de construction (par exemple, rouge ou vert pour nos différentes suites de tests automatisés), les délais de déploiement des changements, les fréquences de déploiement, les promotions d'environnement de test, et l'état de l'environnement.

En ayant une couverture de télémétrie dans tous ces domaines, nous serons capables de voir la santé de tout ce dont notre service dépend, en utilisant des données et des faits au lieu de rumeurs, de blâmes, de reproches, etc.

De plus, nous permettons mieux la détection des événements pertinents pour la sécurité en surveillant toutes les fautes d'application et d'infrastructure (par exemple, les terminaisons de programme anormales, les erreurs et exceptions d'application, et les erreurs de serveur et de stockage). Non seulement cette télémétrie informe mieux le Développement et les Opérations lorsque nos services plantent, mais ces erreurs sont souvent des indicateurs qu'une vulnérabilité de sécurité est en cours d'exploitation active.

En détectant et en corigeant les problèmes plus tôt, nous pouvons les réparer lorsqu'ils sont petits et faciles à corriger, avec moins de clients impactés. De plus, après chaque incident de production, nous devrions identifier toute télémétrie manquante qui aurait pu permettre une détection et une récupération plus rapides ; ou, mieux encore, nous pouvons identifier ces lacunes lors du développement des fonctionnalités dans notre processus de revue par les pairs.

Métriques d'applications et d'affaires

Au niveau de l'application, notre objectif est de nous assurer que nous générerons de la télémétrie non seulement autour de la santé de l'application (par exemple, utilisation de la mémoire, nombre de transactions, etc.), mais aussi de mesurer dans quelle mesure nous atteignons nos objectifs organisationnels (par exemple, nombre de nouveaux utilisateurs, événements de connexion des utilisateurs, durées des sessions des utilisateurs, pourcentage d'utilisateurs actifs, fréquence d'utilisation de certaines fonctionnalités, etc.).

Par exemple, si nous avons un service qui soutient le commerce électronique, nous voulons nous assurer que nous avons de la télémétrie autour de tous les événements utilisateur qui conduisent à une transaction réussie qui génère des revenus. Nous pouvons alors instrumenter toutes les actions des utilisateurs nécessaires pour obtenir les résultats souhaités pour nos clients.

Ces métriques varieront selon les différents domaines et objectifs organisationnels. Par exemple, pour les sites de commerce électronique, nous pouvons vouloir maximiser le temps passé sur le site ; cependant, pour les moteurs de recherche, nous pouvons vouloir réduire le temps passé sur le site, car de longues sessions peuvent indiquer que les utilisateurs rencontrent des difficultés à trouver ce qu'ils recherchent.

En général, les métriques d'affaires feront partie d'un entonnoir d'acquisition de clients, qui est le parcours théorique qu'un client potentiel suivra pour effectuer un achat. Par exemple, sur un site de commerce électronique, les événements de parcours mesurables incluent le temps total passé sur le site, les clics sur les liens produits, les ajouts au panier et les commandes complétées.

Ed Blankenship, Senior Product Manager pour Microsoft Visual Studio Team Services, décrit : "Souvent, les équipes de fonctionnalités définiront leurs objectifs dans un entonnoir d'acquisition, avec pour but que leur fonctionnalité soit utilisée dans le travail quotidien de chaque client. Parfois, ils sont décrits de manière informelle comme des 'testeurs', 'utilisateurs actifs', 'utilisateurs engagés' et 'utilisateurs profondément engagés', avec une télémétrie supportant chaque étape."

Notre objectif est que chaque métrique d'affaires soit exploitable - ces principales métriques devraient aider à informer comment changer notre produit et être propices aux expérimentations et tests A/B. Lorsque les métriques ne sont pas exploitables, elles sont probablement des métriques de vanité qui fournissent peu d'informations utiles - nous voulons les stocker, mais probablement pas les afficher, encore moins les alerter.

Idéalement, toute personne visualisant nos radiateurs d'information pourra comprendre les informations que nous montrons dans le contexte des résultats organisationnels souhaités, tels que les objectifs de revenus, l'acquisition d'utilisateurs, les taux de conversion, etc. Nous devrions définir et lier chaque métrique à une métrique de résultat d'affaires dès les premières étapes de la définition et du développement des fonctionnalités, et mesurer les résultats après leur déploiement en production. De plus, cela aide les propriétaires de produit à décrire le contexte commercial de chaque fonctionnalité pour tout le monde dans le flux de valeur.

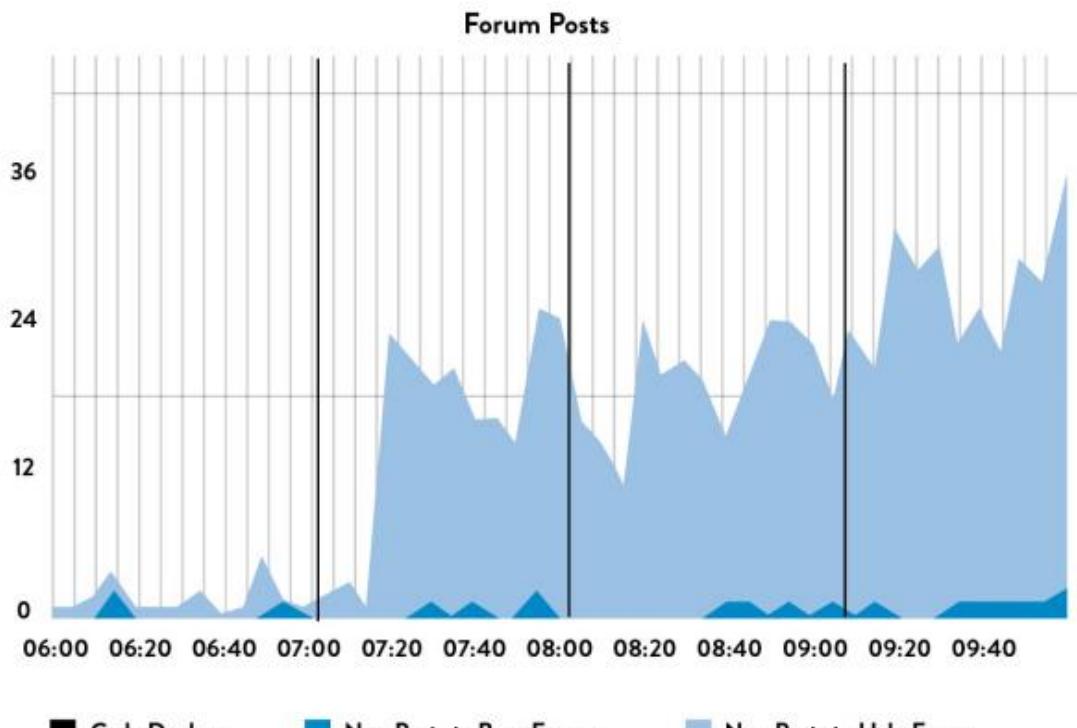


Figure 28: Amount of user excitement of new features in user forum posts after deployments (Source: Mike Brittain, “Tracking Every Release,” [CodeasCraft.com](https://codeascraft.com/2010/12/08/track-every-release/), December 8, 2010, <https://codeascraft.com/2010/12/08/track-every-release/>.)

Un contexte commercial supplémentaire peut être créé en étant conscient des périodes pertinentes pour la planification et les opérations commerciales de haut niveau, et en les affichant visuellement, comme les périodes de transaction élevées associées aux saisons de vente de vacances, les périodes de clôture financière de fin de trimestre ou les audits de conformité programmés. Ces informations peuvent servir de rappel pour éviter de programmer des changements risqués lorsque la disponibilité est cruciale ou éviter certaines activités lorsque des audits sont en cours.

En diffusant la manière dont les clients interagissent avec ce que nous construisons dans le contexte de nos objectifs, nous permettons un retour rapide aux équipes de fonctionnalités afin qu'elles puissent voir si les capacités que nous développons sont réellement utilisées et dans quelle mesure elles atteignent les objectifs commerciaux. Par conséquent, nous renforçons les attentes culturelles selon lesquelles l'instrumentation et l'analyse de l'utilisation par les clients font également partie de notre travail quotidien, afin de mieux comprendre comment notre travail contribue à nos objectifs organisationnels.

Métriques d'infrastructure

Tout comme nous l'avons fait pour les métriques d'application, notre objectif pour l'infrastructure de production et de non-production est de nous assurer que nous générions suffisamment de télémétrie pour que, si un problème survient dans n'importe quel environnement, nous puissions rapidement déterminer si l'infrastructure est une cause contributive du problème. De plus, nous devons être capables d'identifier exactement ce qui dans l'infrastructure contribue au problème (par exemple, base de données, système d'exploitation, stockage, réseau, etc.).

Nous voulons rendre visible autant de télémétrie d'infrastructure que possible, à travers tous les intervenants technologiques, idéalement organisés par service ou application. En d'autres termes, lorsque quelque chose ne va pas dans notre environnement, nous devons savoir exactement quelles applications et services pourraient être ou sont affectés.

Connecter les services et l'infrastructure

Dans les décennies passées, créer des liens entre un service et l'infrastructure de production dont il dépendait était souvent un effort manuel (comme les CMDB ITIL ou la création de définitions de configuration dans des outils d'alerte tels que Nagios). Cependant, de plus en plus, ces liens sont maintenant enregistrés automatiquement au sein de nos services, qui sont ensuite découverts dynamiquement et utilisés en production via des outils tels que Zookeeper, Etcd, Consul, etc.

Ces outils permettent aux services de s'enregistrer eux-mêmes, en stockant les informations dont d'autres services ont besoin pour interagir avec eux (par exemple, adresse IP, numéros de

port, URI). Cela résout la nature manuelle du CMDB ITIL et est absolument nécessaire lorsque les services sont composés de centaines (ou de milliers, voire de millions) de noeuds, chacun avec des adresses IP assignées dynamiquement.

Visualisation des métriques d'affaires, d'application et d'infrastructure

Quel que soit le degré de simplicité ou de complexité de nos services, le fait de représenter graphiquement nos métriques d'affaires aux côtés de nos métriques d'application et d'infrastructure nous permet de détecter lorsque des problèmes surviennent. Par exemple, nous pouvons voir que les nouvelles inscriptions de clients chutent à 20% des normes quotidiennes, puis immédiatement constater que toutes nos requêtes de base de données prennent cinq fois plus de temps que d'habitude, nous permettant ainsi de concentrer notre résolution de problèmes.

De plus, les métriques d'affaires créent un contexte pour nos métriques d'infrastructure, permettant au Développement et aux Opérations de mieux travailler ensemble vers des objectifs communs. Comme l'observe Jody Mulkey, CTO de Ticketmaster/LiveNation, « au lieu de mesurer les opérations en fonction de la quantité de temps d'arrêt, je trouve qu'il est beaucoup mieux de mesurer à la fois le développement et les opérations en fonction des conséquences commerciales réelles du temps d'arrêt : combien de revenus aurions-nous dû atteindre, mais n'avons pas atteints. »

Télémétrie pour les environnements de préproduction

Notez qu'en plus de surveiller nos services de production, nous avons également besoin de télémétrie pour ces services dans nos environnements de préproduction (par exemple, développement, test, staging, etc.). Cela nous permet de détecter et de corriger les problèmes avant qu'ils ne passent en production, comme détecter lorsque nous avons des temps d'insertion de base de données en constante augmentation en raison d'un index de table manquant.

Superposer d'autres informations pertinentes sur nos métriques

Même après avoir créé notre pipeline de déploiement qui nous permet de faire des changements de production petits et fréquents, les changements créent encore intrinsèquement des risques. Les effets secondaires opérationnels ne sont pas seulement des pannes, mais aussi des perturbations importantes et des écarts par rapport aux opérations standard.

Pour rendre les changements visibles, nous rendons le travail visible en superposant toutes les activités de déploiement de production sur nos graphiques. Par exemple, pour un service qui gère un grand nombre de transactions entrantes, les changements de production peuvent entraîner une période de stabilisation significative, où les performances se dégradent considérablement à mesure que toutes les recherches de cache échouent.

Pour mieux comprendre et préserver la qualité de service, nous voulons comprendre à quelle vitesse les performances reviennent à la normale, et si nécessaire, prendre des mesures pour améliorer les performances.

Conclusion

Les améliorations permises par la télémétrie de production de Etsy et LinkedIn nous montrent à quel point il est crucial de voir les problèmes lorsqu'ils se produisent, afin que nous puissions en rechercher la cause et remédier rapidement à la situation. En faisant émettre de la télémétrie par tous les éléments de notre service, qu'il s'agisse de notre application, de notre base de données ou de notre environnement, et en rendant cette télémétrie largement disponible, nous pouvons détecter et corriger les problèmes bien avant qu'ils ne causent quelque chose de catastrophique, idéalement bien avant qu'un client ne remarque qu'il y a un problème. Le résultat est non seulement des clients plus satisfaits, mais aussi, en réduisant la quantité d'incidents et de crises lorsque les choses tournent mal, nous avons un lieu de travail plus heureux et plus productif avec moins de stress et des niveaux d'épuisement professionnel plus faibles.

Analyser la télémétrie pour mieux anticiper les problèmes et atteindre les objectifs

Comme nous l'avons vu dans le chapitre précédent, nous avons besoin d'une télémétrie de production suffisante dans nos applications et notre infrastructure pour voir et résoudre les problèmes lorsqu'ils surviennent. Dans ce chapitre, nous allons créer des outils qui nous permettront de découvrir des variations et des signaux de défaillance de plus en plus faibles cachés dans notre télémétrie de production afin de prévenir les défaillances catastrophiques. De nombreuses techniques statistiques seront présentées, ainsi que des études de cas démontrant leur utilisation.

Un excellent exemple d'analyse de la télémétrie pour trouver et résoudre de manière proactive des problèmes avant que les clients ne soient impactés peut-être vu chez Netflix, un fournisseur mondial de films et de séries télévisées en streaming. Netflix a généré des revenus de 6,2 milliards de dollars grâce à soixante-quinze millions d'abonnés en 2015. L'un de leurs objectifs est de fournir la meilleure expérience possible à ceux qui regardent des vidéos en ligne dans le monde entier, ce qui nécessite une infrastructure de diffusion robuste, évolutive et résiliente. Roy Rapoport décrit l'un des défis de la gestion du service de diffusion vidéo basé sur le cloud de Netflix : "Étant donné un troupeau de bétail qui devrait tous se ressembler et agir de la même manière, quels bovins sont différents des autres ? Ou plus concrètement, si nous avons un cluster de calcul sans état de mille nœuds, tous exécutant le même logiciel et soumis à une charge de trafic approximativement identique, notre défi est de trouver les nœuds qui ne ressemblent pas aux autres."

L'une des techniques statistiques utilisées par l'équipe de Netflix en 2012 était la détection des anomalies, définie par Victoria J. Hodge et Jim Austin de l'Université de York comme la détection "des conditions de fonctionnement anormales susceptibles de provoquer une dégradation significative des performances, telle qu'un défaut de rotation d'un moteur d'avion ou un problème d'écoulement dans un pipeline."

Rapoport explique que Netflix "utilisait la détection des anomalies de manière très simple, en calculant d'abord ce qui était le 'normal actuel' en ce moment, étant donné la population de nœuds dans un cluster de calcul. Puis nous identifions quels nœuds ne correspondaient pas à ce schéma, et retirions ces nœuds de la production."

Rapoport poursuit : "Nous pouvons automatiquement signaler les nœuds défaillants sans avoir à définir de quelque manière que ce soit ce qu'est le comportement 'correct'. Et puisque nous sommes conçus pour fonctionner de manière résiliente dans le cloud, nous ne demandons à personne dans les Opérations de faire quoi que ce soit – au lieu de cela, nous éliminons simplement le nœud de calcul malade ou défaillant, puis nous le journalisons ou notifions les ingénieurs de la manière qu'ils souhaitent."

En mettant en œuvre le processus de détection des anomalies des serveurs, Rapoport déclare que Netflix a "massivement réduit l'effort de détection des serveurs défaillants et, plus important encore, a massivement réduit le temps nécessaire pour les réparer, ce qui a amélioré la qualité du service. Le bénéfice de l'utilisation de ces techniques pour préserver la santé mentale des employés, l'équilibre entre vie professionnelle et vie privée, et la qualité du service ne peut être surestimé."

Le travail accompli chez Netflix met en évidence une manière très spécifique d'utiliser la télémétrie pour atténuer les problèmes avant qu'ils n'impactent nos clients.

Tout au long de ce chapitre, nous explorerons de nombreuses techniques statistiques et de visualisation (y compris la détection des anomalies) que nous pouvons utiliser pour analyser notre télémétrie afin de mieux anticiper les problèmes. Cela nous permet de résoudre les problèmes plus rapidement, moins cher et plus tôt que jamais, avant que notre client ou quiconque dans notre organisation ne soit impacté ; de plus, nous créerons également plus de contexte pour nos données afin de nous aider à prendre de meilleures décisions et à atteindre nos objectifs organisationnels.

Utiliser les moyennes et les écarts types pour détecter les problèmes potentiels

L'une des techniques statistiques les plus simples que nous puissions utiliser pour analyser une métrique de production est de calculer sa moyenne (ou moyenne) et ses écarts types. En faisant cela, nous pouvons créer un filtre qui détecte quand cette métrique est significativement différente de sa norme, et même configurer nos alertes afin que nous puissions prendre des mesures correctives (par exemple, notifier le personnel de production d'astreinte à 2 h du matin pour enquêter lorsque les requêtes de la base de données sont significativement plus lentes que la moyenne).

Lorsque des services de production critiques ont des problèmes, réveiller les gens à 2 h du matin peut être la bonne chose à faire. Cependant, lorsque nous créons des alertes qui ne sont pas exploitables ou qui sont des faux positifs, nous avons inutilement réveillé des personnes en plein milieu de la nuit. Comme l'a observé John Vincent, un des premiers leaders du mouvement DevOps, "La fatigue des alertes est le plus gros problème que nous ayons actuellement... Nous devons être plus intelligents avec nos alertes ou nous allons tous devenir fous."

Nous créons de meilleures alertes en augmentant le rapport signal/bruit, en nous concentrant sur les variations ou les anomalies qui comptent. Supposons que nous analysions le nombre de

tentatives de connexion non autorisées par jour. Nos données collectées ont une distribution gaussienne (c'est-à-dire une distribution normale ou en cloche) qui correspond au graphique de la figure 29. La ligne verticale au milieu de la courbe en cloche est la moyenne, et les premières, deuxièmes et troisièmes déviations standard indiquées par les autres lignes verticales contiennent respectivement 68 %, 95 % et 99,7 % des données.

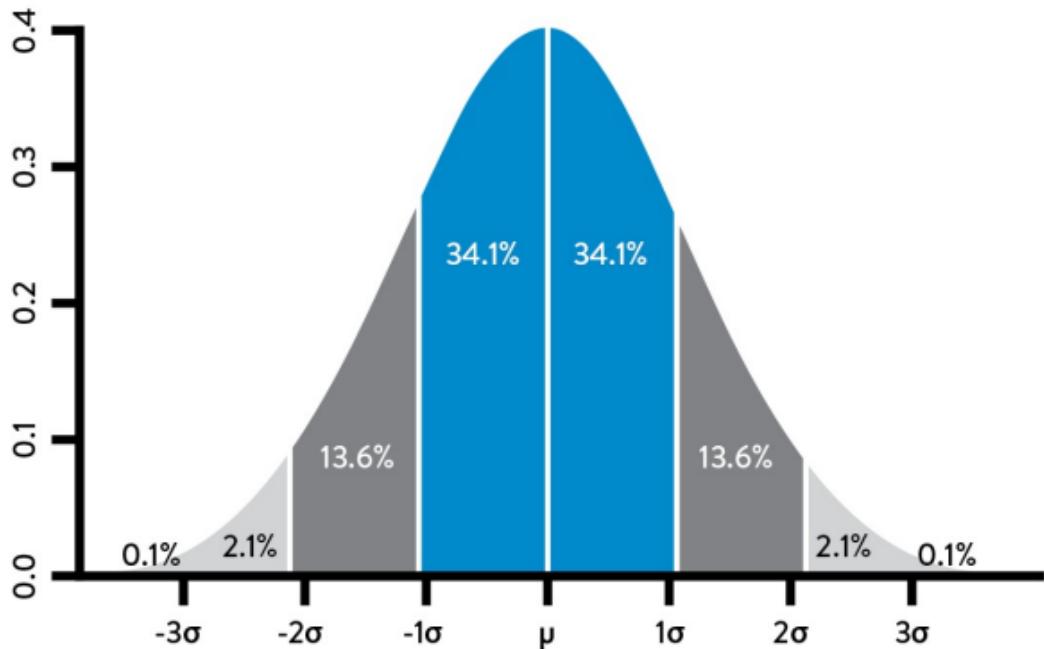


Figure 29: Standard deviations (σ) & mean (μ) with Gaussian distribution (Source: Wikipedia's "Normal Distribution" entry, https://en.wikipedia.org/wiki/Normal_distribution.)

Une utilisation courante des écarts types consiste à inspecter périodiquement l'ensemble des données pour une métrique et à alerter si elle s'écarte de manière significative de la moyenne. Par exemple, nous pouvons définir une alerte lorsque le nombre de tentatives de connexion non autorisées par jour dépasse de trois écarts types la moyenne. À condition que cet ensemble de données suive une distribution gaussienne, nous nous attendons à ce que seuls 0,3 % des points de données déclenchent l'alerte.

Même ce type simple d'analyse statistique est précieux, car personne n'a besoin de définir une valeur seuil statique, ce qui est impraticable si nous suivons des milliers ou des centaines de milliers de métriques de production.

Pour le reste de ce livre, nous utiliserons les termes télémétrie, métrique et ensembles de données de manière interchangeable - en d'autres termes, une métrique (par exemple, "temps de chargement des pages") se traduit par un ensemble de données (par exemple, 2 ms, 8 ms, 11 ms, etc.), le terme utilisé par les statisticiens pour décrire une matrice de points de données où

chaque colonne représente une variable sur laquelle des opérations statistiques sont effectuées.

Instrumenter et alerter sur les résultats indésirables

Tom Limoncelli, co-auteur de *The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems* et ancien ingénieur en fiabilité de site chez Google, raconte l'histoire suivante sur la surveillance :

"Quand les gens me demandent des recommandations sur ce qu'il faut surveiller, je plaisante en disant que dans un monde idéal, nous supprimerions toutes les alertes que nous avons actuellement dans notre système de surveillance. Ensuite, après chaque panne visible par l'utilisateur, nous demanderions quels indicateurs auraient prédit cette panne et les ajouterions à notre système de surveillance, en alertant au besoin. Répéter. Maintenant, nous n'avons plus que des alertes qui préviennent les pannes, au lieu d'être bombardés par des alertes après qu'une panne se soit déjà produite."

Dans cette étape, nous allons reproduire les résultats d'un tel exercice. L'une des façons les plus simples de le faire est d'analyser nos incidents les plus graves des trente derniers jours et de créer une liste de télémétrie qui aurait pu permettre une détection et un diagnostic plus précoces et plus rapides du problème, ainsi qu'une confirmation plus facile et plus rapide qu'une solution efficace avait été mise en œuvre.

Par exemple, si nous avions un problème où notre serveur web NGINX cessait de répondre aux demandes, nous examinerions les indicateurs principaux qui auraient pu nous avertir plus tôt que nous commençons à dévier des opérations standards, tels que :

- **Niveau application** : augmentation des temps de chargement des pages web, etc.
- **Niveau OS** : mémoire libre du serveur faible, espace disque faible, etc.
- **Niveau base de données** : temps de transaction de la base de données plus longs que d'habitude, etc.
- **Niveau réseau** : diminution du nombre de serveurs fonctionnels derrière le répartiteur de charge, etc.

Chacune de ces métriques est un précurseur potentiel d'un incident de production. Pour chacune, nous configurerions nos systèmes d'alerte pour les notifier lorsqu'elles s'écartent suffisamment de la moyenne, afin que nous puissions prendre des mesures correctives.

En répétant ce processus sur des signaux de défaillance de plus en plus faibles, nous détectons les problèmes de plus en plus tôt dans le cycle de vie, ce qui entraîne moins d'incidents affectant les clients et de quasi-accidents. En d'autres termes, nous prévenons les problèmes tout en permettant une détection et une correction plus rapides.

Problèmes qui surviennent lorsque nos données de télémétrie ont une distribution non gaussienne

Utiliser les moyennes et les écarts types pour détecter les variations peut être extrêmement utile. Cependant, l'utilisation de ces techniques sur de nombreux ensembles de données de télémétrie que nous utilisons dans les Opérations ne générera pas les résultats escomptés. Comme l'observe le Dr Toufic Boubez, "Non seulement nous recevrons des appels de réveil à 2 heures du matin, mais nous les recevrons à 2 h 37, 4 h 13, 5 h 17. Cela se produit lorsque les données sous-jacentes que nous surveillons n'ont pas une distribution gaussienne."

En d'autres termes, lorsque la distribution de l'ensemble de données n'a pas la courbe en cloche gaussienne décrite précédemment, les propriétés associées aux écarts types ne s'appliquent pas. Par exemple, considérons le scénario dans lequel nous surveillons le nombre de téléchargements de fichiers par minute depuis notre site web. Nous voulons détecter les périodes où nous avons un nombre inhabituellement élevé de téléchargements, comme lorsque notre taux de téléchargement dépasse de trois écarts types notre moyenne, afin que nous puissions ajouter de manière proactive plus de capacité.

La figure 30 montre notre nombre de téléchargements simultanés par minute au fil du temps, avec une barre superposée. Lorsque la barre est noire, le nombre de téléchargements dans une période donnée (parfois appelée "fenêtre glissante") est d'au moins trois écarts types par rapport à la moyenne. Sinon, elle est grise.**

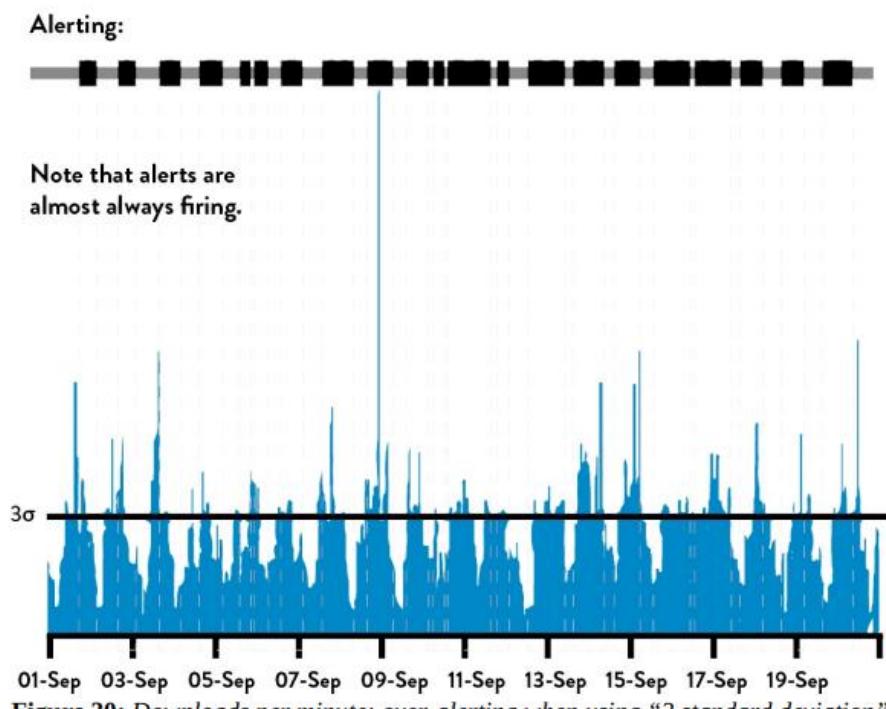


Figure 30: Downloads per minute: over-alerting when using “3 standard deviation” rule

(Source: Dr. Toufic Boubez, “Simple math for anomaly detection.”)

Le problème évident que montre le graphique est que nous alertons presque tout le temps. Cela est dû au fait que, dans presque n'importe quelle période donnée, nous avons des instances où le nombre de téléchargements dépasse notre seuil de trois écarts-types. Pour le confirmer, lorsque nous créons un histogramme (voir figure 31) qui montre la fréquence des téléchargements par minute, nous pouvons voir qu'il n'a pas la forme classique et symétrique de la courbe en cloche. Au lieu de cela, il est évident que la distribution est biaisée vers le bas, montrant que la majorité du temps nous avons très peu de téléchargements par minute, mais que le nombre de téléchargements dépasse fréquemment de trois écarts-types.

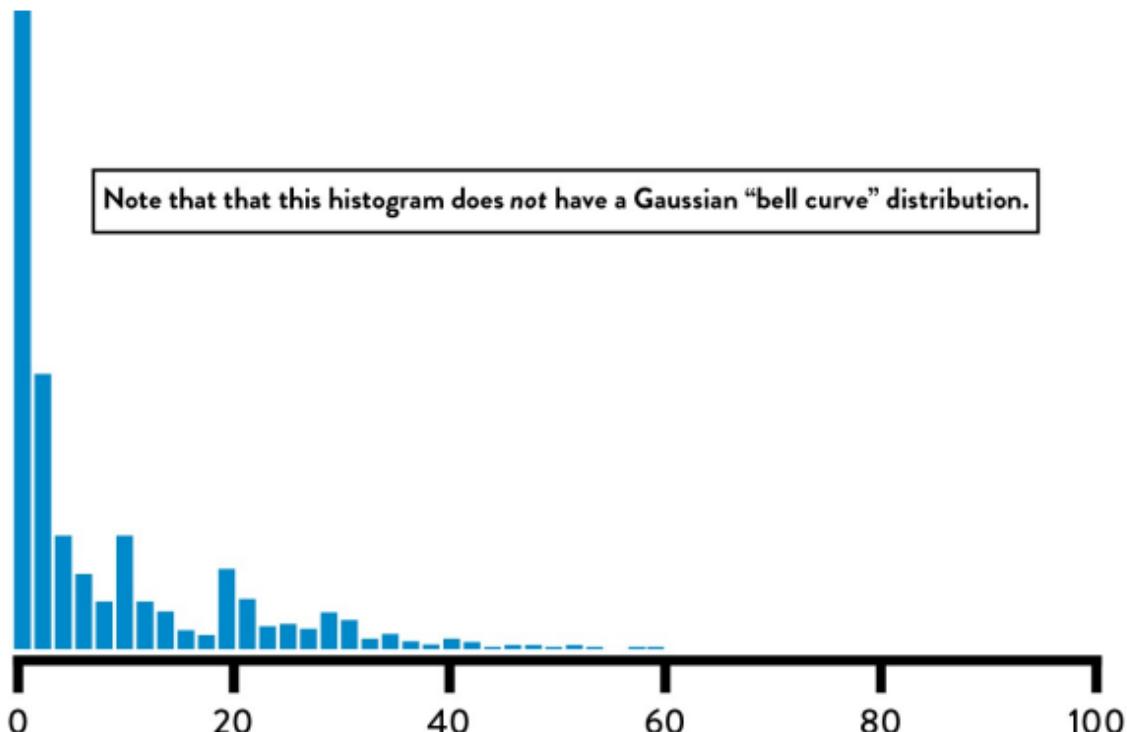


Figure 31: Downloads per minute: histogram of data showing non-Gaussian distribution

(Source: Dr. Toufic Boubez, “Simple math for anomaly detection.”)

De nombreux ensembles de données de production ont une distribution non gaussienne. Dr. Nicole Forsgren explique : « En opérations, beaucoup de nos ensembles de données ont ce que nous appelons une distribution 'chi-carré'. Utiliser des écarts-types pour ces données entraîne non seulement une sur-alerte ou une sous-alerte, mais également des résultats absurdes. » Elle continue : « Lorsque vous calculez le nombre de téléchargements simultanés qui sont trois écarts-types en dessous de la moyenne, vous obtenez un nombre négatif, ce qui n'a évidemment aucun sens. »

La sur-alerte amène les ingénieurs des opérations à être réveillés au milieu de la nuit pendant de longues périodes, même lorsqu'ils ont peu d'actions à prendre. Le problème associé à la sous-alerte est tout aussi significatif. Par exemple, supposons que nous surveillons le nombre de

transactions complètes, et que le nombre de transactions complètes diminue de 50 % en milieu de journée en raison d'une défaillance d'un composant logiciel. Si cela se situe toujours dans les trois écarts-types de la moyenne, aucune alerte ne sera générée, ce qui signifie que nos clients découvriront le problème avant nous, et à ce moment-là, le problème peut être beaucoup plus difficile à résoudre.

Heureusement, il existe des techniques que nous pouvons utiliser pour détecter des anomalies même dans des ensembles de données non gaussiens, qui sont décrites ci-après.

Étude de cas : Auto-Scaling Capacity chez Netflix (2012)

Un autre outil développé chez Netflix pour améliorer la qualité du service, Scryer, répond à certaines des lacunes d'Amazon Auto Scaling (AAS), qui augmente et diminue dynamiquement le nombre de serveurs de calcul AWS en fonction des données de charge de travail. Scryer fonctionne en prédisant les demandes des clients en se basant sur les schémas d'utilisation historiques et en provisionnant la capacité nécessaire.

Scryer a résolu trois problèmes avec AAS. Le premier était de faire face aux pics de demande rapides. Comme les temps de démarrage des instances AWS peuvent être de dix à quarante-cinq minutes, la capacité de calcul supplémentaire était souvent livrée trop tard pour gérer les pics de demande. Le deuxième problème était qu'après les pannes, la diminution rapide de la demande des clients conduisait AAS à supprimer trop de capacité de calcul pour gérer la future demande entrante. Le troisième problème était qu'AAS ne prenait pas en compte les schémas de trafic connus lors de la planification de la capacité de calcul.

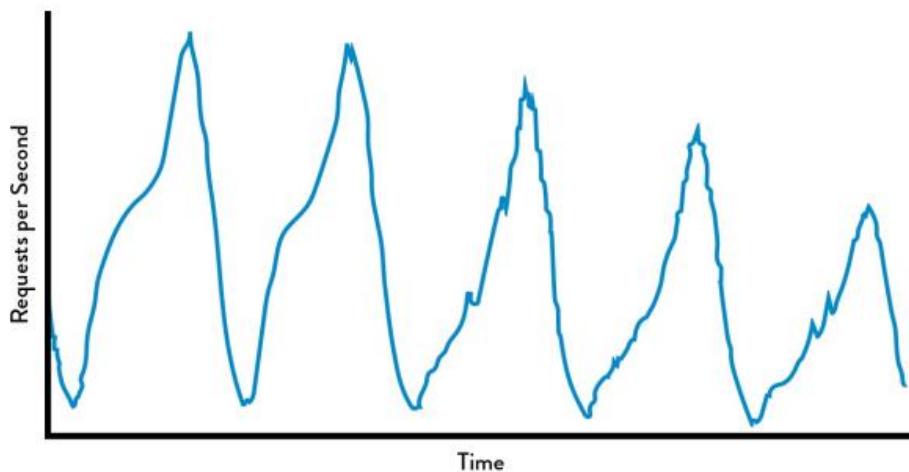


Figure 32: Netflix customer viewing demand for five days (Source: Daniel Jacobson, Danny Yuan, and Neeraj Joshi, “Scryer: Netflix’s Predictive Auto Scaling Engine,” The Netflix Tech Blog, November 5, 2013, <http://techblog.netflix.com/2013/11/scryer-netflixs-predictive-auto-scaling.html>.)

Netflix a tiré parti du fait que les habitudes de visionnage de leurs consommateurs étaient étonnamment cohérentes et prévisibles, malgré l'absence de distributions gaussiennes. Ci-dessous se trouve un graphique reflétant les requêtes des clients par seconde tout au long de la semaine de travail, montrant des habitudes de visionnage régulières et cohérentes du lundi au vendredi. Scryer utilise une combinaison de détections d'anomalies pour éliminer les points de données aberrants, puis utilise des techniques telles que la transformation de Fourier rapide (FFT) et la régression linéaire pour lisser les données tout en préservant les pics de trafic légitimes qui se reproduisent dans leurs données. Le résultat est que Netflix peut prévoir la demande de trafic avec une précision surprenante.

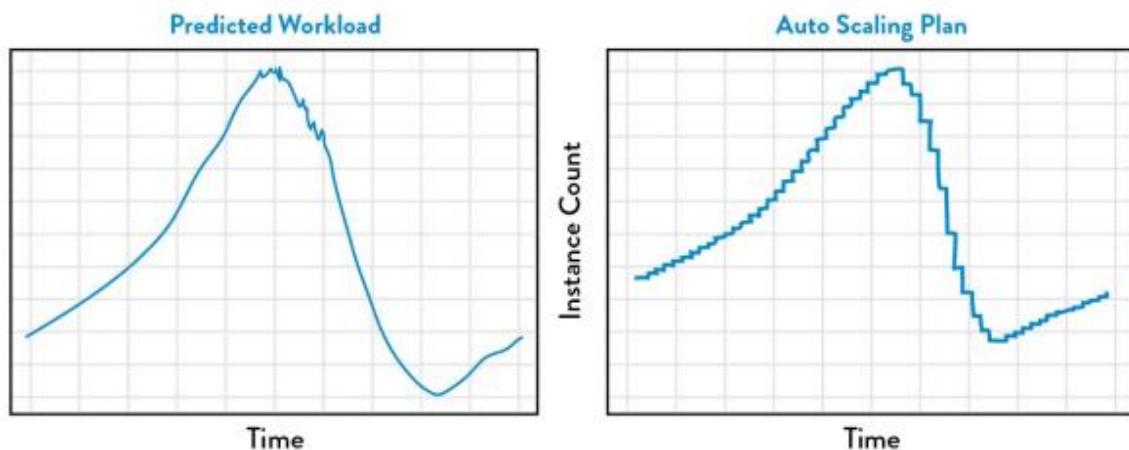


Figure 33: Netflix Scryer forecasting customer traffic and the resulting AWS schedule of compute resources (Source: Jacobson, Yuan, Joshi, “Scryer: Netflix’s Predictive Auto Scaling Engine.”)

Seulement quelques mois après avoir utilisé Scryer en production, Netflix a considérablement amélioré l’expérience de visionnage de ses clients, amélioré la disponibilité du service et réduit les coûts d’Amazon EC2.

Utilisation des techniques de détection d'anomalies

Lorsque nos données n'ont pas de distribution gaussienne, nous pouvons encore trouver des variations notables en utilisant une variété de méthodes. Ces techniques sont largement catégorisées comme détection d'anomalies, souvent définie comme « la recherche d'éléments ou d'événements qui ne se conforment pas à un schéma attendu ». Certaines de ces capacités peuvent être trouvées dans nos outils de surveillance, tandis que d'autres peuvent nécessiter l'aide de personnes ayant des compétences en statistiques.

Tarun Reddy, vice-président du développement et des opérations chez Rally Software, préconise activement cette collaboration entre les opérations et les statistiques, en observant : « Pour mieux assurer la qualité du service, nous mettons toutes nos métriques de production dans Tableau, un logiciel d'analyse statistique. Nous avons même un ingénieur Ops formé en statistiques qui écrit du code R (un autre package statistique) - cet ingénieur a son propre

backlog, rempli de demandes d'autres équipes au sein de l'entreprise qui veulent trouver des variations de plus en plus tôt, avant qu'elles ne causent une variation encore plus grande qui pourrait affecter nos clients. »

Une des techniques statistiques que nous pouvons utiliser est appelée lissage, qui est particulièrement appropriée si nos données sont une série temporelle, ce qui signifie que chaque point de données a un horodatage (par exemple, des événements de téléchargement, des événements de transaction complétée, etc.). Le lissage implique souvent l'utilisation de moyennes mobiles (ou moyennes glissantes), qui transforment nos données en moyennant chaque point avec toutes les autres données dans notre fenêtre glissante. Cela a pour effet de lisser les fluctuations à court terme et de mettre en évidence les tendances ou cycles à plus long terme.

Un exemple de cet effet de lissage est montré dans la figure 34. La ligne noire représente les données brutes, tandis que la ligne bleue indique la moyenne mobile sur trente jours (c'est-à-dire la moyenne des trente jours précédents).

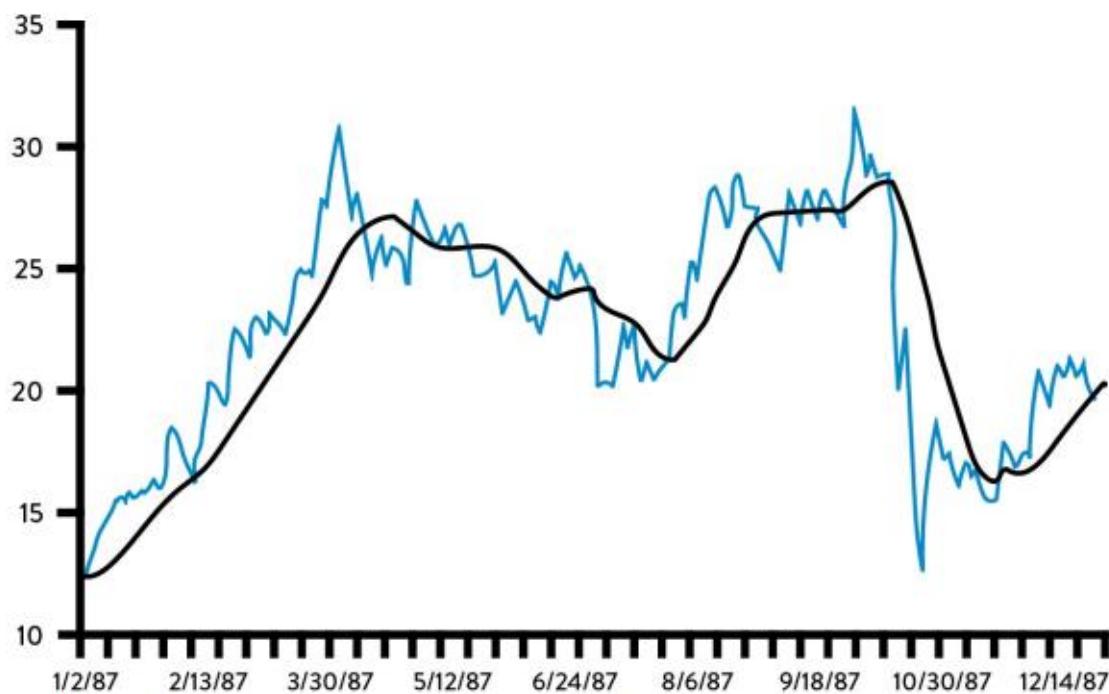


Figure 34: Autodesk share price and thirty day moving average filter (Source: Jacobson, Yuan, Joshi, "Scryer: Netflix's Predictive Auto Scaling Engine. ")

Il existe des techniques de filtrage plus exotiques, telles que les transformations de Fourier rapides, largement utilisées en traitement d'image, et le test de Kolmogorov-Smirnov (utilisé dans Graphite et Grafana), souvent employé pour trouver des similarités ou des différences dans des données métriques périodiques ou saisonnières.

Nous pouvons nous attendre à ce qu'un pourcentage élevé de télémétrie concernant les données des utilisateurs présente des similarités périodiques ou saisonnières - le trafic web, les transactions de vente au détail, le visionnage de films, et de nombreux autres comportements des utilisateurs ont des patterns quotidiens, hebdomadaires et annuels très réguliers et étonnamment prévisibles. Cela nous permet de détecter des situations qui diffèrent des normes historiques, comme lorsque notre taux de transactions de commandes un mardi après-midi chute à 50 % de nos normes hebdomadaires. En raison de l'utilité de ces techniques dans la prévision, il se peut que nous trouvions des personnes dans les départements de Marketing ou de Business Intelligence ayant les connaissances et compétences nécessaires pour analyser ces données. Il pourrait être bénéfique de les chercher et d'explorer une collaboration afin d'identifier des problèmes communs et d'utiliser une détection améliorée des anomalies et des prédictions d'incidents pour les résoudre.

Étude de cas : Détection avancée des anomalies (2014)

Lors du Monitorama en 2014, le Dr Toufic Boubez a décrit la puissance des techniques de détection des anomalies, mettant spécifiquement en avant l'efficacité du test de Kolmogorov-Smirnov, une technique couramment utilisée en statistique pour déterminer si deux ensembles de données diffèrent de manière significative et présente dans les outils populaires Graphite et Grafana.

Le but de présenter cette étude de cas ici n'est pas d'en faire un tutoriel, mais de démontrer comment une classe de techniques statistiques peut être utilisée dans notre travail, ainsi que comment elle est probablement utilisée dans nos organisations pour des applications complètement différentes.

La Figure 35 montre le nombre de transactions par minute sur un site de commerce électronique. Remarquez la périodicité hebdomadaire du graphique, avec une baisse du volume des transactions le week-end. Par une inspection visuelle, nous pouvons constater qu'il se passe quelque chose d'étrange lors de la quatrième semaine, où le volume normal des transactions ne revient pas à des niveaux normaux le lundi. Cela suggère un événement que nous devrions investiguer.

Alerting:

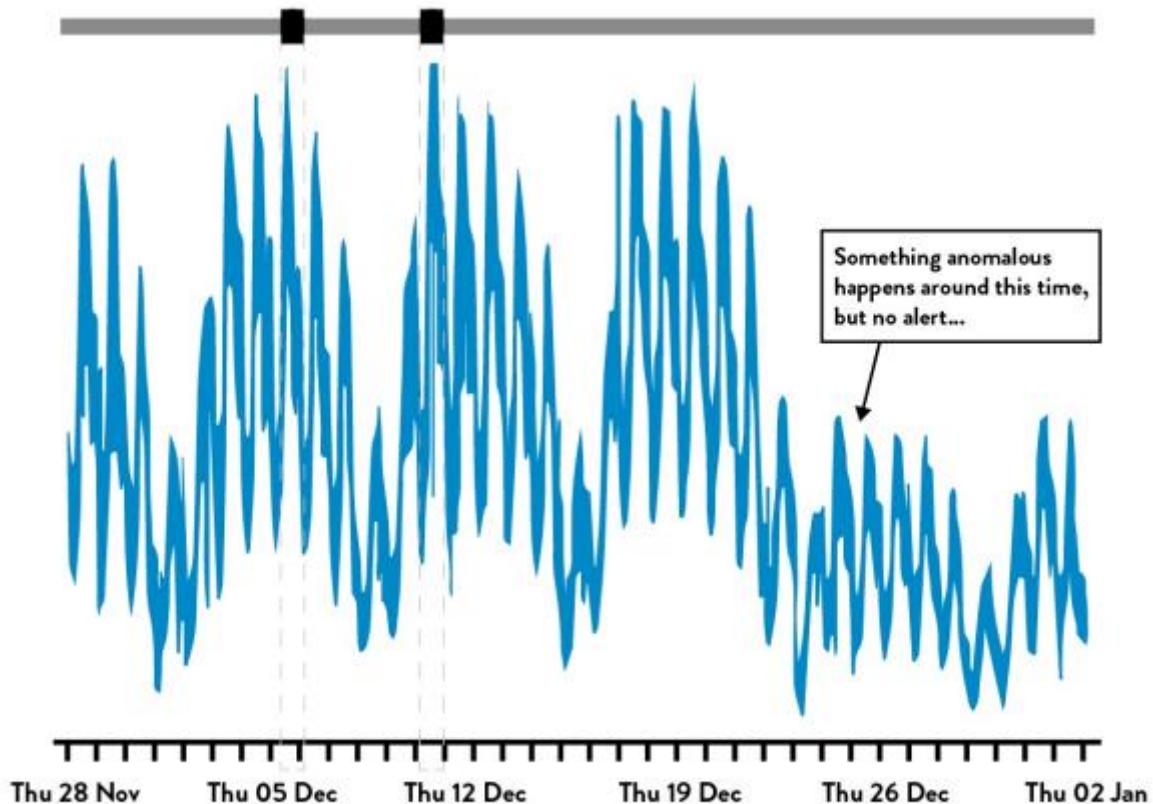


Figure 35: Transaction volume: under-alerting using “3 standard deviation” rule
(Source: Dr. Toufic Boubez, “Simple math for anomaly detection.”)

Utiliser la règle des trois écarts-types ne nous alerterait que deux fois, ce qui manquerait la baisse critique du volume des transactions le lundi. Idéalement, nous voudrions également être alertés lorsque les données s'écartent de notre modèle attendu pour le lundi.

"Même dire 'Kolmogorov-Smirnov' est une excellente manière d'impressionner tout le monde", plaisante le Dr Boubez. "Mais ce que les ingénieurs Ops devraient dire aux statisticiens, c'est que ce type de techniques non paramétriques est idéal pour les données opérationnelles, car elles ne font aucune supposition sur la normalité ou toute autre distribution de probabilité, ce qui est crucial pour comprendre ce qui se passe dans nos systèmes très complexes. Ces techniques comparent deux distributions de probabilité, ce qui nous permet de comparer des données périodiques ou saisonnières, ce qui nous aide à détecter des variations dans des données qui varient de jour en jour ou de semaine en semaine."

La Figure 36, à la page suivante, montre le même jeu de données avec l'application du filtre K-S, la troisième zone mettant en évidence le lundi anormal où le volume des transactions n'est pas revenu à des niveaux normaux. Cela nous aurait alertés d'un problème dans notre système qui aurait été pratiquement impossible à détecter par inspection visuelle ou en utilisant des écarts-

types. Dans ce scénario, cette détection précoce aurait pu éviter un événement impactant pour les clients, tout en nous permettant de mieux atteindre nos objectifs organisationnels.

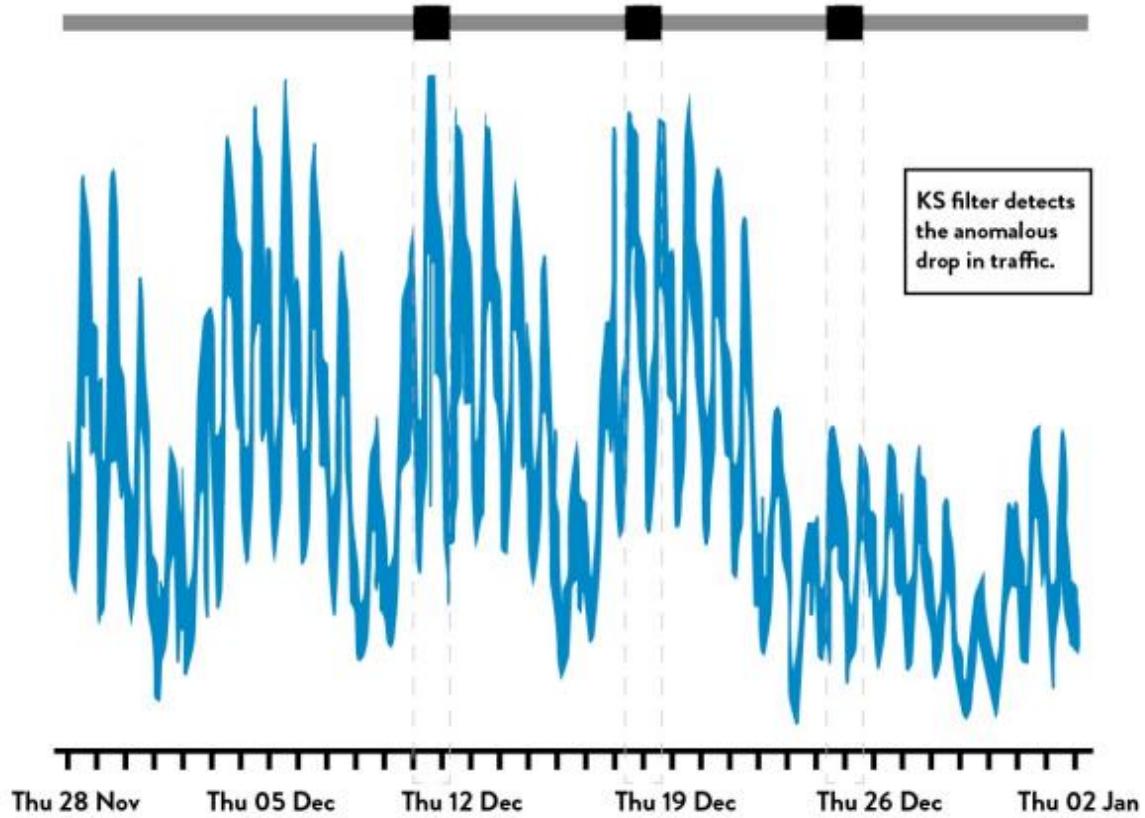


Figure 36: Transaction volume: using Kolmogorov-Smirnov test to alert on

Conclusion

Dans ce chapitre, nous avons exploré plusieurs techniques statistiques différentes qui peuvent être utilisées pour analyser notre télémétrie de production afin de détecter et résoudre les problèmes plus tôt que jamais, souvent lorsqu'ils sont encore mineurs et bien avant qu'ils ne provoquent des résultats catastrophiques. Cela nous permet de repérer des signaux de défaillance de plus en plus faibles sur lesquels nous pouvons agir, créant ainsi un système de travail de plus en plus sûr et renforçant notre capacité à atteindre nos objectifs.

Des études de cas spécifiques ont été présentées, notamment la manière dont Netflix a utilisé ces techniques pour retirer de manière proactive des serveurs de calcul de la production et mettre à l'échelle automatiquement leur infrastructure de calcul. Nous avons également discuté de l'utilisation d'une moyenne mobile et du filtre de Kolmogorov-Smirnov, tous deux disponibles dans des outils graphiques de télémétrie populaires.

Dans le prochain chapitre, nous décrirons comment intégrer la télémétrie de production dans le travail quotidien du développement afin de rendre les déploiements plus sûrs et d'améliorer le système dans son ensemble.

Permettre des retours pour que le développement et les opérations puissent déployer le code en toute sécurité

En 2006, Nick Galbreath était VP de l'ingénierie chez Right Media, responsable des départements de développement et d'opérations pour une plateforme de publicité en ligne affichant et diffusant plus de dix milliards d'impressions quotidiennes.

Galbreath a décrit le paysage concurrentiel dans lequel ils évoluaient :

Dans notre secteur, les niveaux d'inventaire publicitaire étaient extrêmement dynamiques, nous devions donc réagir aux conditions du marché en quelques minutes. Cela signifiait que le développement devait pouvoir apporter rapidement des modifications au code et les mettre en production dès que possible, sinon nous perdions face à des concurrents plus rapides. Nous avons constaté que disposer d'un groupe séparé pour les tests, et même pour le déploiement, était tout simplement trop lent. Nous devions intégrer toutes ces fonctions dans un seul groupe, avec des responsabilités et des objectifs partagés. Croyez-le ou non, notre plus grand défi a été d'amener les développeurs à surmonter leur peur de déployer leur propre code !

Il y a une ironie intéressante ici : les développeurs se plaignent souvent que les opérations craignent de déployer du code. Mais dans ce cas, lorsqu'on leur a donné le pouvoir de déployer leur propre code, les développeurs sont devenus tout aussi effrayés de réaliser des déploiements de code.

La peur de déployer du code partagée par les développeurs et les opérations chez Right Media n'est pas inhabituelle. Cependant, Galbreath a observé que fournir des retours plus rapides et plus fréquents aux ingénieurs effectuant les déploiements (qu'ils soient en développement ou en opérations), ainsi que réduire la taille des lots de leur travail, créait de la sécurité puis de la confiance.

Après avoir observé de nombreuses équipes traverser cette transformation, Galbreath décrit leur progression comme suit :

Nous commençons avec personne en développement ou en opérations ne voulant appuyer sur le bouton "déployer le code" que nous avons construit et qui automatise tout le processus de déploiement de code, en raison de la peur paralysante d'être la première personne à potentiellement faire tomber tous les systèmes de production. Finalement, lorsqu'une personne est assez courageuse pour se porter volontaire pour pousser son code en production, inévitablement, en raison de mauvaises hypothèses ou de subtilités de production qui n'ont pas été pleinement appréciées, le premier déploiement en production ne se passe pas bien - et comme nous n'avons pas suffisamment de télémétrie de production, nous ne découvrons les problèmes que lorsque les clients nous le disent.

Pour résoudre le problème, notre équipe corrige de toute urgence le code et le pousse en production, mais cette fois avec plus de télémétrie de production ajoutée à nos applications et à notre environnement. De cette façon, nous pouvons réellement confirmer que notre correction a rétabli le service correctement, et nous pourrons détecter ce type de problème avant qu'un client ne nous le dise la prochaine fois.

Plus tard, d'autres développeurs commencent à pousser leur propre code en production. Et comme nous travaillons dans un système complexe, nous casserons probablement encore quelque chose en production, mais cette fois nous pourrons rapidement voir quelle fonctionnalité a été cassée, et décider rapidement de revenir en arrière ou de corriger en avant, résolvant ainsi le problème. C'est une grande victoire pour toute l'équipe et tout le monde célèbre - nous sommes maintenant sur la bonne voie.

Cependant, l'équipe veut améliorer les résultats de ses déploiements, alors les développeurs obtiennent de manière proactive plus de révisions par les pairs de leurs modifications de code (décris au chapitre 18), et tout le monde aide à écrire de meilleurs tests automatisés pour que nous puissions trouver les erreurs avant le déploiement. Et parce que tout le monde sait maintenant que plus nos modifications en production sont petites, moins nous aurons de problèmes, les développeurs commencent à vérifier des incrémentations de code de plus en plus petits plus fréquemment dans le pipeline de déploiement, s'assurant que leur changement fonctionne correctement en production avant de passer à leur prochaine modification.

Nous déployons maintenant du code plus fréquemment que jamais, et la stabilité du service est meilleure que jamais aussi. Nous avons redécouvert que le secret d'un flux continu et fluide réside dans la réalisation de petits changements fréquents que tout le monde peut inspecter et comprendre facilement.

Galbreath observe que la progression ci-dessus profite à tout le monde, y compris au développement, aux opérations et à la sécurité de l'information. "En tant que personne également responsable de la sécurité, il est rassurant de savoir que nous pouvons déployer rapidement des corrections en production, car des changements sont déployés en production tout au long de la journée. De plus, il m'épate toujours de voir à quel point chaque ingénieur s'intéresse à la sécurité lorsque vous trouvez des problèmes dans leur code dont ils sont responsables et qu'ils peuvent corriger rapidement eux-mêmes."

L'histoire de Right Media montre qu'il ne suffit pas de simplement automatiser le processus de déploiement - nous devons également intégrer la surveillance de la télémétrie de production dans notre travail de déploiement, ainsi qu'établir les normes culturelles selon lesquelles tout le monde est également responsable de la santé de l'ensemble de la chaîne de valeur.

Dans ce chapitre, nous créons les mécanismes de retour qui nous permettent d'améliorer la santé de la chaîne de valeur à chaque étape du cycle de vie du service, depuis la conception du produit jusqu'au développement et au déploiement, en passant par l'exploitation et finalement la mise hors service. En faisant cela, nous nous assurons que nos services sont "prêts pour la production", même aux premières étapes du projet, ainsi que d'intégrer les enseignements de chaque version et problème de production dans notre travail futur, ce qui se traduit par une meilleure sécurité et productivité pour tous.

Utiliser la télémétrie pour rendre les développements plus sûrs

Dans cette étape, nous nous assurons que nous surveillons activement notre télémétrie de production lorsque quelqu'un effectue un déploiement en production, comme illustré dans l'histoire de Right Media. Cela permet à quiconque effectue le déploiement, qu'il soit en développement ou en opérations, de déterminer rapidement si les fonctionnalités fonctionnent comme prévu après la mise en production de la nouvelle version. Après tout, nous ne devrions jamais considérer notre déploiement de code ou notre changement de production comme terminé tant qu'il ne fonctionne pas comme prévu dans l'environnement de production.

Nous faisons cela en surveillant activement les métriques associées à notre fonctionnalité pendant notre déploiement pour nous assurer que nous n'avons pas involontairement cassé notre service - ou pire, que nous n'avons pas cassé un autre service. Si notre changement casse ou altère une fonctionnalité, nous travaillons rapidement pour rétablir le service, en faisant appel à qui que ce soit d'autre nécessaire pour diagnostiquer et résoudre le problème.

Comme décrit dans la Partie III, notre objectif est de détecter les erreurs dans notre pipeline de déploiement avant qu'elles n'atteignent la production. Cependant, il y aura toujours des erreurs que nous ne détectons pas, et nous comptons sur la télémétrie de production pour rétablir rapidement le service. Nous pouvons choisir de désactiver les fonctionnalités cassées avec des bascules de fonctionnalités (ce qui est souvent l'option la plus simple et la moins risquée car cela n'implique aucun déploiement en production), ou de corriger en avant (c'est-à-dire apporter des modifications au code pour corriger le défaut, qui sont ensuite poussées en production via le pipeline de déploiement), ou de revenir en arrière (par exemple, revenir à la version précédente en utilisant des bascules de fonctionnalités ou en retirant les serveurs cassés de la rotation en utilisant les modèles de déploiement blue-green ou canary, etc.)

Bien que la correction en avant puisse souvent être dangereuse, elle peut être extrêmement sûre lorsque nous avons des tests automatisés et des processus de déploiement rapides, et une télémétrie suffisante qui nous permet de confirmer rapidement si tout fonctionne correctement en production.

La Figure 37 montre un déploiement de changement de code PHP chez Etsy qui a généré un pic dans les avertissements d'exécution PHP - dans ce cas, le développeur a rapidement remarqué le problème en quelques minutes, et a généré une correction et l'a déployée en production, résolvant le problème en moins de dix minutes.

Étant donné que les déploiements en production sont l'une des principales causes de problèmes en production, chaque événement de déploiement et de changement est superposé sur nos graphiques de métriques pour que tout le monde dans la chaîne de valeur soit conscient des activités pertinentes, permettant une meilleure communication et coordination, ainsi qu'une détection et une récupération plus rapides.

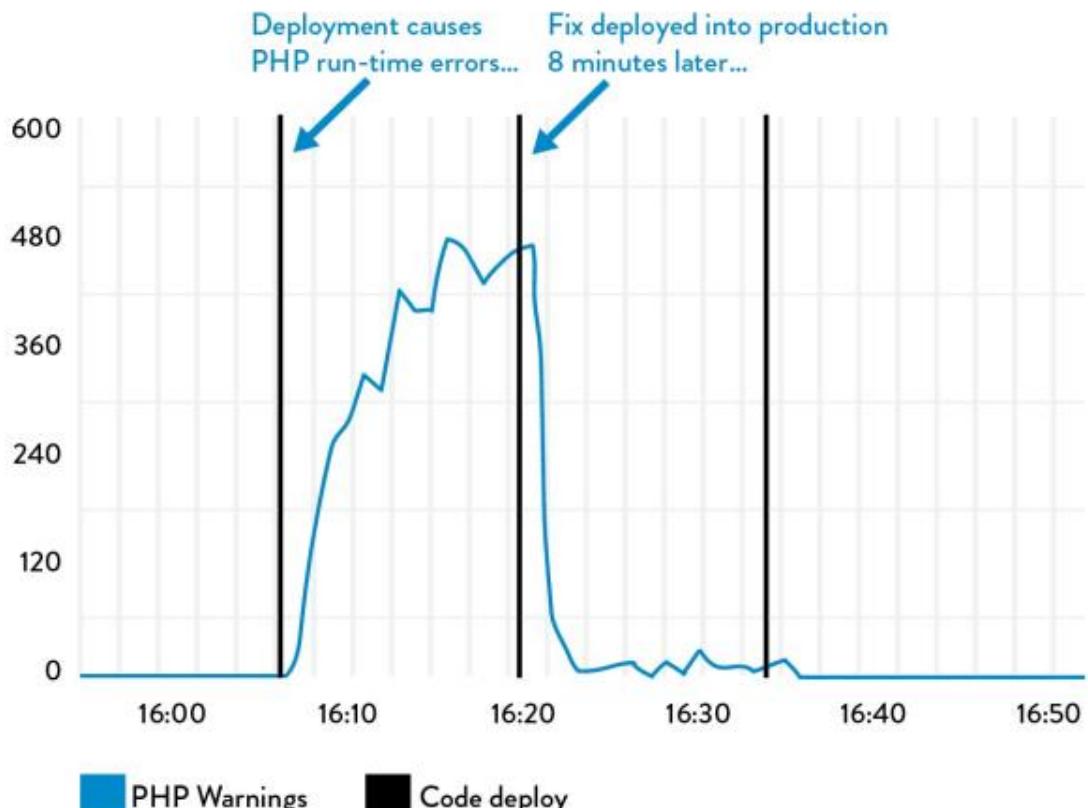


Figure 37: Deployment to Etsy.com causes PHP runtime warnings and is quickly fixed

(Source: Mike Brittain, "Tracking Every Release. ")

Partager les tâches de rotation de téléavertisseur (Pager) entre Dev et Ops

Même lorsque nos déploiements et nos mises en production se déroulent parfaitement, dans tout service complexe, nous aurons toujours des problèmes inattendus, tels que des incidents et des pannes qui se produisent à des moments inopportun (tous les soirs à 2 heures du

matin). Si ces problèmes ne sont pas résolus, ils peuvent causer des problèmes récurrents et des souffrances pour les ingénieurs Ops en aval, surtout lorsque ces problèmes ne sont pas rendus visibles pour les ingénieurs en amont responsables de leur création.

Même si le problème résulte en un défaut assigné à l'équipe de fonctionnalité, il peut être priorisé en dessous de la livraison de nouvelles fonctionnalités. Le problème peut persister pendant des semaines, des mois, voire des années, causant un chaos et une perturbation constants pour les opérations. C'est un exemple de la manière dont les centres de travail en amont peuvent s'optimiser localement mais dégrader en réalité la performance de l'ensemble de la chaîne de valeur.

Pour éviter cela, nous ferons en sorte que tout le monde dans la chaîne de valeur partage les responsabilités en aval de gestion des incidents opérationnels. Nous pouvons y parvenir en mettant les développeurs, les responsables du développement et les architectes en rotation de télémétrie, comme Pedro Canahuati, directeur de l'ingénierie de production chez Facebook, l'a fait en 2009. Cela garantit que tout le monde dans la chaîne de valeur reçoit des retours directs sur les décisions architecturales et de codage qu'ils prennent en amont.

En faisant cela, les opérations ne luttent plus seules et isolées avec les problèmes de production liés au code; au lieu de cela, tout le monde aide à trouver le bon équilibre entre la correction des défauts de production et le développement de nouvelles fonctionnalités, peu importe où nous nous trouvons dans la chaîne de valeur. Comme l'a observé Patrick Lightbody, SVP de la gestion des produits chez New Relic, en 2011, "Nous avons constaté que lorsque nous réveillions les développeurs à 2 heures du matin, les défauts étaient corrigés plus rapidement que jamais."

Un effet secondaire de cette pratique est qu'elle aide la direction du développement à voir que les objectifs commerciaux ne sont pas atteints simplement parce que des fonctionnalités ont été marquées comme "terminées". Au lieu de cela, la fonctionnalité n'est terminée que lorsqu'elle fonctionne comme prévu en production, sans causer de surcharges ou de travail imprévu excessif pour le développement ou les opérations.

Cette pratique est également applicable aux équipes orientées marché, responsables à la fois du développement de la fonctionnalité et de son fonctionnement en production, ainsi qu'aux équipes fonctionnelles. Comme l'a observé Arup Chakrabarti, responsable de l'ingénierie des opérations chez PagerDuty, lors d'une présentation en 2014, "Il devient de moins en moins courant pour les entreprises d'avoir des équipes de garde dédiées; au lieu de cela, tous ceux qui touchent au code et aux environnements de production sont censés être joignables en cas de panne."

Peu importe comment nous avons organisé nos équipes, les principes sous-jacents restent les mêmes: lorsque les développeurs reçoivent des retours sur la performance de leurs applications en production, y compris en les corrigent lorsqu'elles tombent en panne, ils se rapprochent des clients, ce qui crée une adhésion dont tout le monde dans la chaîne de valeur bénéficie.

Faire suivre le travail en aval par les développeurs

L'une des techniques les plus puissantes en conception d'interaction et d'expérience utilisateur (UX) est l'enquête contextuelle. C'est lorsque l'équipe produit observe un client utiliser l'application dans son environnement naturel, souvent à son bureau. Cela permet souvent de découvrir des façons étonnantes dont les clients luttent avec l'application, comme le fait de nécessiter des dizaines de clics pour effectuer des tâches simples dans leur travail quotidien, de copier et coller du texte de plusieurs écrans, ou de prendre des notes sur papier. Toutes ces situations sont des exemples de comportements compensatoires et de contournements pour des problèmes de convivialité.

La réaction la plus courante des développeurs après avoir participé à une observation client est la consternation, déclarant souvent "à quel point il était horrible de voir les nombreuses façons dont nous infligeons de la douleur à nos clients". Ces observations des clients aboutissent presque toujours à un apprentissage significatif et à un désir fervent d'améliorer la situation pour le client.

Notre objectif est d'utiliser cette même technique pour observer comment notre travail affecte nos clients internes. Les développeurs devraient suivre leur travail en aval, pour voir comment les centres de travail en aval doivent interagir avec leur produit pour le faire fonctionner en production.

Les développeurs veulent suivre leur travail en aval - en voyant les difficultés des clients de première main, ils prennent de meilleures décisions et mieux informées dans leur travail quotidien.

En faisant cela, nous créons des retours sur les aspects non fonctionnels de notre code - tous les éléments qui ne sont pas liés à la fonctionnalité orientée client - et identifions les moyens d'améliorer la déployabilité, la gérabilité, l'exploitabilité, etc.

L'observation UX a souvent un impact puissant sur les observateurs. En décrivant sa première observation client, Gene Kim, fondateur et CTO de Tripwire pendant treize ans et co-auteur de ce livre, a déclaré:

"L'un des pires moments de ma carrière professionnelle a été en 2006 lorsque j'ai passé toute une matinée à regarder un de nos clients utiliser notre produit. Je le regardais effectuer une opération que nous attendions que les clients fassent chaque semaine, et, à notre horreur extrême, nous avons découvert que cela nécessitait soixante-trois clics. Cette personne n'arrêtait pas de s'excuser, en disant des choses comme, 'Désolé, il y a probablement une meilleure façon de faire cela.'

Malheureusement, il n'y avait pas de meilleure façon de faire cette opération. Un autre client a décrit comment la configuration initiale du produit prenait 1 300 étapes. Soudain, j'ai compris pourquoi la tâche de gestion de notre produit était toujours assignée au nouvel ingénieur de l'équipe - personne ne voulait le travail de faire fonctionner notre produit.

C'était l'une des raisons pour lesquelles j'ai aidé à créer la pratique UX dans ma société, pour expier la douleur que nous infligions à nos clients."

L'observation UX permet de créer de la qualité à la source et entraîne une bien plus grande empathie pour les membres de l'équipe dans la chaîne de valeur. Idéalement, l'observation UX nous aide à créer des exigences non fonctionnelles codifiées à ajouter à notre backlog partagé de travail, nous permettant finalement de les intégrer de manière proactive dans chaque service que nous construisons, ce qui est une partie importante de la création d'une culture de travail DevOps.

Faire en sorte que les développeurs gèrent initialement eux-mêmes leur service en production

Même lorsque les développeurs écrivent et exécutent leur code dans des environnements similaires à la production dans leur travail quotidien, les opérations peuvent encore rencontrer des déploiements catastrophiques en production car c'est la première fois que nous voyons réellement comment notre code se comporte lors d'un déploiement et dans des conditions réelles de production. Ce résultat se produit parce que les apprentissages opérationnels surviennent souvent trop tard dans le cycle de vie du logiciel.

Si ce problème n'est pas résolu, il en résulte souvent un logiciel de production difficile à exploiter. Comme l'a dit un ingénieur Ops anonyme : « Dans notre groupe, la plupart des administrateurs système ne duraient que six mois. Les choses se cassaient toujours en production, les heures étaient insensées et les déploiements d'applications étaient incroyablement douloureux - le pire était de coupler les clusters de serveurs d'applications, ce qui nous prenait six heures. À chaque instant, nous avions tous l'impression que les développeurs nous détestaient personnellement. »

Cela peut être le résultat d'un manque d'ingénieurs Ops pour soutenir toutes les équipes produit et les services que nous avons déjà en production, ce qui peut arriver dans les équipes fonctionnelles et orientées marché.

Une contre-mesure potentielle consiste à faire ce que fait Google, à savoir que les groupes de développement gèrent eux-mêmes leurs services en production avant qu'ils ne soient éligibles à une gestion par un groupe Ops centralisé. En rendant les développeurs responsables du déploiement et du support en production, nous sommes beaucoup plus susceptibles d'avoir une transition fluide vers les opérations.

Pour éviter que des services autogérés problématiques ne passent en production et ne créent des risques organisationnels, nous pouvons définir des exigences de lancement qui doivent être respectées pour que les services interagissent avec de vrais clients et soient exposés à un trafic réel en production. De plus, pour aider les équipes produit, les ingénieurs Ops devraient agir en tant que consultants pour les aider à rendre leurs services prêts pour la production.

En créant des directives de lancement, nous aidons à garantir que chaque équipe produit bénéficie de l'expérience cumulative et collective de toute l'organisation, en particulier des opérations. Les directives et exigences de lancement incluront probablement les éléments suivants :

- Nombre et gravité des défauts : L'application fonctionne-t-elle réellement comme prévu ?
- Type/fréquence des alertes de pager : L'application génère-t-elle un nombre insupportable d'alertes en production ?
- Couverture de la surveillance : La couverture de la surveillance est-elle suffisante pour restaurer le service en cas de problème ?
- Architecture du système : Le service est-il suffisamment découplé pour supporter un taux élevé de modifications et de déploiements en production ?
- Processus de déploiement : Existe-t-il un processus prévisible, déterministe et suffisamment automatisé pour déployer du code en production ?
- Hygiène de production : Y a-t-il suffisamment de bonnes pratiques de production pour permettre à n'importe qui de gérer le support en production ?

Superficiellement, ces exigences peuvent sembler similaires aux listes de contrôle traditionnelles de production que nous avons utilisées dans le passé. Cependant, les différences clés sont que nous exigeons une surveillance efficace, des déploiements fiables et déterministes, et une architecture qui supporte des déploiements rapides et fréquents.

Si des lacunes sont constatées lors de l'examen, l'ingénieur Ops assigné devrait aider l'équipe de fonctionnalité à résoudre les problèmes ou même aider à réingénier le service si nécessaire, afin qu'il puisse être facilement déployé et géré en production.

À ce stade, nous voudrons peut-être également savoir si ce service est soumis à des objectifs de conformité réglementaire ou s'il est susceptible de l'être à l'avenir :

- Le service génère-t-il une quantité significative de revenus ? (Par exemple, s'il représente plus de 5 % des revenus totaux d'une société cotée en bourse aux États-Unis, il s'agit d'un « compte significatif » et il est soumis à la conformité avec la section 404 de la loi Sarbanes-Oxley de 2002 [SOX].)
- Le service a-t-il un trafic utilisateur élevé ou des coûts élevés en cas de panne/dégradation ? (c'est-à-dire que les problèmes opérationnels risquent de créer des risques de disponibilité ou de réputation ?)
- Le service stocke-t-il des informations sensibles, telles que des numéros de carte de crédit ou des informations personnellement identifiables, telles que des numéros de sécurité sociale ou des dossiers de soins aux patients ? Existe-t-il d'autres problèmes de sécurité pouvant créer des risques réglementaires, contractuels, de confidentialité ou de réputation ?
- Le service a-t-il d'autres exigences de conformité réglementaire ou contractuelle associées, telles que les réglementations américaines sur l'exportation, PCI-DSS, HIPAA, etc. ?

Ces informations aident à garantir que nous gérons efficacement non seulement les risques techniques associés à ce service, mais également tous les risques en matière de sécurité et de conformité. Elles fournissent également des informations essentielles sur la conception de l'environnement de contrôle de production.

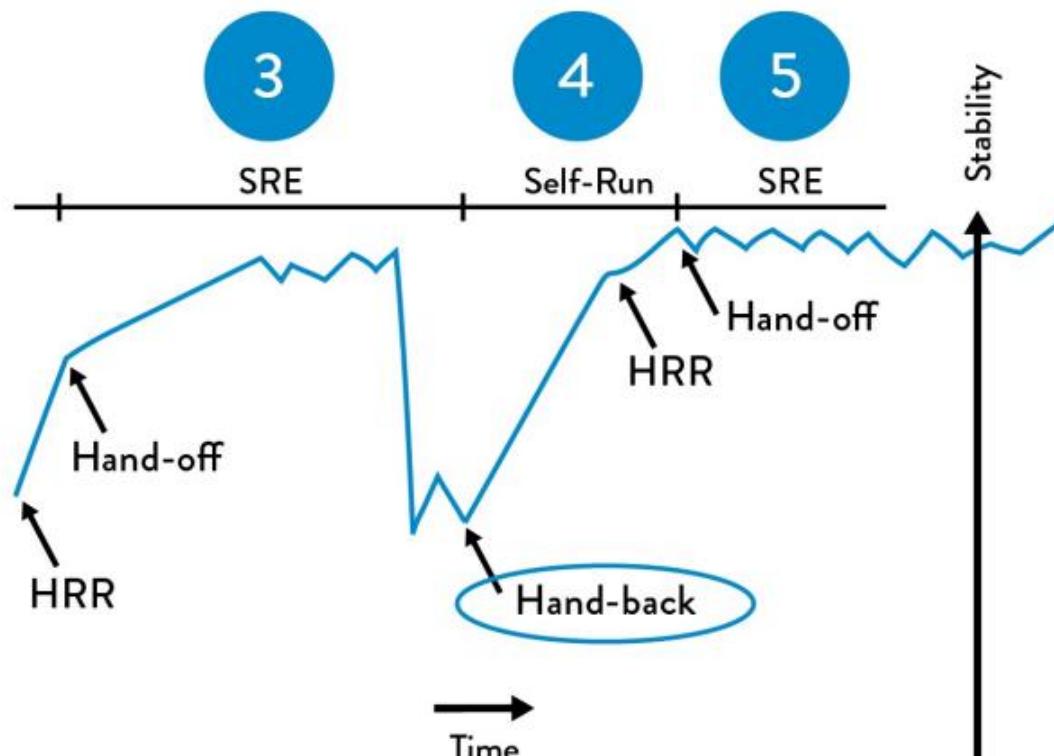


Figure 38: The “Service Handback” at Google (Source: “SRE@Google: Thousands of DevOps Since 2004,” YouTube video, 45:57, posted by USENIX, January 12, 2012, <https://www.youtube.com/watch?v=iIuTnhdTzK0>.)

En intégrant les exigences d'opérabilité dès les premières étapes du processus de développement et en permettant à l'équipe de développement de gérer initialement leurs propres applications et services, le processus de transition des nouveaux services en production devient plus fluide, plus facile et plus prévisible à accomplir. Cependant, pour les services déjà en production, nous avons besoin d'un mécanisme différent pour s'assurer que les opérations ne se retrouvent jamais coincées avec un service insupportable en production. Cela est particulièrement pertinent pour les organisations d'opérations orientées fonctionnellement.

À cette étape, nous pouvons créer un mécanisme de renvoi de service – en d'autres termes, lorsque qu'un service de production devient suffisamment fragile, les opérations ont la capacité de renvoyer la responsabilité du support en production au développement. Lorsque qu'un service retourne à un état géré par les développeurs, le rôle des opérations passe du support en production à la consultation, aidant l'équipe à rendre le service prêt pour la production.

Ce mécanisme sert de soupape de sécurité, garantissant que les opérations ne se retrouvent jamais dans une situation où elles sont piégées à gérer un service fragile tandis qu'une dette technique croissante les ensevelit et amplifie un problème local en un problème global. Ce mécanisme aide également à s'assurer que les opérations ont suffisamment de capacité pour travailler sur des projets d'amélioration et de prévention. Le renvoi reste une pratique de longue date chez Google et est peut-être l'une des meilleures démonstrations du respect mutuel entre les ingénieurs Dev et Ops. En faisant cela, le développement est capable de générer rapidement de nouveaux services, les ingénieurs Ops rejoignant l'équipe lorsque les services deviennent stratégiquement importants pour l'entreprise et, dans de rares cas, les renvoyant lorsqu'ils deviennent trop difficiles à gérer en production.

La cas d'étude suivante sur l'ingénierie de fiabilité de site chez Google décrit comment les processus de révision de préparation au lancement et de préparation au transfert ont évolué et les avantages qui en ont résulté.

Étude de cas - La révision de préparation au lancement et au transfert chez Google (2010)

L'un des nombreux faits surprenants à propos de Google est qu'ils ont une orientation fonctionnelle pour leurs ingénieurs Ops, appelés « ingénieurs de fiabilité de site » (SRE), un terme inventé par Ben Treynor Sloss en 2004. Cette année-là, Treynor Sloss a commencé avec une équipe de sept SREs qui a grandi pour dépasser les 1 200 SREs en 2014. Comme l'a dit Treynor Sloss, « Si Google tombe en panne, c'est ma faute. » Treynor Sloss a résisté à créer une définition en une seule phrase de ce que sont les SREs, mais il a décrit les SREs comme « ce qui se passe lorsqu'un ingénieur logiciel est chargé de ce qu'on appelait autrefois les opérations. »

Chaque SRE relève de l'organisation de Treynor Sloss pour aider à assurer la cohérence de la qualité du personnel et des recrutements, et ils sont intégrés dans les équipes produit à travers Google (qui fournissent également leur financement). Cependant, les SREs sont encore si rares qu'ils sont assignés uniquement aux équipes produit qui ont la plus grande importance pour l'entreprise ou celles qui doivent se conformer aux exigences réglementaires. De plus, ces services doivent avoir une faible charge opérationnelle. Les produits qui ne répondent pas aux critères nécessaires restent dans un état géré par les développeurs.

Même lorsque de nouveaux produits deviennent suffisamment importants pour l'entreprise pour justifier l'affectation d'un SRE, les développeurs doivent toujours avoir auto-géré leur service en production pendant au moins six mois avant de devenir éligibles pour qu'un SRE soit assigné à l'équipe.

Pour aider à s'assurer que ces équipes produit auto-gérées peuvent toujours bénéficier de l'expérience collective de l'organisation SRE, Google a créé deux ensembles de vérifications de sécurité pour deux étapes critiques de la mise en service de nouveaux services appelées la révision de préparation au lancement et la révision de préparation au transfert (LRR et HRR, respectivement). La LRR doit être réalisée et approuvée avant que tout nouveau service Google soit mis à disposition du public et reçoive du trafic de production en direct, tandis que la HRR est réalisée lorsque le service est transféré à un état géré par les Ops, généralement des mois après la LRR. Les listes de contrôle de la LRR et de la HRR sont similaires, mais la HRR est beaucoup plus stricte et a des normes d'acceptation plus élevées, tandis que la LRR est auto-déclarée par les équipes produit.

Toute équipe produit passant par une LRR ou une HRR se voit attribuer un SRE pour les aider à comprendre les exigences et les aider à les atteindre. Les listes de contrôle de la LRR et de la HRR ont évolué au fil du temps afin que chaque équipe puisse bénéficier des expériences collectives de tous les lancements précédents, qu'ils soient réussis ou non. Tom Limoncelli a noté lors de sa présentation « SRE@Google: Thousands of DevOps Since 2004 » en 2012, « Chaque fois que nous faisons un lancement, nous apprenons quelque chose. Il y aura toujours des personnes moins expérimentées que d'autres lors des lancements et des mises en production. Les listes de contrôle LRR et HRR sont un moyen de créer cette mémoire organisationnelle. »

Exiger des équipes produit qu'elles auto-gèrent leurs propres services en production force le développement à se mettre à la place des Ops, mais guidé par la LRR et la HRR, ce qui non seulement facilite et rend plus prévisible la transition des services, mais aide également à créer de l'empathie entre les centres de travail en amont et en aval.

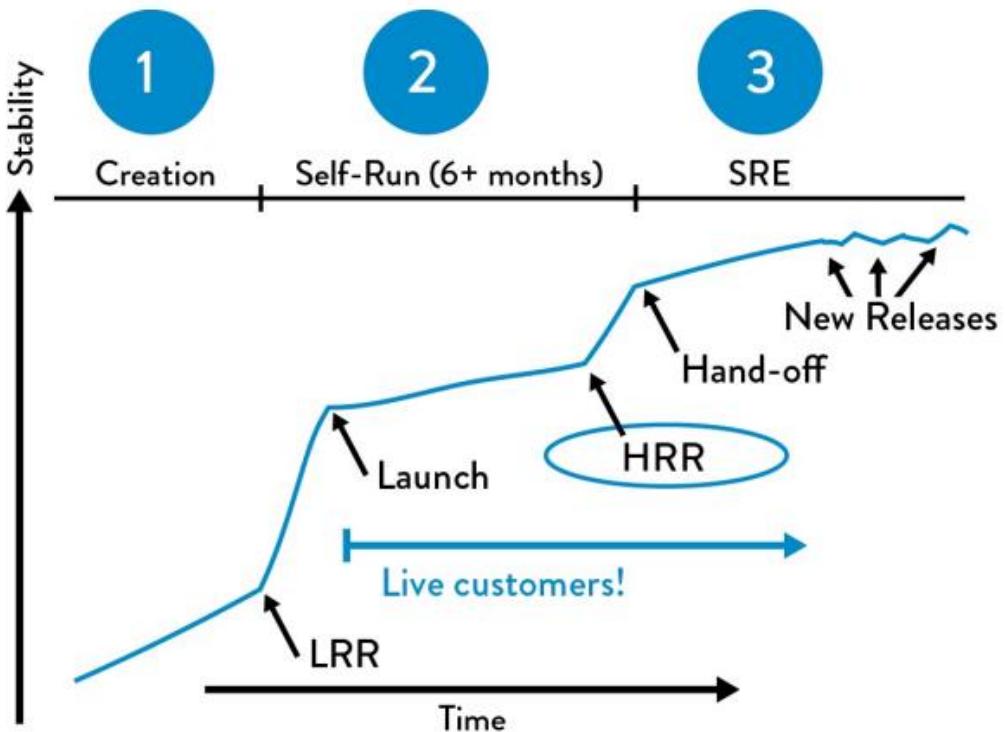


Figure 39: The “Launch readiness review and hand-offs readiness review” at Google
 (Source: “SRE@Google: Thousands of DevOps Since 2004,” YouTube video, 45:57, posted by USENIX, January 12, 2012, <https://www.youtube.com/watch?v=iIuTnhdTzK0>.)

Limoncelli a noté : « Dans le meilleur des cas, les équipes produit utilisent la liste de contrôle LRR comme ligne directrice, travaillant à sa réalisation en parallèle du développement de leur service, et faisant appel aux SREs pour obtenir de l'aide quand elles en ont besoin. »

De plus, Limoncelli a observé : « Les équipes qui obtiennent le plus rapidement l'approbation de production HRR sont celles qui ont travaillé avec les SREs dès les premières étapes de la conception jusqu'au lancement. Et la bonne nouvelle, c'est qu'il est toujours facile de trouver un SRE volontaire pour aider avec votre projet. Chaque SRE voit de la valeur à donner des conseils aux équipes projet dès le début, et sera probablement volontaire pour consacrer quelques heures ou jours à cela. »

La pratique des SREs aidant tôt les équipes produit est une norme culturelle importante continuellement renforcée chez Google. Limoncelli a expliqué : « Aider les équipes produit est un investissement à long terme qui portera ses fruits plusieurs mois plus tard lorsqu'il sera temps de lancer. C'est une forme de 'bonne citoyenneté' et de 'service communautaire' qui est valorisée, elle est régulièrement prise en compte lors de l'évaluation des ingénieurs pour les promotions SRE. »

Conclusion

Dans ce chapitre, nous avons discuté des mécanismes de rétroaction qui nous permettent d'améliorer notre service à chaque étape de notre travail quotidien, que ce soit le déploiement des changements en production, la correction du code lorsque les ingénieurs sont alertés, le suivi du travail des développeurs en aval, la création de spécifications non fonctionnelles qui aident les équipes de développement à écrire un code plus prêt pour la production, voire le renvoi de services problématiques pour qu'ils soient gérés par le développement.

En créant ces boucles de rétroaction, nous rendons les déploiements en production plus sûrs, augmentons la préparation à la production du code créé par le développement, et contribuons à améliorer la relation de travail entre le développement et les opérations en renforçant les objectifs communs, les responsabilités et l'empathie.

Dans le prochain chapitre, nous explorerons comment la télémétrie peut permettre un développement axé sur l'hypothèse et des tests A/B pour mener des expériences qui nous aident à atteindre nos objectifs organisationnels et à réussir sur le marché.

Intégrer le développement piloté par hypothèses et les tests A/B dans notre travail quotidien

Trop souvent dans les projets logiciels, les développeurs travaillent sur des fonctionnalités pendant des mois ou des années, couvrant plusieurs versions, sans jamais confirmer si les résultats commerciaux souhaités sont atteints, comme savoir si une fonctionnalité particulière atteint les résultats désirés ou est utilisée du tout.

Pire encore, même lorsque nous découvrons qu'une fonctionnalité donnée ne produit pas les résultats escomptés, apporter des corrections à la fonctionnalité peut être repoussé par d'autres nouvelles fonctionnalités, assurant que la fonctionnalité sous-performante n'atteindra jamais son objectif commercial prévu. En général, Jez Humble observe que "le moyen le plus inefficace de tester un modèle commercial ou une idée de produit est de construire le produit complet pour voir si la demande prédictive existe réellement".

Avant de construire une fonctionnalité, nous devrions nous poser rigoureusement la question : "Devrions-nous la construire, et pourquoi ?" Nous devrions ensuite réaliser les expériences les moins chères et les plus rapides possibles pour valider à travers la recherche utilisateur si la fonctionnalité prévue atteindra effectivement les résultats souhaités. Nous pouvons utiliser des techniques telles que le développement piloté par hypothèses, les tunnels d'acquisition client et les tests A/B, des concepts que nous explorons tout au long de ce chapitre. Intuit, Inc. fournit un exemple marquant de la façon dont les organisations utilisent ces techniques pour créer des produits que les clients adorent, pour favoriser l'apprentissage organisationnel et pour réussir sur le marché.

Intuit se concentre sur la création de solutions de gestion d'entreprise et financière pour simplifier la vie des petites entreprises, des consommateurs et des professionnels de la comptabilité. En 2012, ils ont réalisé un chiffre d'affaires de 4,5 milliards de dollars avec 8 500 employés, et leurs produits phares incluent QuickBooks, TurboTax, Mint, et jusqu'à récemment, Quicken.

Scott Cook, le fondateur d'Intuit, prône depuis longtemps la création d'une culture de l'innovation, encourageant les équipes à adopter une approche expérimentale du développement de produits et incitant la direction à les soutenir. Comme il l'a dit : "Au lieu de se concentrer sur le vote du patron... l'accent est mis sur le fait de faire agir de vraies personnes dans de vraies expériences, et de baser vos décisions là-dessus." C'est l'apogée d'une approche scientifique du développement de produits.

Cook explique que ce dont nous avons besoin, c'est "d'un système où chaque employé peut mener des expériences rapides à haute vitesse... Dan Maurer dirige notre division grand

public... [qui] gère le site Web TurboTax. Lorsqu'il a pris les commandes, nous avons réalisé environ sept expériences par an."

Il continue : "En installant une culture d'innovation effrénée en 2010, ils réalisent maintenant 165 expériences pendant les trois mois de la saison fiscale [aux États-Unis]. Résultat commercial ? Le taux de conversion du site Web a augmenté de 50 %... Les gens [membres de l'équipe] l'adorent simplement, car maintenant leurs idées peuvent atteindre le marché."

Outre l'effet sur le taux de conversion du site Web, l'un des éléments les plus surprenants de cette histoire est que TurboTax a réalisé des expériences de production pendant leurs périodes de trafic intense. Pendant des décennies, surtout dans le commerce de détail, le risque de pannes impactant les revenus pendant la saison des fêtes était si élevé que nous mettions souvent en place un gel des changements de mi-octobre à mi-janvier. Cependant, en rendant les déploiements et les mises en production rapides et sûrs, l'équipe TurboTax a fait des expérimentations utilisateur en ligne et des changements de production nécessaires une activité à faible risque pouvant être réalisée pendant les périodes de trafic et de génération de revenus les plus élevées.

Cela souligne l'idée que la période où l'expérimentation a la plus grande valeur est pendant les saisons de trafic intense. Si l'équipe TurboTax avait attendu le 16 avril, lendemain de la date limite de dépôt des impôts aux États-Unis, pour mettre en œuvre ces changements, l'entreprise aurait pu perdre de nombreux clients potentiels, voire certains de ses clients existants, au profit de la concurrence.

Plus nous pouvons expérimenter, itérer et intégrer les commentaires dans notre produit ou service rapidement, plus nous pouvons apprendre rapidement et surpasser la concurrence en matière d'expérimentation. Et la vitesse à laquelle nous pouvons intégrer nos commentaires dépend de notre capacité à déployer et à publier des logiciels.

L'exemple d'Intuit montre que l'équipe TurboTax d'Intuit a réussi à tirer parti de cette situation et a remporté la victoire sur le marché en conséquence.

Une brève histoire des tests A/B

Comme le souligne l'histoire de TurboTax d'Intuit, une technique extrêmement puissante de recherche utilisateur consiste à définir le tunnel d'acquisition client et à réaliser des tests A/B. Les techniques de tests A/B ont été pionnières dans le marketing de réponse directe, qui est l'une des deux principales catégories de stratégies marketing. L'autre catégorie est le marketing de masse ou de marque, qui repose souvent sur la diffusion du plus grand nombre possible d'impressions publicitaires pour influencer les décisions d'achat.

Dans les ères précédentes, avant le courriel et les médias sociaux, le marketing de réponse directe consistait à envoyer des milliers de cartes postales ou de dépliants par la poste, et à demander aux prospects d'accepter une offre en appelant un numéro de téléphone, en retournant une carte postale ou en passant une commande.

Dans ces campagnes, des expériences étaient menées pour déterminer quelle offre avait les taux de conversion les plus élevés. Ils expérimentaient en modifiant et en adaptant l'offre, en reformulant l'offre, en variant les styles de rédaction, la conception et la typographie, l'emballage, etc., pour déterminer ce qui était le plus efficace pour générer l'action désirée (par exemple, appeler un numéro de téléphone, commander un produit). Chaque expérience nécessitait souvent une nouvelle conception et une nouvelle impression, l'envoi de milliers d'offres par courrier, et l'attente de semaines pour que les réponses reviennent. Chaque expérience coûtait généralement des dizaines de milliers de dollars par essai et nécessitait des semaines ou des mois pour être complétée. Cependant, malgré les coûts, les tests itératifs étaient rentables s'ils augmentaient significativement les taux de conversion (par exemple, le pourcentage de répondants commandant un produit passant de 3 % à 12 %).

Des cas bien documentés de tests A/B incluent la collecte de fonds de campagne, le marketing Internet et la méthodologie Lean Startup. De manière intéressante, cela a également été utilisé par le gouvernement britannique pour déterminer quelles lettres étaient les plus efficaces pour collecter les impôts dus auprès des citoyens délinquants.

Intégrer les tests A/B dans nos tests de fonctionnalité

La technique A/B la plus couramment utilisée dans la pratique moderne de l'UX implique un site web où les visiteurs sont sélectionnés au hasard pour voir l'une des deux versions d'une page, soit un contrôle (le "A") soit un traitement (le "B"). Sur la base de l'analyse statistique du comportement subséquent de ces deux cohortes d'utilisateurs, nous démontrons s'il existe une différence significative dans les résultats des deux, établissant ainsi un lien de causalité entre le traitement (par exemple, un changement dans une fonctionnalité, un élément de conception, une couleur de fond) et le résultat (par exemple, le taux de conversion, la taille moyenne de la commande).

Par exemple, nous pouvons mener une expérience pour voir si la modification du texte ou de la couleur d'un bouton "acheter" augmente les revenus, ou si le ralentissement du temps de réponse d'un site web (en introduisant un délai artificiel comme traitement) réduit les revenus. Ce type de tests A/B nous permet d'établir une valeur en dollars sur les améliorations de performance.

Parfois, les tests A/B sont également appelés expériences contrôlées en ligne et tests fractionnés. Il est également possible de réaliser des expériences avec plusieurs variables. Cela nous permet de voir comment les variables interagissent, une technique connue sous le nom de test multivarié.

Les résultats des tests A/B sont souvent surprenants. Ronny Kohavi, ingénieur distingué et directeur général du groupe d'analyse et d'expérimentation chez Microsoft, a observé qu'après "évaluation d'expériences bien conçues et exécutées visant à améliorer une métrique clé, seulement environ un tiers ont réussi à améliorer cette métrique clé!" En d'autres termes, deux tiers des fonctionnalités ont soit un impact négligeable, soit empêrent les choses. Kohavi ajoute que toutes ces fonctionnalités étaient initialement considérées comme raisonnables et bonnes, renforçant ainsi la nécessité des tests utilisateur par rapport à l'intuition et aux opinions d'experts.

Les implications des données de Kohavi sont impressionnantes. Si nous ne réalisons pas de recherche utilisateur, il est probable que deux tiers des fonctionnalités que nous développons n'apportent aucune valeur ou ont même un impact négatif sur notre organisation, tout en complexifiant notre base de code, augmentant ainsi nos coûts de maintenance et rendant notre logiciel plus difficile à modifier. De plus, l'effort déployé pour développer ces fonctionnalités se fait souvent au détriment de la livraison de fonctionnalités qui apporteraient de la valeur (c'est-à-dire le coût d'opportunité). Jez Humble plaisantait : "Poussé à l'extrême, l'organisation et les clients auraient été mieux lotis en donnant à toute l'équipe des vacances, au lieu de construire une de ces fonctionnalités sans valeur ajoutée."

Notre contre-mesure est d'intégrer les tests A/B dans la manière dont nous concevons, implémentons, testons et déployons nos fonctionnalités. La réalisation de recherches utilisateurs et d'expériences significatives assure que nos efforts contribuent à atteindre nos objectifs clients et organisationnels, et nous aident à réussir sur le marché.

Intégrer les tests A/B dans notre planification des fonctionnalités

Une fois que nous avons l'infrastructure pour supporter le déploiement et le test des fonctionnalités en A/B, nous devons nous assurer que les responsables de produit considèrent chaque fonctionnalité comme une hypothèse et utilisent nos déploiements en production comme des expériences avec de vrais utilisateurs pour prouver ou réfuter cette hypothèse. La construction des expériences doit être conçue dans le contexte de l'entonnoir global d'acquisition de clients. Barry O'Reilly, co-auteur de "Lean Enterprise: How High Performance Organizations Innovate at Scale", décrit comment formuler des hypothèses dans le développement de fonctionnalités sous la forme suivante :

- Nous croyons qu'augmenter la taille des images d'hôtel sur la page de réservation conduira à une meilleure implication et conversion des clients

- Nous aurons confiance pour procéder lorsque nous verrons une augmentation de 5 % des clients qui consultent les images d'hôtel, puis passent à la réservation dans les quarante-huit heures.

Adopter une approche expérimentale du développement de produit nécessite non seulement de découper le travail en petites unités (stories ou exigences), mais aussi de valider si chaque unité de travail produit les résultats attendus. Si ce n'est pas le cas, nous modifions notre feuille de route de travail avec des chemins alternatifs qui atteindront effectivement ces résultats.

Étude de cas - Doublement de la croissance du revenu grâce à un cycle de mise en œuvre rapide

Expérimentation chez Yahoo! Answers (2010)

Plus nous pouvons itérer rapidement et intégrer les retours dans le produit ou service que nous offrons aux clients, plus vite nous pouvons apprendre et plus grand impact nous pouvons créer. L'impact dramatique que peuvent avoir des temps de cycle plus rapides était évident chez Yahoo! Answers lorsqu'ils sont passés d'une publication toutes les six semaines à plusieurs publications chaque semaine.

En 2009, Jim Stoneham était directeur général du groupe des Communautés Yahoo! qui comprenait Flickr et Answers. Auparavant, il était principalement responsable de Yahoo! Answers, en concurrence avec d'autres entreprises de questions-réponses telles que Quora, Aardvark et Stack Exchange.

À ce moment-là, Answers comptait environ 140 millions de visiteurs mensuels, avec plus de vingt millions d'utilisateurs actifs répondant à des questions dans plus de vingt langues différentes. Cependant, la croissance des utilisateurs et des revenus avait stagné, et les scores d'engagement des utilisateurs étaient en baisse.

Stoneham observe que "Yahoo! Answers était et reste l'un des plus grands jeux sociaux sur Internet ; des dizaines de millions de personnes essaient activement de 'monter en niveau' en fournissant des réponses de qualité aux questions plus rapidement que les autres membres de la communauté. Il y avait de nombreuses opportunités pour ajuster le mécanisme du jeu, les boucles virales et d'autres interactions communautaires. Lorsque vous travaillez avec ces comportements humains, vous devez pouvoir effectuer des itérations et des tests rapides pour voir ce qui fonctionne avec les gens."

Il continue, "Ces [expérimentations] sont les choses que Twitter, Facebook et Zynga faisaient si bien. Ces organisations réalisaient des expériences au moins deux fois par semaine, voire plus. Elles examinaient même les changements apportés avant leurs déploiements, pour s'assurer qu'elles étaient toujours sur la bonne voie. Donc, ici je suis, à la tête du plus grand site de questions-réponses du marché, voulant faire des tests itératifs rapides de fonctionnalités, mais nous ne pouvons pas publier plus rapidement qu'une fois toutes les 4 semaines. En revanche, les autres acteurs du marché avaient une boucle de rétroaction 10 fois plus rapide que la nôtre."

Stoneham a observé que, aussi souvent que les propriétaires de produits et les développeurs parlent de se concentrer sur les métriques, si les expériences ne sont pas réalisées fréquemment (quotidiennement ou hebdomadairement), le travail quotidien se concentre simplement sur la fonctionnalité sur laquelle ils travaillent, plutôt que sur les résultats pour les clients.

Lorsque l'équipe de Yahoo! Answers a pu passer à des déploiements hebdomadaires, puis à plusieurs déploiements par semaine, leur capacité à expérimenter de nouvelles fonctionnalités a augmenté de manière spectaculaire. Leurs réalisations et apprentissages stupéfiant au cours des douze mois suivants d'expérimentation comprenaient une augmentation des visites mensuelles de 72 %, une multiplication par trois de l'engagement des utilisateurs et un doublement de leurs revenus. Pour continuer leur succès, l'équipe s'est concentrée sur l'optimisation des principales métriques suivantes :

- Temps avant la première réponse : À quelle vitesse une réponse a-t-elle été publiée pour une question utilisateur ?
- Temps avant la meilleure réponse : À quelle vitesse la communauté d'utilisateurs a-t-elle attribué une meilleure réponse ?
- Upvotes par réponse : Combien de fois une réponse a-t-elle été upvotée par la communauté d'utilisateurs ?
- Réponses/semaine/personne : Combien de réponses les utilisateurs créaient-ils ?
- Taux de deuxième recherche : À quelle fréquence les visiteurs ont-ils dû effectuer une nouvelle recherche pour obtenir une réponse ? (Plus bas est mieux.)

Stoneham conclut, "C'était exactement l'apprentissage dont nous avions besoin pour réussir sur le marché - et cela a changé bien plus que notre vitesse de fonctionnalités. Nous sommes passés d'une équipe de salariés à une équipe de propriétaires. Lorsque vous avancez à cette vitesse, en examinant les chiffres et les résultats quotidiennement, votre niveau d'investissement change radicalement."

Conclusion

Le succès nécessite non seulement de déployer et publier des logiciels rapidement, mais aussi de surpasser nos concurrents par l'expérimentation. Des techniques telles que le développement piloté par l'hypothèse, la définition et la mesure de notre entonnoir d'acquisition de clients, et les tests A/B nous permettent d'effectuer des expériences utilisateur de manière sûre et efficace, nous permettant de libérer la créativité, l'innovation, et de créer un apprentissage organisationnel. Et, bien que le succès soit important, l'apprentissage organisationnel issu de l'expérimentation donne également aux employés un sentiment d'appartenance aux objectifs commerciaux et à la satisfaction client. Dans le prochain chapitre, nous examinerons et créerons des processus de revue et de coordination pour améliorer la qualité de notre travail actuel.

Créer des processus de révision et de coordination pour améliorer la qualité de notre travail actuel

Dans les chapitres précédents, nous avons créé la télémétrie nécessaire pour identifier et résoudre les problèmes en production et à toutes les étapes de notre pipeline de déploiement, et nous avons mis en place des boucles de rétroaction rapides de la part des clients pour améliorer l'apprentissage organisationnel - un apprentissage qui encourage la prise en charge et la responsabilité de la satisfaction des clients et des performances des fonctionnalités, ce qui nous aide à réussir.

Notre objectif dans ce chapitre est de permettre à Développement et Opérations de réduire le risque des changements en production avant qu'ils ne soient effectués. Traditionnellement, lorsque nous révisons les changements avant le déploiement, nous avons tendance à nous appuyer fortement sur les révisions, les inspections et les approbations juste avant le déploiement. Souvent, ces approbations sont données par des équipes externes qui sont souvent trop éloignées du travail pour prendre des décisions éclairées sur le caractère risqué d'un changement, et le temps nécessaire pour obtenir toutes les approbations nécessaires allonge également nos délais de changement.

Le processus de révision par les pairs chez GitHub est un exemple frappant de la manière dont l'inspection peut améliorer la qualité, sécuriser les déploiements et être intégrée au flux de travail quotidien de chacun. Ils ont fait émerger le processus appelé pull request, l'une des formes de révision par les pairs les plus populaires qui englobe Dev et Ops.

Scott Chacon, CIO et co-fondateur de GitHub, a écrit sur son site web que les pull requests sont le mécanisme qui permet aux ingénieurs d'informer les autres des modifications qu'ils ont poussées vers un dépôt sur GitHub. Une fois qu'une pull request est envoyée, les parties intéressées peuvent examiner l'ensemble des modifications, discuter des modifications potentielles et même pousser des commits supplémentaires si nécessaire. Les ingénieurs soumettant une pull request demanderont souvent un "+1," "+2," ou plus, selon le nombre de révisions dont ils ont besoin, ou "@mentionner" des ingénieurs dont ils souhaitent obtenir des avis.

Chez GitHub, les pull requests sont également le mécanisme utilisé pour déployer le code en production à travers un ensemble de pratiques collectives qu'ils appellent "GitHub Flow" - c'est ainsi que les ingénieurs demandent des révisions de code, recueillent et intègrent des commentaires, et annoncent que le code sera déployé en production (c'est-à-dire sur la branche "master").

```

27      +     opts[:options] [:stripnl] || = false
35      28
36      29     timeout opts.delete(:timeout) || DEFAULT_TIMEOUT do
37      30     begin
38      31       Pygments.highlight(text, opts)

```

2

brianmario repo collab

So what are the defaults here if no encoding or lexer is passed?

Also there's at least one other place where the API is expected to take an :encoding key (not nested under an :options key/hash) -
<https://github.com/github/github/blob/master/app/models/gist.rb#L114>

Only reason I did it that way was to sorta abstract the fact that we're using pygments for colorizing currently (not that we have plans to change that anytime soon...)

Josh repo collab

Alright, I'll push that down to colorize.

Add a line note

Figure 40: Comments and suggestions on a GitHub pull request
 (Source: Scott Chacon, “GitHub Flow,” ScottChacon.com, August 31, 2011,
[http://scottchacon.com/2011/08/31/github-flow.html. \)](http://scottchacon.com/2011/08/31/github-flow.html.)

GitHub Flow se compose de cinq étapes :

- Pour travailler sur quelque chose de nouveau, l'ingénieur crée une branche nommée de manière descriptive à partir de master (par exemple, "new-oauth2-scopes").
- L'ingénieur fait des commits sur cette branche localement, en poussant régulièrement son travail vers la même branche nommée sur le serveur.
- Lorsqu'il a besoin de retours ou d'aide, ou lorsqu'il pense que la branche est prête pour la fusion, il ouvre une pull request.
- Lorsque les révisions souhaitées sont obtenues et que les approbations nécessaires sont reçues, ils fusionnent la branche dans master.
- Une fois que les modifications de code sont fusionnées et poussées vers master, l'ingénieur peut alors fusionner cette branche dans master.

Ces pratiques, qui intègrent la révision et la coordination dans le travail quotidien, ont permis à GitHub de livrer rapidement et de manière fiable des fonctionnalités sur le marché avec une haute qualité et sécurité. Par exemple, en 2012, ils ont réalisé l'incroyable chiffre de 12 602 déploiements. En particulier, le 23 août, après un sommet d'entreprise où de nombreuses idées passionnantes ont été brainstormées et discutées, l'entreprise a eu son jour de déploiement le

plus chargé de l'année, avec 563 builds et 175 déploiements réussis en production, tous rendus possibles grâce au processus de pull request.

Tout au long de ce chapitre, nous intégrerons des pratiques, telles que celles utilisées chez GitHub, pour nous éloigner de la dépendance aux inspections et approbations périodiques, en passant à des révisions par les pairs intégrées, effectuées continuellement dans le cadre de notre travail quotidien. Notre objectif est de garantir que Développement, Opérations et Infosec collaborent en permanence pour que les changements que nous apportons à nos systèmes fonctionnent de manière fiable, sécurisée, sûre et conforme aux attentes.

Les dangers des processus d'approbation des changements

L'échec de Knight Capital est l'une des erreurs de déploiement logiciel les plus marquantes de ces dernières années. Une erreur de déploiement de quinze minutes a entraîné une perte de 440 millions de dollars en transactions, pendant laquelle les équipes d'ingénierie n'ont pas pu désactiver les services de production. Les pertes financières ont mis en péril les opérations de l'entreprise et ont forcé la vente de la société pendant le week-end pour qu'elle puisse continuer à fonctionner sans mettre en danger l'ensemble du système financier.

John Allspaw a observé que lorsque des incidents de grande envergure se produisent, comme l'accident de déploiement de Knight Capital, il existe généralement deux récits contrefactuels pour expliquer pourquoi l'accident s'est produit.

Le premier récit est que l'accident était dû à un échec du contrôle des changements, ce qui semble valable car nous pouvons imaginer une situation où de meilleures pratiques de contrôle des changements auraient pu détecter le risque plus tôt et empêcher le changement d'être mis en production. Et si nous n'avions pas pu l'empêcher, nous aurions pu prendre des mesures pour permettre une détection et une récupération plus rapides.

Le deuxième récit est que l'accident était dû à un échec des tests. Cela semble également valable, avec de meilleures pratiques de test, nous aurions pu identifier le risque plus tôt et annuler le déploiement risqué, ou nous aurions pu au moins prendre des mesures pour permettre une détection et une récupération plus rapides.

La réalité surprenante est que dans les environnements de faible confiance, avec des cultures de commandement et de contrôle, les résultats de ces types de contre-mesures de contrôle des changements et de tests entraînent souvent une probabilité accrue que des problèmes se reproduisent, potentiellement avec des conséquences encore pires.

Gene Kim (co-auteur de ce livre) décrit sa prise de conscience que les contrôles de changement et de test peuvent potentiellement avoir l'effet inverse de celui escompté comme "l'un des moments les plus importants de ma carrière professionnelle. Ce moment 'aha' était le résultat d'une conversation en 2013 avec John Allspaw et Jez Humble à propos de l'accident de Knight Capital, me faisant remettre en question certaines de mes croyances fondamentales formées au cours des dix dernières années, surtout ayant été formé en tant qu'auditeur."

Il continue : "Aussi bouleversant que cela ait été, ce fut aussi un moment très formateur pour moi. Non seulement ils m'ont convaincu qu'ils avaient raison, mais nous avons testé ces croyances dans le rapport State of DevOps de 2014, ce qui a conduit à des découvertes étonnantes qui renforcent que la construction de cultures de haute confiance est probablement le plus grand défi de gestion de cette décennie."

Dangers des « Changements trop contrôlés »

Les contrôles de changement traditionnels peuvent entraîner des résultats inattendus, tels que l'allongement des délais et la réduction de la rapidité et de l'immédiateté des retours du processus de déploiement. Pour comprendre comment cela se produit, examinons les contrôles que nous mettons souvent en place lorsque des échecs de contrôle des changements se produisent :

- Ajouter plus de questions à répondre dans le formulaire de demande de changement
- Exiger plus d'autorisations, comme un niveau supplémentaire d'approbation de la gestion (par exemple, au lieu que seul le VP des Opérations approuve, nous exigeons désormais que le CIO approuve également) ou plus de parties prenantes (par exemple, ingénierie réseau, comités d'examen de l'architecture, etc.)
- Exiger plus de temps de préparation pour les approbations de changement afin que les demandes de changement puissent être correctement évaluées

Ces contrôles ajoutent souvent plus de friction au processus de déploiement en multipliant le nombre d'étapes et d'approbations, et en augmentant la taille des lots et les délais de déploiement, ce qui réduit la probabilité de résultats réussis en production pour Dev et Ops. Ces contrôles réduisent également la rapidité avec laquelle nous obtenons des retours sur notre travail.

L'une des croyances fondamentales du système de production Toyota est que "les personnes les plus proches d'un problème en savent généralement le plus à son sujet." Cela devient plus prononcé à mesure que le travail effectué et le système dans lequel le travail se déroule deviennent plus complexes et dynamiques, comme c'est typiquement le cas dans les flux de valeur DevOps. Dans ces cas, créer des étapes d'approbation par des personnes situées de plus en plus loin du travail peut en fait réduire la probabilité de succès. Comme cela a été prouvé à maintes reprises, plus la distance entre la personne effectuant le travail (c'est-à-dire le changeur) et la personne décidant d'effectuer le travail (c'est-à-dire l'autorisateur du changement) est grande, plus le résultat est mauvais.

Dans le rapport State of DevOps de Puppet Labs de 2014, l'une des principales conclusions était que les organisations performantes se reposaient davantage sur la révision par les pairs et moins sur l'approbation externe des changements. La figure 41 montre que plus les organisations se reposent sur les approbations de changement, plus leurs performances IT sont mauvaises en termes de stabilité (temps moyen de restauration du service et taux d'échec des changements) et de débit (délais de déploiement, fréquence des déploiements).

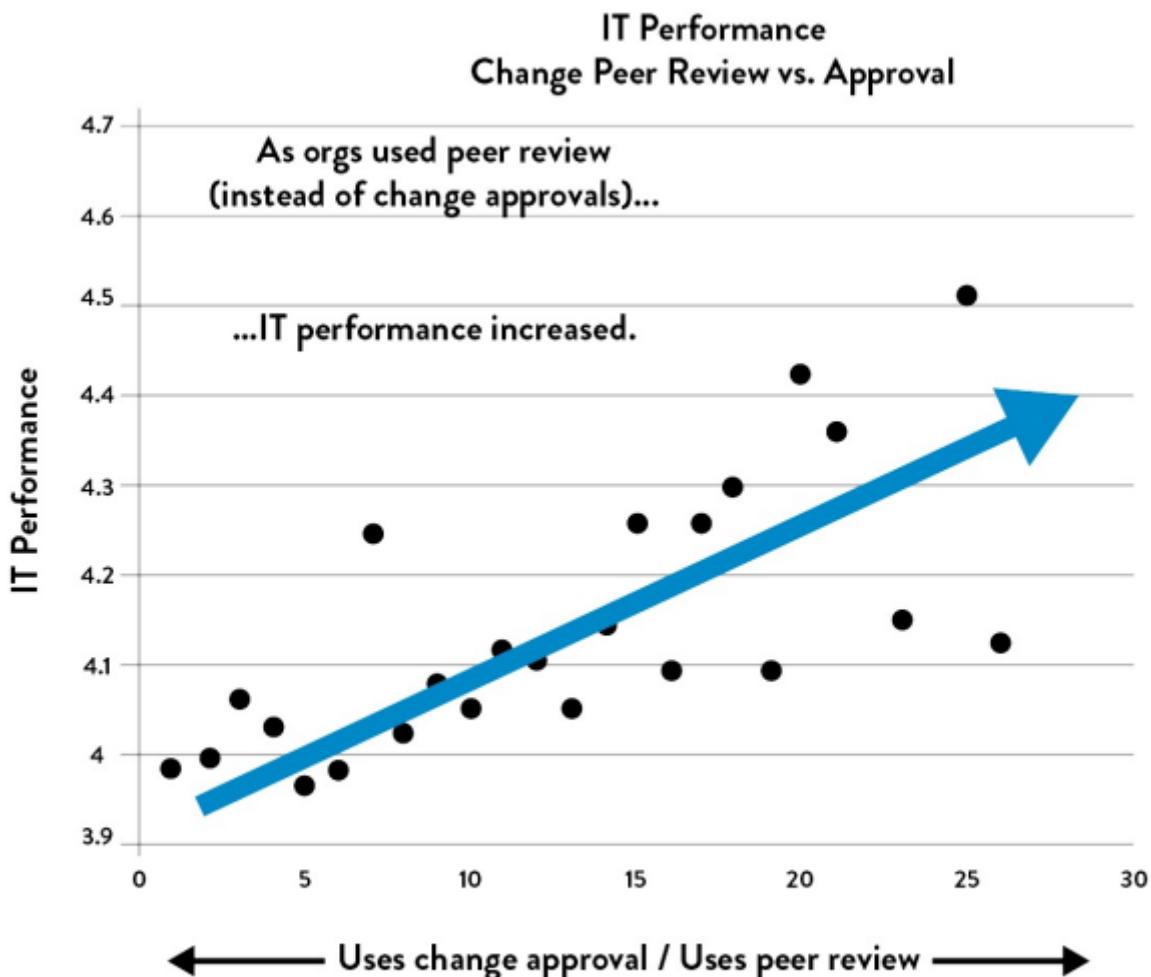


Figure 41: Organizations that rely on peer review outperform those with change approvals (Source: Puppet Labs, DevOps Survey Of Practice 2014)

Dans de nombreuses organisations, les comités consultatifs sur les changements jouent un rôle important dans la coordination et la gouvernance du processus de livraison, mais leur travail ne devrait pas être d'évaluer manuellement chaque changement, et ITIL ne mandate pas une telle pratique.

Pour comprendre pourquoi c'est le cas, considérez la situation d'être dans un comité consultatif sur les changements, en train de revoir un changement complexe composé de centaines de milliers de lignes de code modifiées, et créé par des centaines d'ingénieurs.

À une extrémité, nous ne pouvons pas prédire de manière fiable si un changement sera réussi en lisant une description de cent mots du changement ou en validant simplement qu'une liste de contrôle a été complétée. À l'autre extrémité, examiner minutieusement des milliers de lignes de code modifiées est peu susceptible de révéler de nouvelles idées. Cela fait partie de la nature des changements dans un système complexe. Même les ingénieurs qui travaillent dans la base de code dans le cadre de leur travail quotidien sont souvent surpris par les effets secondaires de ce qui devrait être des changements à faible risque.

Pour toutes ces raisons, nous devons créer des pratiques de contrôle efficaces qui ressemblent davantage à des revues par les pairs, réduisant notre dépendance à l'égard des organismes externes pour autoriser nos changements. Nous devons également coordonner et planifier les changements de manière efficace. Nous explorons ces deux aspects dans les deux sections suivantes.

Permettre la coordination et la planification des changements

Chaque fois que nous avons plusieurs groupes travaillant sur des systèmes qui partagent des dépendances, nos changements devront probablement être coordonnés pour s'assurer qu'ils ne se gênent pas mutuellement (par exemple, en organisant, regroupant et séquençant les changements). En général, plus notre architecture est faiblement couplée, moins nous avons besoin de communiquer et de coordonner avec d'autres équipes de composants - lorsque l'architecture est véritablement orientée service, les équipes peuvent apporter des modifications avec un haut degré d'autonomie, où les changements locaux sont peu susceptibles de créer des perturbations globales. Cependant, même dans une architecture faiblement couplée, lorsque de nombreuses équipes effectuent des centaines de déploiements indépendants par jour, il peut y avoir un risque que les changements interfèrent les uns avec les autres (par exemple, des tests A/B simultanés). Pour atténuer ces risques, nous pouvons utiliser des salles de chat pour annoncer les changements et détecter de manière proactive les collisions éventuelles.

Pour les organisations plus complexes et celles ayant des architectures plus étroitement couplées, nous pourrions devoir programmer délibérément nos changements, où des représentants des équipes se réunissent, non pas pour autoriser les changements, mais pour programmer et séquencer leurs changements afin de minimiser les accidents. Cependant, certains domaines, tels que les changements d'infrastructure globale (par exemple, les changements de commutateurs réseau principaux) auront toujours un risque plus élevé associé à eux. Ces changements nécessiteront toujours des contre-mesures techniques, telles que la redondance, le basculement, les tests complets et (idéalement) la simulation.

Permettre la révision par les pairs des changements

Au lieu d'exiger l'approbation d'un organisme externe avant le déploiement, nous pouvons exiger des ingénieurs qu'ils obtiennent des révisions par les pairs de leurs changements. En développement, cette pratique est appelée revue de code, mais elle est également applicable à tout changement que nous apportons à nos applications ou environnements, y compris les serveurs, les réseaux et les bases de données.

Le but est de trouver des erreurs en faisant examiner nos changements par des collègues ingénieurs proches du travail. Cette révision améliore la qualité de nos changements, ce qui crée également les avantages de la formation croisée, de l'apprentissage par les pairs et de l'amélioration des compétences.

Un endroit logique pour exiger des révisions est avant de valider du code dans la branche principale du contrôle de source, où les changements pourraient potentiellement avoir un impact sur l'équipe ou à l'échelle mondiale. Au minimum, les collègues ingénieurs devraient réviser nos changements, mais pour les domaines à risque plus élevé, tels que les changements de base de données ou les composants critiques pour l'entreprise avec une couverture de test automatisée insuffisante, nous pouvons exiger une révision supplémentaire par un expert en la matière (par exemple, ingénieur en sécurité de l'information, ingénieur de base de données) ou plusieurs révisions (par exemple, "+2" au lieu de simplement "+1").

Le principe des petites tailles de lots s'applique également aux revues de code. Plus la taille du changement à réviser est grande, plus il faut de temps pour comprendre et plus le fardeau pour l'ingénieur révisant est lourd. Comme l'a observé Randy Shoup, "Il existe une relation non linéaire entre la taille du changement et le risque d'intégration de ce changement - lorsque vous passez d'un changement de dix lignes de code à cent lignes de code, le risque que quelque chose tourne mal est plus de dix fois plus élevé, et ainsi de suite." C'est pourquoi il est essentiel que les développeurs travaillent par petites étapes incrémentales plutôt que sur des branches de fonctionnalités à long terme.

De plus, notre capacité à critiquer de manière significative les changements de code diminue à mesure que la taille du changement augmente. Comme l'a tweeté Giray Özil, "Demandez à un programmeur de réviser dix lignes de code, il trouvera dix problèmes. Demandez-lui de réviser cinq cents lignes, et il dira que ça a l'air bon."

Les directives pour les revues de code incluent :

- Tout le monde doit avoir quelqu'un pour réviser ses changements (par exemple, au code, à l'environnement, etc.) avant de valider dans la branche principale.

- Tout le monde doit surveiller le flux de validation de leurs coéquipiers afin que les conflits potentiels puissent être identifiés et révisés.
- Définir quels changements sont qualifiés de haut risque et peuvent nécessiter une révision par un expert désigné en la matière (par exemple, changements de base de données, modules sensibles à la sécurité tels que l'authentification, etc.).
- Si quelqu'un soumet un changement trop grand pour être compris facilement- en d'autres termes, si vous ne pouvez pas comprendre son impact après l'avoir lu plusieurs fois ou si vous devez demander des éclaircissements au soumetteur - il doit être divisé en plusieurs petits changements pouvant être compris d'un coup d'œil.

Pour nous assurer que nous ne faisons pas simplement des révisions de pure forme, nous pouvons également vouloir inspecter les statistiques de révision de code pour déterminer le nombre de changements proposés approuvés par rapport à ceux non approuvés, et peut-être échantillonner et inspecter des révisions de code spécifiques.

Les revues de code se présentent sous différentes formes :

- **Programmation en binôme** : les programmeurs travaillent en binômes (voir section ci-dessous)
- **"Par-dessus l'épaule"** : un développeur regarde par-dessus l'épaule de l'auteur pendant que ce dernier parcourt le code.
- **Passation par courriel** : un système de gestion de code source envoie automatiquement le code aux réviseurs après la validation du code.
- **Revue de code assistée par outil** : les auteurs et les réviseurs utilisent des outils spécialisés conçus pour la révision de code par les pairs (par exemple, Gerrit, les pull requests GitHub, etc.) ou les fonctionnalités fournies par les dépôts de code source (par exemple, GitHub, Mercurial, Subversion, ainsi que d'autres plateformes comme Gerrit, Atlassian Stash et Atlassian Crucible).

Un examen minutieux des changements sous de nombreuses formes est efficace pour détecter des erreurs précédemment négligées. Les revues de code peuvent faciliter l'augmentation des validations de code et des déploiements en production, et soutenir le déploiement basé sur la branche principale et la livraison continue à grande échelle, comme nous le verrons dans l'étude de cas suivante.

Étude de cas - Revues de code chez Google (2010)

Google est un excellent exemple d'entreprise qui pratique le développement basé sur le tronc commun et la livraison continue à grande échelle. Comme mentionné précédemment dans ce livre, Eran Messeri a décrit qu'en 2013, les processus chez Google permettaient à plus de treize mille développeurs de travailler à partir du tronc commun d'un seul arbre de code source, effectuant plus de 5 500 validations de code par semaine, ce qui se traduisait par des centaines de déploiements en production par semaine. En 2010, plus de 20 modifications étaient validées dans le tronc commun chaque minute, ce qui entraînait des changements sur 50 % de la base de code chaque mois.

Cela demande une discipline considérable de la part des membres de l'équipe chez Google et des revues de code obligatoires, qui couvrent les domaines suivants :

- Lisibilité du code pour les langages (application des guides de style)
- Attribution des propriétaires pour les sous-arbres de code afin de maintenir la cohérence et la correction
- Transparence du code et contributions de code entre les équipes

La Figure 42 montre comment les délais de revue de code sont affectés par la taille des changements. Sur l'axe des x se trouve la taille du changement, et sur l'axe des y se trouve le délai nécessaire pour le processus de revue de code. En général, plus le changement soumis pour la revue de code est important, plus le délai nécessaire pour obtenir les validations nécessaires est long. Les points de données dans le coin supérieur gauche représentent les changements les plus complexes et potentiellement risqués nécessitant une plus grande délibération et discussion.

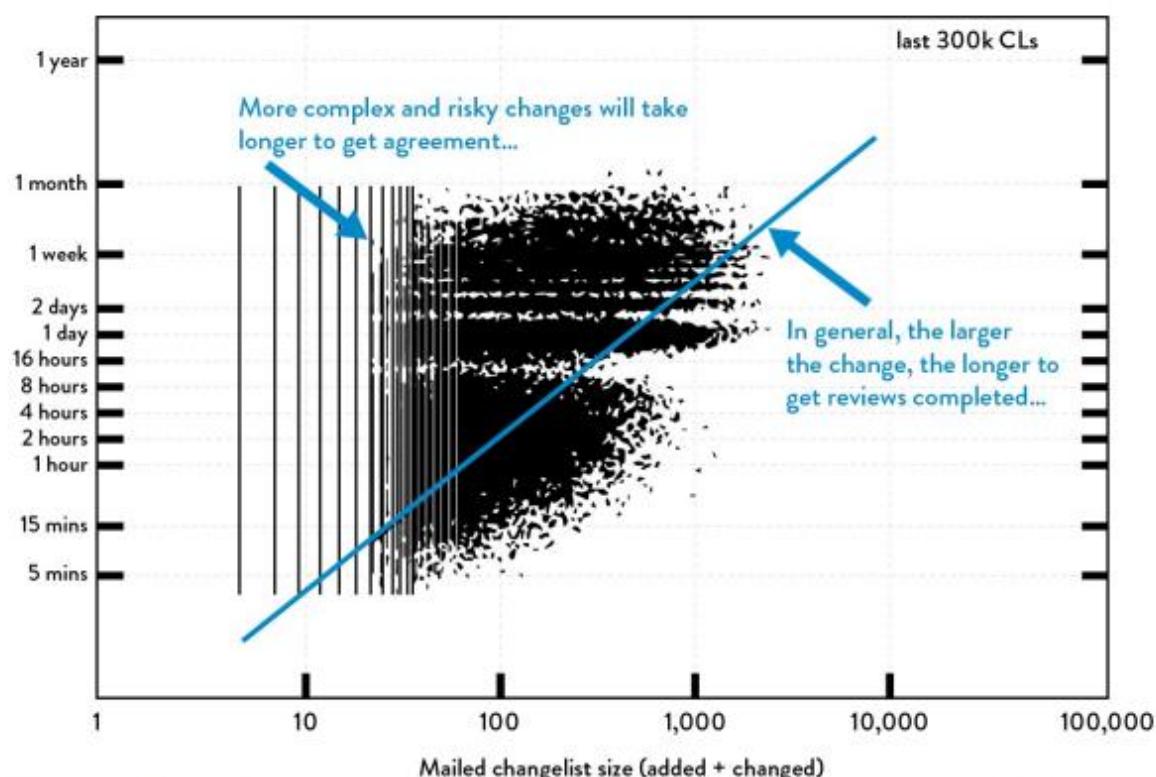


Figure 42: Size of change vs. lead time for reviews at Google (Source: Ashish Kumar, “Development at the Speed and Scale of Google,” presentation at QCon, San Francisco, CA, 2010, https://qconsf.com/sf2010/dl/qcon-sanfran-2010/slides/AshishKumar_DevelopingProductsattheSpeedandScaleofGoogle.pdf.)

Pendant qu'il travaillait en tant que directeur de l'ingénierie chez Google, Randy Shoup a lancé un projet personnel pour résoudre un problème technique auquel l'organisation était

confrontée. Il a déclaré : "J'ai travaillé sur ce projet pendant des semaines et j'ai finalement demandé à un expert du sujet de passer en revue mon code. C'était près de trois mille lignes de code, ce qui a pris plusieurs jours de travail au relecteur pour tout examiner. Il m'a dit : 'S'il te plaît, ne me refais pas ça.' J'ai été reconnaissant que cet ingénieur ait pris le temps de le faire. C'est aussi à ce moment-là que j'ai appris à intégrer les revues de code dans mon travail quotidien."

Potentiels dangers de faire plus de tests manuels et de gel des modifications

Maintenant que nous avons mis en place des revues par les pairs qui réduisent notre risque, raccourcissent les délais associés aux processus d'approbation des changements et permettent la livraison continue à grande échelle, comme nous l'avons vu dans l'étude de cas de Google, examinons les effets de la contre-mesure de test qui peut parfois se retourner contre nous. Lorsque des échecs de test surviennent, notre réaction habituelle est de faire plus de tests. Cependant, si nous effectuons simplement plus de tests vers la fin du projet, nous risquons d'aggraver nos résultats.

C'est particulièrement vrai si nous faisons des tests manuels, car les tests manuels sont naturellement plus lents et plus fastidieux que les tests automatisés, et effectuer des "tests supplémentaires" a souvent pour conséquence de prendre beaucoup plus de temps pour les tests, ce qui signifie que nous déployons moins fréquemment, augmentant ainsi la taille de nos lots de déploiement. Et nous savons tant dans la théorie que dans la pratique que lorsque nous augmentons la taille de nos lots de déploiement, nos taux de réussite des changements diminuent et nos incidents et notre MTTR (Mean Time To Recovery) augmentent - l'effet inverse de ce que nous souhaitons.

Au lieu d'effectuer des tests sur de grands lots de changements planifiés autour des périodes de gel des modifications, nous voulons intégrer pleinement le test de notre travail quotidien comme faisant partie d'un flux continu et fluide vers la production, et augmenter notre fréquence de déploiement. En faisant cela, nous intégrons la qualité, ce qui nous permet de tester, déployer et publier dans des lots de plus en plus petits.

Utiliser la programmation en binôme pour améliorer tous nos changements

La programmation en binôme consiste à ce que deux ingénieurs travaillent ensemble sur la même station de travail, une méthode popularisée par Extreme Programming et Agile au début des années 2000. Comme pour les revues de code, cette pratique a commencé dans le développement mais s'applique tout autant au travail que tout ingénieur réalise dans notre chaîne de valeur. Dans ce livre, nous utiliserons les termes "pairing" et "pair programming" de manière interchangeable pour indiquer que cette pratique n'est pas réservée aux développeurs.

Dans un schéma courant de pair programming, un ingénieur endosse le rôle du "driver", celui qui écrit réellement le code, tandis que l'autre ingénieur joue le rôle de "navigateur", observateur ou pointeur, celui qui examine le travail pendant qu'il est réalisé. En examinant, l'observateur peut également considérer la direction stratégique du travail, proposer des idées d'amélioration et anticiper les problèmes futurs probables. Cela permet au "driver" de se concentrer entièrement sur les aspects tactiques de l'achèvement de la tâche, en utilisant l'observateur comme filet de sécurité et guide. Lorsque les deux ont des spécialités différentes, les compétences sont transférées comme effet secondaire automatique, que ce soit par une formation ad hoc ou par le partage de techniques et de solutions de contournement.

Un autre schéma de pair programming renforce le développement piloté par les tests (TDD) en faisant écrire à un ingénieur le test automatisé et à l'autre implémenter le code. Jeff Atwood, l'un des fondateurs de Stack Exchange, a écrit : "Je me demande si le pair programming n'est rien de plus que la revue de code sous stéroïdes... L'avantage du pair programming est son immédiateté captivante : il est impossible d'ignorer le relecteur lorsqu'il est assis juste à côté de vous." Il a ajouté : "La plupart des gens opteront passivement pour ne pas faire de revue de code s'ils en ont le choix. Avec le pair programming, ce n'est pas possible. Chaque moitié de la paire doit comprendre le code, là et maintenant, au fur et à mesure de son écriture. Le pair programming peut être envahissant, mais il peut aussi imposer un niveau de communication que vous n'atteindriez pas autrement."

Le Dr Laurie Williams a réalisé une étude en 2001 qui a montré que "les programmeurs en binôme sont 15 % plus lents que deux programmeurs individuels indépendants, tandis que le code 'sans erreur' est passé de 70 % à 85 %. Comme les tests et le débogage sont souvent beaucoup plus coûteux que la programmation initiale, c'est un résultat impressionnant. Les paires envisagent généralement plus d'alternatives de conception que les programmeurs travaillant seuls et parviennent à des conceptions plus simples et plus faciles à maintenir ; elles repèrent également tôt les défauts de conception." Le Dr Williams a également rapporté que 96 % de ses répondants ont déclaré qu'ils appréciaient davantage leur travail lorsqu'ils programmaient en binôme plutôt que lorsqu'ils programmaient seuls.

Le pair programming présente également l'avantage supplémentaire de diffuser les connaissances dans toute l'organisation et d'augmenter le flux d'information au sein de l'équipe. Lorsque des ingénieurs plus expérimentés passent en revue le travail d'un ingénieur moins expérimenté pendant que celui-ci code, cela constitue également une manière efficace d'enseigner et d'apprendre.

Étude de cas - Le Pair Programming remplace les processus de revue de code défaillants chez Pivotal Labs (2011)

Elisabeth Hendrickson, vice-présidente de l'ingénierie chez Pivotal Software, Inc. et auteure de "Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing", a longuement parlé de rendre chaque équipe responsable de sa propre qualité, plutôt que de confier cette responsabilité à des départements distincts. Elle soutient que cela non seulement améliore la qualité, mais augmente également considérablement le flux de travail vers la production.

Dans sa présentation au DevOps Enterprise Summit de 2015, elle a décrit comment en 2011, il y avait deux méthodes acceptées de revue de code chez Pivotal : le pair programming (qui garantissait que chaque ligne de code était inspectée par deux personnes) ou un processus de revue de code géré par Gerrit (qui garantissait que chaque validation de code était approuvée par deux personnes désignées "+1" avant d'être intégrée dans le tronc commun).

Le problème observé par Hendrickson avec le processus de revue de code Gerrit était qu'il fallait souvent toute une semaine aux développeurs pour obtenir les revues nécessaires. Pire encore, les développeurs expérimentés vivaient "une expérience frustrante et décourageante de ne pas pouvoir intégrer de simples changements dans la base de code, car nous avions involontairement créé des goulots d'étranglement intolérables."

Hendrickson se lamentait : "les seules personnes ayant la capacité d'approuver les changements étaient les ingénieurs seniors, qui avaient de nombreuses autres responsabilités et ne se souciaient souvent pas autant des corrections sur lesquelles travaillaient les développeurs juniors ou de leur productivité. Cela a créé une situation terrible : pendant que vous attendiez que vos changements soient revus, d'autres développeurs intégraient leurs changements. Ainsi, pendant une semaine, vous deviez fusionner tous leurs changements de code sur votre ordinateur portable, relancer tous les tests pour s'assurer que tout fonctionnait toujours, et parfois vous deviez soumettre à nouveau vos changements pour revue !"

Pour résoudre le problème et éliminer tous ces retards, ils ont fini par démanteler entièrement le processus de revue de code Gerrit, en exigeant à la place le pair programming pour implémenter les changements de code dans le système. En faisant cela, ils ont réduit le temps nécessaire pour faire examiner le code de semaines à quelques heures.

Hendrickson souligne rapidement que les revues de code fonctionnent bien dans de nombreuses organisations, mais cela nécessite une culture qui valorise autant la revue de code que l'écriture du code en premier lieu. Lorsque cette culture n'est pas encore en place, le pair programming peut servir de pratique intérimaire précieuse.

Évaluer l'efficacité des processus de pull request

Étant donné que le processus de revue par les pairs est une partie importante de notre environnement de contrôle, nous devons être en mesure de déterminer s'il fonctionne efficacement ou non. Une méthode consiste à examiner les pannes de production et à étudier le processus de revue par les pairs pour les changements pertinents.

Une autre méthode provient de Ryan Tomayko, CIO et co-fondateur de GitHub, ainsi que l'un des inventeurs du processus de pull request. Lorsqu'on lui a demandé de décrire la différence entre une mauvaise et une bonne pull request, il a déclaré que cela avait peu à voir avec le résultat de production. Au contraire, une mauvaise pull request est celle qui manque de contexte pour le lecteur, avec peu ou pas de documentation sur ce que le changement est censé accomplir. Par exemple, une pull request qui se contente du texte suivant : "Correction des problèmes #3616 et #3841."

Il s'agissait d'une véritable pull request interne de GitHub, que Tomayko a critiquée en disant : "Cela a probablement été écrit par un nouvel ingénieur ici. Tout d'abord, aucun ingénieur spécifique n'a été mentionné spécifiquement - au minimum, l'ingénieur aurait dû mentionner leur mentor ou un expert dans le domaine qu'ils modifient pour s'assurer qu'une personne appropriée examine leur changement. Pire encore, il n'y a aucune explication sur ce que sont réellement les changements, pourquoi ils sont importants, ou aucune exposition de la réflexion de l'exécutant."

D'autre part, lorsqu'on lui demande de décrire une excellente pull request indiquant un processus de revue efficace, Tomayko énumère rapidement les éléments essentiels : il doit y avoir suffisamment de détails sur pourquoi le changement est réalisé, comment il a été effectué, ainsi que tout risque identifié et les contre-mesures correspondantes.

Tomayko recherche également une bonne discussion sur le changement, facilitée par tout le contexte fourni par la pull request. Souvent, des risques supplémentaires sont signalés, des idées sur de meilleures façons de mettre en œuvre le changement désiré sont proposées, des suggestions sur la manière de mieux atténuer le risque, etc. De plus, si quelque chose de mauvais ou d'inattendu se produit lors du déploiement, il est ajouté à la pull request avec un lien vers le problème correspondant. Toute discussion se déroule sans chercher à blâmer ; au contraire, il y a une conversation franche sur la manière de prévenir le problème à l'avenir.

À titre d'exemple, Tomayko a présenté une autre pull request interne de GitHub pour une migration de base de données. Elle était longue de plusieurs pages, avec de longues discussions sur les risques, aboutissant à la déclaration suivante de l'auteur de la pull request : "Je lance cela maintenant. Les builds échouent maintenant pour la branche, en raison d'une colonne manquante dans les serveurs CI. (Lien vers le Post-Mortem : Panne MySQL)"

L'auteur du changement s'est ensuite excusé pour l'incident, décrivant les conditions et les hypothèses erronées qui ont conduit à l'accident, ainsi qu'une liste de contre-mesures proposées pour éviter que cela ne se reproduise. Cela a été suivi de pages et de pages de discussion. En lisant la pull request, Tomayko a souri : "Voilà une excellente pull request."

Comme décrit ci-dessus, nous pouvons évaluer l'efficacité de notre processus de revue par les pairs en échantillonnant et en examinant des pull requests, soit dans toute la population des pull requests, soit celles qui sont pertinentes pour les incidents de production.

Couper sans peur les procédures bureaucratiques

Jusqu'à présent, nous avons discuté des processus de revue par les pairs et de programmation en binôme qui nous permettent d'améliorer la qualité de notre travail sans dépendre d'approbations externes pour les changements. Cependant, de nombreuses entreprises ont encore des processus d'approbation de longue date qui nécessitent des mois pour être navigués. Ces processus d'approbation peuvent considérablement augmenter les délais, non seulement en nous empêchant de livrer rapidement de la valeur aux clients, mais aussi en augmentant potentiellement le risque pour nos objectifs organisationnels. Lorsque cela se produit, nous devons revoir nos processus pour atteindre nos objectifs plus rapidement et en toute sécurité.

Comme l'a observé Adrian Cockcroft, "Une excellente métrique à publier largement est le nombre de réunions et de tickets de travail obligatoires pour effectuer une release - l'objectif est de réduire sans relâche l'effort requis pour que les ingénieurs effectuent leur travail et le livrent aux clients."

De manière similaire, le Dr Tapabrata Pal, fellow technique chez Capital One, a décrit un programme chez Capital One appelé "Got Goo?", qui implique une équipe dédiée à supprimer les obstacles, y compris les outils, les processus et les approbations, qui entravent l'achèvement du travail. Jason Cox, directeur principal de l'ingénierie des systèmes chez Disney, a décrit dans sa présentation au DevOps Enterprise Summit en 2015 un programme appelé "Join The Rebellion" visant à éliminer les tâches ingrates et les obstacles du travail quotidien.

Chez Target en 2012, une combinaison du processus d'adoption technologique de l'entreprise et du Lead Architecture Review Board (processus TEAP-LARB) a entraîné des délais d'approbation complexes et longs pour quiconque tentait d'introduire de nouvelles technologies. Le formulaire TEAP devait être rempli par toute personne souhaitant proposer l'adoption de nouvelles technologies, telles qu'une nouvelle base de données ou des technologies de surveillance. Ces propositions étaient évaluées, et celles jugées appropriées étaient inscrites à l'ordre du jour mensuel de la réunion LARB.

Heather Mickman et Ross Clanton, respectivement directrice du développement et directeur des opérations chez Target, Inc., ont contribué à diriger le mouvement DevOps chez Target. Au cours de leur initiative DevOps, Mickman avait identifié une technologie nécessaire pour soutenir une initiative des lignes de business (en l'occurrence, Tomcat et Cassandra). La décision du LARB était que les Opérations ne pouvaient pas la soutenir à ce moment-là. Cependant, parce que Mickman était convaincue de l'importance de cette technologie, elle a proposé que son équipe de développement soit responsable du support de service ainsi que de l'intégration, de la disponibilité et de la sécurité, au lieu de dépendre de l'équipe des Opérations.

"Alors que nous traversions le processus, je voulais mieux comprendre pourquoi le processus TEAP-LARB prenait autant de temps à franchir, et j'ai utilisé la technique des 'cinq pourquoi'... Ce qui a finalement conduit à la question de savoir pourquoi le TEAP-LARB existait en premier lieu. La chose surprenante était que personne ne le savait, en dehors d'une notion vague selon laquelle nous avions besoin d'un processus de gouvernance quelconque. Beaucoup savaient qu'il y avait eu une sorte de catastrophe qui ne pourrait jamais se reproduire il y a des années, mais personne ne pouvait se rappeler exactement quelle catastrophe c'était," a observé Mickman.

Mickman a conclu que ce processus n'était pas nécessaire pour son groupe s'il était responsable des responsabilités opérationnelles de la technologie qu'elle introduisait. Elle a ajouté, "J'ai informé tout le monde que les futures technologies que nous soutiendrions ne devraient pas non plus passer par le processus TEAP-LARB."

Le résultat a été que Cassandra a été introduite avec succès chez Target et finalement largement adoptée. De plus, le processus TEAP-LARB a finalement été démantelé. En signe de reconnaissance, son équipe a décerné à Mickman le prix Achievement Award pour avoir supprimé les obstacles à la réalisation du travail technologique au sein de Target.

Conclusion

Dans ce chapitre, nous avons discuté de la façon d'intégrer des pratiques dans notre travail quotidien qui augmentent la qualité de nos changements et réduisent le risque de résultats médiocres lors du déploiement, en réduisant notre dépendance aux processus d'approbation. Les études de cas de GitHub et de Target montrent que ces pratiques améliorent non seulement nos résultats, mais réduisent également considérablement les délais et augmentent la productivité des développeurs. Pour mener à bien ce type de travail, une culture de haute confiance est nécessaire.

Considérons une histoire que John Allspaw a racontée à propos d'une jeune ingénierie récemment embauchée : L'ingénierie a demandé s'il était acceptable de déployer un petit changement HTML, et Allspaw a répondu, "Je ne sais pas, est-ce le cas ?" Il a ensuite demandé,

"Avez-vous fait vérifier votre changement par quelqu'un ? Savez-vous qui est la meilleure personne à solliciter pour des changements de ce type ? Avez-vous fait tout ce que vous pouviez pour vous assurer que ce changement fonctionne en production comme prévu ? Si oui, alors ne me demandez pas - faites simplement le changement."

En répondant de cette manière, Allspaw a rappelé à l'ingénierie qu'elle était seule responsable de la qualité de son changement : si elle avait fait tout ce qu'elle pouvait pour se donner confiance que le changement fonctionnerait, alors elle n'avait pas besoin de demander l'approbation de quelqu'un d'autre, elle devait faire le changement.

Créer les conditions qui permettent aux implementeurs de changement de posséder pleinement la qualité de leurs changements est une partie essentielle de la culture générative de haute confiance que nous nous efforçons de construire. De plus, ces conditions nous permettent de créer un système de travail de plus en plus sécurisé, où nous nous aidons tous mutuellement à atteindre nos objectifs, en franchissant toutes les frontières nécessaires pour y parvenir.

Conclusion de la partie IV

La partie IV nous a montré qu'en mettant en œuvre des boucles de rétroaction, nous pouvons permettre à tous de travailler ensemble vers des objectifs communs, de voir les problèmes lorsqu'ils surviennent et, avec une détection et une récupération rapides, nous assurer que les fonctionnalités non seulement fonctionnent comme prévu en production, mais atteignent également les objectifs organisationnels et l'apprentissage organisationnel. Nous avons également examiné comment permettre des objectifs communs couvrant Dev et Ops afin qu'ils puissent améliorer la santé de l'ensemble du flux de valeur.

Nous sommes maintenant prêts à entrer dans la partie V : Le Troisième Chemin, Les Pratiques Techniques de l'Apprentissage, afin que nous puissions créer des opportunités d'apprentissage qui se produisent plus tôt, plus rapidement et à moindre coût, et pour que nous puissions libérer une culture d'innovation et d'expérimentation qui permet à chacun d'accomplir un travail significatif qui aide notre organisation à réussir.

Partie V - La troisième voie - Les pratiques d'apprentissage continu et d'expérimentation

Dans la Partie III, La Première Voie : Les Pratiques Techniques de Flux, nous avons discuté de la mise en œuvre des pratiques nécessaires pour créer un flux rapide dans notre chaîne de valeur. Dans la Partie IV, La Deuxième Voie : Les Pratiques Techniques de Feedback, notre objectif était de créer autant de feedback que possible, provenant de nombreuses zones de notre système, et ce, de manière plus précoce, plus rapide et moins coûteuse.

Dans la Partie V, La Troisième Voie : Les Pratiques Techniques d'Apprentissage, nous présentons les pratiques qui créent des opportunités d'apprentissage, de manière rapide, fréquente, économique et aussi tôt que possible. Cela inclut l'apprentissage à partir d'accidents et d'échecs, qui sont inévitables lorsque nous travaillons dans des systèmes complexes, ainsi que l'organisation et la conception de nos systèmes de travail de manière à expérimenter et apprendre continuellement, rendant ainsi nos systèmes plus sûrs. Les résultats incluent une résilience accrue et une connaissance collective croissante de la manière dont notre système fonctionne réellement, nous permettant ainsi d'atteindre plus efficacement nos objectifs.

Dans les chapitres suivants, nous allons institutionnaliser des rituels qui augmentent la sécurité, l'amélioration continue et l'apprentissage en faisant ce qui suit :

- Établir une culture juste pour rendre la sécurité possible
- Injecter des échecs de production pour créer de la résilience
- Convertir les découvertes locales en améliorations globales
- Réserver du temps pour créer des améliorations organisationnelles et de l'apprentissage

Nous allons également créer des mécanismes pour que tout nouvel apprentissage généré dans une zone de l'organisation puisse être rapidement utilisé dans l'ensemble de l'organisation, transformant ainsi les améliorations locales en avancées globales. De cette manière, non seulement nous apprenons plus rapidement que nos concurrents, nous aidant à gagner sur le marché, mais nous créons également une culture de travail plus sûre et plus résiliente, que les gens sont enthousiastes de rejoindre et qui les aide à atteindre leur plein potentiel.

Activer et injecter l'apprentissage dans le travail quotidien

Lorsque nous travaillons au sein d'un système complexe, il est impossible de prédire toutes les conséquences des actions que nous entreprenons. Cela contribue à des accidents inattendus et parfois catastrophiques, même lorsque nous utilisons des outils préventifs statiques tels que des listes de contrôle et des runbooks, qui codifient notre compréhension actuelle du système.

Pour nous permettre de travailler en toute sécurité au sein de systèmes complexes, nos organisations doivent devenir de plus en plus douées en autodiagnostic et en auto-amélioration, et être habiles à détecter les problèmes, à les résoudre et à multiplier les effets en rendant les solutions disponibles dans toute l'organisation. Cela crée un système dynamique d'apprentissage qui nous permet de comprendre nos erreurs et de traduire cette compréhension en actions visant à empêcher ces erreurs de se reproduire à l'avenir.

Le résultat est ce que le Dr Steven Spear décrit comme des organisations résilientes, qui sont "habiles à détecter les problèmes, à les résoudre et à multiplier les effets en rendant les solutions disponibles dans toute l'organisation." Ces organisations peuvent se guérir elles-mêmes. "Pour une telle organisation, répondre aux crises n'est pas un travail idiosyncratique. C'est quelque chose qui est fait tout le temps. C'est cette réactivité qui est leur source de fiabilité."

Un exemple frappant de l'incroyable résilience qui peut résulter de ces principes et pratiques a été observé le 21 avril 2011, lorsque toute la zone de disponibilité US-EAST d'Amazon AWS a été mise hors service, affectant pratiquement tous leurs clients qui en dépendaient, y compris Reddit et Quora.

Cependant, Netflix était une exception surprenante, apparemment pas affectée par cette énorme panne AWS. À la suite de l'événement, il y a eu beaucoup de spéculations sur la manière dont Netflix a maintenu ses services en ligne. Une théorie populaire était que puisque Netflix était l'un des plus grands clients d'Amazon Web Services, il bénéficiait d'un traitement spécial qui lui permettait de continuer à fonctionner. Cependant, un article de blog d'ingénierie de Netflix a expliqué que leurs décisions architecturales prises en 2009 ont permis leur résilience exceptionnelle.

En 2008, le service de diffusion de vidéos en ligne de Netflix fonctionnait sur une application monolithique J2EE hébergée dans l'un de leurs centres de données. Cependant, à partir de 2009, ils ont commencé à restructurer ce système pour le rendre ce qu'ils appelaient "cloud native" : conçu pour fonctionner entièrement dans le cloud public d'Amazon et être suffisamment résilient pour survivre à des pannes importantes.

Un de leurs objectifs de conception spécifiques était de s'assurer que les services Netflix continuaient à fonctionner même si une zone de disponibilité entière d'AWS tombait en panne, comme cela s'est produit avec US-EAST. Pour cela, leur système devait être faiblement couplé,

chaque composant ayant des délais d'attente agressifs pour s'assurer que les composants défaillants ne fassent pas tomber l'ensemble du système. Au lieu de cela, chaque fonctionnalité et composant était conçu pour se dégrader gracieusement. Par exemple, lors de pics de trafic entraînant des pics d'utilisation CPU, au lieu d'afficher une liste de films personnalisés pour l'utilisateur, ils affichaient du contenu statique, tel que des résultats mis en cache ou non personnalisés, nécessitant moins de calcul.

De plus, l'article de blog expliquait qu'en plus de mettre en œuvre ces modèles architecturaux, ils avaient également développé et utilisé un service étonnant et audacieux appelé Chaos Monkey, qui simulait les pannes AWS en tuant constamment et aléatoirement des serveurs de production. Ils l'ont fait parce qu'ils voulaient que toutes les "équipes d'ingénierie s'habituent à un niveau constant d'échec dans le cloud", de sorte que les services puissent "se rétablir automatiquement sans aucune intervention manuelle".

En d'autres termes, l'équipe Netflix a lancé Chaos Monkey pour avoir l'assurance d'avoir atteint leurs objectifs de résilience opérationnelle, en injectant constamment des pannes dans leurs environnements de préproduction et de production.

Comme on pourrait s'y attendre, lorsqu'ils ont d'abord lancé Chaos Monkey dans leurs environnements de production, les services ont échoué de manière qu'ils n'auraient jamais pu prédire ou imaginer. En trouvant et en corrigeant constamment ces problèmes pendant les heures de travail normales, les ingénieurs de Netflix ont rapidement et itérativement créé un service plus résilient, tout en générant simultanément des apprentissages organisationnels (pendant les heures de travail normales !) qui leur ont permis de faire évoluer leurs systèmes bien au-delà de leurs concurrents.

Chaos Monkey n'est qu'un exemple de la manière dont l'apprentissage peut être intégré dans le travail quotidien. L'histoire montre également comment les organisations apprenantes considèrent les échecs, les accidents et les erreurs comme une opportunité d'apprentissage et non comme quelque chose à punir. Ce chapitre explore comment créer un système d'apprentissage et comment établir une culture juste, ainsi que comment répéter régulièrement et délibérément créer des échecs pour accélérer l'apprentissage.

Établir une culture juste et apprenante

L'un des prérequis pour une culture d'apprentissage est que lorsque des accidents se produisent (ce qui arrivera inévitablement), la réponse à ces accidents soit considérée comme "juste". Le Dr Sidney Dekker, qui a contribué à codifier certains des éléments clés de la culture de la sécurité et a inventé le terme de culture juste, écrit : "Lorsque les réponses aux incidents et aux accidents sont perçues comme injustes, cela peut entraver les enquêtes sur la sécurité, favoriser la peur plutôt que la vigilance chez les personnes qui effectuent un travail critique pour

la sécurité, rendre les organisations plus bureaucratiques au lieu d'être plus prudentes, et cultiver le secret professionnel, l'évasion et l'autoprotection."

Cette notion de punition est présente, de manière subtile ou évidente, dans la façon dont de nombreux gestionnaires ont opéré au cours du dernier siècle. La pensée sous-jacente est que, pour atteindre les objectifs de l'organisation, les dirigeants doivent commander, contrôler, établir des procédures pour éliminer les erreurs et faire respecter la conformité à ces procédures.

Le Dr Dekker appelle cette notion d'élimination des erreurs en éliminant les personnes responsables des erreurs la théorie de la mauvaise pomme. Il affirme que cette idée est invalide, car "l'erreur humaine n'est pas la cause de nos problèmes ; au contraire, l'erreur humaine est une conséquence de la conception des outils que nous leur avons donnés."

Si les accidents ne sont pas causés par des "mauvaises pommes", mais plutôt par des problèmes de conception inévitables dans le système complexe que nous avons créé, alors au lieu de "nommer, blâmer et honte", notre objectif devrait toujours être de maximiser les opportunités d'apprentissage organisationnel, en renforçant continuellement que nous valorisons les actions qui exposent et partagent plus largement les problèmes dans notre travail quotidien. C'est ce qui nous permet d'améliorer la qualité et la sécurité du système dans lequel nous opérons et de renforcer les relations entre tous ceux qui opèrent dans ce système.

En transformant l'information en connaissance et en intégrant les résultats de l'apprentissage dans nos systèmes, nous commençons à atteindre les objectifs d'une culture juste, équilibrant les besoins de sécurité et de responsabilité. Comme le dit John Allspaw, CTO d'Etsy : "Notre objectif chez Etsy est de considérer les erreurs, les défaillances, les lapsus, etc., avec une perspective d'apprentissage."

Lorsque les ingénieurs commettent des erreurs et se sentent en sécurité en fournissant des détails à ce sujet, ils sont non seulement disposés à assumer la responsabilité, mais ils sont également enthousiastes à aider le reste de l'entreprise à éviter la même erreur à l'avenir.

Deux pratiques efficaces pour créer une culture juste et axée sur l'apprentissage sont les post-mortems sans blâme et l'introduction contrôlée des échecs en production afin de créer des opportunités de pratique pour les problèmes inévitables qui surviennent dans les systèmes complexes. Nous commencerons par examiner les post-mortems sans blâme, puis explorerons pourquoi l'échec peut être bénéfique.

Planifier les réunions de post-mortem sans blâme après les accidents

Pour favoriser une culture juste, lorsque des accidents ou des incidents significatifs surviennent (par exemple, un déploiement échoué, un problème de production affectant les clients), nous devrions organiser un post-mortem sans blâme après la résolution de l'incident.

Les post-mortems sans blâme, terme inventé par John Allspaw, nous aident à examiner "les erreurs en mettant l'accent sur les aspects situationnels du mécanisme d'échec et le processus de prise de décision des individus proches de l'échec."

Pour ce faire, nous planifions le post-mortem dès que possible après l'accident et avant que les souvenirs et les liens entre cause et effet ne s'estompent ou que les circonstances ne changent. (Bien entendu, nous attendons que le problème soit résolu pour ne pas distraire les personnes qui travaillent encore activement sur le problème.)

Lors de la réunion de post-mortem sans blâme, nous ferons ce qui suit :

- Construire une chronologie et recueillir des détails à partir de plusieurs perspectives sur les échecs, en veillant à ne pas punir les personnes pour leurs erreurs
- Autoriser tous les ingénieurs à améliorer la sécurité en leur permettant de fournir des comptes rendus détaillés de leurs contributions aux échecs
- Permettre et encourager les personnes qui commettent des erreurs à devenir les experts qui éduquent le reste de l'organisation sur la manière de ne pas les reproduire à l'avenir
- Accepter qu'il existe toujours un espace discrétionnaire où les humains peuvent décider d'agir ou non, et que le jugement de ces décisions réside dans la rétrospective
- Proposer des contre-mesures pour éviter qu'un accident similaire ne se reproduise à l'avenir et veiller à ce que ces contre-mesures soient enregistrées avec une date cible et un responsable pour le suivi

Pour nous permettre d'atteindre cette compréhension, les parties prenantes suivantes doivent être présentes lors de la réunion :

- Les personnes impliquées dans les décisions qui ont peut-être contribué au problème
- Les personnes qui ont identifié le problème
- Les personnes qui ont répondu au problème
- Les personnes qui ont diagnostiqué le problème
- Les personnes qui ont été affectées par le problème
- Et toute personne intéressée à assister à la réunion.

Notre première tâche lors de la réunion de post-mortem sans blâme est d'enregistrer notre meilleure compréhension de la chronologie des événements pertinents tels qu'ils se sont produits. Cela inclut toutes les actions que nous avons entreprises et leur heure (idéalement appuyées par des journaux de chat, comme IRC ou Slack), les effets observés (idéalement sous forme de mesures spécifiques provenant de notre télémétrie de production, plutôt que de

simples récits subjectifs), tous les chemins d'investigation que nous avons suivis, et les solutions envisagées.

Pour atteindre ces résultats, nous devons être rigoureux dans l'enregistrement des détails et renforcer une culture où l'information peut être partagée sans crainte de punition ou de représailles. Pour cette raison, surtout pour nos premiers post-mortems, il peut être utile que la réunion soit dirigée par un facilitateur formé qui n'était pas impliqué dans l'accident.

Pendant la réunion et la résolution qui s'ensuit, nous devrions explicitement interdire les expressions "aurait dû" ou "aurait pu", car ce sont des déclarations contrefactuelles qui résultent de notre tendance humaine à créer des alternatives possibles aux événements déjà survenus.

Les déclarations contrefactuelles, telles que "J'aurais pu..." ou "Si j'avais su cela, j'aurais dû...", encadrent le problème en termes du système tel qu'il est imaginé plutôt qu'en termes du système qui existe réellement, ce qui est le contexte sur lequel nous devons nous restreindre.

L'un des résultats potentiellement surprenants de ces réunions est que les gens se blâment souvent pour des choses qui échappent à leur contrôle ou remettent en question leurs propres capacités. Ian Malpass, un ingénieur chez Etsy, observe : "À ce moment-là, lorsque nous faisons quelque chose qui cause la panne de tout le site, nous ressentons ce sentiment de 'l'eau glacée dans le dos', et probablement la première pensée qui nous traverse la tête est, 'Je suis nul et je n'ai aucune idée de ce que je fais.' Nous devons nous empêcher de faire cela, car cela mène à la folie, au désespoir et à des sentiments d'imposture, ce que nous ne pouvons pas permettre aux bons ingénieurs. La meilleure question à se poser est plutôt : 'Pourquoi cela avait-il du sens pour moi lorsque j'ai pris cette mesure ?'"

Dans la réunion, nous devons réservé suffisamment de temps pour le brainstorming et la décision des contre-mesures à mettre en œuvre. Une fois les contre-mesures identifiées, elles doivent être priorisées et avoir un responsable ainsi qu'une échéance pour leur mise en œuvre. Faire cela démontre davantage que nous valorisons l'amélioration de notre travail quotidien plus que le travail quotidien lui-même.

Dan Milstein, l'un des ingénieurs principaux chez Hubspot, écrit qu'il commence toutes les réunions de post-mortem sans blâme "en disant : 'Nous essayons de nous préparer à un avenir où nous sommes aussi stupides qu'aujourd'hui.'" En d'autres termes, il n'est pas acceptable d'avoir une contre-mesure pour simplement "être plus prudent" ou "être moins stupide" - nous devons plutôt concevoir de véritables contre-mesures pour éviter que ces erreurs ne se reproduisent.

Des exemples de telles contre-mesures incluent de nouveaux tests automatisés pour détecter les conditions dangereuses dans notre pipeline de déploiement, l'ajout de nouvelles télémétries de production, l'identification des catégories de changements nécessitant un examen par les pairs supplémentaire, et la réalisation de répétitions de cette catégorie d'échecs dans le cadre d'exercices régulièrement programmés de "Game Day".

Publier nos post-mortems le plus largement possible

Après avoir mené une réunion de post-mortem sans blâme, nous devrions annoncer largement la disponibilité des notes de réunion et de tout artefact associé (par exemple, les chronologies, les journaux de discussion IRC, les communications externes). Ces informations devraient idéalement être placées dans un endroit centralisé accessible à toute notre organisation afin que chacun puisse en tirer des leçons à la suite de l'incident. Il est si important de mener des post-mortems que nous pourrions même interdire la clôture des incidents de production avant que la réunion de post-mortem ne soit terminée.

Cette pratique nous aide à transformer les apprentissages et améliorations locaux en apprentissages et améliorations globaux. Randy Shoup, ancien directeur de l'ingénierie pour Google App Engine, décrit comment la documentation des réunions de post-mortem peut avoir une valeur considérable pour les autres au sein de l'organisation : « Comme vous pouvez l'imaginer chez Google, tout est recherchable. Tous les documents de post-mortem sont disponibles pour les autres Googlers. Et croyez-moi, lorsque tout groupe rencontre un incident similaire à un précédent, ces documents de post-mortem sont parmi les premiers à être lus et étudiés ».

La publication large des post-mortems et l'encouragement à leur lecture au sein de l'organisation augmentent l'apprentissage organisationnel. Il est également de plus en plus courant pour les entreprises de services en ligne de publier des post-mortems concernant les pannes impactant les clients. Cela accroît souvent la transparence envers nos clients internes et externes, renforçant ainsi leur confiance en nous.

Le désir de mener autant de réunions de post-mortem sans blâme que nécessaire chez Etsy a conduit à certains problèmes : au fil de quatre années, Etsy a accumulé un grand nombre de notes de réunions de post-mortem sur des pages de wiki, devenant de plus en plus difficiles à rechercher, sauvegarder et collaborer.

Pour résoudre ce problème, ils ont développé un outil appelé Morgue pour enregistrer facilement différents aspects de chaque accident, tels que le MTTR (Mean Time To Recover) de l'incident et sa gravité, mieux gérer les fuseaux horaires (ce qui est devenu pertinent avec la croissance du nombre d'employés d'Etsy travaillant à distance), et inclure d'autres données comme du texte enrichi au format Markdown, des images intégrées, des tags et un historique.

Morgue a été conçu pour faciliter l'enregistrement des informations suivantes par l'équipe :

- Savoir si le problème était dû à un incident planifié ou non planifié
- Le responsable du post-mortem

- Les journaux de discussion IRC pertinents (particulièrement importants pour les problèmes survenant à 3 heures du matin, lorsque la prise de notes précises peut ne pas être facile)
- Les tickets JIRA pertinents pour les actions correctives et leurs dates d'échéance (information particulièrement importante pour la direction)
- Les liens vers les posts des forums clients (où les clients se plaignent des problèmes)

Après avoir développé et utilisé Morgue, le nombre de post-mortems enregistrés chez Etsy a considérablement augmenté par rapport à l'utilisation des pages de wiki, en particulier pour les incidents de gravité P2, P3 et P4 (c'est-à-dire les problèmes de gravité moindre). Ce résultat a renforcé l'hypothèse que rendre plus facile la documentation des post-mortems grâce à des outils comme Morgue inciterait davantage de personnes à enregistrer et détailler les résultats de leurs réunions de post-mortem, favorisant ainsi un meilleur apprentissage organisationnel.

La Dr. Amy C. Edmondson, Professeur Novartis de Leadership et de Management à la Harvard Business School et co-auteur de Building the Future: Big Teaming for Audacious Innovation, écrit : "Encore une fois, le remède - qui n'implique pas nécessairement beaucoup de temps et de dépenses - est de réduire la stigmatisation de l'échec. Eli Lilly le fait depuis le début des années 1990 en organisant des "fêtes de l'échec" pour honorer les expériences scientifiques intelligentes et de haute qualité qui échouent à atteindre les résultats désirés. Ces fêtes ne coûtent pas cher, et redéployer des ressources précieuses - en particulier des scientifiques - vers de nouveaux projets plus tôt plutôt que plus tard peut économiser des centaines de milliers de dollars, sans parler de stimuler de potentielles nouvelles découvertes."

Diminuer les tolérances aux incidents pour détecter des signaux de défaillance toujours plus faible

Inévitablement, à mesure que les organisations apprennent à voir et résoudre efficacement les problèmes, elles doivent réduire le seuil de ce qui constitue un problème pour continuer à apprendre. Pour ce faire, nous cherchons à amplifier les signaux faibles de défaillance. Par exemple, comme décrit au chapitre 4, lorsque Alcoa a réussi à réduire le taux d'accidents du travail pour qu'ils ne soient plus monnaie courante, Paul O'Neill, PDG d'Alcoa, a commencé à être informé des incidents presque manqués en plus des accidents réels survenant sur le lieu de travail.

Le Dr. Spear résume les accomplissements de O'Neill chez Alcoa en écrivant : "Bien que cela ait commencé en se concentrant sur les problèmes liés à la sécurité au travail, Alcoa a vite compris que les problèmes de sécurité reflétaient une ignorance des processus et que cette ignorance se manifestait également dans d'autres problèmes tels que la qualité, la ponctualité et le rendement par rapport aux rebuts".

Lorsque nous travaillons dans des systèmes complexes, il est crucial d'amplifier les signaux faibles de défaillance pour éviter les échecs catastrophiques. La façon dont la NASA a géré les signaux de défaillance à l'époque des navettes spatiales en est un exemple illustratif : en 2003, seize jours après le lancement de la navette spatiale Columbia, celle-ci a explosé lors de son

retour dans l'atmosphère terrestre. Nous savons maintenant qu'un morceau de mousse isolante s'était détaché du réservoir externe de carburant pendant le décollage.

Cependant, avant la rentrée de Columbia, quelques ingénieurs de la NASA de niveau intermédiaire avaient rapporté cet incident, mais leurs voix étaient restées inaudibles. Ils avaient observé le décollage de mousse sur les écrans vidéo lors d'une session de révision après le lancement et avaient immédiatement informé les gestionnaires de la NASA, mais on leur avait dit que le problème de la mousse n'était rien de nouveau. Le délogement de mousse avait endommagé des navettes lors de lancements précédents, mais cela n'avait jamais entraîné d'accident. C'était considéré comme un problème de maintenance et n'avait pas été pris en compte jusqu'à ce qu'il soit trop tard.

Michael Roberto, Richard M.J. Bohmer et Amy C. Edmondson ont écrit dans un article de 2006 pour la Harvard Business Review comment la culture de la NASA a contribué à ce problème. Ils décrivent comment les organisations sont généralement structurées selon l'un de deux modèles : un modèle standardisé, où la routine et les systèmes gouvernent tout, y compris le respect strict des délais et des budgets, ou un modèle expérimental, où chaque jour chaque exercice et chaque nouvelle information sont évalués et débattus dans une culture qui ressemble à un laboratoire de recherche et développement (R&D).

Ils observent : « Les entreprises rencontrent des problèmes lorsqu'elles appliquent le mauvais état d'esprit à une organisation [ce qui dicte comment elles répondent aux menaces ambiguës ou, dans la terminologie de ce livre, aux signaux de défaillance faibles].... Dans les années 1970, la NASA avait créé une culture de standardisation rigide, en promouvant auprès du Congrès la navette spatiale comme un vaisseau spatial bon marché et réutilisable ». La NASA favorisait la conformité stricte aux processus plutôt qu'un modèle expérimental où chaque information devait être évaluée dès son apparition sans préjugé. L'absence d'apprentissage continu et d'expérimentation a entraîné des conséquences désastreuses. Les auteurs concluent que la culture et l'état d'esprit sont importants, pas seulement "être prudent" - comme ils l'écrivent, "la vigilance seule ne préviendra pas les menaces ambiguës [les signaux de défaillance faibles] de se transformer en échecs coûteux (et parfois tragiques)".

Notre travail dans le flux de valeur technologique, tout comme le voyage spatial, doit être abordé comme une entreprise fondamentalement expérimentale et gérée de cette manière. Chaque travail que nous faisons est potentiellement une hypothèse importante et une source de données, plutôt qu'une application et validation routinières des pratiques passées. Plutôt que de traiter le travail technologique comme entièrement standardisé, où nous visons la conformité aux processus, nous devons continuellement chercher à identifier des signaux de défaillance de plus en plus faibles afin de mieux comprendre et gérer le système dans lequel nous opérons.

Redéfinir l'échec et encourager la prise de risques calculés

Les leaders d'une organisation, qu'ils le fassent délibérément ou involontairement, renforcent la culture organisationnelle et les valeurs par leurs actions. Les experts en audit, comptabilité et éthique ont depuis longtemps observé que le "ton au sommet" prédit la probabilité de fraude et d'autres pratiques non éthiques. Pour renforcer notre culture d'apprentissage et de prise de risques calculés, nous avons besoin que les leaders rappellent constamment que chacun devrait se sentir à l'aise et responsable de mettre en lumière et d'apprendre des échecs.

À propos des échecs, Roy Rapoport de Netflix observe : « Ce que le rapport State of DevOps de 2014 m'a prouvé, c'est que les organisations DevOps performantes échoueront et feront plus souvent des erreurs. Non seulement c'est acceptable, mais c'est ce dont les organisations ont besoin ! Vous pouvez le constater dans les données : si les hauts performants exécutent trente fois plus de changements avec seulement la moitié du taux d'échec, ils ont évidemment plus d'échecs ».

Il continue : « Je discutais avec un collègue à propos d'une panne massive que nous venons de subir à Netflix - elle a été causée, franchement, par une erreur stupide. En fait, elle a été causée par un ingénieur qui a fait tomber Netflix deux fois au cours des dix-huit derniers mois. Mais, bien sûr, c'est une personne que nous ne renverrions jamais. Au cours de ces dix-huit mois, cet ingénieur a fait avancer l'état de nos opérations et de notre automatisation non pas de quelques kilomètres, mais de années-lumière. Ce travail nous a permis de déployer en toute sécurité quotidiennement, et cette personne a personnellement effectué un grand nombre de déploiements en production ».

Il conclut : « DevOps doit permettre ce genre d'innovation et les risques résultants des erreurs humaines. Oui, vous aurez plus d'échecs en production. Mais c'est une bonne chose, et cela ne devrait pas être puni ».

Injecter des échecs en production pour permettre la résilience et l'apprentissage

Comme nous l'avons vu dans l'introduction du chapitre, injecter des défauts dans l'environnement de production (comme Chaos Monkey) est une façon d'accroître notre résilience. Dans cette section, nous décrivons les processus impliqués dans la répétition et l'injection de défaillances dans notre système pour confirmer que nous avons conçu et architecturé nos systèmes de manière appropriée, de sorte que les défaillances se produisent de manière spécifique et contrôlée. Nous le faisons en effectuant régulièrement (voire continuellement) des tests pour nous assurer que nos systèmes échouent de manière élégante.

Comme le commente Michael Nygard, auteur de "Release It! Design and Deploy Production-Ready Software" : « Comme construire des zones de déformation dans les voitures pour absorber les impacts et protéger les passagers, vous pouvez décider quelles fonctionnalités du système sont indispensables et construire des modes de défaillance qui éloignent les fissures de ces fonctionnalités. Si vous ne concevez pas vos modes de défaillance, alors vous obtiendrez ce que vous obtiendrez de manière imprévisible - et généralement dangereuse. »

La résilience nécessite d'abord que nous définissions nos modes de défaillance, puis que nous effectuons des tests pour nous assurer que ces modes de défaillance fonctionnent comme prévu. Une façon de procéder est d'injecter des défauts dans notre environnement de production et de répéter des défaillances à grande échelle afin d'être certains que nous pouvons récupérer des accidents lorsqu'ils se produisent, idéalement sans même affecter nos clients.

L'histoire de 2012 sur Netflix et la panne d'Amazon AWS-EAST présentée dans l'introduction est juste un exemple. Un exemple encore plus intéressant de résilience chez Netflix a eu lieu lors de la "Grande réinitialisation d'Amazon de 2014", lorsque près de 10 % de l'ensemble de la flotte de serveurs Amazon EC2 a dû être redémarré pour appliquer un correctif de sécurité d'urgence Xen. Comme l'a rappelé Christos Kalantzis de Netflix Cloud Database Engineering : « Quand nous avons reçu la nouvelle des redémarrages d'urgence d'EC2, nous sommes restés bouche bée. Quand nous avons reçu la liste des nœuds Cassandra affectés, j'ai eu un malaise. Mais, continue Kalantzis, puis je me suis rappelé tous les exercices de Chaos Monkey que nous avions traversés. Ma réaction a été : "Amenez-le !" »

Une fois de plus, les résultats ont été étonnants. Sur les plus de 2 700 nœuds Cassandra utilisés en production, 218 ont été redémarrés et vingt-deux n'ont pas redémarré avec succès. Comme l'ont écrit Kalantzis et Bruce Wong de Netflix Chaos Engineering : « Netflix n'a connu aucune interruption ce week-end-là. Exercer de manière répétée et régulière l'échec, même au niveau de la couche de persistance [base de données], devrait faire partie de la planification de résilience de chaque entreprise. Si ce n'était pas pour la participation de Cassandra à Chaos Monkey, cette histoire aurait eu une tout autre fin. »

Encore plus surprenant, non seulement personne chez Netflix ne travaillait sur des incidents actifs dus à des nœuds Cassandra défaillants, mais personne n'était même au bureau - ils étaient à Hollywood pour une fête célébrant une étape d'acquisition. C'est un autre exemple démontrant que se concentrer proactivement sur la résilience signifie souvent qu'une entreprise peut gérer des événements qui pourraient causer des crises pour la plupart des organisations de manière routinière et banale.

Instituer des journées de jeu pour répéter les défaillances

Dans cette section, nous décrivons des répétitions spécifiques de récupération après sinistre appelées Journées de Jeu, un terme popularisé par Jesse Robbins, l'un des fondateurs de la communauté Velocity Conference et co-fondateur de Chef, pour le travail qu'il a effectué chez Amazon, où il était responsable des programmes visant à assurer la disponibilité du site et était largement connu en interne sous le nom de "Maître du Désastre". Le concept de Journées de Jeu provient de la discipline de l'ingénierie de la résilience. Robbins définit l'ingénierie de la résilience comme "un exercice conçu pour accroître la résilience grâce à l'injection à grande échelle de défaillances à travers les systèmes critiques".

Robbins observe que "chaque fois que vous lancez dans l'ingénierie d'un système à grande échelle, le mieux que vous puissiez espérer est de construire une plateforme logicielle fiable sur des composants qui sont totalement peu fiables. Cela vous place dans un environnement où les défaillances complexes sont à la fois inévitables et imprévisibles."

Par conséquent, nous devons nous assurer que les services continuent de fonctionner lorsque des défaillances se produisent, potentiellement à travers tout notre système, idéalement sans crise ou même intervention manuelle. Comme le dit Robbins, "un service n'est pas vraiment testé tant que nous ne l'avons pas cassé en production."

Notre objectif pour les Journées de Jeu est d'aider les équipes à simuler et à répéter des accidents pour leur donner la capacité de s'entraîner. Tout d'abord, nous planifions un événement catastrophique, comme la destruction simulée d'un centre de données entier, qui se produira à un moment donné dans le futur. Ensuite, nous donnons aux équipes le temps de se préparer, d'éliminer tous les points de défaillance uniques, et de créer les procédures de surveillance nécessaires, les procédures de basculement, etc.

Notre équipe de Journées de Jeu définit et exécute des exercices, comme la réalisation de basculements de base de données (c'est-à-dire simuler une défaillance de base de données et s'assurer que la base de données secondaire fonctionne) ou l'arrêt d'une connexion réseau importante pour mettre en évidence des problèmes dans les processus définis. Tout problème ou difficulté rencontré est identifié, traité et testé à nouveau.

À l'heure prévue, nous exécutons ensuite la panne. Comme le décrit Robbins, chez Amazon, ils "éteignaient littéralement une installation - sans préavis - puis laissaient les systèmes échouer naturellement et [permettaient] aux personnes de suivre leurs processus où qu'ils les mènent."

En faisant cela, nous commençons à exposer les défauts latents de notre système, qui sont les problèmes qui apparaissent seulement parce que nous avons injecté des défauts dans le système. Robbins explique : "Vous pourriez découvrir que certains systèmes de surveillance ou de gestion cruciaux pour le processus de récupération se retrouvent éteints en tant que partie

de la défaillance que vous avez orchestrée. [Ou] vous pourriez trouver des points de défaillance uniques dont vous ne connaissiez pas l'existence de cette manière." Ces exercices sont ensuite réalisés de manière de plus en plus intense et complexe dans le but de les rendre aussi banals que possible.

En exécutant des Journées de Jeu, nous créons progressivement un service plus résilient et un degré plus élevé d'assurance que nous pouvons reprendre les opérations lorsque des événements inopportunus se produisent, tout en générant davantage d'apprentissages et une organisation plus résiliente.

Un excellent exemple de simulation de catastrophe est le programme de récupération après sinistre (DiRT) de Google. Kripa Krishnan est directrice de programme technique chez Google et, au moment de cette rédaction, elle dirige le programme depuis plus de sept ans. Pendant cette période, ils ont simulé un tremblement de terre dans la Silicon Valley, ce qui a entraîné la déconnexion complète du campus de Mountain View de Google ; des pertes totales de puissance dans les principaux centres de données ; et même des attaques d'aliens sur des villes où résidaient des ingénieurs.

Comme l'a écrit Krishnan : « Une zone souvent négligée des tests est celle des processus métier et des communications. Les systèmes et les processus sont étroitement liés, et séparer les tests des systèmes des tests des processus métier n'est pas réaliste : une défaillance d'un système métier affectera le processus métier, et inversement, un système fonctionnel n'est pas très utile sans le personnel adéquat. »

Certains des apprentissages tirés de ces catastrophes comprenaient :

- Lorsque la connectivité était perdue, la bascule vers les postes de travail des ingénieurs ne fonctionnait pas.
- Les ingénieurs ne savaient pas comment accéder à une passerelle de conférence téléphonique, ou la passerelle n'avait de la capacité que pour cinquante personnes, ou ils avaient besoin d'un nouveau fournisseur de conférences téléphoniques qui leur permettrait d'éjecter les ingénieurs ayant soumis toute la conférence à de la musique d'attente.
- Lorsque les centres de données étaient à court de diesel pour les générateurs de secours, personne ne connaissait les procédures pour effectuer des achats d'urgence auprès du fournisseur, ce qui a conduit quelqu'un à utiliser une carte de crédit personnelle pour acheter 50 000 dollars de diesel.

En créant des défaillances dans une situation contrôlée, nous pouvons pratiquer et créer les livres de jeux dont nous avons besoin. Un des autres résultats des Journées de Jeu est que les gens savent réellement qui appeler et avec qui parler - en faisant cela, ils développent des relations avec des personnes d'autres départements afin de pouvoir travailler ensemble lors d'un incident, transformant des actions conscientes en actions inconscientes qui peuvent devenir routinières.

Conclusion

Pour créer une culture juste qui favorise l'apprentissage organisationnel, nous devons recontextualiser les soi-disant échecs. Lorsqu'ils sont traités correctement, les erreurs inhérentes aux systèmes complexes peuvent créer un environnement d'apprentissage dynamique où tous les intervenants se sentent suffisamment en sécurité pour avancer avec des idées et des observations, et où les groupes rebondissent plus facilement des projets qui ne se déroulent pas comme prévu.

Tant les post-mortems sans blâme que l'injection de défaillances en production renforcent une culture où chacun devrait se sentir à l'aise et responsable de mettre en lumière et d'apprendre des échecs. En fait, lorsque nous réduisons suffisamment le nombre d'accidents, nous diminuons notre tolérance pour pouvoir continuer à apprendre. Comme le dit Peter Senge : « Le seul avantage concurrentiel durable est la capacité d'une organisation à apprendre plus vite que la concurrence. »

Convertir les découvertes locales en améliorations globales

Dans le chapitre précédent, nous avons discuté de la mise en place d'une culture d'apprentissage sécurisée en encourageant tout le monde à parler des erreurs et des incidents grâce à des post-mortems sans blâme. Nous avons également exploré la recherche et la correction de signaux de défaillance de plus en plus faibles, ainsi que le renforcement et la récompense de l'expérimentation et de la prise de risques. De plus, nous avons contribué à rendre notre système de travail plus résilient en planifiant et en testant de manière proactive des scénarios de défaillance, rendant nos systèmes plus sûrs en trouvant et en corrigeant des défauts latents.

Dans ce chapitre, nous allons créer des mécanismes permettant de capturer et de partager à l'échelle de toute l'organisation les nouvelles connaissances et améliorations découvertes localement, multipliant ainsi l'effet des connaissances et des améliorations globales. En faisant cela, nous élevons l'état de la pratique de l'ensemble de l'organisation afin que chacun puisse bénéficier de l'expérience cumulative de l'organisation.

Utiliser des salles de discussion et des bots de discussion pour automatiser et capturer les connaissances organisationnelles

De nombreuses organisations ont créé des salles de discussion pour faciliter la communication rapide au sein des équipes. Cependant, les salles de discussion sont également utilisées pour déclencher l'automatisation. Cette technique a été pionnière dans le parcours ChatOps chez GitHub. L'objectif était d'intégrer des outils d'automatisation dans la conversation de leurs salles de discussion, aidant à créer de la transparence et à documenter leur travail. Comme Jesse Newland, un ingénieur système chez GitHub, le décrit : « Même lorsque vous êtes nouveau dans l'équipe, vous pouvez consulter les journaux de discussion et voir comment tout est fait. C'est comme si vous faisiez du pair-programming avec eux tout le temps. »

Ils ont créé Hubot, une application logicielle qui interagissait avec l'équipe Ops dans leurs salles de discussion, où elle pouvait être instruite pour effectuer des actions simplement en lui envoyant une commande (par exemple, « @hubot déployer owl en production »). Les résultats étaient également envoyés dans la salle de discussion.

Faire effectuer ce travail par l'automatisation dans la salle de discussion (plutôt que d'exécuter des scripts automatisés via la ligne de commande) avait de nombreux avantages, notamment :

- Tout le monde voyait tout ce qui se passait.
- Les ingénieurs dès leur premier jour de travail pouvaient voir à quoi ressemblait le travail quotidien et comment il était effectué.
- Les gens étaient plus enclins à demander de l'aide lorsqu'ils voyaient d'autres s'entraider.
- L'apprentissage organisationnel rapide était facilité et accumulé.

De plus, au-delà des avantages testés ci-dessus, les salles de discussion enregistrent et rendent toutes les communications publiques par nature ; en revanche, les courriels sont privés par défaut, et les informations qu'ils contiennent ne peuvent pas être facilement découvertes ou propagées au sein d'une organisation.

Intégrer notre automatisation dans les salles de discussion aide à documenter et partager nos observations et nos résolutions de problèmes comme une partie inhérente à l'exécution de notre travail. Cela renforce une culture de transparence et de collaboration dans tout ce que nous faisons. C'est également un moyen extrêmement efficace de convertir les apprentissages locaux en connaissances globales. Chez GitHub, tout le personnel des opérations travaillait à distance - en fait, aucun des ingénieurs ne travaillait dans la même ville. Comme le rappelle Mark Imbriaco, ancien VP des opérations chez GitHub, « Il n'y avait pas de point de rencontre physique chez GitHub. La salle de discussion était le point de rencontre. »

GitHub a permis à Hubot de déclencher leurs technologies d'automatisation, notamment Puppet, Capistrano, Jenkins, resque (une bibliothèque supportée par Redis pour créer des tâches en arrière-plan), et graphme (qui génère des graphiques à partir de Graphite). Les actions effectuées via Hubot incluaient la vérification de la santé des services, l'exécution de push Puppet ou de déploiements de code en production, et la mise en sourdine des alertes lorsque les services entraient en mode maintenance. Les actions effectuées plusieurs fois, telles que la consultation des journaux de tests de fumée lorsqu'un déploiement échouait, la mise hors rotation des serveurs de production, le retour à la version principale pour les services frontaux de production, ou même les excuses aux ingénieurs de garde, devenaient également des actions Hubot.

De même, les commits dans le dépôt de code source et les commandes qui déclenchent les déploiements en production émettent tous deux des messages dans la salle de discussion. De plus, à mesure que les changements progressent dans le pipeline de déploiement, leur statut est publié dans la salle de discussion.

Un échange typique dans une salle de discussion rapide pourrait ressembler à ceci :

« @sr : @jnewland, comment obtenir cette liste de gros dépôts ? disk_hogs ou quelque chose comme ça ? »

« @jnewland : /disk-hogs »

Newland observe que certaines questions qui étaient auparavant posées au cours d'un projet ne le sont plus. Par exemple, les ingénieurs peuvent se demander entre eux, « Comment se passe ce déploiement ? » ou « Est-ce que tu déploies ça, ou devrais-je le faire ? » ou « À quoi ressemble la charge ? »

Parmi tous les avantages que Newland décrit, y compris une intégration plus rapide des nouveaux ingénieurs et une augmentation de la productivité de tous les ingénieurs, le résultat

qu'il considère comme le plus important est que le travail des Ops est devenu plus humain, car les ingénieurs Ops peuvent découvrir les problèmes et s'entraider rapidement et facilement.

GitHub a créé un environnement propice à l'apprentissage collaboratif local qui pouvait être transformé en apprentissages à travers toute l'organisation. Tout au long de ce chapitre, nous explorerons des moyens de créer et d'accélérer la diffusion des nouveaux apprentissages organisationnels.

Automatiser les processus standardisés dans les logiciels pour la réutilisation

Trop souvent, nous codifions nos normes et processus pour l'architecture, les tests, le déploiement et la gestion de l'infrastructure sous forme de texte, les stockant dans des documents Word qui sont téléchargés quelque part. Le problème est que les ingénieurs qui construisent de nouvelles applications ou environnements ne savent souvent pas que ces documents existent, ou ils n'ont pas le temps de mettre en œuvre les normes documentées. Le résultat est qu'ils créent leurs propres outils et processus, avec tous les résultats décevants que l'on peut attendre : des applications et environnements fragiles, non sécurisés et non maintenables qui sont coûteux à exécuter, à maintenir et à faire évoluer.

Au lieu de mettre notre expertise dans des documents Word, nous devons transformer ces normes et processus documentés, qui englobent la somme de nos apprentissages et connaissances organisationnelles, en une forme exécutable qui les rend plus faciles à réutiliser. L'une des meilleures façons de rendre ces connaissances réutilisables est de les intégrer dans un dépôt de code source centralisé, rendant l'outil disponible pour que tout le monde puisse le rechercher et l'utiliser.

Justin Arbuckle était architecte en chef chez GE Capital en 2013 lorsqu'il a dit : « Nous avions besoin de créer un mécanisme qui permettrait aux équipes de se conformer facilement aux politiques—réglementations nationales, régionales et sectorielles à travers des dizaines de cadres réglementaires, couvrant des milliers d'applications fonctionnant sur des dizaines de milliers de serveurs dans des dizaines de centres de données. »

Le mécanisme qu'ils ont créé s'appelait ArchOps, qui « permettait à nos ingénieurs d'être des constructeurs, pas des maçons. En mettant nos normes de conception dans des plans automatisés qui pouvaient être facilement utilisés par n'importe qui, nous avons obtenu de la cohérence comme sous-produit. »

En encodant nos processus manuels dans du code automatisé et exécuté, nous permettons au processus d'être largement adopté, offrant de la valeur à tous ceux qui les utilisent. Arbuckle a

conclu que « la conformité réelle d'une organisation est proportionnelle au degré auquel ses politiques sont exprimées sous forme de code. »

En faisant de ce processus automatisé le moyen le plus simple d'atteindre l'objectif, nous permettons aux pratiques d'être largement adoptées—nous pouvons même envisager de les transformer en services partagés soutenus par l'organisation.

Créer un dépôt de code source unique et partagé pour toute notre organisation

Un dépôt de code source partagé à l'échelle de l'entreprise est l'un des mécanismes les plus puissants pour intégrer les découvertes locales à travers toute l'organisation. Lorsque nous mettons à jour quoi que ce soit dans le dépôt de code source (par exemple, une bibliothèque partagée), cela se propage rapidement et automatiquement à tous les autres services qui utilisent cette bibliothèque, et est intégré via le pipeline de déploiement de chaque équipe.

Google est l'un des plus grands exemples d'utilisation d'un dépôt de code source partagé à l'échelle de l'organisation. En 2015, Google avait un dépôt de code source unique avec plus d'un milliard de fichiers et plus de deux milliards de lignes de code. Ce dépôt est utilisé par chacun de leurs vingt-cinq mille ingénieurs et couvre toutes les propriétés de Google, y compris Google Search, Google Maps, Google Docs, Google+, Google Calendar, Gmail et YouTube.

L'un des résultats précieux de cela est que les ingénieurs peuvent tirer parti de l'expertise diversifiée de tout le monde dans l'organisation. Rachel Potvin, une responsable de l'ingénierie chez Google supervisant le groupe Developer Infrastructure, a déclaré à Wired que chaque ingénieur de Google peut accéder à "une richesse de bibliothèques" parce que "presque tout a déjà été fait".

De plus, comme l'explique Eran Messeri, un ingénieur du groupe Developer Infrastructure de Google, l'un des avantages d'utiliser un dépôt unique est qu'il permet aux utilisateurs d'accéder facilement à tout le code dans sa forme la plus à jour, sans besoin de coordination.

Nous mettons dans notre dépôt de code source partagé non seulement le code source, mais aussi d'autres artefacts qui encodent les connaissances et les apprentissages, y compris :

- Normes de configuration pour nos bibliothèques, infrastructures et environnements (recettes Chef, manifestes Puppet, etc.)
- Outils de déploiement
- Normes et outils de test, y compris la sécurité
- Outils de pipeline de déploiement

- Outils de surveillance et d'analyse
- Tutoriels et normes

Encoder les connaissances et les partager à travers ce dépôt est l'un des mécanismes les plus puissants que nous ayons pour propager les connaissances. Comme le décrit Randy Shoup, "Le mécanisme le plus puissant pour prévenir les échecs chez Google est le dépôt de code unique. Chaque fois que quelqu'un enregistre quoi que ce soit dans le dépôt, cela se traduit par une nouvelle construction, qui utilise toujours la dernière version de tout. Tout est construit à partir du code source plutôt que lié dynamiquement au moment de l'exécution - il y a toujours une seule version d'une bibliothèque qui est la version actuelle utilisée, qui est ce qui est lié statiquement pendant le processus de construction."

Tom Limoncelli est le co-auteur de *The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems* et un ancien ingénieur de fiabilité des sites chez Google. Dans son livre, il déclare que la valeur d'avoir un dépôt unique pour toute une organisation est si puissante qu'il est difficile de l'expliquer.

Vous pouvez écrire un outil une seule fois et le rendre utilisable pour tous les projets. Vous avez une connaissance exacte à 100% de qui dépend d'une bibliothèque ; par conséquent, vous pouvez la refactoriser et être sûr à 100% de qui sera affecté et qui doit tester pour détecter les ruptures. Je pourrais probablement lister cent autres exemples. Je ne peux pas exprimer en mots combien cela représente un avantage concurrentiel pour Google.

Chez Google, chaque bibliothèque (par exemple, libc, OpenSSL, ainsi que les bibliothèques développées en interne comme les bibliothèques de threading Java) a un propriétaire qui est responsable de s'assurer que la bibliothèque non seulement se compile, mais passe également avec succès les tests pour tous les projets qui en dépendent, un peu comme un bibliothécaire dans le monde réel. Ce propriétaire est également responsable de la migration de chaque projet d'une version à l'autre.

Considérez l'exemple réel d'une organisation qui exécute quatre-vingt-un différentes versions de la bibliothèque Java Struts en production - toutes sauf une de ces versions ont des vulnérabilités de sécurité critiques, et maintenir toutes ces versions, chacune avec ses propres particularités et idiosyncrasies, crée un fardeau opérationnel et du stress significatifs. De plus, toute cette variance rend les mises à jour de version risquées et dangereuses, ce qui à son tour décourage les développeurs de faire des mises à jour. Et le cycle continue.

Le dépôt de source unique résout une grande partie de ce problème, tout en ayant des tests automatisés qui permettent aux équipes de migrer vers de nouvelles versions de manière sûre et confiante.

Si nous ne pouvons pas tout construire à partir d'un seul arbre de source, nous devons trouver un autre moyen de maintenir des versions connues et bonnes des bibliothèques et de leurs dépendances. Par exemple, nous pouvons avoir un dépôt d'organisation tel que Nexus, Artifactory, ou un dépôt Debian ou RPM, que nous devons alors mettre à jour en cas de vulnérabilités connues, à la fois dans ces dépôts et dans les systèmes de production.

Diffuser les connaissances en utilisant les tests automatisés comme documentation et les communautés de pratique

Lorsque nous avons des bibliothèques partagées utilisées dans toute l'organisation, nous devrions permettre une propagation rapide de l'expertise et des améliorations. S'assurer que chacune de ces bibliothèques inclut une quantité significative de tests automatisés signifie que ces bibliothèques deviennent auto-documentantes et montrent aux autres ingénieurs comment les utiliser. Cet avantage sera presque automatique si nous avons des pratiques de développement piloté par les tests (TDD) en place, où les tests automatisés sont écrits avant que nous n'écrivions le code. Cette discipline transforme nos suites de tests en une spécification vivante et à jour du système. Tout ingénieur souhaitant comprendre comment utiliser le système peut consulter la suite de tests pour trouver des exemples fonctionnels de l'utilisation de l'API du système.

Idéalement, chaque bibliothèque aura un seul propriétaire ou une seule équipe de support, représentant où se trouvent les connaissances et l'expertise sur la bibliothèque. De plus, nous devrions (idéalement) permettre l'utilisation d'une seule version en production, garantissant que ce qui est en production tire parti des meilleures connaissances collectives de l'organisation.

Dans ce modèle, le propriétaire de la bibliothèque est également responsable de la migration sécurisée de chaque groupe utilisant le dépôt d'une version à l'autre. Cela nécessite à son tour une détection rapide des erreurs de régression grâce à des tests automatisés complets et une intégration continue pour tous les systèmes utilisant la bibliothèque.

Afin de propager plus rapidement les connaissances, nous pouvons également créer des groupes de discussion ou des salles de chat pour chaque bibliothèque ou service, afin que toute personne ayant des questions puisse obtenir des réponses d'autres utilisateurs, souvent plus rapides à répondre que les développeurs.

En utilisant ce type d'outil de communication au lieu d'avoir des poches isolées d'expertise réparties dans toute l'organisation, nous facilitons un échange de connaissances et d'expériences, en nous assurant que les travailleurs peuvent s'aider mutuellement avec des problèmes et de nouveaux modèles.

Concevoir pour les opérations à travers des exigences non fonctionnelles codifiées

Lorsque le développement suit son travail en aval et participe aux activités de résolution d'incidents en production, l'application devient de mieux en mieux conçue pour les opérations. De plus, à mesure que nous commençons à concevoir délibérément notre code et notre application pour qu'ils puissent accueillir un flux rapide et une déployabilité, nous identifierons probablement un ensemble d'exigences non fonctionnelles que nous voudrons intégrer dans tous nos services de production.

La mise en œuvre de ces exigences non fonctionnelles permettra à nos services d'être faciles à déployer et à maintenir en production, où nous pouvons détecter et corriger rapidement les problèmes, et garantir qu'ils se dégradent de manière gracieuse lorsque des composants échouent.

Des exemples d'exigences non fonctionnelles incluent la garantie d'avoir :

- Une télémétrie de production suffisante dans nos applications et environnements
- La capacité de suivre précisément les dépendances
- Des services résilients et se dégradant de manière gracieuse
- Une compatibilité avant et arrière entre les versions
- La capacité d'archiver les données pour gérer la taille de l'ensemble de données de production
- La capacité de rechercher et comprendre facilement les messages de journal à travers les services
- La capacité de tracer les demandes des utilisateurs à travers plusieurs services
- Une configuration d'exécution simple et centralisée en utilisant des indicateurs de fonctionnalités, etc.

En codifiant ces types d'exigences non fonctionnelles, nous facilitons l'utilisation des connaissances et de l'expérience collectives de l'organisation pour tous nos services nouveaux et existants. Ce sont toutes des responsabilités de l'équipe construisant le service.

Construire des histoires utilisateurs d'opérations réutilisables dans le développement

Lorsqu'il y a du travail opérationnel qui ne peut pas être entièrement automatisé ou rendu en libre-service, notre objectif est de rendre ce travail récurrent aussi reproductible et déterministe que possible. Nous le faisons en standardisant le travail nécessaire, en automatisant autant que possible, et en documentant notre travail afin de permettre aux équipes produits de mieux planifier et allouer les ressources nécessaires à cette activité.

Plutôt que de construire manuellement des serveurs et de les mettre en production selon des listes de contrôle manuelles, nous devrions automatiser autant que possible ce travail. Là où certaines étapes ne peuvent pas être automatisées (par exemple, l'installation manuelle d'un serveur et sa connexion par une autre équipe), nous devrions définir collectivement les transferts aussi clairement que possible pour réduire les délais et les erreurs. Cela nous permettra également de mieux planifier et programmer ces étapes à l'avenir. Par exemple, nous pouvons utiliser des outils tels que Rundeck pour automatiser et exécuter des workflows, ou des systèmes de ticket comme JIRA ou ServiceNow.

Idéalement, pour tout notre travail opérationnel récurrent, nous devrions connaître ce qui est nécessaire, qui est nécessaire pour le réaliser, quelles sont les étapes pour le terminer, et ainsi de suite. Par exemple, "Nous savons qu'un déploiement à haute disponibilité comporte quatorze étapes, nécessitant du travail de la part de quatre équipes différentes, et les cinq dernières fois que nous l'avons fait, cela a pris en moyenne trois jours."

Tout comme nous créons des histoires utilisateurs dans le développement que nous plaçons dans le backlog et que nous tirons pour travailler dessus, nous pouvons créer des "histoires utilisateurs d'Ops" bien définies qui représentent des activités de travail pouvant être réutilisées dans tous nos projets (par exemple, déploiement, capacité, sécurité, etc.). En créant ces histoires utilisateurs d'Ops bien définies, nous exposons le travail opérationnel informatique répétable d'une manière qui apparaît aux côtés du travail de développement, permettant une meilleure planification et des résultats plus reproductibles.

Veiller à ce que les choix techniques aident à atteindre les objectifs opérationnels

Lorsque l'un de nos objectifs est de maximiser la productivité des développeurs et que nous avons des architectures orientées service, de petites équipes de service peuvent potentiellement construire et gérer leur service dans le langage ou le framework qui répond le mieux à leurs besoins spécifiques. Dans certains cas, c'est ce qui nous permet le mieux d'atteindre nos objectifs organisationnels.

Cependant, il y a des scénarios où le contraire se produit, comme lorsque l'expertise pour un service critique réside uniquement dans une équipe, et que seule cette équipe peut apporter des modifications ou résoudre des problèmes, créant ainsi un goulet d'étranglement. En d'autres termes, nous avons peut-être optimisé la productivité des équipes mais entravé involontairement la réalisation des objectifs organisationnels.

Cela se produit souvent lorsque nous avons un groupe d'opérations orienté fonctionnellement qui est responsable de tous les aspects du support des services. Dans ces scénarios, pour nous assurer de valoriser les compétences approfondies dans des technologies spécifiques, nous

voulons nous assurer que les opérations peuvent influencer le choix des composants utilisés en production, ou leur donner la possibilité de ne pas être responsables des plateformes non prises en charge.

Si nous n'avons pas de liste des technologies que les opérations prendront en charge, générée collectivement par le développement et les opérations, nous devrions systématiquement passer en revue l'infrastructure de production et les services, ainsi que toutes leurs dépendances actuellement prises en charge, pour identifier celles qui créent une demande de défaillance disproportionnée et un travail non planifié. Notre objectif est d'identifier les technologies qui :

- Entravent ou ralentissent le flux de travail
- Crètent de manière disproportionnée des niveaux élevés de travail non planifié
- Crètent de manière disproportionnée un grand nombre de demandes de support
- Sont les moins conformes à nos résultats architecturaux souhaités (par exemple, débit, stabilité, sécurité, fiabilité, continuité des activités)

En éliminant ces infrastructures et plates-formes problématiques de la liste des technologies prises en charge par les opérations, nous leur permettons de se concentrer sur l'infrastructure qui aide le mieux à atteindre les objectifs globaux de l'organisation.

Comme le décrit Tom Limoncelli, "Lorsque j'étais chez Google, nous avions un langage compilé officiel, un langage de script officiel, et un langage d'interface utilisateur officiel. Oui, d'autres langages étaient supportés d'une manière ou d'une autre, mais rester avec 'les trois grands' signifiait des bibliothèques de support, des outils, et une manière plus facile de trouver des collaborateurs."

Ces normes étaient également renforcées par le processus de revue de code, ainsi que par les langages supportés par leurs plates-formes internes.

Dans une présentation qu'il a donnée avec Olivier Jacques et Rafael Garcia lors du DevOps Enterprise Summit 2015, Ralph Loura, CIO de HP, a déclaré :

"En interne, nous avons décrit notre objectif comme celui de créer des 'balises, pas des frontières'. Au lieu de tracer des frontières strictes que tout le monde doit respecter, nous mettons en place des balises qui indiquent les zones profondes du canal où vous êtes en sécurité et soutenu. Vous pouvez dépasser les balises tant que vous suivez les principes organisationnels. Après tout, comment pouvons-nous jamais découvrir la prochaine innovation qui nous aide à gagner si nous n'explorons pas et ne testons pas aux limites ?

En tant que leaders, nous devons naviguer dans le canal, le marquer, et permettre aux gens d'explorer au-delà."

ÉTUDE DE CAS - Standardisation d'une nouvelle pile technologique chez Etsy (2010)

Dans de nombreuses organisations adoptant DevOps, une histoire courante que racontent les développeurs est : "Ops ne nous fournissait pas ce dont nous avions besoin, alors nous l'avons simplement construit et soutenu nous-mêmes." Cependant, aux premiers stades de la transformation d'Etsy, la direction technologique a adopté une approche opposée, réduisant considérablement le nombre de technologies prises en charge en production.

En 2010, après une saison des fêtes presque désastreuse, l'équipe d'Etsy a décidé de réduire massivement le nombre de technologies utilisées en production, choisissant quelques-unes que toute l'organisation pourrait pleinement soutenir et éradiquant les autres.

Leur objectif était de standardiser et de réduire très délibérément l'infrastructure et les configurations prises en charge. Une des premières décisions a été de migrer toute la plateforme d'Etsy vers PHP et MySQL. Il s'agissait principalement d'une décision philosophique plutôt que technologique : ils voulaient que Dev et Ops puissent comprendre entièrement la pile technologique afin que tout le monde puisse contribuer à une plateforme unique, ainsi que permettre à chacun de lire, réécrire et corriger le code des autres. Au cours des années suivantes, comme le rappelle Michael Rembetsy, qui était directeur des opérations d'Etsy à l'époque : "Nous avons retiré certaines technologies majeures, les retirant complètement de la production", y compris lighttpd, Postgres, MongoDB, Scala, CoffeeScript, Python, et bien d'autres.

De manière similaire, Dan McKinley, développeur dans l'équipe fonctionnelle qui a introduit MongoDB chez Etsy en 2010, écrit sur son blog que tous les avantages d'avoir une base de données sans schéma ont été annulés par tous les problèmes opérationnels que l'équipe a dû résoudre. Cela incluait des problèmes de journalisation, de création de graphiques, de surveillance, de télémétrie en production, de sauvegarde et de restauration, ainsi que de nombreux autres problèmes auxquels les développeurs n'ont généralement pas besoin de se préoccuper. Le résultat a été d'abandonner MongoDB, en portant le nouveau service pour utiliser l'infrastructure de base de données MySQL déjà prise en charge.

Conclusion

Les techniques décrites dans ce chapitre permettent à chaque nouvel apprentissage d'être incorporé dans le savoir collectif de l'organisation, multipliant ainsi son effet. Nous y parvenons en communiquant activement et largement les nouvelles connaissances, par exemple à travers des salles de discussion et des technologies telles que l'architecture en tant que code, les référentiels de code source partagés, la standardisation des technologies, et ainsi de suite. En faisant cela, nous élevons l'état de la pratique non seulement de Dev et Ops, mais aussi de toute l'organisation, de sorte que chacun qui effectue un travail le fasse avec l'expérience cumulative de toute l'organisation.

Réserver du temps pour créer un apprentissage et une amélioration organisationnels

Une des pratiques qui fait partie du système de production Toyota s'appelle le blitz d'amélioration (ou parfois un blitz kaizen), défini comme une période dédiée et concentrée pour aborder un problème particulier, souvent sur plusieurs jours. Le Dr Spear explique : «... les blitz prennent souvent cette forme : un groupe est rassemblé pour se concentrer intensément sur un processus avec des problèmes... Le blitz dure quelques jours, l'objectif est l'amélioration du processus, et les moyens sont l'utilisation concentrée de personnes extérieures au processus pour conseiller celles normalement à l'intérieur du processus. »

Spear observe que le résultat de l'équipe de blitz d'amélioration sera souvent une nouvelle approche pour résoudre un problème, comme de nouvelles dispositions d'équipements, de nouveaux moyens de transporter les matériaux et les informations, un espace de travail plus organisé ou un travail standardisé. Ils peuvent également laisser une liste de choses à faire de changements à apporter ultérieurement.

Un exemple de blitz d'amélioration DevOps est le programme de défi mensuel au Dojo DevOps de Target. Ross Clanton, directeur des opérations chez Target, est responsable de l'accélération de l'adoption de DevOps. Un de ses principaux mécanismes pour cela est le Centre d'Innovation Technologique, plus connu sous le nom de Dojo DevOps.

Le Dojo DevOps occupe environ dix-huit mille pieds carrés d'espace de bureau ouvert, où des coachs DevOps aident des équipes de toute l'organisation technologique de Target à éléver leur pratique. Le format le plus intensif est ce qu'ils appellent les "Défis de 30 jours", où des équipes de développement internes viennent pendant un mois et travaillent ensemble avec des coachs et des ingénieurs dédiés du Dojo. L'équipe apporte son travail avec elle, dans le but de résoudre un problème interne avec lequel elle a du mal et de créer une percée en trente jours.

Pendant les trente jours, ils travaillent intensivement avec les coachs du Dojo sur le problème - planifiant, travaillant et faisant des démonstrations en sprints de deux jours. Lorsque le Défi de 30 jours est terminé, les équipes internes retournent à leurs lignes de métier, non seulement en ayant résolu un problème significatif, mais en ramenant leurs nouvelles connaissances à leurs équipes.

Clanton décrit : « Nous avons actuellement la capacité d'avoir huit équipes faisant des Défis de 30 jours en même temps, nous nous concentrerons donc sur les projets les plus stratégiques de l'organisation. Jusqu'à présent, certaines de nos capacités les plus critiques sont passées par le Dojo, y compris des équipes de Point de Vente (POS), Inventaire, Tarification et Promotion. »

En ayant du personnel du Dojo affecté à plein temps et en se concentrant sur un seul objectif, les équipes qui suivent un Défi de 30 jours font des améliorations incroyables.

Ravi Pandey, un responsable de développement chez Target qui a suivi ce programme, explique : « Autrefois, nous devions attendre six semaines pour obtenir un environnement de test. Maintenant, nous l'obtenons en quelques minutes, et nous travaillons côté à côté avec des ingénieurs Ops qui nous aident à augmenter notre productivité et à construire des outils pour nous aider à atteindre nos objectifs. » Clanton développe cette idée : « Il n'est pas rare que les équipes accomplissent en quelques jours ce qui leur prendrait habituellement trois à six mois. Jusqu'à présent, deux cents apprenants sont passés par le Dojo, ayant terminé quatorze défis. »

Le Dojo soutient également des modèles d'engagement moins intensifs, y compris les Flash Builds, où des équipes se réunissent pour des événements d'un à trois jours, avec l'objectif de livrer un produit minimal viable (MVP) ou une capacité à la fin de l'événement. Ils organisent également des Open Labs toutes les deux semaines, où n'importe qui peut visiter le Dojo pour parler aux coachs du Dojo, assister à des démonstrations ou recevoir une formation.

Dans ce chapitre, nous décrirons cela et d'autres façons de réserver du temps pour l'apprentissage et l'amélioration organisationnels, institutionnalisant davantage la pratique de consacrer du temps à l'amélioration du travail quotidien.

Institutionnaliser des rituels pour rembourser la dette technique

Dans cette section, nous planifions des rituels qui aident à renforcer la pratique de réserver du temps pour Dev et Ops pour le travail d'amélioration, tels que les exigences non fonctionnelles, l'automatisation, etc. L'une des façons les plus simples de le faire est de programmer et de mener des blitz d'amélioration d'un jour ou d'une semaine, où tout le monde dans une équipe (ou dans toute l'organisation) s'auto-organise pour résoudre les problèmes qui les concernent - aucun travail de fonctionnalité n'est autorisé. Cela pourrait être une zone problématique du code, de l'environnement, de l'architecture, des outils, et ainsi de suite. Ces équipes couvrent l'ensemble du flux de valeur, combinant souvent des ingénieurs de développement, d'opérations et de sécurité de l'information. Les équipes qui ne travaillent généralement pas ensemble combinent leurs compétences et leurs efforts pour améliorer une zone choisie et montrent ensuite leur amélioration au reste de l'entreprise.

En plus des termes orientés Lean comme kaizen blitz et blitz d'amélioration, la technique des rituels dédiés au travail d'amélioration a également été appelée nettoyages de printemps ou d'automne et semaines d'inversion des files d'attente de tickets. D'autres termes ont également été utilisés, comme les jours de hack, les hackathons et le temps d'innovation de 20%. Malheureusement, ces rituels spécifiques se concentrent parfois sur l'innovation produit et le prototypage de nouvelles idées de marché, plutôt que sur le travail d'amélioration, et pire, ils

sont souvent réservés aux développeurs - ce qui est considérablement différent des objectifs d'un blitz d'amélioration.

Notre objectif pendant ces blitz n'est pas simplement d'expérimenter et d'innover pour tester de nouvelles technologies, mais d'améliorer notre travail quotidien, comme résoudre nos contournements quotidiens. Bien que les expériences puissent également mener à des améliorations, les blitz d'amélioration sont très concentrés sur la résolution de problèmes spécifiques que nous rencontrons dans notre travail quotidien.

Nous pouvons programmer des blitz d'amélioration d'une semaine qui priorisent Dev et Ops travaillant ensemble vers des objectifs d'amélioration. Ces blitz d'amélioration sont simples à administrer : une semaine est sélectionnée où tout le monde dans l'organisation technologique travaille sur une activité d'amélioration en même temps. À la fin de la période, chaque équipe fait une présentation à ses pairs qui discute du problème qu'ils abordaient et de ce qu'ils ont construit. Cette pratique renforce une culture dans laquelle les ingénieurs travaillent sur l'ensemble du flux de valeur pour résoudre les problèmes. De plus, elle renforce la résolution des problèmes comme partie intégrante de notre travail quotidien et démontre que nous valorisons le remboursement de la dette technique.

Ce qui rend les blitz d'amélioration si puissants, c'est que nous donnons le pouvoir à ceux qui sont les plus proches du travail d'identifier et de résoudre continuellement leurs propres problèmes. Considérons un instant que notre système complexe est comme une toile d'araignée, avec des brins entrelacés qui s'affaiblissent et se brisent constamment. Si la bonne combinaison de brins se brise, toute la toile s'effondre. Il n'y a pas de gestion de commande et de contrôle qui puisse diriger les travailleurs pour réparer chaque brin un par un. Au lieu de cela, nous devons créer la culture organisationnelle et les normes qui conduisent tout le monde à trouver et réparer continuellement les brins cassés dans le cadre de notre travail quotidien. Comme l'observe le Dr Spear, « Pas étonnant alors que les araignées réparent les déchirures et les trous dans la toile dès qu'ils se produisent, sans attendre que les échecs s'accumulent. »

Un excellent exemple du succès du concept de blitz d'amélioration est décrit par Mark Zuckerberg, PDG de Facebook. Dans une interview avec Jessica Stillman de Inc.com, il dit, « Tous les quelques mois, nous avons un hackathon, où tout le monde crée des prototypes pour de nouvelles idées qu'ils ont. À la fin, toute l'équipe se réunit et examine tout ce qui a été construit. Beaucoup de nos produits les plus réussis sont issus de hackathons, y compris Timeline, le chat, la vidéo, notre cadre de développement mobile et certaines de nos infrastructures les plus importantes comme le compilateur HipHop. »

Un exemple particulièrement intéressant est le compilateur HipHop PHP. En 2008, Facebook faisait face à des problèmes de capacité significatifs, avec plus de cent millions d'utilisateurs actifs et une croissance rapide, créant d'énormes problèmes pour toute l'équipe d'ingénierie. Pendant un jour de hack, Haiping Zhao, ingénieur principal des serveurs chez Facebook, a

commencé à expérimenter la conversion du code PHP en code C++ compilable, avec l'espoir d'augmenter significativement la capacité de leur infrastructure existante. Au cours des deux années suivantes, une petite équipe a été constituée pour construire ce qui est devenu connu sous le nom de compilateur HipHop, convertissant tous les services de production de Facebook du PHP interprété en binaires C++ compilés. HipHop a permis à la plateforme de Facebook de gérer des charges de production six fois plus élevées que le PHP natif.

Dans une interview avec Cade Metz de Wired, Drew Paroski, l'un des ingénieurs qui a travaillé sur le projet, a noté, « Il y a eu un moment où, si HipHop n'avait pas été là, nous aurions été dans de beaux draps. Nous aurions probablement eu besoin de plus de machines pour servir le site que nous n'aurions pu obtenir à temps. C'était une tentative désespérée qui a fonctionné. »

Plus tard, Paroski et ses collègues ingénieurs Keith Adams et Jason Evans ont décidé qu'ils pouvaient surpasser les performances de l'effort du compilateur HipHop et réduire certaines de ses limitations qui diminuaient la productivité des développeurs. Le projet résultant était le projet de machine virtuelle HipHop (« HHVM »), adoptant une approche de compilation à la volée. En 2012, HHVM avait complètement remplacé le compilateur HipHop en production, avec près de vingt ingénieurs contribuant au projet.

En réalisant des blitz d'amélioration et des semaines de hack régulièrement programmés, nous permettons à tout le monde dans le flux de valeur de prendre fierté et responsabilité dans les innovations qu'ils créent, et nous intégrons continuellement des améliorations dans notre système, permettant ainsi plus de sécurité, de fiabilité et d'apprentissage.

Permettre à chacun d'enseigner et d'apprendre

Une culture dynamique d'apprentissage crée des conditions pour que tout le monde puisse non seulement apprendre, mais aussi enseigner, que ce soit par des méthodes didactiques traditionnelles (par exemple, suivre des cours, assister à des formations) ou des méthodes plus expérientielles ou ouvertes (par exemple, des conférences, des ateliers, du mentorat). Une façon de favoriser cet enseignement et cet apprentissage est de consacrer du temps organisationnel à cela.

Steve Farley, vice-président des technologies de l'information chez Nationwide Insurance, a déclaré : « Nous avons cinq mille professionnels de la technologie, que nous appelons 'associés'. Depuis 2011, nous nous engageons à créer une culture d'apprentissage - une partie de cela est quelque chose que nous appelons Teaching Thursday, où chaque semaine, nous consacrons du temps pour que nos associés apprennent. Pendant deux heures, chaque associé est censé enseigner ou apprendre. Les sujets sont ceux que nos associés veulent apprendre - certains concernent la technologie, les nouvelles techniques de développement logiciel ou d'amélioration des processus, ou même la manière de mieux gérer leur carrière. La

chose la plus précieuse que tout associé puisse faire est de mentor ou apprendre d'autres associés. »

Comme cela a été démontré tout au long de ce livre, certaines compétences deviennent de plus en plus nécessaires pour tous les ingénieurs, pas seulement pour les développeurs. Par exemple, il devient plus important pour tous les ingénieurs en opérations et en test de se familiariser avec les techniques, rituels et compétences de développement, tels que le contrôle de version, les tests automatisés, les pipelines de déploiement, la gestion de configuration et la création d'automatisation. La familiarité avec les techniques de développement aide les ingénieurs en opérations à rester pertinents à mesure que de plus en plus de flux de valeur technologique adoptent les principes et les modèles DevOps.

Bien que la perspective d'apprendre quelque chose de nouveau puisse être intimidante ou provoquer un sentiment d'embarras ou de honte, elle ne devrait pas l'être. Après tout, nous sommes tous des apprenants à vie, et l'un des meilleurs moyens d'apprendre est de nos pairs. Karthik Gaekwad, qui a fait partie de la transformation DevOps de National Instruments, a déclaré : « Pour les personnes des opérations qui essaient d'apprendre l'automatisation, cela ne devrait pas être effrayant - demandez simplement à un développeur sympathique, car il serait ravi d'aider. »

Nous pouvons encore aider à enseigner des compétences à travers notre travail quotidien en réalisant conjointement des revues de code qui incluent les deux parties afin que nous apprenions en faisant, ainsi qu'en faisant travailler ensemble le développement et les opérations pour résoudre de petits problèmes. Par exemple, nous pourrions demander au développement de montrer aux opérations comment authentifier une application, se connecter et exécuter des tests automatisés sur diverses parties de l'application pour s'assurer que les composants critiques fonctionnent correctement (par exemple, la fonctionnalité clé de l'application, les transactions de base de données, les files de messages). Nous intégrerions ensuite ce nouveau test automatisé dans notre pipeline de déploiement et l'exécuterions périodiquement, en envoyant les résultats à nos systèmes de surveillance et d'alerte afin d'obtenir une détection précoce lorsque des composants critiques échouent.

Comme Glenn O'Donnell de Forrester Research l'a plaisanté dans sa présentation au DevOps Enterprise Summit de 2014, « Pour tous les professionnels de la technologie qui aiment innover, qui aiment le changement, il y a un avenir merveilleux et vibrant devant nous. »

Partagez vos expériences des conférences DevOps

Dans de nombreuses organisations axées sur les coûts, les ingénieurs sont souvent découragés d'assister à des conférences et d'apprendre de leurs pairs. Pour aider à bâtir une organisation apprenante, nous devrions encourager nos ingénieurs (tant du Développement que des

Opérations) à assister à des conférences, à y donner des conférences et, si nécessaire, à créer et organiser eux-mêmes des conférences internes ou externes.

DevOpsDays reste aujourd'hui l'une des séries de conférences auto-organisées les plus dynamiques. De nombreuses pratiques DevOps ont été partagées et propagées lors de ces événements. Elles sont restées gratuites ou presque gratuites, soutenues par une communauté dynamique de praticiens et de vendeurs.

Le DevOps Enterprise Summit a été créé en 2014 pour que les leaders technologiques partagent leurs expériences d'adoption des principes et pratiques DevOps dans des organisations grandes et complexes. Le programme est principalement organisé autour de rapports d'expérience des leaders technologiques sur le parcours DevOps, ainsi que d'experts en la matière sur des sujets sélectionnés par la communauté.

Étude de cas - Conférences technologiques internes chez Nationwide Insurance, Capital One et Target (2014)

En plus d'assister à des conférences externes, de nombreuses entreprises, y compris celles décrites dans cette section, organisent des conférences internes pour leurs employés en technologie.

Nationwide Insurance est un fournisseur leader de services d'assurance et financiers, opérant dans des industries fortement réglementées. Leurs nombreuses offres incluent l'assurance auto et habitation, et ils sont le principal fournisseur de plans de retraite pour le secteur public et d'assurance pour animaux de compagnie. En 2014, ils géraient 195 milliards de dollars d'actifs, avec 24 milliards de dollars de revenus. Depuis 2005, Nationwide adopte les principes Agile et Lean pour améliorer les pratiques de leurs cinq mille professionnels de la technologie, permettant ainsi l'innovation de terrain.

Steve Farley, vice-président des technologies de l'information, se souvient : « Des conférences technologiques passionnantes commençaient à apparaître à cette époque, comme la conférence nationale Agile. En 2011, la direction technologique de Nationwide a convenu que nous devrions créer une conférence technologique, appelée TechCon. En organisant cet événement, nous voulions créer une meilleure manière de nous former nous-mêmes, ainsi que garantir que tout avait un contexte Nationwide, contrairement à envoyer tout le monde à une conférence externe. »

Capital One, l'une des plus grandes banques des États-Unis avec plus de 298 milliards de dollars d'actifs et 24 milliards de dollars de revenus en 2015, a organisé sa première conférence interne de génie logiciel en 2015 dans le but de créer une organisation technologique de classe

mondiale. La mission était de promouvoir une culture de partage et de collaboration, de bâtir des relations entre les professionnels de la technologie et de permettre l'apprentissage. La conférence comptait treize pistes d'apprentissage et cinquante-deux sessions, et plus de 1 200 employés internes y ont assisté.

Le Dr Tapabrata Pal, fellow technique chez Capital One et l'un des organisateurs de l'événement, décrit : « Nous avions même un hall d'exposition, où nous avions vingt-huit stands où des équipes internes de Capital One présentaient toutes les capacités étonnantes sur lesquelles elles travaillaient. Nous avons même décidé très délibérément qu'il n'y aurait pas de vendeurs, car nous voulions maintenir l'accent sur les objectifs de Capital One. »

Target est le sixième plus grand détaillant aux États-Unis, avec 72 milliards de dollars de revenus en 2014 et 1 799 magasins de détail et 347 000 employés dans le monde. Heather Mickman, directrice du Développement, et Ross Clanton ont organisé six événements internes DevOpsDays depuis 2014 et ont plus de 975 abonnés au sein de leur communauté technologique interne, calquée sur les DevOpsDays tenus chez ING à Amsterdam en 2013.

Après que Mickman et Clanton aient assisté au DevOps Enterprise Summit en 2014, ils ont organisé leur propre conférence interne, invitant de nombreux conférenciers de sociétés extérieures afin de recréer leur expérience pour leur direction. Clanton décrit : « 2015 a été l'année où nous avons attiré l'attention des cadres et où nous avons gagné en momentum. Après cet événement, de nombreuses personnes sont venues nous voir, demandant comment elles pouvaient s'impliquer et comment elles pouvaient aider. »

Créer un système de consultation et de coaching interne pour propager les pratiques

Créer une organisation interne de coaching et de consultation est une méthode couramment utilisée pour diffuser l'expertise au sein d'une organisation. Cela peut prendre différentes formes. Chez Capital One, des experts désignés tiennent des heures de bureau où chacun peut les consulter, poser des questions, etc.

Plus tôt dans le livre, nous avons commencé l'histoire de comment le Testing Grouplet a construit une culture de test automatisé de classe mondiale chez Google à partir de 2005. Leur histoire continue ici, alors qu'ils tentent d'améliorer l'état des tests automatisés dans tout Google en utilisant des blitz d'amélioration dédiés, des coachs internes et même un programme de certification interne.

Bland a déclaré qu'à cette époque, il existait une politique de 20 % de temps d'innovation chez Google, permettant aux développeurs de passer environ un jour par semaine sur un projet lié à

Google en dehors de leur domaine de responsabilité principal. Certains ingénieurs ont choisi de former des grouplets, des équipes ad hoc d'ingénieurs partageant les mêmes idées qui voulaient mettre en commun leur temps de 20 %, leur permettant ainsi de faire des blitz d'amélioration ciblés.

Un grouplet de test a été formé par Bharat Mediratta et Nick Lesiecki, avec pour mission de promouvoir l'adoption des tests automatisés chez Google. Bien qu'ils n'aient eu ni budget ni autorité formelle, comme l'a décrit Mike Bland, "aucune contrainte explicite ne nous a été imposée. Et nous en avons profité."

Ils ont utilisé plusieurs mécanismes pour favoriser l'adoption, mais l'un des plus célèbres était Testing on the Toilet (ou TotT), leur périodique de test hebdomadaire. Chaque semaine, ils publiaient un bulletin dans presque toutes les toilettes de presque tous les bureaux de Google dans le monde entier. Bland a déclaré : "Le but était d'élever le niveau de connaissance et de sophistication en matière de tests dans toute l'entreprise. Il est douteux qu'une publication uniquement en ligne aurait impliqué autant de personnes."

Bland poursuit : "L'un des épisodes TotT les plus significatifs était celui intitulé 'Test Certified : Lousy Name, Great Results', car il exposait deux initiatives qui ont eu un succès significatif dans l'avancement de l'utilisation des tests automatisés."

Test Certified (TC) fournissait une feuille de route pour améliorer l'état des tests automatisés. Comme Bland le décrit, "il était destiné à exploiter les priorités axées sur la mesure de la culture Google... et à surmonter le premier obstacle effrayant de ne pas savoir où ou comment commencer. Le niveau 1 consistait à établir rapidement une métrique de référence, le niveau 2 consistait à définir une politique et à atteindre un objectif de couverture de test automatisé, et le niveau 3 consistait à viser un objectif de couverture à long terme."

La deuxième capacité consistait à fournir des mentors Test Certified à toute équipe souhaitant des conseils ou de l'aide, et des Test Mercenaries (c'est-à-dire une équipe à temps plein de coachs et consultants internes) pour travailler directement avec les équipes afin d'améliorer leurs pratiques de test et la qualité de leur code. Les Mercenaries le faisaient en appliquant les connaissances, outils et techniques du Testing Grouplet au propre code de l'équipe, en utilisant TC comme guide et objectif. Bland a finalement été un leader du Testing Grouplet de 2006 à 2007 et un membre des Test Mercenaries de 2007 à 2009.

Bland poursuit : "Notre objectif était d'amener chaque équipe au niveau TC 3, qu'elles soient inscrites à notre programme ou non. Nous avons également collaboré étroitement avec les équipes d'outils de test internes, fournissant des retours d'expérience alors que nous relevions les défis de test avec les équipes de produits. Nous étions sur le terrain, appliquant les outils

que nous avions construits, et finalement, nous avons pu éliminer 'je n'ai pas le temps de tester' comme excuse légitime."

Il poursuit : "Les niveaux TC exploitaient la culture de Google axée sur les métriques – les trois niveaux de test étaient quelque chose dont les gens pouvaient discuter et se vanter lors des évaluations de performance. Le Testing Grouplet a finalement obtenu un financement pour les Test Mercenaries, une équipe à temps plein de consultants internes. Cela a été une étape importante, car la direction était désormais entièrement à bord, non pas avec des édits, mais avec un financement réel."

Un autre concept important était de tirer parti des blitz d'amélioration "Fixit" à l'échelle de l'entreprise. Bland décrit les Fixits comme "lorsque des ingénieurs ordinaires avec une idée et un sens de la mission recrutent toute l'ingénierie de Google pour des sprints intensifs d'une journée de réforme du code et d'adoption d'outils." Il a organisé quatre Fixits à l'échelle de l'entreprise, deux Fixits purement axés sur les tests et deux plus orientés vers les outils, le dernier impliquant plus de cent bénévoles dans plus de vingt bureaux dans treize pays. Il a également dirigé le Fixit Grouplet de 2007 à 2008.

Ces Fixits, comme le décrit Bland, signifient que nous devrions fournir des missions ciblées à des moments critiques pour générer de l'excitation et de l'énergie, ce qui aide à faire progresser l'état de l'art. Cela aidera la mission de changement de culture à long terme à atteindre un nouveau plateau avec chaque grand effort visible.

Les résultats de la culture du test sont évidents dans les résultats étonnantes que Google a obtenus, présentés tout au long du livre.

Conclusion

Ce chapitre a décrit comment nous pouvons instituer des rituels qui aident à renforcer la culture selon laquelle nous sommes tous des apprenants à vie et que nous valorisons l'amélioration du travail quotidien par rapport au travail quotidien lui-même. Nous faisons cela en réservant du temps pour rembourser la dette technique, créer des forums permettant à chacun d'apprendre et d'enseigner aux autres, tant à l'intérieur qu'à l'extérieur de notre organisation. Et nous rendons des experts disponibles pour aider les équipes internes, soit par le biais de coaching, de consultation ou même simplement en tenant des heures de bureau pour répondre aux questions.

En aidant chacun à apprendre des autres dans notre travail quotidien, nous apprenons plus vite que la concurrence, nous aidant à gagner sur le marché. Mais nous aidons également chacun à atteindre son plein potentiel en tant qu'être humain.

Conclusion de la partie V

Au cours de la Partie V, nous avons exploré les pratiques qui aident à créer une culture d'apprentissage et d'expérimentation dans votre organisation. Apprendre des incidents, créer des dépôts partagés et partager les apprentissages sont essentiels lorsque nous travaillons dans des systèmes complexes, contribuant à rendre notre culture de travail plus juste et nos systèmes plus sûrs et résilients.

Dans la Partie VI, nous explorerons comment étendre le flux, les retours d'information, ainsi que l'apprentissage et l'expérimentation en les utilisant pour nous aider simultanément à atteindre nos objectifs en matière de sécurité de l'information.

Partie VI - Les pratiques d'intégration de la sécurité de l'information, de la gestion du changement et de la conformité

La sécurité de l'information comme tâche de tous, tous les jours

L'une des principales objections à la mise en œuvre des principes et des modèles DevOps a été : "La sécurité de l'information et la conformité ne nous le permettront pas." Et pourtant, DevOps pourrait être l'un des meilleurs moyens d'intégrer davantage la sécurité de l'information dans le travail quotidien de tous dans la chaîne de valeur technologique.

Lorsque la sécurité de l'information est organisée en silo, séparée du Développement et des Opérations, de nombreux problèmes surviennent. James Wickett, l'un des créateurs de l'outil de sécurité Gauntlet et organisateur de DevOpsDays Austin et de la conférence Lonestar Application Security, a observé :

"Une interprétation de DevOps est qu'il est né du besoin de permettre aux développeurs d'être productifs, car à mesure que le nombre de développeurs augmentait, il n'y avait pas assez de personnes Ops pour gérer tout le travail de déploiement résultant. Cette pénurie est encore pire dans la sécurité de l'information - le ratio d'ingénieurs en Développement, Opérations et Sécurité de l'information dans une organisation technologique typique est de 100:10:1. Lorsque la sécurité de l'information est si minoritaire, sans automatisation et intégration de la sécurité de l'information dans le travail quotidien des Dev et Ops, la sécurité de l'information ne peut se contenter que de vérifier la conformité, ce qui est l'opposé de l'ingénierie de la sécurité - et en plus, cela nous fait détester par tout le monde."

James Wickett et Josh Corman, ancien CTO de Sonatype et chercheur respecté en sécurité de l'information, ont écrit sur l'incorporation des objectifs de sécurité de l'information dans DevOps, un ensemble de pratiques et de principes appelé Rugged DevOps. Des idées similaires ont été créées par le Dr Tapabrata Pal, Directeur et Membre Technique en Ingénierie des Plateformes chez Capital One, et l'équipe de Capital One, qui décrivent leurs processus comme DevOpsSec, où la sécurité de l'information est intégrée à toutes les étapes du cycle de vie du développement logiciel (SDLC). Rugged DevOps tire une partie de son histoire de Visible Ops Security, écrit par Gene Kim, Paul Love et George Spafford.

Tout au long du DevOps Handbook, nous avons exploré comment intégrer pleinement les objectifs de QA et d'Opérations dans toute notre chaîne de valeur technologique. Dans ce chapitre, nous décrivons comment intégrer de manière similaire les objectifs de sécurité de l'information dans notre travail quotidien, où nous pouvons augmenter la productivité des développeurs et des opérations, accroître la sécurité et augmenter notre sécurité.

Intégrer la sécurité dans les démonstrations d'itération de développement

L'un de nos objectifs est de faire en sorte que les équipes de fonctionnalités soient impliquées avec la sécurité de l'information le plus tôt possible, plutôt que de s'impliquer principalement à la fin du projet. Une manière de le faire est d'inviter la sécurité de l'information aux démonstrations de produit à la fin de chaque intervalle de développement afin qu'ils puissent mieux comprendre les objectifs de l'équipe dans le contexte des objectifs organisationnels, observer leurs implantations au fur et à mesure de leur construction, et fournir des conseils et des retours dès les premières étapes du projet, lorsque le temps et la liberté pour faire des corrections sont les plus grands.

Justin Arbuckle, ancien architecte en chef chez GE Capital, observe :

"En ce qui concerne la sécurité de l'information et la conformité, nous avons constaté que les blocages en fin de projet étaient beaucoup plus coûteux qu'au début - et les blocages de la sécurité de l'information étaient parmi les pires. La 'conformité par démonstration' est devenue l'un des rituels que nous avons utilisés pour déplacer toute cette complexité plus tôt dans le processus."

Il continue :

"En impliquant la sécurité de l'information tout au long de la création de toute nouvelle capacité, nous avons pu réduire considérablement notre utilisation des listes de contrôle statiques et nous appuyer davantage sur leur expertise tout au long du processus de développement logiciel."

Cela a aidé l'organisation à atteindre ses objectifs. Snehal Antani, ancien CIO de l'architecture d'entreprise chez GE Capital Americas, a décrit leurs trois principales mesures commerciales clés comme étant "la vitesse de développement (c'est-à-dire la rapidité de livraison des fonctionnalités au marché), les interactions client échouées (c'est-à-dire les pannes, erreurs), et le temps de réponse de conformité (c'est-à-dire le délai entre la demande d'audit et la livraison de toutes les informations quantitatives et qualitatives requises pour répondre à la demande)."

Lorsque la sécurité de l'information fait partie de l'équipe, même si elle ne font qu'être informés et observer le processus, ils acquièrent le contexte commercial dont ils ont besoin pour prendre de meilleures décisions basées sur les risques. De plus, la sécurité de l'information est capable d'aider les équipes de fonctionnalités à apprendre ce qui est nécessaire pour atteindre les objectifs de sécurité et de conformité.

Intégrer la sécurité dans le suivi des défauts et les retours d'expérience

Dans la mesure du possible, nous voulons suivre toutes les questions de sécurité ouvertes dans le même système de suivi des travaux que celui utilisé par le Développement et les Opérations, en veillant à ce que le travail soit visible et puisse être priorisé par rapport à tous les autres travaux. Cela est très différent de la manière dont la sécurité de l'information (Infosec) fonctionnait traditionnellement, où toutes les vulnérabilités de sécurité étaient stockées dans un outil GRC (gouvernance, risque et conformité) auquel seule la sécurité de l'information avait accès. Au lieu de cela, nous mettrons tout travail nécessaire dans les systèmes utilisés par le Développement et les Opérations.

Dans une présentation lors des DevOpsDays à Austin en 2012, Nick Galbreath, qui a dirigé la sécurité de l'information chez Etsy pendant de nombreuses années, décrit comment ils traitaient les problèmes de sécurité : "Nous avons mis tous les problèmes de sécurité dans JIRA, que tous les ingénieurs utilisent dans leur travail quotidien, et ils étaient soit 'P1' soit 'P2', ce qui signifiait qu'ils devaient être corrigés immédiatement ou d'ici la fin de la semaine, même si le problème ne concernait qu'une application interne."

De plus, il déclare : "Chaque fois que nous avions un problème de sécurité, nous faisions un retour d'expérience, car cela permettait de mieux éduquer nos ingénieurs sur la façon de prévenir ce type de problème à l'avenir, ainsi que de transférer efficacement les connaissances en matière de sécurité à nos équipes d'ingénieurs."

Intégrer les contrôles de sécurité préventifs dans les répertoires de code source partagés les services partagés

Dans le chapitre 20, nous avons créé un dépôt de code source partagé qui permet à chacun de découvrir et de réutiliser les connaissances collectives de notre organisation, non seulement pour notre code, mais aussi pour nos chaînes d'outils, notre pipeline de déploiement, nos standards, etc. En faisant cela, chacun peut bénéficier de l'expérience cumulative de tous les membres de l'organisation.

Nous ajouterons maintenant à notre dépôt de code source partagé tous les mécanismes ou outils qui nous aident à garantir que nos applications et environnements sont sécurisés. Nous ajouterons des bibliothèques approuvées par la sécurité pour atteindre des objectifs spécifiques de sécurité de l'information, telles que des bibliothèques et services d'authentification et de chiffrement. Comme tous les acteurs de la chaîne de valeur DevOps utilisent le contrôle de version pour tout ce qu'ils construisent ou supportent, mettre nos artefacts de sécurité de l'information à cet endroit rend beaucoup plus facile d'influencer le travail quotidien du Développement et des Opérations, car tout ce que nous créons est disponible, consultable et réutilisable. Le contrôle de version sert également de mécanisme de communication omnidirectionnelle pour tenir toutes les parties informées des changements en cours.

Si nous avons une organisation centralisée de services partagés, nous pouvons également collaborer avec elle pour créer et exploiter des plateformes partagées pertinentes pour la sécurité, telles que des services d'authentification, d'autorisation, de journalisation et autres services de sécurité et d'audit dont le Développement et les Opérations ont besoin. Lorsque les ingénieurs utilisent l'une de ces bibliothèques ou services prédéfinis, ils n'ont pas besoin de planifier une revue de conception de sécurité séparée pour ce module ; ils utiliseront les directives que nous avons créées concernant le durcissement de la configuration, les paramètres de sécurité de la base de données, les longueurs de clé, etc.

Pour augmenter encore la probabilité que les services et bibliothèques que nous fournissons soient utilisés correctement, nous pouvons offrir une formation à la sécurité au Développement et aux Opérations, ainsi que revoir ce qu'ils ont créé pour aider à garantir que les objectifs de sécurité sont correctement mis en œuvre, en particulier pour les équipes utilisant ces outils pour la première fois.

En fin de compte, notre objectif est de fournir les bibliothèques ou services de sécurité dont chaque application ou environnement moderne a besoin, tels que l'activation de l'authentification des utilisateurs, l'autorisation, la gestion des mots de passe, le chiffrement des données, etc. De plus, nous pouvons fournir au Développement et aux Opérations des paramètres de configuration spécifiques à la sécurité pour les composants qu'ils utilisent dans leurs piles applicatives, tels que pour la journalisation, l'authentification et le chiffrement. Nous pouvons inclure des éléments tels que :

- Bibliothèques de code et leurs configurations recommandées (par exemple, bibliothèque d'authentification à deux facteurs [2FA], hachage de mot de passe bcrypt, journalisation)
- Gestion des secrets (par exemple, paramètres de connexion, clés de chiffrement) en utilisant des outils tels que Vault, Sneaker, Keywhiz, Credstash, Trousseau, Red October, etc.
- Packages et constructions OS (par exemple, NTP pour la synchronisation temporelle, versions sécurisées d'OpenSSL avec configurations correctes, OSSEC ou Tripwire pour la surveillance de l'intégrité des fichiers, configuration de syslog pour garantir la journalisation des informations de sécurité critiques dans notre pile ELK centralisée)

En mettant tout cela dans notre dépôt de code source partagé, nous facilitons la création et l'utilisation correctes des standards de journalisation et de chiffrement dans leurs applications et environnements par n'importe quel ingénieur, sans travail supplémentaire de notre part.

Nous devrions également collaborer avec les équipes des Opérations pour créer un livre de recettes de base ou une image de construction de notre OS, des bases de données et autres infrastructures (par exemple, NGINX, Apache, Tomcat), montrant qu'ils sont dans un état connu, sécurisé et à risque réduit. Notre dépôt partagé devient non seulement l'endroit où nous pouvons obtenir les dernières versions, mais aussi un endroit où nous pouvons collaborer avec d'autres ingénieurs et surveiller et alerter sur les modifications apportées aux modules sensibles à la sécurité.

Intégrer la sécurité dans notre pipeline de déploiement

À des époques antérieures, pour renforcer et sécuriser notre application, nous commençions notre revue de sécurité après la fin du développement. Souvent, le résultat de cette revue était des centaines de pages de vulnérabilités dans un PDF que nous remettions au Développement et aux Opérations, qui restaient complètement non traitées en raison de la pression des dates limites de projet ou de problèmes découverts trop tard dans le cycle de vie du logiciel pour être facilement corrigés.

À cette étape, nous allons automatiser autant que possible nos tests de sécurité de l'information, afin qu'ils soient exécutés en parallèle avec tous nos autres tests automatisés dans notre pipeline de déploiement, idéalement à chaque commit de code par le Développement ou les Opérations, et même aux premiers stades d'un projet logiciel.

Notre objectif est de fournir au Développement et aux Opérations un retour d'information rapide sur leur travail afin qu'ils soient avertis dès qu'ils valident des changements potentiellement non sécurisés. En faisant cela, nous leur permettons de détecter et de corriger rapidement les problèmes de sécurité dans le cadre de leur travail quotidien, ce qui favorise l'apprentissage et prévient les erreurs futures.

Idéalement, ces tests de sécurité automatisés seront exécutés dans notre pipeline de déploiement aux côtés des autres outils d'analyse de code statique.

Des outils comme Gauntlet ont été conçus pour s'intégrer dans les pipelines de déploiement, exécutant des tests de sécurité automatisés sur nos applications, nos dépendances applicatives, notre environnement, etc. Fait remarquable, Gauntlet met même tous ses tests de sécurité dans des scripts de test syntaxiques Gherkin, largement utilisés par les développeurs pour les tests unitaires et fonctionnels. Cela place les tests de sécurité dans un cadre avec lequel ils sont probablement déjà familiers. Cela permet également aux tests de sécurité de s'exécuter facilement dans un pipeline de déploiement à chaque changement validé, tels que l'analyse de code statique, la vérification des dépendances vulnérables ou les tests dynamiques.

En faisant cela, nous fournissons à tout le monde dans la chaîne de valeur le retour d'information le plus rapide possible sur la sécurité de ce qu'ils créent, permettant aux ingénieurs Dev et Ops de trouver et de corriger rapidement les problèmes.

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

Figure 43: Jenkins running automated security testing (Source: James Wicket and Gareth Rushgrove, “Battle-tested code without the battle,” Velocity 2014 conference presentation, posted to Speakerdeck.com, June 24, 2014, <https://speakerdeck.com/garethr/battle-tested-code-without-the-battle.>)

Assurer la sécurité de l'application

Souvent, les tests de Développement se concentrent sur la justesse de la fonctionnalité, en examinant les flux logiques positifs. Ce type de test est souvent appelé le "happy path", qui valide les parcours utilisateur (et parfois les chemins alternatifs) où tout se passe comme prévu, sans exceptions ni conditions d'erreur.

En revanche, les pratiques efficaces de QA, Infosec et de lutte contre la fraude se concentrent souvent sur les "sad paths", qui se produisent lorsque les choses vont mal, notamment en ce qui concerne les conditions d'erreur liées à la sécurité. (Ces types de conditions spécifiques à la sécurité sont parfois plaisamment appelés les "bad paths".)

Par exemple, supposons que nous avons un site de commerce électronique avec un formulaire de saisie client qui accepte les numéros de carte de crédit dans le cadre de la génération d'une commande client. Nous voulons définir tous les chemins "sad" et "bad" nécessaires pour garantir que les cartes de crédit invalides sont correctement rejetées afin de prévenir la fraude et les exploits de sécurité, tels que les injections SQL, les dépassements de tampon et d'autres résultats indésirables.

Au lieu d'effectuer ces tests manuellement, nous les générerions idéalement dans le cadre de nos tests unitaires ou fonctionnels automatisés afin qu'ils puissent être exécutés en continu dans notre pipeline de déploiement. Dans le cadre de nos tests, nous souhaiterons inclure les éléments suivants :

- **Analyse statique** : Il s'agit des tests que nous effectuons dans un environnement non-exécution, idéalement dans le pipeline de déploiement. Typiquement, un outil d'analyse statique inspecte le code source du programme pour tous les comportements possibles à l'exécution et recherche les défauts de codage, les portes dérobées et le code potentiellement malveillant (c'est parfois connu sous le nom de "testing from the inside-out"). Des exemples d'outils incluent Brakeman, Code Climate, et la recherche de fonctions de code interdites (par exemple, "exec()").
- **Analyse dynamique** : Contrairement aux tests statiques, l'analyse dynamique consiste en des tests exécutés pendant que le programme est en fonctionnement. Les tests dynamiques surveillent des éléments tels que la mémoire système, le comportement fonctionnel, le temps de réponse et la performance globale du système. Cette méthode (parfois appelée "testing from the outside-in") est similaire à la manière dont un tiers malveillant pourrait interagir avec une application. Des exemples incluent Arachni et OWASP ZAP (Zed Attack Proxy).

Certains types de tests de pénétration peuvent également être effectués de manière automatisée et devraient être inclus dans le cadre de l'analyse dynamique en utilisant des outils tels que Nmap et Metasploit. Idéalement, nous devrions effectuer des tests dynamiques automatisés pendant la phase de tests fonctionnels automatisés de notre pipeline de déploiement, ou même contre nos services lorsqu'ils sont en production. Pour garantir un traitement correct de la sécurité, des outils comme OWASP ZAP peuvent être configurés pour attaquer nos services via un proxy de navigateur web et inspecter le trafic réseau dans notre environnement de test.

- **Analyse des dépendances** : Un autre type de test statique que nous effectuons normalement au moment de la construction dans notre pipeline de déploiement consiste à inventorier toutes nos dépendances pour les binaires et les exécutables, et à garantir que ces dépendances, sur lesquelles nous n'avons souvent pas de contrôle, sont exemptes de vulnérabilités ou de binaires malveillants. Des exemples incluent Gemnasium et bundler audit pour Ruby, Maven pour Java, et l'OWASP Dependency-Check.
- **Intégrité du code source et signature du code** : Tous les développeurs devraient avoir leur propre clé PGP, peut-être créée et gérée dans un système tel que keybase.io. Tous les commits dans le contrôle de version devraient être signés, ce qui est facile à configurer en utilisant les outils open source gpg et git. De plus, tous les packages créés par le processus CI devraient être signés et leur hash enregistré dans le service de journalisation centralisé à des fins d'audit.

De plus, nous devrions définir des modèles de conception pour aider les développeurs à écrire du code pour prévenir les abus, comme mettre en place des limites de taux pour nos services et

griser les boutons de soumission après qu'ils ont été pressés. L'OWASP publie une grande quantité de conseils utiles tels que la série Cheat Sheet, qui comprend :

- Comment stocker les mots de passe
- Comment gérer les mots de passe oubliés
- Comment gérer la journalisation
- Comment prévenir les vulnérabilités de type cross-site scripting (XSS)

Étude de Cas - Tests de Sécurité Statique chez Twitter (2009)

La présentation « 10 Deploys per Day: Dev and Ops Cooperation at Flickr » de John Allspaw et Paul Hammond est célèbre pour avoir catalysé la communauté Dev et Ops en 2009. L'équivalent pour la communauté de la sécurité de l'information est probablement la présentation donnée par Justin Collins, Alex Smolen, et Neil Matatall sur leur travail de transformation de la sécurité de l'information chez Twitter lors de la conférence AppSecUSA en 2012.

Twitter a rencontré de nombreux défis en raison de sa croissance rapide. Pendant des années, la célèbre page d'erreur Fail Whale était affichée lorsque Twitter n'avait pas suffisamment de capacité pour répondre à la demande des utilisateurs, montrant un graphique d'une baleine soulevée par huit oiseaux. L'ampleur de la croissance des utilisateurs était époustouflante : entre janvier et mars 2009, le nombre d'utilisateurs actifs de Twitter est passé de 2,5 millions à 10 millions.

Twitter a également eu des problèmes de sécurité durant cette période. Début 2009, deux violations de sécurité graves se sont produites. Tout d'abord, en janvier, le compte Twitter @BarackObama a été piraté. Ensuite, en avril, les comptes administratifs de Twitter ont été compromis par une attaque par force brute de dictionnaire. Ces événements ont conduit la Federal Trade Commission à juger que Twitter induisait ses utilisateurs en erreur en leur faisant croire que leurs comptes étaient sécurisés et a émis un ordre de consentement de la FTC.

L'ordre de consentement exigeait que Twitter se conforme dans les soixante jours en instituant un ensemble de processus devant être appliqués pour les vingt années suivantes et faisant ce qui suit :

- Désigner un ou des employés responsables du plan de sécurité de l'information de Twitter
- Identifier les risques raisonnablement prévisibles, internes et externes, pouvant conduire à un incident d'intrusion et créer et mettre en œuvre un plan pour traiter ces risques
- Maintenir la confidentialité des informations des utilisateurs, non seulement de sources extérieures mais aussi en interne, avec un aperçu des sources possibles de vérification et de test de la sécurité et de la correction de ces mises en œuvre

Le groupe d'ingénieurs chargé de résoudre ce problème devait intégrer la sécurité dans le travail quotidien de Dev et Ops et combler les failles de sécurité ayant permis les violations initiales.

Dans leur présentation précédemment mentionnée, Collins, Smolen et Matatall ont identifié plusieurs problèmes qu'ils devaient résoudre :

- **Prévenir la répétition des erreurs de sécurité** : Ils ont constaté qu'ils corrigeaient les mêmes défauts et vulnérabilités encore et encore. Ils devaient modifier le système de travail et les outils d'automatisation pour éviter que les problèmes ne se reproduisent.
- **Intégrer les objectifs de sécurité dans les outils de développement existants** : Ils ont identifié dès le début que la principale source de vulnérabilités était les problèmes de code. Ils ne pouvaient pas exécuter un outil générant un énorme rapport PDF et ensuite l'envoyer par courriel à quelqu'un dans Développement ou Opérations. Au lieu de cela, ils devaient fournir au développeur ayant créé la vulnérabilité les informations exactes nécessaires pour la corriger.
- **Préserver la confiance du Développement** : Ils devaient gagner et maintenir la confiance de Développement. Cela signifiait qu'ils devaient savoir quand ils envoyoyaient des faux positifs à Développement, afin de pouvoir corriger l'erreur ayant généré le faux positif et éviter de perdre du temps à Développement.
- **Maintenir un flux rapide à travers Infosec grâce à l'automatisation** : Même lorsque le scan des vulnérabilités de code était automatisé, Infosec devait encore effectuer beaucoup de travail manuel et attendre. Ils devaient attendre la fin du scan, recevoir la pile de rapports, interpréter les rapports, puis trouver la personne responsable de la correction. Et lorsque le code changeait, il fallait tout recommencer. En automatisant le travail manuel, ils effectuaient moins de tâches « d'appui sur des boutons », leur permettant d'utiliser plus de créativité et de jugement.
- **Rendre tout ce qui est lié à la sécurité en libre-service, si possible** : Ils faisaient confiance au fait que la plupart des gens voulaient faire ce qu'il fallait, donc il était nécessaire de leur fournir tout le contexte et les informations dont ils avaient besoin pour résoudre les problèmes.
- **Adopter une approche holistique pour atteindre les objectifs de sécurité de l'information** : Leur objectif était d'analyser tous les angles : le code source, l'environnement de production, et même ce que leurs clients voyaient.

La première grande avancée pour l'équipe Infosec s'est produite lors d'une semaine de hacking à l'échelle de l'entreprise, lorsqu'ils ont intégré l'analyse statique du code dans le processus de construction de Twitter. L'équipe a utilisé Brakeman, qui scanne les applications Ruby on Rails pour détecter les vulnérabilités. L'objectif était d'intégrer le scan de sécurité dès les premières étapes du processus de développement, et pas seulement lorsque le code était validé dans le dépôt de code source.

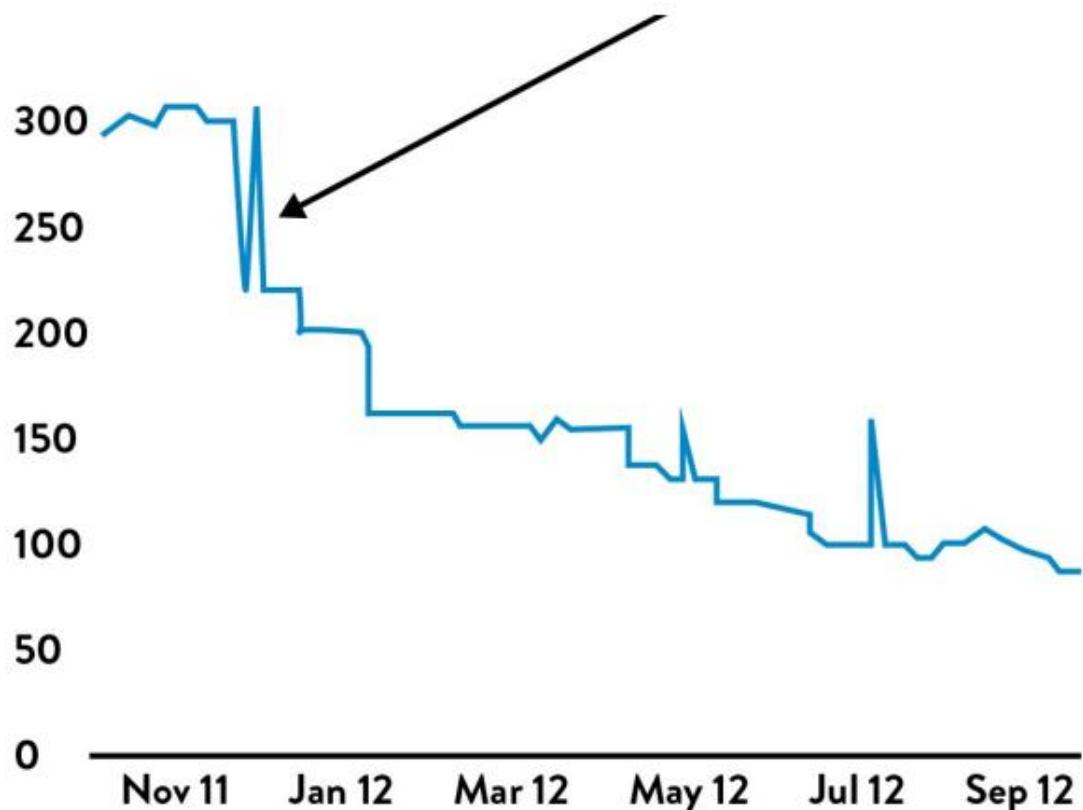


Figure 44: Number of Brakeman security vulnerabilities detected

Les résultats de l'intégration des tests de sécurité dans le processus de développement ont été époustouflants. Au fil des ans, en créant des retours rapides pour les développeurs lorsqu'ils écrivent du code non sécurisé et en leur montrant comment corriger les vulnérabilités, Brakeman a réduit le taux de vulnérabilités trouvées de 60 %, comme le montre la figure 44. (Les pics sont généralement associés à de nouvelles versions de Brakeman.)

Cette étude de cas illustre à quel point il est nécessaire d'intégrer la sécurité dans le travail quotidien et les outils de DevOps et à quel point cela peut être efficace. Cela permet de réduire le risque de sécurité, diminue la probabilité de vulnérabilités dans le système et aide à enseigner aux développeurs à écrire un code plus sécurisé.

Assurer la sécurité de notre chaîne d'approvisionnement logiciel

Josh Corman a observé qu'en tant que développeurs, « nous ne créons plus de logiciels personnalisés - nous assemblons plutôt ce dont nous avons besoin à partir de pièces open source, qui sont devenues la chaîne d'approvisionnement logicielle sur laquelle nous comptons beaucoup ». En d'autres termes, lorsque nous utilisons des composants ou des bibliothèques - qu'ils soient commerciaux ou open source - dans notre logiciel, nous héritons non seulement de leur fonctionnalité, mais aussi de toutes les vulnérabilités de sécurité qu'ils contiennent.

Lors de la sélection de logiciels, nous détectons lorsque nos projets logiciels reposent sur des composants ou des bibliothèques ayant des vulnérabilités connues, et aidons les développeurs à choisir les composants qu'ils utilisent de manière délibérée et avec soin, en sélectionnant ceux (par exemple, les projets open source) qui ont un historique démontré de correction rapide des vulnérabilités logicielles. Nous recherchons également plusieurs versions de la même bibliothèque utilisées dans notre paysage de production, en particulier la présence de versions plus anciennes de bibliothèques contenant des vulnérabilités connues.

Examiner les violations de données des titulaires de cartes montre à quel point la sécurité des composants open source que nous choisissons peut être importante. Depuis 2008, le rapport annuel Verizon PCI Data Breach Investigation Report (DBIR) est la voix la plus autoritaire sur les violations de données où les informations des titulaires de cartes ont été perdues ou volées. Dans le rapport 2014, ils ont étudié plus de quatre-vingt-cinq mille violations pour mieux comprendre d'où venaient les attaques, comment les données des titulaires de cartes étaient volées et les facteurs conduisant à la violation.

Le DBIR a constaté que dix vulnérabilités (c'est-à-dire des CVE) représentaient presque 97 % des exploits utilisés dans les violations de données des titulaires de cartes étudiées en 2014. Parmi ces dix vulnérabilités, huit avaient plus de dix ans.

Le rapport Sonatype State of the Software Supply Chain 2015 a analysé plus en détail les données de vulnérabilités du Nexus Central Repository. En 2015, ce dépôt fournissait les artefacts de construction pour plus de 605 000 projets open source, traitant plus de dix-sept milliards de demandes de téléchargement d'artefacts et de dépendances principalement pour la plateforme Java, en provenance de 106 000 organisations.

Le rapport a inclus ces résultats surprenants :

- L'organisation typique reposait sur 7 601 artefacts de construction (c'est-à-dire, des fournisseurs de logiciels ou des composants) et utilisait 18 614 versions différentes (c'est-à-dire, des parties logicielles).
- Parmi ces composants utilisés, 7,5 % avaient des vulnérabilités connues, avec plus de 66 % de ces vulnérabilités ayant plus de deux ans sans avoir été résolues.
- La dernière statistique confirme une autre étude de sécurité de l'information réalisée par Dr. Dan Geer et Josh Corman, qui a montré que parmi les projets open source ayant des vulnérabilités connues enregistrées dans la National Vulnerability Database, seulement 41 % ont été corrigés et ont nécessité en moyenne 390 jours pour publier un correctif. Pour les vulnérabilités classées au niveau de gravité le plus élevé (c'est-à-dire celles notées CVSS niveau 10), les correctifs ont nécessité 224 jours.

Assurer la sécurité de l'environnement

À cette étape, nous devons faire tout ce qui est nécessaire pour garantir que les environnements sont dans un état durci et à risque réduit. Bien que nous ayons peut-être déjà créé des configurations connues et bonnes, nous devons mettre en place des contrôles de surveillance pour nous assurer que toutes les instances de production correspondent à ces états connus.

Nous procérons en générant des tests automatisés pour garantir que tous les paramètres appropriés ont été correctement appliqués pour le durcissement de la configuration, les paramètres de sécurité de la base de données, les longueurs de clé, etc. De plus, nous utiliserons des tests pour scanner nos environnements à la recherche de vulnérabilités connues.

Une autre catégorie de vérification de la sécurité est la compréhension des environnements réels (c'est-à-dire « tels qu'ils sont réellement »). Des exemples d'outils pour cela incluent Nmap pour s'assurer que seuls les ports attendus sont ouverts et Metasploit pour garantir que nous avons suffisamment durci nos environnements contre les vulnérabilités connues, comme le scan avec des attaques par injection SQL. Les résultats de ces outils doivent être enregistrés dans notre dépôt d'artefacts et comparés avec la version précédente dans le cadre de notre processus de test fonctionnel. Cela nous aidera à détecter tout changement indésirable dès qu'il se produit.

Étude de cas - 18F : Automatisation de la conformité pour le gouvernement fédéral avec Compliance Masonry

Les agences du gouvernement fédéral américain étaient censées dépenser près de 80 milliards de dollars en TI en 2016, soutenant la mission de toutes les agences exécutives. Peu importe l'agence, pour passer d'un système de « développement terminé » à « en production », il est nécessaire d'obtenir une Autorisation d'Opération (ATO) d'une Autorité d'Approbation Désignée (DAA).

Les lois et politiques qui régissent la conformité dans le gouvernement sont constituées de dizaines de documents totalisant plus de quatre mille pages, jonchées d'acronymes tels que FISMA, FedRAMP, et FITARA. Même pour les systèmes nécessitant uniquement des niveaux faibles de confidentialité, d'intégrité et de disponibilité, plus de cent contrôles doivent être mis en œuvre, documentés et testés. Il faut généralement entre huit et quatorze mois pour qu'une ATO soit accordée après le « développement terminé ».

L'équipe 18F de l'Administration des Services Généraux (GSA) du gouvernement fédéral a adopté une approche multi-facettes pour résoudre ce problème. Mike Bland explique : « 18F a été créé au sein de l'Administration des Services Généraux pour capitaliser sur l'élan généré par

la récupération de Healthcare.gov afin de réformer la manière dont le gouvernement construit et achète des logiciels. »

Un des efforts de 18F est une plateforme en tant que service appelée Cloud.gov, créée à partir de composants open source. Cloud.gov fonctionne actuellement sur AWS GovCloud. Non seulement la plateforme gère de nombreuses préoccupations opérationnelles que les équipes de livraison devraient autrement prendre en charge, telles que la journalisation, la surveillance, l'alerte et la gestion du cycle de vie des services, mais elle prend également en charge la majorité des préoccupations de conformité. En fonctionnant sur cette plateforme, une grande majorité des contrôles que les systèmes gouvernementaux doivent mettre en œuvre peuvent être gérés au niveau de l'infrastructure et de la plateforme. Ensuite, seuls les contrôles restants au niveau de l'application doivent être documentés et testés, ce qui réduit considérablement le fardeau de la conformité et le temps nécessaire pour obtenir une ATO.

AWS GovCloud a déjà été approuvé pour une utilisation pour les systèmes gouvernementaux de tous types, y compris ceux nécessitant des niveaux élevés de confidentialité, d'intégrité et de disponibilité. Au moment où vous lirez ce livre, il est prévu que Cloud.gov sera approuvé pour tous les systèmes nécessitant des niveaux modérés de confidentialité, d'intégrité et de disponibilité.

De plus, l'équipe Cloud.gov construit un cadre pour automatiser la création des plans de sécurité des systèmes (SSP), qui sont des « descriptions complètes de l'architecture du système, des contrôles mis en œuvre et de la posture de sécurité générale... [qui sont] souvent incroyablement complexes, s'étendant sur plusieurs centaines de pages. » Ils ont développé un outil prototype appelé compliance masonry afin que les données SSP soient stockées en YAML lisible par machine et converties automatiquement en GitBooks et PDF.

18F est dédié à travailler en toute transparence et publie son travail en open source dans le domaine public. Vous pouvez trouver compliance masonry et les composants qui composent Cloud.gov dans les dépôts GitHub de 18F - vous pouvez même créer votre propre instance de Cloud.gov. Le travail sur la documentation open source pour les SSP est réalisé en étroite collaboration avec la communauté OpenControl.

Intégrer la sécurité de l'information dans la télémétrie de production

Marcus Sachs, l'un des chercheurs sur les violations de données chez Verizon, a observé en 2010 : « Année après année, dans la grande majorité des violations de données de titulaire de carte, l'organisation détectait la violation de sécurité des mois ou des trimestres après qu'elle s'était produite. Pire encore, la manière dont la violation était détectée n'était pas un contrôle interne de surveillance, mais était bien plus probablement quelqu'un en dehors de l'organisation, généralement un partenaire commercial ou le client qui remarque des

transactions frauduleuses. L'une des raisons principales est que personne dans l'organisation ne passait régulièrement en revue les fichiers journaux. »

En d'autres termes, les contrôles de sécurité internes sont souvent inefficaces pour détecter les violations de manière opportune, soit en raison de points aveugles dans notre surveillance, soit parce que personne dans notre organisation n'examine la télémétrie pertinente dans son travail quotidien.

Dans le chapitre 14, nous avons discuté de la création d'une culture dans Dev et Ops où tout le monde dans la chaîne de valeur crée de la télémétrie et des métriques de production, les rendant visibles dans des endroits publics et proéminents pour que tout le monde puisse voir comment nos services fonctionnent en production. De plus, nous avons exploré la nécessité de rechercher sans relâche des signaux de défaillance de plus en plus faibles afin de pouvoir trouver et résoudre les problèmes avant qu'ils ne conduisent à une défaillance catastrophique.

Ici, nous déployons la surveillance, la journalisation et l'alerte nécessaires pour atteindre nos objectifs de sécurité de l'information à travers nos applications et environnements, ainsi que pour garantir qu'elles sont adéquatement centralisées pour faciliter une analyse et une réponse faciles et significatives.

Nous le faisons en intégrant notre télémétrie de sécurité dans les mêmes outils que ceux utilisés par le Développement, le QA et les Opérations, donnant ainsi à tout le monde dans la chaîne de valeur une visibilité sur la manière dont leur application et leurs environnements fonctionnent dans un environnement de menace hostile où les attaquants tentent constamment d'exploiter les vulnérabilités, d'accéder sans autorisation, de planter des portes dérobées, de commettre des fraudes, de réaliser des attaques par déni de service, etc.

En diffusant comment nos services sont attaqués dans l'environnement de production, nous renforçons le besoin pour tout le monde de réfléchir aux risques de sécurité et de concevoir des contre-mesures dans leur travail quotidien.

Créer une télémétrie de sécurité dans nos applications

Pour détecter un comportement utilisateur problématique qui pourrait être un indicateur ou un facilitateur de fraude et d'accès non autorisé, nous devons créer la télémétrie pertinente dans nos applications.

Les exemples peuvent inclure :

- Connexions réussies et échouées des utilisateurs
- Réinitialisations de mot de passe des utilisateurs
- Réinitialisations des adresses électroniques des utilisateurs
- Changements de carte de crédit des utilisateurs

Par exemple, comme indicateur précoce des tentatives de connexion par force brute pour obtenir un accès non autorisé, nous pourrions afficher le ratio des tentatives de connexion échouées par rapport aux connexions réussies. Et bien sûr, nous devrions créer des alertes autour des événements importants pour garantir que nous pouvons détecter et corriger rapidement les problèmes.

Créer une télémétrie de sécurité dans notre environnement

En plus d'instrumenter notre application, nous devons également créer une télémétrie suffisante dans nos environnements afin de pouvoir détecter les indicateurs précoce d'accès non autorisé, en particulier dans les composants qui fonctionnent sur des infrastructures que nous ne contrôlons pas (par exemple, environnements d'hébergement, dans le cloud). Nous devons surveiller et potentiellement alerter sur les éléments suivants :

- Changements du système d'exploitation (par exemple, en production, dans notre infrastructure de construction)
- Changements des groupes de sécurité
- Changements de configurations (par exemple, OSSEC, Puppet, Chef, Tripwire)
- Changements dans l'infrastructure cloud (par exemple, VPC, groupes de sécurité, utilisateurs et priviléges)
- Tentatives de XSS (c'est-à-dire, "attaques par injection de script inter-sites")
- Tentatives de SQLi (c'est-à-dire, "attaques par injection SQL")
- Erreurs du serveur web (par exemple, erreurs 4XX et 5XX)

Nous voulons également confirmer que nous avons correctement configuré notre journalisation afin que toute la télémétrie soit envoyée au bon endroit. Lorsque nous détectons des attaques, en plus de journaliser qu'elles se sont produites, nous pouvons également choisir de bloquer l'accès et de stocker des informations sur la source pour nous aider à choisir les meilleures actions de mitigation.

Étude de cas - Instrumenter l'environnement chez Etsy (2010)

En 2010, Nick Galbreath était directeur de l'ingénierie chez Etsy et responsable de la sécurité de l'information, du contrôle de la fraude et de la confidentialité. Galbreath définissait la fraude comme lorsque « le système fonctionne incorrectement, permettant l'entrée de données invalides ou non inspectées dans le système, causant des pertes financières, des pertes/vols de données, des temps d'arrêt du système, du vandalisme ou une attaque contre un autre système. »

Pour atteindre ces objectifs, Galbreath n'a pas créé un département séparé de contrôle de la fraude ou de sécurité de l'information ; au lieu de cela, il a intégré ces responsabilités tout au long du flux de valeur DevOps.

Galbreath a créé des télémétries liées à la sécurité qui étaient affichées aux côtés de toutes les autres métriques plus orientées Dev et Ops, que chaque ingénieur d'Etsy voyait régulièrement :

- Interruption anormale des programmes en production (par exemple, fautes de segmentation, vidages de noyau, etc.) : « Il était particulièrement préoccupant de comprendre pourquoi certains processus continuaient de provoquer des vidages de noyau dans notre environnement de production, déclenchés par du trafic provenant du même adresse IP, encore et encore. Il en était de même pour les erreurs HTTP ‘500 Internal Server Errors’. Ce sont des indicateurs qu'une vulnérabilité était exploitée pour obtenir un accès non autorisé à nos systèmes, et qu'un correctif devait être appliqué de toute urgence. »
- Erreur de syntaxe de base de données : « Nous cherchions toujours des erreurs de syntaxe de base de données dans notre code - celles-ci pouvaient soit permettre des attaques par injection SQL, soit être des attaques en cours. Pour cette raison, nous avions une tolérance zéro pour les erreurs de syntaxe de base de données dans notre code, car elles restent l'un des principaux vecteurs d'attaque utilisés pour compromettre les systèmes. »
- Indications d'attaques par injection SQL : « C'était un test ridiculement simple - nous alertions simplement chaque fois que ‘UNION ALL’ apparaissait dans les champs de saisie utilisateur, car cela indique presque toujours une attaque par injection SQL. Nous avons également ajouté des tests unitaires pour nous assurer que ce type de saisie utilisateur incontrôlée ne pouvait jamais être autorisé dans nos requêtes de base de données. »

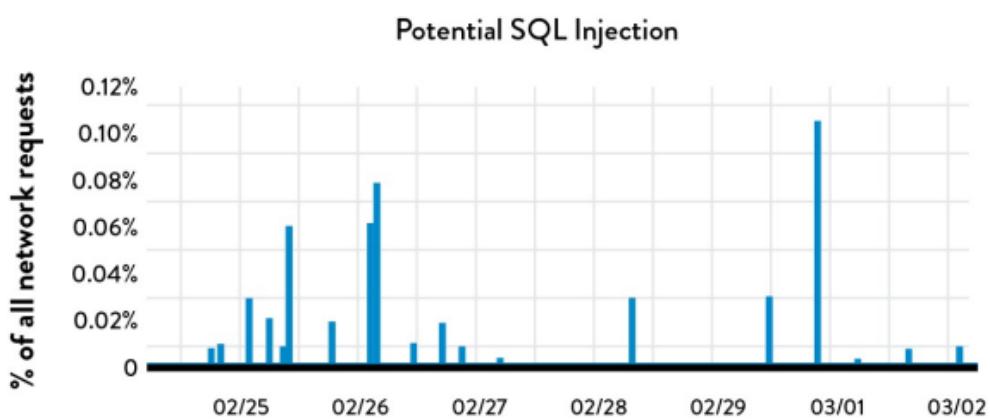


Figure 45: Developers would see SQL injection attempts in Graphite at Etsy (Source: “DevOpsSec: Applying DevOps Principles to Security, DevOpsDays Austin 2012,” SlideShare.net, posted by Nick Galbreath, April 12, 2012, <http://www.slideshare.net/nickgsuperstar/devopssec-apply-devops-principles-to-security>.)

La Figure 45 est un exemple de graphique que chaque développeur verrait, montrant le nombre d'attaques potentielles par injection SQL qui ont été tentées dans l'environnement de production. Comme l'a observé Galbreath, « Rien n'aide les développeurs à comprendre à quel point l'environnement opérationnel est hostile comme de voir leur code attaqué en temps réel. »

Galbreath a observé : « L'un des résultats de l'affichage de ce graphique a été que les développeurs ont réalisé qu'ils étaient attaqués tout le temps ! Et c'était formidable, car cela a changé la façon dont les développeurs pensaient à la sécurité de leur code pendant qu'ils écrivaient le code. »

Protéger notre pipeline de déploiement

L'infrastructure qui soutient nos processus d'intégration continue et de déploiement continu présente également une nouvelle surface vulnérable aux attaques. Par exemple, si quelqu'un compromet les serveurs exécutant le pipeline de déploiement qui détient les identifiants pour notre système de gestion de version, cela pourrait permettre de voler du code source. Pire encore, si le pipeline de déploiement a des droits d'écriture, un attaquant pourrait également injecter des modifications malveillantes dans notre dépôt de gestion de version, et donc injecter des modifications malveillantes dans notre application et nos services.

Comme l'a observé Jonathan Claudius, ancien Senior Security Tester chez TrustWave SpiderLabs, « Les serveurs de construction et de test continu sont géniaux, et je les utilise moi-même. Mais j'ai commencé à réfléchir à des moyens d'utiliser CI/CD comme un moyen d'injecter du code malveillant. Ce qui m'a conduit à la question de savoir où il serait bon de cacher du code malveillant ? La réponse était évidente : dans les tests unitaires. Personne ne regarde réellement les tests unitaires, et ils sont exécutés chaque fois que quelqu'un commet du code dans le dépôt. »

Cela démontre que pour protéger adéquatement l'intégrité de nos applications et environnements, nous devons également atténuer les vecteurs d'attaque sur notre pipeline de déploiement. Les risques incluent les développeurs introduisant du code permettant un accès non autorisé (ce que nous avons atténué par des contrôles tels que les tests de code, les révisions de code et les tests de pénétration) et les utilisateurs non autorisés accédant à notre code ou à notre environnement (ce que nous avons atténué par des contrôles tels que l'assurance que les configurations correspondent à des états connus comme bons, et un patching efficace).

Cependant, pour protéger notre pipeline de construction, d'intégration ou de déploiement continu, nos stratégies d'atténuation peuvent inclure :

- Renforcer les serveurs de construction et d'intégration continue et s'assurer que nous pouvons les reproduire de manière automatisée, tout comme nous le ferions pour l'infrastructure qui soutient les services de production destinés aux clients, pour empêcher que nos serveurs de construction et d'intégration continue ne soient compromis.
- Examiner tous les changements introduits dans le contrôle de version, soit par la programmation en binôme au moment de l'engagement, soit par un processus de révision de code entre l'engagement et la fusion dans la branche principale, pour éviter que les serveurs d'intégration continue n'exécutent du code incontrôlé (par exemple, les tests unitaires peuvent contenir du code malveillant permettant ou facilitant un accès non autorisé).
- Instrumenter notre dépôt pour détecter lorsque du code de test contenant des appels API suspects (par exemple, des tests unitaires accédant au système de fichiers ou au réseau) est vérifié dans le dépôt, éventuellement en le mettant en quarantaine et en déclenchant une révision de code immédiate.
- S'assurer que chaque processus CI fonctionne sur son propre conteneur ou VM isolé.
- S'assurer que les identifiants de contrôle de version utilisés par le système CI sont en lecture seule.

Conclusion

Tout au long de ce chapitre, nous avons décrit des moyens d'intégrer les objectifs de sécurité de l'information dans toutes les étapes de notre travail quotidien. Nous le faisons en intégrant les contrôles de sécurité dans les mécanismes que nous avons déjà créés, en veillant à ce que tous les environnements à la demande soient également dans un état durci et réduit en risques - en intégrant les tests de sécurité dans le pipeline de déploiement et en veillant à la création de télémétrie de sécurité dans les environnements pré-production et de production.

Ce faisant, nous permettons d'augmenter la productivité des développeurs et des opérations tout en augmentant simultanément notre sécurité globale. Notre prochaine étape est de protéger le pipeline de déploiement.

Protection de la pipeline de déploiement

Tout au long de ce chapitre, nous examinerons comment protéger notre pipeline de déploiement, ainsi que comment atteindre les objectifs de sécurité et de conformité dans notre environnement de contrôle, y compris la gestion des changements et la séparation des responsabilités.

Intégrer la sécurité et la conformité dans les processus d'approbation des changements

Presque toute organisation informatique de taille significative aura des processus de gestion des changements existants, qui sont les principaux contrôles pour réduire les risques opérationnels et de sécurité. Les responsables de la conformité et de la sécurité se fient aux processus de gestion des changements pour les exigences de conformité, et ils demandent généralement des preuves que tous les changements ont été correctement autorisés.

Si nous avons correctement construit notre pipeline de déploiement de manière que les déploiements soient à faible risque, la majorité de nos changements n'auront pas besoin de passer par un processus d'approbation manuelle, car nous nous appuierons sur des contrôles tels que les tests automatisés et la surveillance proactive de la production.

À cette étape, nous ferons ce qui est nécessaire pour garantir que nous pouvons intégrer avec succès la sécurité et la conformité dans tout processus de gestion des changements existant. Des politiques de gestion des changements efficaces reconnaîtront qu'il existe différents risques associés à différents types de changements et que ces changements sont tous traités différemment. Ces processus sont définis dans l'ITIL, qui divise les changements en trois catégories :

- **Changements standards** : Ce sont des changements à faible risque qui suivent un processus établi et approuvé, mais peuvent également être préapprouvés. Ils incluent les mises à jour mensuelles des tables de taxes d'application ou des codes pays, les modifications de contenu et de style de site web, et certains types de correctifs d'application ou de système d'exploitation dont l'impact est bien compris. Le proposant du changement n'a pas besoin d'approbation avant de déployer le changement, et les déploiements de changement peuvent être entièrement automatisés et doivent être enregistrés pour assurer une traçabilité.
- **Changements normaux** : Ce sont des changements à risque plus élevé qui nécessitent une révision ou une approbation de l'autorité de changement convenue. Dans de nombreuses organisations, cette responsabilité est placée de manière inappropriée sur le comité consultatif des changements (CAB) ou le comité consultatif des changements d'urgence (ECAB), qui peuvent manquer de l'expertise requise pour comprendre pleinement l'impact du changement, conduisant souvent à des délais inacceptablement longs. Ce problème est particulièrement pertinent pour les grands

déploiements de code, qui peuvent contenir des centaines de milliers (ou même des millions) de lignes de nouveau code, soumises par des centaines de développeurs sur plusieurs mois. Pour que les changements normaux soient autorisés, le CAB aura presque certainement un formulaire de demande de changement (RFC) bien défini qui précise les informations nécessaires pour la décision d'approbation. Le formulaire RFC inclut généralement les résultats commerciaux souhaités, l'utilité et la garantie planifiées, un dossier commercial avec des risques et des alternatives, et un calendrier proposé.

- **Changements urgents** : Ce sont des changements d'urgence, et donc potentiellement à haut risque, qui doivent être mis en production immédiatement (par exemple, un correctif de sécurité urgent, une restauration de service). Ces changements nécessitent souvent l'approbation de la direction, mais permettent que la documentation soit effectuée après coup. Un objectif clé des pratiques DevOps est de rationaliser notre processus de changement normal pour qu'il soit également adapté aux changements d'urgence.

Recatégoriser la majorité de nos changements à faible risque comme des changements standards

Idéalement, en ayant un pipeline de déploiement fiable en place, nous aurons déjà acquis une réputation de déploiements rapides, fiables et non dramatiques. À ce stade, nous devrions chercher à obtenir l'accord des opérations et des autorités de changement concernées pour que nos changements soient suffisamment démontrés comme étant à faible risque pour être définis comme des changements standards, préapprouvés par le CAB. Cela nous permet de déployer en production sans besoin d'approbation supplémentaire, bien que les changements doivent toujours être correctement enregistrés.

Une façon de soutenir l'affirmation que nos changements sont à faible risque est de montrer un historique des changements sur une période significative (par exemple, des mois ou des trimestres) et de fournir une liste complète des problèmes de production pendant cette même période. Si nous pouvons montrer des taux de succès élevés des changements et un MTTR bas, nous pouvons affirmer que nous avons un environnement de contrôle qui empêche efficacement les erreurs de déploiement, ainsi que prouver que nous pouvons détecter et corriger efficacement et rapidement tout problème résultant.

Même lorsque nos changements sont catégorisés comme des changements standards, ils doivent toujours être visibles et enregistrés dans nos systèmes de gestion des changements (par exemple, Remedy ou ServiceNow). Idéalement, les déploiements seront effectués automatiquement par nos outils de gestion de configuration et de pipeline de déploiement (par exemple, Puppet, Chef, Jenkins) et les résultats seront automatiquement enregistrés. En faisant cela, tout le monde dans notre organisation (DevOps ou non) aura une visibilité sur nos changements en plus de tous les autres changements se produisant dans l'organisation.

Nous pouvons automatiquement lier ces enregistrements de demandes de changement à des éléments spécifiques dans nos outils de planification de travail (par exemple, JIRA, Rally, LeanKit, ThoughtWorks Mingle), ce qui nous permet de créer plus de contexte pour nos changements, tels que des liens vers des défauts de fonctionnalités, des incidents de production, ou des user stories. Cela peut être accompli de manière légère en incluant des numéros de ticket des outils de planification dans les commentaires associés aux validations de contrôle de version. En faisant cela, nous pouvons tracer un déploiement en production aux changements dans le contrôle de version et, de là, les tracer plus loin jusqu'aux tickets des outils de planification.

Créer cette traçabilité et ce contexte devrait être facile et ne devrait pas créer une charge trop lourde ou chronophage pour les ingénieurs. Le lien vers des user stories, des exigences ou des défauts est presque certainement suffisant - tout détail supplémentaire, tel que l'ouverture d'un ticket pour chaque validation dans le contrôle de version, est probablement inutile, et donc indésirable, car cela imposerait un niveau de friction significatif sur leur travail quotidien.

Que faire lorsque les changements sont classés comme des changements normaux

Pour les changements que nous ne pouvons pas faire classer comme des changements standards, ils seront considérés comme des changements normaux et nécessiteront l'approbation d'au moins un sous-ensemble du CAB avant le déploiement. Dans ce cas, notre objectif est toujours de garantir que nous pouvons déployer rapidement, même si ce n'est pas entièrement automatisé.

Dans ce cas, nous devons nous assurer que toute demande de changement soumise est aussi complète et précise que possible, en fournissant au CAB tout ce dont il a besoin pour évaluer correctement notre changement. Après tout, si notre demande de changement est mal formée ou incomplète, elle nous sera renvoyée, augmentant le temps nécessaire pour passer en production et jetant le doute sur notre compréhension des objectifs du processus de gestion des changements.

Nous pouvons presque certainement automatiser la création de RFCs complètes et précises, en remplissant le ticket avec des détails sur ce qui doit être changé. Par exemple, nous pourrions automatiquement créer un ticket de changement ServiceNow avec un lien vers la user story JIRA, ainsi que les manifestes de construction et les résultats des tests de notre outil de pipeline de déploiement et des liens vers les scripts Puppet/Chef qui seront exécutés.

Étant donné que nos changements soumis seront évalués manuellement par des personnes, il est encore plus important de décrire le contexte du changement. Cela inclut l'identification de la raison pour laquelle nous apportons le changement (par exemple, en fournissant un lien vers les fonctionnalités, les défauts ou les incidents), qui est affecté par le changement, et ce qui va être changé.

Notre objectif est de partager les preuves et les artefacts qui nous donnent confiance que le changement fonctionnera en production comme prévu. Bien que les RFCs aient généralement des champs de texte libre, nous devrions fournir des liens vers des données lisibles par machine pour permettre à d'autres d'intégrer et de traiter nos données (par exemple, des liens vers des fichiers JSON).

Dans de nombreuses chaînes d'outils, cela peut être fait de manière conforme et entièrement automatisée. Par exemple, Mingle et Go de ThoughtWorks peuvent automatiquement lier ces informations ensemble, telles qu'une liste des défauts corrigés et des nouvelles fonctionnalités complétées associées au changement, et les inclure dans un RFC.

Lors de la soumission de notre RFC, les membres pertinents du CAB examineront, traiteront et approuveront ces changements comme ils le feraient pour toute autre demande de changement soumise. Si tout se passe bien, les autorités de changement apprécieront la rigueur et les détails de nos changements soumis, car nous leur avons permis de valider rapidement l'exactitude des informations que nous avons fournies (par exemple, en visualisant les liens vers les artefacts de nos outils de pipeline de déploiement). Cependant, notre objectif devrait être de montrer continuellement un historique exemplaire de changements réussis, afin que nous puissions finalement obtenir leur accord pour que nos changements automatisés soient classés en tant que changements standards.

Étude de cas - Changements d'infrastructure automatisés en tant que changements standards chez Salesforce.com (2012)

Salesforce a été fondée en 2000 dans le but de rendre la gestion de la relation client facilement disponible et livrable en tant que service. Les offres de Salesforce ont été largement adoptées par le marché, conduisant à une introduction en bourse réussie en 2004. En 2007, l'entreprise comptait plus de cinquante-neuf mille clients d'entreprise, traitant des centaines de millions de transactions par jour, avec un revenu annuel de 497 millions de dollars.

Cependant, à peu près à la même époque, leur capacité à développer et à déployer de nouvelles fonctionnalités à leurs clients semblait s'arrêter. En 2006, ils ont eu quatre grandes versions pour les clients, mais en 2007, ils n'ont pu faire qu'une seule version pour les clients malgré l'embauche de plus d'ingénieurs. Le résultat a été que le nombre de fonctionnalités

livrées par équipe a continué de diminuer et les jours entre les versions majeures ont continué d'augmenter.

Et parce que la taille de lot de chaque version continuait d'augmenter, les résultats des déploiements devenaient également de plus en plus mauvais. Karthik Rajan, alors vice-président de l'ingénierie des infrastructures, rapporte dans une présentation de 2013 que 2007 a marqué "la dernière année où le logiciel a été créé et expédié en utilisant un processus en cascade et lorsque nous avons fait notre transition vers un processus de livraison plus incrémental."

Lors du DevOps Enterprise Summit 2014, Dave Mangot et Reena Mathew ont décrit la transformation DevOps pluriannuelle qui a commencé en 2009. Selon Mangot et Mathew, en mettant en œuvre des principes et des pratiques DevOps, l'entreprise a réduit ses délais de déploiement de six jours à cinq minutes en 2013. En conséquence, ils ont pu augmenter plus facilement la capacité, leur permettant de traiter plus d'un milliard de transactions par jour.

L'un des principaux thèmes de la transformation de Salesforce était de faire de l'ingénierie de qualité le travail de tout le monde, que ce soit au sein du développement, des opérations ou de la sécurité de l'information. Pour ce faire, ils ont intégré des tests automatisés à toutes les étapes de la création d'applications et d'environnements, ainsi que dans le processus d'intégration et de déploiement continu, et ont créé l'outil open source Ruster pour effectuer des tests fonctionnels de leurs modules Puppet.

Ils ont également commencé à effectuer régulièrement des tests destructifs, un terme utilisé dans la fabrication pour désigner les tests d'endurance prolongés dans les conditions de fonctionnement les plus sévères jusqu'à ce que le composant testé soit détruit. L'équipe de Salesforce a commencé à tester régulièrement leurs services sous des charges de plus en plus élevées jusqu'à ce que le service se casse, ce qui les a aidés à comprendre leurs modes de défaillance et à apporter les corrections appropriées. Sans surprise, le résultat a été une qualité de service nettement supérieure avec des charges de production normales.

La sécurité de l'information a également travaillé avec l'ingénierie de qualité dès les premières étapes de leur projet, collaborant continuellement dans des phases critiques telles que la conception de l'architecture et des tests, ainsi qu'intégrant correctement les outils de sécurité dans le processus de test automatisé.

Pour Mangot et Mathew, l'un des principaux succès de toute la répétabilité et de la rigueur qu'ils ont conçues dans le processus a été de se faire dire par leur groupe de gestion des changements que "les changements d'infrastructure effectués via Puppet seraient désormais traités comme des 'changements standards,' nécessitant beaucoup moins, voire aucune autre

approbation du CAB." De plus, ils ont noté que "les changements manuels de l'infrastructure nécessiteraient encore des approbations."

En faisant cela, ils ont non seulement intégré leurs processus DevOps avec le processus de gestion des changements, mais ont également créé une motivation supplémentaire pour automatiser le processus de changement pour une plus grande partie de leur infrastructure.

Réduire la dépendance à la séparation des tâches

Pendant des décennies, nous avons utilisé la séparation des tâches comme l'un de nos principaux contrôles pour réduire le risque de fraude ou d'erreurs dans le processus de développement logiciel. Il a été courant dans la plupart des SDLCs d'exiger que les modifications des développeurs soient soumises à un bibliothécaire de code, qui examinerait et approuverait la modification avant que les opérations informatiques ne promeuvent la modification en production.

Il existe de nombreux autres exemples moins controversés de séparation des tâches dans le travail des opérations, comme les administrateurs de serveurs pouvant idéalement consulter les journaux mais ne pouvant pas les supprimer ou les modifier, afin d'empêcher quiconque ayant un accès privilégié de supprimer des preuves de fraude ou d'autres problèmes.

Lorsque nous faisions des déploiements en production moins fréquemment (par exemple, annuellement) et lorsque notre travail était moins complexe, compartimenter notre travail et effectuer des transferts étaient des façons viables de mener nos activités. Cependant, à mesure que la complexité et la fréquence des déploiements augmentent, réussir les déploiements en production nécessite de plus en plus que tout le monde dans la chaîne de valeur voie rapidement les résultats de ses actions.

La séparation des tâches peut souvent entraver cela en ralentissant et en réduisant les retours que les ingénieurs reçoivent sur leur travail. Cela empêche les ingénieurs de prendre pleine responsabilité de la qualité de leur travail et réduit la capacité de l'entreprise à créer un apprentissage organisationnel.

Par conséquent, dans la mesure du possible, nous devrions éviter d'utiliser la séparation des tâches comme un contrôle. Au lieu de cela, nous devrions choisir des contrôles tels que la programmation en binôme, l'inspection continue des validations de code et la revue de code. Ces contrôles peuvent nous donner la tranquillité d'esprit nécessaire concernant la qualité de notre travail. De plus, en mettant en place ces contrôles, si la séparation des tâches est nécessaire, nous pouvons montrer que nous atteignons des résultats équivalents avec les contrôles que nous avons créés.

Étude de cas - Conformité PCI et un conte de précaution sur la séparation des tâches chez Etsy (2014)

Bill Massie est un responsable du développement chez Etsy et est responsable de l'application de paiement appelée ICHT (une abréviation de "I Can Haz Tokens"). ICHT prend les commandes de crédit des clients via un ensemble d'applications de traitement des paiements développées en interne qui gèrent la saisie des commandes en ligne en prenant les données des titulaires de carte saisies par les clients, les tokenisant, communiquant avec le processeur de paiement et complétant la transaction de commande.

Étant donné que la portée des normes de sécurité des données de l'industrie des cartes de paiement (PCI DSS) pour l'environnement des données des titulaires de carte (CDE) est "les personnes, les processus et la technologie qui stockent, traitent ou transmettent les données des titulaires de carte ou les données d'authentification sensibles", y compris tous les composants de système connectés, l'application ICHT est incluse dans la portée du PCI DSS.

Pour limiter la portée du PCI DSS, l'application ICHT est physiquement et logiquement séparée du reste de l'organisation Etsy et est gérée par une équipe d'application complètement distincte composée de développeurs, d'ingénieurs de bases de données, d'ingénieurs réseau et d'ingénieurs ops. Chaque membre de l'équipe reçoit deux ordinateurs portables : un pour ICHT (qui est configuré différemment pour répondre aux exigences DSS, ainsi que verrouillé dans un coffre-fort lorsqu'il n'est pas utilisé) et un pour le reste d'Etsy.

En faisant cela, ils ont pu découpler l'environnement CDE du reste de l'organisation Etsy, limitant la portée des réglementations PCI DSS à une zone séparée. Les systèmes qui forment le CDE sont séparés (et gérés différemment) du reste des environnements d'Etsy au niveau physique, réseau, code source et infrastructure logique. De plus, le CDE est construit et exploité par une équipe interfonctionnelle qui est uniquement responsable du CDE.

L'équipe ICHT a dû modifier ses pratiques de livraison continue pour tenir compte de la nécessité d'approuver le code. Selon la section 6.3.2 du PCI DSS v3.1, les équipes doivent examiner :

Tout code personnalisé avant la mise en production ou la mise à disposition des clients afin d'identifier toute vulnérabilité potentielle de codage (en utilisant des processus manuels ou automatisés) comme suit :

- Les modifications de code sont-elles examinées par des personnes autres que l'auteur initial du code, et par des personnes connaissant les techniques de revue de code et les pratiques de codage sécurisé ?
- Les revues de code garantissent-elles que le code soit développé conformément aux directives de codage sécurisé ?
- Les corrections appropriées sont-elles mises en œuvre avant la mise en production ?
- Les résultats des revues de code sont-ils examinés et approuvés par la direction avant la mise en production ?

Pour répondre à cette exigence, l'équipe a initialement décidé de désigner Massie comme l'approbateur des changements responsable du déploiement de toutes les modifications en production. Les déploiements souhaités seraient signalés dans JIRA, et Massie les marquerait comme examinés et approuvés, et les déployerait manuellement dans la production ICHT.

Cela a permis à Etsy de répondre à ses exigences PCI DSS et d'obtenir son rapport de conformité signé par ses évaluateurs. Cependant, en ce qui concerne l'équipe, des problèmes importants ont surgi.

Massie observe qu'un effet secondaire troublant "est un niveau de 'compartimentation' qui se produit dans l'équipe ICHT qu'aucun autre groupe n'a chez Etsy. Depuis que nous avons mis en place la séparation des tâches et d'autres contrôles requis par la conformité PCI DSS, personne ne peut être un ingénieur full-stack dans cet environnement."

En conséquence, tandis que le reste des équipes de développement et des opérations chez Etsy travaillent ensemble étroitement et déploient des changements en douceur et en toute confiance, Massie note que "dans notre environnement PCI, il y a une peur et une réticence autour du déploiement et de la maintenance parce que personne n'a de visibilité en dehors de sa partie de la pile logicielle. Les changements apparemment mineurs que nous avons apportés à notre façon de travailler semblent avoir créé un mur impénétrable entre les développeurs et les ops, et créent une tension indéniable que personne chez Etsy n'a connue depuis 2008. Même si vous avez confiance en votre partie, il est impossible d'avoir confiance que le changement de quelqu'un d'autre ne va pas casser votre partie de la pile."

Cette étude de cas montre que la conformité est possible dans les organisations utilisant DevOps. Cependant, le conte de précaution potentiel ici est que toutes les vertus que nous associons aux équipes DevOps performantes sont fragiles : même une équipe qui a des expériences partagées avec une grande confiance et des objectifs communs peut commencer à lutter lorsque des mécanismes de contrôle à faible confiance sont mis en place.

Assurer la documentation et les preuves pour les auditeurs et les agents de conformité

Alors que les organisations technologiques adoptent de plus en plus les modèles DevOps, la tension entre l'informatique et l'audit est plus forte que jamais. Ces nouveaux modèles DevOps remettent en question la pensée traditionnelle sur l'audit, les contrôles et l'atténuation des risques.

Comme le remarque Bill Shinn, architecte principal de solutions de sécurité chez Amazon Web Services, « DevOps consiste à combler le fossé entre Dev et Ops. D'une certaine manière, le défi

de combler le fossé entre DevOps et les auditeurs et agents de conformité est encore plus grand. Par exemple, combien d'auditeurs peuvent lire du code et combien de développeurs ont lu le NIST 800-37 ou la loi Gramm-Leach-Bliley ? Cela crée un fossé de connaissances, et la communauté DevOps doit aider à combler ce fossé. »

Étude de Cas - Prouver la Conformité dans les Environnements Réglementés (2015)

Aider les grands clients d'entreprise à prouver qu'ils peuvent toujours se conformer à toutes les lois et réglementations pertinentes fait partie des responsabilités de Bill Shinn en tant qu'architecte principal de solutions de sécurité chez Amazon Web Services. Au fil des ans, il a passé du temps avec plus d'un millier de clients d'entreprise, notamment Hearst Media, GE, Phillips et Pacific Life, qui ont publiquement mentionné leur utilisation des clouds publics dans des environnements hautement réglementés.

Shinn note : « L'un des problèmes est que les auditeurs ont été formés à des méthodes qui ne sont pas très adaptées aux modèles de travail DevOps. Par exemple, si un auditeur voyait un environnement avec dix mille serveurs de production, il a été traditionnellement formé à demander un échantillon de mille serveurs, accompagné de preuves sous forme de captures d'écran de la gestion des actifs, des paramètres de contrôle d'accès, des installations d'agents, des journaux de serveurs, etc. »

« Cela était acceptable pour les environnements physiques, » continue Shinn. « Mais lorsque l'infrastructure est du code, et que l'auto-scaling fait apparaître et disparaître les serveurs tout le temps, comment échantillonner cela ? Vous rencontrez les mêmes problèmes lorsque vous avez un pipeline de déploiement, très différent du processus traditionnel de développement logiciel, où un groupe écrit le code et un autre groupe déploie ce code en production. »

Il explique : « Lors des travaux de terrain d'audit, les méthodes les plus courantes de collecte de preuves sont encore les captures d'écran et les fichiers CSV remplis de paramètres de configuration et de journaux. Notre objectif est de créer des méthodes alternatives de présentation des données qui montrent clairement aux auditeurs que nos contrôles sont opérationnels et efficaces. »

Pour aider à combler ce fossé, il fait travailler des équipes avec des auditeurs dans le processus de conception des contrôles. Ils utilisent une approche itérative, assignant un contrôle unique pour chaque sprint afin de déterminer ce qui est nécessaire en termes de preuves d'audit. Cela a permis de garantir que les auditeurs obtiennent les informations dont ils ont besoin lorsque le service est en production, entièrement à la demande.

Shinn déclare que le meilleur moyen d'y parvenir est de « envoyer toutes les données dans nos systèmes de télémétrie, tels que Splunk ou Kibana. De cette manière, les auditeurs peuvent obtenir ce dont ils ont besoin, en libre-service complet. Ils n'ont pas besoin de demander un échantillon de données, ils se connectent à Kibana, puis recherchent les preuves d'audit dont ils ont besoin pour une période donnée. Idéalement, ils verront très rapidement qu'il y a des preuves pour montrer que nos contrôles fonctionnent. »

Shinn continue : « Avec la journalisation moderne des audits, les salles de discussion et les pipelines de déploiement, il y a une visibilité et une transparence sans précédent sur ce qui se passe en production, surtout comparé à la manière dont les opérations étaient faites auparavant, avec une bien plus faible probabilité d'erreurs et de failles de sécurité. Donc, le défi est de transformer toutes ces preuves en quelque chose qu'un auditeur reconnaît. »

Cela nécessite de dériver les exigences techniques des réglementations réelles. Shinn explique : « Pour découvrir ce que HIPAA exige d'un point de vue de la sécurité de l'information, vous devez consulter la législation du CFR Part 160, aller dans les sous-parties A et C de la partie 164. Même alors, vous devez continuer à lire jusqu'à atteindre les 'sauvegardes techniques et les contrôles d'audit'. Ce n'est qu'à ce moment que vous verrez que ce qui est requis, c'est que nous déterminions les activités qui seront suivies et auditées en rapport avec les informations de santé des patients, que nous documentons et mettons en œuvre ces contrôles, sélectionnons des outils, et enfin que nous examinons et capturons les informations appropriées. »

Shinn continue : « Comment remplir cette exigence est la discussion qui doit avoir lieu entre les agents de conformité et de réglementation, et les équipes de sécurité et de DevOps, spécifiquement autour de la manière de prévenir, détecter et corriger les problèmes. Parfois, cela peut être rempli par un paramètre de configuration dans le contrôle de version. D'autres fois, c'est un contrôle de surveillance. »

Shinn donne un exemple : « Nous pouvons choisir de mettre en œuvre l'un de ces contrôles en utilisant AWS CloudWatch, et nous pouvons tester que le contrôle fonctionne avec une seule ligne de commande. De plus, nous devons montrer où vont les journaux—idéalement, nous poussons tout cela dans notre cadre de journalisation, où nous pouvons lier les preuves d'audit à l'exigence de contrôle réelle. »

Pour aider à résoudre ce problème, le DevOps Audit Defense Toolkit décrit le récit de bout en bout du processus de conformité et d'audit pour une organisation fictive (Parts Unlimited de The Phoenix Project). Il commence par décrire les objectifs organisationnels de l'entité, les processus métier, les principaux risques et l'environnement de contrôle en résultant, ainsi que la manière dont la direction pourrait prouver avec succès que les contrôles existent et sont efficaces. Un ensemble d'objections d'audit est également présenté, ainsi que la manière de les surmonter.

Le document décrit comment les contrôles pourraient être conçus dans un pipeline de déploiement pour atténuer les risques indiqués, et fournit des exemples d'attestations de contrôle et d'artefacts de contrôle pour démontrer l'efficacité des contrôles. Il était destiné à être général pour tous les objectifs de contrôle, y compris pour soutenir la précision des rapports financiers, la conformité réglementaire (par exemple, SEC SOX-404, HIPAA, FedRAMP, EU Model Contracts et les règlements proposés SEC Reg-SCI), les obligations contractuelles (par exemple, PCI DSS, DOD DISA), et les opérations efficaces et efficientes.

Étude de Cas - S'appuyer sur la Télémétrie de Production pour les Systèmes de Distributeurs Automatiques de Billets

Mary Smith (un pseudonyme) dirige l'initiative DevOps pour la branche bancaire de détail d'une grande organisation financière américaine. Elle a observé que la sécurité de l'information, les auditeurs et les régulateurs se fient souvent trop aux revues de code pour détecter les fraudes. Au lieu de cela, ils devraient compter sur les contrôles de surveillance en production en plus d'utiliser des tests automatisés, des revues de code et des approbations pour atténuer efficacement les risques associés aux erreurs et à la fraude.

Elle a observé :

Il y a de nombreuses années, nous avions un développeur qui avait introduit une porte dérobée dans le code que nous déployons sur nos distributeurs automatiques de billets. Il a pu mettre les distributeurs en mode maintenance à certains moments, ce qui lui permettait de retirer de l'argent des machines. Nous avons pu détecter la fraude très rapidement, et ce n'était pas par le biais d'une revue de code. Ces types de portes dérobées sont difficiles, voire impossibles, à détecter lorsque les auteurs ont des moyens, des motifs et des opportunités suffisants.

Cependant, nous avons rapidement détecté la fraude lors de notre réunion régulière de révision des opérations, lorsque quelqu'un a remarqué que les distributeurs automatiques dans une ville étaient mis en mode maintenance à des moments non prévus. Nous avons découvert la fraude même avant le processus de vérification de l'argent prévu, lorsque l'on réconcilie le montant d'argent dans les distributeurs avec les transactions autorisées.

Dans cette étude de cas, la fraude s'est produite malgré la séparation des fonctions entre Développement et Opérations et un processus d'approbation des changements, mais a été rapidement détectée et corrigée grâce à une télémétrie de production efficace.

Conclusion

Tout au long de ce chapitre, nous avons discuté des pratiques qui font de la sécurité de l'information la responsabilité de chacun, où tous nos objectifs de sécurité de l'information sont intégrés dans le travail quotidien de chaque personne dans le flux de valeur. En faisant cela, nous améliorons considérablement l'efficacité de nos contrôles, afin de mieux prévenir les violations de sécurité, ainsi que de les détecter et de nous en remettre plus rapidement. Et nous réduisons considérablement le travail associé à la préparation et à la réussite des audits de conformité.

Partie VI Conclusion

Tout au long des chapitres précédents, nous avons exploré comment appliquer les principes DevOps à la sécurité de l'information, en nous aidant à atteindre nos objectifs, et en veillant à ce que la sécurité fasse partie du travail de chacun, chaque jour. Une meilleure sécurité garantit que nous protégeons nos données de manière défendable et judicieuse, que nous pouvons nous remettre des problèmes de sécurité avant qu'ils ne deviennent catastrophiques, et, surtout, que nous pouvons améliorer la sécurité de nos systèmes et de nos données comme jamais auparavant.

Un Appel à l'Action - Conclusion du DevOps Handbook

Nous avons atteint la fin d'une exploration détaillée des principes et des pratiques techniques du DevOps. À une époque où chaque leader technologique est confronté à la nécessité d'assurer la sécurité, la fiabilité et l'agilité, et où les violations de sécurité, le délai de mise sur le marché et les transformations technologiques massives se multiplient, le DevOps offre une solution. Nous espérons que ce livre a fourni une compréhension approfondie du problème et une feuille de route pour créer des solutions pertinentes.

Comme nous l'avons exploré tout au long du DevOps Handbook, nous savons que, laissée sans gestion, un conflit inhérent peut exister entre le Développement et les Opérations, créant des problèmes de plus en plus graves, ce qui entraîne un délai de mise sur le marché plus long pour les nouveaux produits et fonctionnalités, une qualité médiocre, une augmentation des pannes et de la dette technique, une productivité réduite des ingénieurs, ainsi qu'une insatisfaction et un épuisement professionnels accrus.

Les principes et les modèles DevOps nous permettent de briser ce conflit chronique. Après avoir lu ce livre, nous espérons que vous verrez comment une transformation DevOps peut permettre la création d'organisations apprenantes dynamiques, atteignant des résultats incroyables tels qu'un flux rapide et une fiabilité et une sécurité de classe mondiale, ainsi qu'une compétitivité accrue et une satisfaction des employés améliorée.

Le DevOps nécessite potentiellement de nouvelles normes culturelles et de gestion, ainsi que des changements dans nos pratiques techniques et notre architecture. Cela nécessite une coalition qui englobe la direction de l'entreprise, la gestion des produits, le développement, l'assurance qualité, les opérations informatiques, la sécurité de l'information, et même le marketing, où de nombreuses initiatives technologiques prennent naissance. Lorsque toutes ces équipes travaillent ensemble, nous pouvons créer un système de travail sûr, permettant à de petites équipes de développer et de valider rapidement et indépendamment du code qui peut être déployé en toute sécurité chez les clients. Cela se traduit par une productivité maximale des développeurs, un apprentissage organisationnel, une satisfaction élevée des employés et la capacité de réussir sur le marché.

Notre objectif en écrivant ce livre était de codifier suffisamment les principes et les pratiques DevOps pour que les résultats incroyables obtenus au sein de la communauté DevOps puissent être reproduits par d'autres. Nous espérons accélérer l'adoption des initiatives DevOps et soutenir leur mise en œuvre réussie tout en réduisant l'énergie d'activation nécessaire à leur réalisation.

Nous connaissons les dangers du report des améliorations et de la satisfaction des solutions de contournement quotidiennes, ainsi que les difficultés de changer la manière dont nous priorisons et effectuons notre travail quotidien. De plus, nous comprenons les risques et les efforts nécessaires pour amener les organisations à adopter une manière différente de travailler, ainsi que la perception que le DevOps est une mode passagère, bientôt remplacée par le prochain mot à la mode.

Nous affirmons que le DevOps est transformateur pour la façon dont nous effectuons le travail technologique, tout comme le Lean a transformé à jamais la manière dont le travail de fabrication était effectué dans les années 1980. Ceux qui adoptent le DevOps réussiront sur le marché, au détriment de ceux qui ne le feront pas. Ils créeront des organisations dynamiques et en apprentissage continu qui surpasseront et innoveront davantage que leurs concurrents.

De ce fait, le DevOps n'est pas seulement une nécessité technologique, mais aussi une nécessité organisationnelle. En fin de compte, le DevOps est applicable et pertinent pour toutes les organisations qui doivent augmenter le flux de travail prévu à travers l'organisation technologique, tout en maintenant la qualité, la fiabilité et la sécurité pour nos clients.

Notre appel à l'action est le suivant : peu importe le rôle que vous jouez dans votre organisation, commencez à trouver des personnes autour de vous qui veulent changer la manière dont le travail est effectué. Montrez ce livre à d'autres et créez une coalition de penseurs partageant les mêmes idées pour sortir de la spirale descendante. Demandez aux dirigeants organisationnels de soutenir ces efforts ou, mieux encore, parrainez et dirigez ces efforts vous-même.

Enfin, puisque vous êtes arrivé jusque-là, nous avons un secret à révéler. Dans de nombreuses études de cas, après l'obtention des résultats révolutionnaires présentés, de nombreux agents du changement ont été promus - mais, dans certains cas, il y a eu un changement de direction qui a entraîné le départ de nombreuses personnes impliquées, accompagné d'un retour en arrière des changements organisationnels qu'ils avaient créés.

Nous pensons qu'il est important de ne pas être cynique à ce sujet. Les personnes impliquées dans ces transformations savaient d'emblée que ce qu'elles faisaient avait de fortes chances d'échouer, et elles ont quand même continué. En faisant cela, elles ont, peut-être le plus important, inspiré le reste d'entre nous en montrant ce qui peut être accompli. L'innovation est impossible sans prise de risques, et si vous n'avez pas réussi à bouleverser au moins certaines personnes dans la direction, vous n'essayez probablement pas assez fort. Ne laissez pas le système immunitaire de votre organisation vous détourner ou vous distraire de votre vision. Comme le dit Jesse Robbins, anciennement "maître du désastre" chez Amazon, "Ne combattez pas la stupidité, faites plus d'extraordinaire."

Le DevOps bénéficie à chacun d'entre nous dans la chaîne de valeur technologique, que nous soyons Développeurs, Ops, QA, Sécurité de l'information, Propriétaires de produits ou clients. Il rend le développement de produits exceptionnels plus agréable, avec moins de marches de la mort. Il permet des conditions de travail humaines avec moins de week-ends et de fêtes manquées avec nos proches. Il permet aux équipes de travailler ensemble pour survivre, apprendre, prospérer, ravir nos clients et aider notre organisation à réussir.

Nous espérons sincèrement que le DevOps Handbook vous aidera à atteindre ces objectifs.

Annexes

Annexe 1 : la convergence du Devops

Nous croyons que le DevOps bénéficie d'une convergence incroyable de mouvements de gestion, qui sont tous mutuellement renforçants et peuvent aider à créer une coalition puissante pour transformer la manière dont les organisations développent et livrent des produits et services informatiques.

John Willis a nommé cela « la Convergence du DevOps ». Les divers éléments de cette convergence sont décrits ci-dessous dans un ordre chronologique approximatif. (Notez que ces descriptions ne sont pas destinées à être des descriptions exhaustives, mais simplement suffisamment pour montrer la progression de la pensée et les connexions plutôt improbables qui ont conduit au DevOps.)

Le mouvement LEAN

Le Mouvement Lean a commencé dans les années 1980 comme une tentative de codifier le Système de Production Toyota avec la vulgarisation de techniques telles que la Cartographie de la Chaîne de Valeur, les tableaux kanban et la Maintenance Productive Totale.

Deux principes majeurs du Lean étaient la croyance profondément ancrée que le délai de production (c'est-à-dire le temps nécessaire pour convertir les matières premières en biens finis) était le meilleur prédicteur de la qualité, de la satisfaction client et du bonheur des employés ; et que l'un des meilleurs prédicteurs des délais courts était la taille réduite des lots, l'idéal théorique étant le « flux d'une seule pièce » (c'est-à-dire le flux « 1x1 » : inventaire de 1, taille de lot de 1).

Les principes Lean se concentrent sur la création de valeur pour le client : penser systématiquement, créer une constance de but, embrasser la pensée scientifique, créer du flux et tirer (au lieu de pousser), garantir la qualité à la source, diriger avec humilité et respecter chaque individu.

Le mouvement agile

Créé en 2001, le Manifeste Agile a été élaboré par dix-sept des principaux penseurs du développement logiciel, dans le but de transformer des méthodes légères telles que le DP et le DSDM en un mouvement plus large capable de rivaliser avec les processus de développement logiciel lourds comme le développement en cascade et les méthodologies telles que le Processus Unifié Rational.

Un principe clé était de « livrer un logiciel fonctionnel fréquemment, toutes les deux semaines à deux mois, avec une préférence pour les délais plus courts ». Deux autres principes se concentrent sur la nécessité de petites équipes autonomes, travaillant dans un modèle de gestion basé sur la confiance élevée et une emphase sur les tailles de lots réduites. Agile est également associé à un ensemble d'outils et de pratiques tels que Scrum, Standups, et autres.

Le mouvement de la conférence Velocity

Créée en 2007, la Conférence Velocity a été fondée par Steve Souders, John Allspaw et Jesse Robbins pour offrir un espace au groupe IT Operations et Web Performance. Lors de la conférence Velocity 2009, John Allspaw et Paul Hammond ont présenté la conférence pionnière « 10 Deployés par Jour : Coopération Dev et Ops chez Flickr ».

Le mouvement de l'infrastructure agile

Lors de la conférence Agile Toronto 2008, Patrick Dubois et Andrew Schafer ont tenu une session « oiseaux de même plumage » sur l'application des principes Agile à l'infrastructure plutôt qu'au code applicatif. Ils ont rapidement gagné un public de penseurs partageant les mêmes idées, y compris John Willis. Plus tard, Dubois, enthousiasmé par la présentation « 10 Deployés par Jour : Coopération Dev et Ops chez Flickr », a créé les premiers DevOpsDays à Gand, en Belgique, en 2009, en inventant le mot « DevOps ».

Le mouvement de la livraison continue

En s'appuyant sur la discipline du développement de build continu, de test et d'intégration, Jez Humble et David Farley ont étendu le concept de livraison continue, qui comprend un « pipeline de déploiement » pour garantir que le code et l'infrastructure sont toujours dans un état déployable et que tout code soumis au dépôt est déployé en production.

Cette idée a été présentée pour la première fois à Agile 2006 et a également été développée indépendamment par Tim Fitz dans un billet de blog intitulé « Continuous Deployment ».

Le mouvement Toyota Kata

En 2009, Mike Rother a écrit Toyota Kata : Managing People for Improvement, Adaptiveness and Superior Results, qui décrivait les apprentissages de son voyage de vingt ans pour comprendre et codifier les mécanismes causaux du Système de Production Toyota. Le Toyota Kata décrit les « routines et pensées managériales invisibles qui se cachent derrière le succès de Toyota en matière d'amélioration continue et d'adaptation... et comment d'autres entreprises développent des routines et des pensées similaires dans leurs organisations ».

Sa conclusion était que la communauté Lean avait manqué la pratique la plus importante de toutes, qu'il décrivait comme le Kata d'Amélioration. Il explique que chaque organisation a des

routines de travail, et le facteur critique chez Toyota était de rendre le travail d'amélioration habituel et de l'intégrer dans le travail quotidien de chacun dans l'organisation. Le Toyota Kata institue une approche itérative, incrémentale et scientifique de la résolution de problèmes dans la poursuite d'un vrai nord organisationnel partagé.

Le mouvement Lean Startup

En 2011, Eric Ries a écrit *The Lean Startup : How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*, en codifiant ses leçons apprises chez IMVU, une startup de la Silicon Valley, qui s'appuyait sur le travail de Steve Blank dans *The Four Steps to the Epiphany* ainsi que sur les techniques de déploiement continu. Eric Ries a également codifié des pratiques et termes connexes, y compris le Produit Minimum Viable, le cycle construire-mesurer-apprendre, et de nombreux modèles techniques de déploiement continu.

Le mouvement Lean UX

En 2013, Jeff Gothelf a écrit *Lean UX : Applying Lean Principles to Improve User Experience*, qui a codifié comment améliorer le « front end flou » et expliqué comment les propriétaires de produits peuvent formuler des hypothèses commerciales, expérimenter et acquérir de la confiance dans ces hypothèses avant d'investir du temps et des ressources dans les fonctionnalités résultantes. En ajoutant Lean UX, nous disposons désormais des outils pour optimiser complètement le flux entre les hypothèses commerciales, le développement de fonctionnalités, les tests, le déploiement et la livraison du service au client.

Le mouvement Rugged Computing

En 2011, Joshua Corman, David Rice et Jeff Williams ont examiné l'apparente futilité de sécuriser les applications et les environnements tard dans le cycle de vie. En réponse, ils ont créé une philosophie appelée « Rugged Computing », qui tente de cadrer les exigences non fonctionnelles de stabilité, évolutivité, disponibilité, survie, durabilité, sécurité, soutien, gestion et défense.

En raison du potentiel élevé de taux de publication, DevOps peut exercer une pression incroyable sur QA et Infosec, car lorsque les taux de déploiement passent de mensuels ou trimestriels à des centaines ou des milliers par jour, les délais de deux semaines pour Infosec ou QA ne sont plus tenables. Le mouvement Rugged Computing a postulé que l'approche actuelle pour combattre le complexe industriel vulnérable employé par la plupart des programmes de sécurité de l'information est désespérée.

Annexe 2 : Théorie des contraintes et conflits nucléaires, chroniques

Le corpus de connaissances de la Théorie des Contraintes discute largement de l'utilisation de la création de nuages de conflits essentiels (souvent appelés « C3 »). Voici le nuage de conflit pour les IT :

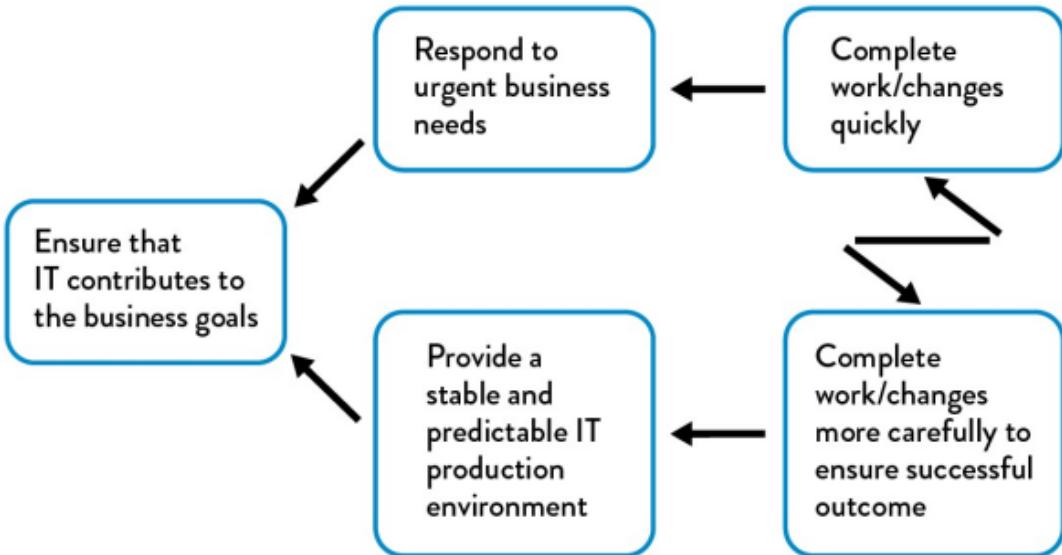


Figure 46: The core, chronic conflict facing every IT organization

Pendant les années 1980, il y avait un conflit de base bien connu dans le secteur manufacturier. Chaque responsable d'usine avait deux objectifs commerciaux valables : protéger les ventes et réduire les coûts. Le problème était qu'afin de protéger les ventes, la direction commerciale était incitée à augmenter les stocks pour s'assurer qu'il était toujours possible de satisfaire la demande des clients.

En revanche, pour réduire les coûts, la direction de la production était incitée à diminuer les stocks pour s'assurer que l'argent n'était pas immobilisé dans le travail en cours qui n'était pas immédiatement expédiable au client sous forme de ventes réalisées.

Ils ont pu résoudre ce conflit en adoptant les principes Lean, tels que la réduction des tailles de lots, la réduction du travail en cours et le raccourcissement et l'amplification des boucles de rétroaction. Cela a entraîné des augmentations spectaculaires de la productivité des usines, de la qualité des produits et de la satisfaction des clients.

Les principes sous-jacents aux modèles de travail DevOps sont les mêmes que ceux qui ont transformé le secteur manufacturier, nous permettant d'optimiser le flux de valeur IT, en convertissant les besoins commerciaux en capacités et services qui apportent de la valeur à nos clients.

Annexe 3 – Formule tabulaire de la spirale descendante

La forme colonne de la spirale descendante décrite dans Le Projet Phoenix est montrée ci-dessous.

IT Operations sees...	Development sees...
Fragile applications are prone to failure	Fragile applications are prone to failure
Long time required to figure out which bit got flipped	More urgent, date-driven projects put into the queue
Detective control is a salesperson	Even more fragile code (less secure) put into production
Too much time required to restore service	More releases have increasingly turbulent installs
Too much firefighting and unplanned work	Release cycles lengthen to amortize cost of deployments
Urgent security rework and remediation	Failing bigger deployments more difficult to diagnose
Planned project work cannot be completed	Most senior and constrained IT Operations resources have less time to fix underlying process problems
Frustrated customers leave	Ever increasing backlog of work that could help the business win
Market share goes down	Ever increasing amount of tension between IT Operations, Development, Design
Business misses Wall Street commitments	
Business makes even larger promises to Wall Street	

Annexe 4 – Les dangers des passages de relais et des files d'attente

Le problème des temps de file d'attente élevés est aggravé lorsqu'il y a de nombreux passages de relais, car c'est à ce moment-là que les files d'attente se créent. La Figure 47 montre le temps d'attente en fonction de l'occupation d'une ressource dans un centre de travail. La courbe asymptotique montre pourquoi un « simple changement de trente minutes » prend souvent des semaines à être complété - des ingénieurs et centres de travail spécifiques deviennent souvent des goulets d'étranglement problématiques lorsqu'ils fonctionnent à une haute utilisation. À mesure qu'un centre de travail approche de 100 % d'utilisation, tout travail requis à partir de celui-ci stagne dans les files d'attente et ne sera pas traité sans que quelqu'un ne le pousse ou n'escalade.

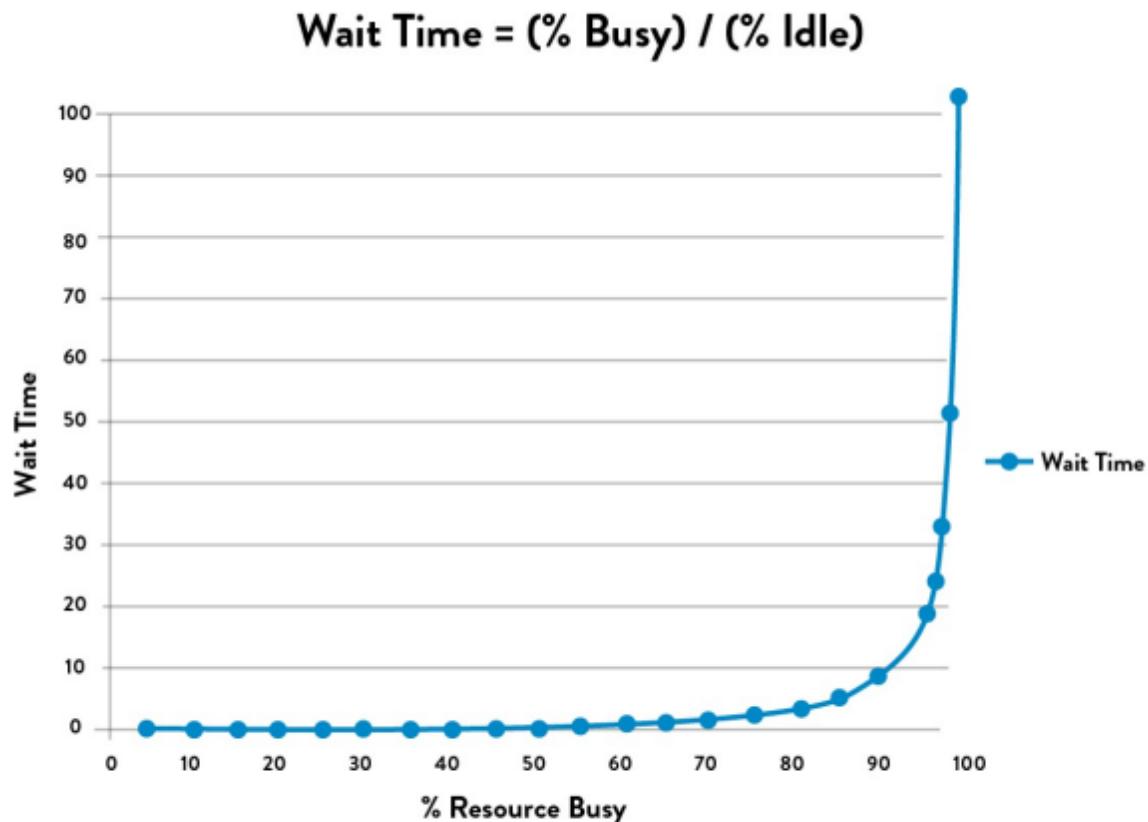


Figure 47: Queue size and wait times as function of percent utilization (Source: Kim, Behr, and Spafford, The Phoenix Project, ePub edition, 557.)

Dans la figure 47, l'axe des x représente le pourcentage d'occupation d'une ressource dans un centre de travail, et l'axe des y représente le temps d'attente approximatif (ou, plus précisément, la longueur de la file d'attente). Ce que la forme de la courbe montre, c'est qu'à mesure que l'utilisation de la ressource dépasse 80 %, le temps d'attente grimpe en flèche.

Dans Le Projet Phoenix, voici comment Bill et son équipe ont réalisé les conséquences dévastatrices de cette propriété sur les délais pour les engagements qu'ils faisaient auprès du bureau de gestion de projet :

« Je leur parle de ce qu'Erik m'a dit au MRP-8, sur la façon dont les temps d'attente dépendent de l'utilisation des ressources. “Le temps d'attente est le ‘pourcentage de temps occupé’ divisé par le ‘pourcentage de temps inactif’. En d'autres termes, si une ressource est occupée à cinquante pour cent, alors elle est inactive à cinquante pour cent. Le temps d'attente est de cinquante pour cent divisé par cinquante pour cent, soit une unité de temps. Appelons cela une heure.

Donc, en moyenne, notre tâche attendrait dans la file d'attente pendant une heure avant d'être traitée.

“D'autre part, si une ressource est occupée à quatre-vingt-dix pour cent, le temps d'attente est de ‘quatre-vingt-dix pour cent divisé par dix pour cent’, soit neuf heures. En d'autres termes, notre tâche attendrait dans la file d'attente neuf fois plus longtemps que si la ressource était inactive à cinquante pour cent.”

Je conclus, “Donc... Pour la tâche Phoenix, en supposant que nous ayons sept passages de relais, et que chacune de ces ressources soit occupée quatre-vingt-dix pour cent du temps, les tâches passeraient dans la file d'attente un total de neuf heures multipliées par les sept étapes...”

“Quoi ? Soixante-trois heures, juste pour le temps d'attente ?” dit Wes, incrédule. “C'est impossible !”

Patty dit avec un sourire en coin, “Oh, bien sûr. Parce que c'est seulement trente secondes de frappe, n'est-ce pas ?”

Bill et son équipe réalisent que leur “tâche simple de trente minutes” nécessite en réalité sept passages de relais (par exemple, équipe serveur, équipe réseau, équipe base de données, équipe virtualisation, et, bien sûr, Brent, l'ingénieur ‘rockstar’).

En supposant que tous les centres de travail étaient occupés à 90 %, la figure nous montre que le temps d'attente moyen à chaque centre de travail est de neuf heures - et comme le travail devait passer par sept centres de travail, le temps d'attente total est sept fois cela : soixante-trois heures.

En d'autres termes, le pourcentage total de temps de valeur ajoutée (parfois connu sous le nom de temps de traitement) n'était que de 0,16 % du délai total (trente minutes divisé par soixante-trois heures). Cela signifie que pour 99,8 % de notre délai total, le travail était simplement en attente dans la file d'attente, attendant d'être traité.

Annexe 5 – Mythes de la sécurité industrielle

Des décennies de recherche sur les systèmes complexes montrent que les contre-mesures reposent sur plusieurs mythes. Dans “Certains mythes sur la sécurité industrielle”, par Denis Besnard et Erik Hollnagel, ils sont résumés comme suit :

Mythe 1 : « L'erreur humaine est la principale cause d'accidents et d'incidents. »

Mythe 2 : « Les systèmes seront sûrs si les gens respectent les procédures qui leur ont été données. »

Mythe 3 : « La sécurité peut être améliorée par des barrières et des protections ; plus de couches de protection entraîne une sécurité accrue. »

Mythe 4 : « L'analyse des accidents peut identifier la cause profonde (la ‘vérité’) de la survenue de l'accident. »

Mythe 5 : « L'enquête sur les accidents est l'identification logique et rationnelle des causes basées sur des faits. »

Mythe 6 : « La sécurité a toujours la priorité absolue et ne sera jamais compromise. »

Les différences entre ce qui est mythe et ce qui est vrai sont montrées ci-dessous :

Myth	Reality
Human error is seen as the cause of failure.	Human error is seen as the effect of systemic vulnerabilities deeper inside the organization.
Saying what people should have done is a satisfying way to describe failure.	Saying what people should have done doesn't explain why it made sense for them to do what they did.
Telling people to be more careful will make the problem go away.	Only by constantly seeking out their vulnerabilities can organizations enhance safety.

Annexe 6 – Le cordon Andon de Toyota

Beaucoup se demandent comment un travail peut être complété si le cordon Andon est tiré plus de cinq mille fois par jour. Pour être précis, chaque tirage du cordon Andon ne conduit pas à l'arrêt complet de la chaîne de montage. En réalité, lorsque le cordon Andon est tiré, le leader d'équipe supervisant le centre de travail spécifié dispose de cinquante secondes pour résoudre le problème. Si le problème n'est pas résolu à l'expiration des cinquante secondes, le véhicule partiellement assemblé franchira une ligne tracée physiquement au sol, et la chaîne de montage sera arrêtée.

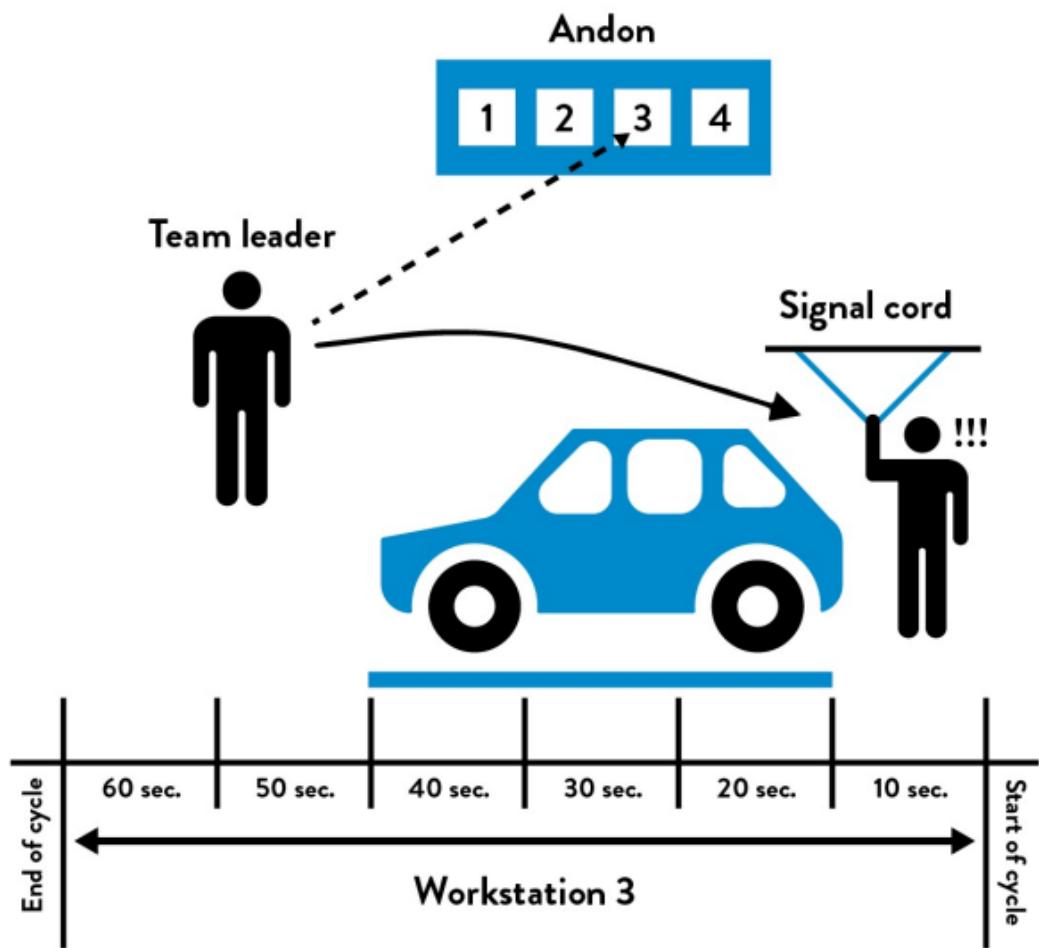


Figure 48: *The Toyota Andon cord*

Annexe 7 - Logiciels commerciaux

Actuellement, pour mettre des logiciels COTS (commerciaux) complexes (par exemple, SAP, IBM WebSphere, Oracle WebLogic) sous contrôle de version, nous devons peut-être éliminer l'utilisation des outils d'installation graphique de type point-and-click fournis par les vendeurs. Pour ce faire, nous devons découvrir ce que fait l'installateur du vendeur, et nous pourrions devoir effectuer une installation sur une image de serveur propre, comparer le système de fichiers, et mettre ces fichiers ajoutés sous contrôle de version. Les fichiers qui ne varient pas selon l'environnement sont placés dans un endroit (« installation de base »), tandis que les fichiers spécifiques à l'environnement sont placés dans leur propre répertoire (« test » ou « production »). Ce faisant, les opérations d'installation des logiciels deviennent simplement une opération de contrôle de version, permettant une meilleure visibilité, répétabilité et rapidité.

Nous pourrions également devoir transformer les paramètres de configuration des applications afin qu'ils soient sous contrôle de version. Par exemple, nous pourrions transformer les configurations d'application stockées dans une base de données en fichiers XML et vice versa.

Annexe 8 - Réunions Post-Mortem

Un exemple d'agenda de la réunion post-mortem est montré ci-dessous :

Une déclaration initiale sera faite par le leader ou le facilitateur de la réunion pour renforcer le fait que cette réunion est une analyse post-mortem sans culpabilité et que nous ne nous concentrerons pas sur les événements passés ni ne spéculerons sur les « aurait dû » ou « aurait pu ».

Les facilitateurs pourraient lire le « Retrospective Prime Directive » du site web Retrospective.com.

De plus, le facilitateur rappellera à tout le monde que toute contre-mesure doit être assignée à quelqu'un, et si l'action corrective ne mérite pas d'être une priorité absolue lorsque la réunion est terminée, alors ce n'est pas une action corrective. (Cela vise à éviter que la réunion ne génère une liste de bonnes idées qui ne sont jamais mises en œuvre.)

Les participants à la réunion s'accorderont sur la chronologie complète de l'incident, y compris quand et qui a détecté le problème, comment il a été découvert (par exemple, surveillance automatique, détection manuelle, client nous a notifiés), quand le service a été restauré de manière satisfaisante, et ainsi de suite. Nous intégrerons également dans la chronologie toutes les communications externes pendant l'incident.

Lorsque nous utilisons le mot « chronologie », il peut évoquer l'image d'une série linéaire d'étapes de la manière dont nous avons compris le problème et l'avons finalement résolu. En réalité, surtout dans les systèmes complexes, il y aura probablement de nombreux événements ayant contribué à l'accident, et de nombreux chemins et actions de dépannage auront été entrepris pour le résoudre. Dans cette activité, nous cherchons à chroniquer tous ces événements et les perspectives des acteurs et à établir des hypothèses concernant les causes et les effets lorsque cela est possible.

L'équipe créera une liste de tous les facteurs ayant contribué à l'incident, qu'ils soient humains ou techniques. Ils pourront ensuite les trier en catégories, telles que « décision de conception », « remédiation », « découverte du problème », etc. L'équipe utilisera des techniques telles que le brainstorming et les « pourquoi infinis » pour approfondir les facteurs contributifs qu'ils jugent particulièrement importants afin de découvrir des niveaux plus profonds de facteurs contributifs. Toutes les perspectives doivent être incluses et respectées - personne ne devrait être autorisé à discuter ou nier la réalité d'un facteur contributif identifié par quelqu'un d'autre. Il est important que le facilitateur post-mortem veuille à ce qu'un temps suffisant soit consacré à cette activité, et que l'équipe n'essaie pas d'engager un comportement convergent tel que tenter d'identifier une ou plusieurs « causes profondes ».

Les participants à la réunion s'accorderont sur la liste des actions correctives qui seront des priorités après la réunion. L'élaboration de cette liste nécessitera du brainstorming et la sélection des meilleures actions potentielles pour soit prévenir le problème, soit permettre une détection ou une récupération plus rapide. D'autres façons d'améliorer les systèmes peuvent également être incluses.

Notre objectif est d'identifier le plus petit nombre d'étapes incrémentielles pour atteindre les résultats souhaités, par opposition aux changements « big bang », qui non seulement prennent plus de temps à mettre en œuvre, mais retardent les améliorations dont nous avons besoin.

Nous générerons également une liste distincte d'idées de moindre priorité et attribuerons un responsable. Si des problèmes similaires se produisent à l'avenir, ces idées pourraient servir de base pour élaborer de futures contre-mesures.

Les participants à la réunion s'accorderont sur les métriques de l'incident et leur impact organisationnel. Par exemple, nous pourrions choisir de mesurer nos incidents selon les métriques suivantes :

- Gravité de l'événement : Quelle a été la gravité de ce problème ? Cela est directement lié à l'impact sur le service et nos clients.
- Temps d'indisponibilité total : Combien de temps les clients ont-ils été incapables d'utiliser le service ?
- Temps de détection : Combien de temps nous a-t-il fallu, à nous ou à nos systèmes, pour savoir qu'il y avait un problème ?
- Temps de résolution : Combien de temps après avoir constaté le problème a-t-il fallu pour restaurer le service ?

Bethany Macri d'Etsy a observé : « La non-culpabilité dans une analyse post-mortem ne signifie pas que personne ne prend de responsabilité. Cela signifie que nous voulons découvrir quelles étaient les circonstances qui ont permis à la personne faisant le changement ou introduisant le problème de le faire. Quel était l'environnement plus large... L'idée est qu'en enlevant la culpabilité, vous enlevez la peur, et en enlevant la peur, vous obtenez l'honnêteté. »

Annexe 9 - L'armée Simienne

Après la panne AWS EAST de 2011, Netflix a eu de nombreuses discussions sur l'ingénierie de leurs systèmes pour gérer automatiquement les pannes. Ces discussions ont évolué en un service appelé « Chaos Monkey ».

Depuis lors, Chaos Monkey a évolué en une famille entière d'outils, connue en interne sous le nom d'« Armée Simienne de Netflix », pour simuler des niveaux de pannes de plus en plus catastrophiques :

- Chaos Gorilla : simule la panne d'une zone de disponibilité entière d'AWS
- Chaos Kong : simule la panne de régions entières d'AWS, telles que l'Amérique du Nord ou l'Europe

D'autres membres de l'Armée Simienne incluent maintenant :

- Latency Monkey : induit des retards ou des temps d'arrêt artificiels dans leur couche de communication client-serveur RESTful pour simuler une dégradation du service et s'assurer que les services dépendants réagissent de manière appropriée
- Conformity Monkey : trouve et ferme les instances AWS qui ne respectent pas les bonnes pratiques (par exemple, lorsque les instances ne font pas partie d'un groupe de mise à l'échelle automatique ou lorsqu'il n'y a pas d'adresse courriel d'ingénieur d'escalade répertoriée dans le catalogue des services)
- Doctor Monkey : accède aux vérifications de santé qui s'exécutent sur chaque instance et trouve les instances non saines, les fermant de manière proactive si les propriétaires ne corrigent pas la cause première à temps
- Janitor Monkey : s'assure que leur environnement cloud est exempt de désordre et de déchets ; recherche les ressources non utilisées et les élimine
- Security Monkey : une extension de Conformity Monkey ; trouve et termine les instances avec des violations de sécurité ou des vulnérabilités, telles que des groupes de sécurité AWS mal configurés

Annexe 10 - Disponibilité transparente

Lenny Rachitsky a écrit sur les avantages de ce qu'il appelait « disponibilité transparente » :

- Vos coûts de support diminuent à mesure que vos utilisateurs peuvent identifier eux-mêmes les problèmes à l'échelle du système sans appeler ou envoyer des courriels à votre département de support. Les utilisateurs n'auront plus à deviner si leurs problèmes sont locaux ou globaux et peuvent rapidement parvenir à la racine du problème avant de se plaindre auprès de vous.
- Vous êtes mieux à même de communiquer avec vos utilisateurs pendant les événements de panne, en tirant parti de la nature de diffusion de l'Internet par rapport à la nature un-à-un des courriels et des appels téléphoniques. Vous passez moins de temps à communiquer la même chose encore et encore et plus de temps à résoudre le problème.
- Vous créez un endroit unique et évident pour que vos utilisateurs se rendent lorsqu'ils rencontrent une panne. Vous économisez le temps de vos utilisateurs actuellement passé à rechercher sur des forums, Twitter ou votre blog.
- La confiance est la pierre angulaire de toute adoption réussie de SaaS. Vos clients parient leur entreprise et leurs moyens de subsistance sur votre service ou votre plateforme. Les clients actuels et potentiels ont besoin de confiance en votre service. Ils doivent savoir qu'ils ne seront pas laissés dans l'ombre, seuls et mal informés, lorsque vous rencontrerez des problèmes. Un aperçu en temps réel des événements inattendus est le meilleur moyen de bâtir cette confiance. Les tenir dans l'ombre et seuls n'est plus une option.
- Ce n'est qu'une question de temps avant que chaque fournisseur de SaaS sérieux n'offre un tableau de bord de santé public. Vos utilisateurs l'exigeront.