LOG680
Introduction à l'approche DevOps

Conception de versions à faible risque

The Dev Ops Handbook

Part III, Chap 13

Objectifs d'apprentissage

- Expliquer en quoi consiste le "Strangler Application Pattern". Expliquer ses avantages et les contextes dans lesquels il peut être appliqué.
- Expliquer les caractéristiques principales des architectures monolithiques et celles basées sur les microservices. Expliquer les avantages et inconvénients de chacune.

Sujets

- Introduction
- Architecture à faible couplage
- Monolithes vs microservices
- Strangler Application Pattern
- Conclusion

- **Introduction**
- Architecture à faible couplage
- Monolithes vs microservices
- Strangler Application Pattern
- Conclusion

Introduction

- Presque tous les exemples bien connus de DevOps ont vécu des expériences proches de la mort en raison de problèmes architecturaux.
  - Exemples présentées sur 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 à leurs besoins organisationnels.
- Principe de l'architecture évolutive –
  - Jez Humble : « l'architecture de tout produit ou organisation performant évoluera nécessairement au cours de son cycle de vie».
  - Randy Shoup (Google, Ingénieur en chef et architecte chez eBay de 2004 à 2011) : **« eBay et Google en sont chacun à leur cinquième réécriture complète de leur architecture**».

    «En rétrospective, certaines technologies [et certains choix architecturaux] paraissent prémonitoires et d'autres à courte vue.

    Chaque décision servait probablement les objectifs organisationnels du moment.

    Si nous avions essayé d'implémenter l'équivalent 1995 des micro-services, nous aurions probablement échoué,

    nous nous serions effondrés sous notre propre poids et aurions probablement emmené toute la compagnie avec nous. »

Introduction

- Le défi consiste à continuer de migrer de l'architecture existante vers l'architecture dont nous avons besoin.
  - Chez eBay, lorsqu'ils avaient besoin d'une nouvelle architecture, ils réalisaient d'abord un projet pilote pour se prouver qu'ils comprenaient suffisamment le problème pour pouvoir s'engager.
  - E.g. lorsque l'équipe de Shoup envisageait de migrer certaines parties du site vers Java à pile intégrale ("full stack") en 2006, elle a cherché la zone qui leur rapporterait le plus, en triant les pages du site en fonction des revenus générés.
  - Ils ont **choisi les secteurs les plus rentables**, s'arrêtant s'il n'y avait pas assez de retour d'affaires pour justifier leurs efforts.
- Exemple classique de conception évolutive, utilisant le "strangler application pattern"
- "**Strangler Application Pattern**"
  - **Utile pour la migration de parties d'une application monolithique ou services fortement couplés vers une application plus faiblement couplée**.
  - Consiste à **placer la fonctionnalité existante derrière un API** et évitez d'y apporter des modifications, plutôt que de la détruire et la redévelopper.
  - **Toutes les nouvelles fonctionnalités sont ensuite implémentées dans les nouveaux services qui utilisent la nouvelle architecture souhaitée**, en appelant l'ancien système si nécessaire.

Introduction

- Dans ce chapitre, nous décrirons les étapes à suivre pour
  - inverser la spirale descendante,
  - passer en revue les principaux archétypes architecturaux,
  - examiner les attributs des architectures permettant la productivité, la testabilité, la déployabilité et la sécurité des développeurs,
  - l'évaluation de stratégies nous permettant de migrer de l'architecture actuelle jusqu'à celle qui nous permet de mieux atteindre nos objectifs organisationnels.

- Introduction
- **Architecture à faible couplage**
- Monolithes vs microservices
- Strangler Application Pattern
- Conclusion

Architectures trop étroitement couplées

- Quels sont les caractéristiques des architectures trop étroitement couplées?
- Quels sont les conséquences des architectures trop étroitement couplées?
- Que peut-on faire pour réduire (éliminer) le couplage trop étroit?

Architectures trop étroitement couplées

- **Conséquences d'architectures trop étroitement couplées ("tight")** :
  - **Chaque fois** que nous essayons d'**intégrer ("commit") du code dans le tronc** ou de le **publier en production**, nous **risquons de créer des échecs globaux**
    - E.g. nous brisons les tests et les fonctionnalités de tout le monde, ou le site entier tombe en panne.
  - Pour éviter cela, **chaque petit changement nécessite des quantités énormes de communication et de coordination** (sur des jours ou des semaines), ainsi que les **approbations** de tout groupe susceptible d'être affecté.
    - Résultat: «Mes développeurs ne passent que 15% de leur temps à coder, le reste de leur temps est consacré aux réunions.»
  - Les **déploiements deviennent également problématiques**: le nombre de modifications groupées pour chaque déploiement augmente, ce qui complique davantage l'intégration et les tests, et accroît la probabilité déjà élevée de problème.
- Tous ces éléments contribuent à un **système de travail dans lequel de petits changements ont des conséquences souvent inconnues et catastrophiques**.
  - Cela contribue également souvent à la peur d'intégrer et de déployer notre code et à la spirale descendante auto-renforçant de déployer moins fréquemment.

Architecture à faible couplage

- Une **architecture à couplage étroit** peut entraver la productivité de tout un chacun et la capacité d'apporter des modifications en toute sécurité.
- Une **architecture à faible couplage** avec des interfaces bien définies assurant la liaison des modules entre eux favorise la productivité et la sécurité.
  - Permet aux petites équipes (2PT) de travailler sur des unités de développement plus petites et plus simples que chaque équipe peut déployer de manière indépendante, rapide et en toute sécurité.
  - Et comme chaque service dispose également d'une API bien définie, elle facilite les tests des services et la création de contrats et de Service Level Agreement (SLA) entre les équipes.
- Randy Shoup

  «Les organisations avec ces types d'architectures, telles que Google et Amazon, montrent comment cela peut avoir un impact sur les structures organisationnelles, en créant [une] flexibilité et une évolutivité.

  Ce sont deux organisations avec des dizaines de milliers de développeurs, où les petites équipes peuvent toujours être incroyablement productives. "

- Introduction
- Architecture à faible couplage
- **Monolithes vs microservices**
- Strangler Application Pattern
- Conclusion

Microservices

- Quels sont les caractéristiques d'une architecture basées sur les microservices?
- Quels sont les avantages et inconvénients d'une architecture basées sur les microservices?

Monolithes vs microservices

- À un moment donné de leur histoire, **la plupart des organisations DevOps ont été entravées par des architectures monolithiques étroitement couplées**.
- Les architectures monolithiques ne sont pas intrinsèquement mauvaises.
  - Constituent souvent le meilleur choix au tout début du cycle de vie d'un produit
- 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 à un ensemble d'exigences et de contraintes, telles que délai de mise sur le marché, facilité de développement des fonctionnalités, mise à l'échelle, etc.

  La fonctionnalité de tout produit ou service évoluera presque assurément au fil du temps - il n'est pas surprenant que nos besoins architecturaux vont également changer.

  Ce qui fonctionne à l'échelle 1x fonctionne rarement à l'échelle 10x ou 100x. »

Table 3: *Architectural archetypes*

	Pros	Cons
Monolithic v1 (All functionality in one application)	• Simple at first • Low inter-process latencies • Single codebase, one deployment unit • Resource-efficient at small scales	• 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")	• Simple at first • Join queries are easy • Single schema, deployment • Resource-efficient at small scales	• 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)	• Each unit is simple • Independent scaling and performance • Independent testing and deployment • Can optimally tune performance (caching, replication, etc.)	• Many cooperating units • Many small repos • Requires more sophisticated tooling and dependency management • Network latencies

(Source: Shoup, "From the Monolith to Micro-services.")

Étude de cas : Amazon (2002)

- Architecture évolutive d'Amazon (2002)
  - L'une des transformations d'architecture les plus étudiées s'est produite chez Amazon.

- Dans une interview avec Jim Gray (lauréat du prix ACM Turing Award et associé technique de Microsoft), Werner Vogels (directeur technique d'Amazon) explique

  « Amazon.com a débuté en 1996 en tant qu'application monolithique s'exécutant sur un serveur Web et communiquant avec une base de données. Cette application, baptisée Obidos, a évolué pour contenir toute la logique commerciale, toute la logique d'affichage et toutes les fonctionnalités qui ont finalement rendu Amazon célèbre: similitudes, recommandations, Listmania, critiques/revues, etc. »

- Résultat: «beaucoup de choses que vous désirez avoir dans un bon environnement logiciel ne pouvaient plus être faites; de nombreux composants logiciels complexes étaient combinés en un seul système. Obidos ne pouvait plus évoluer. »

Étude de cas : Amazon (2002)

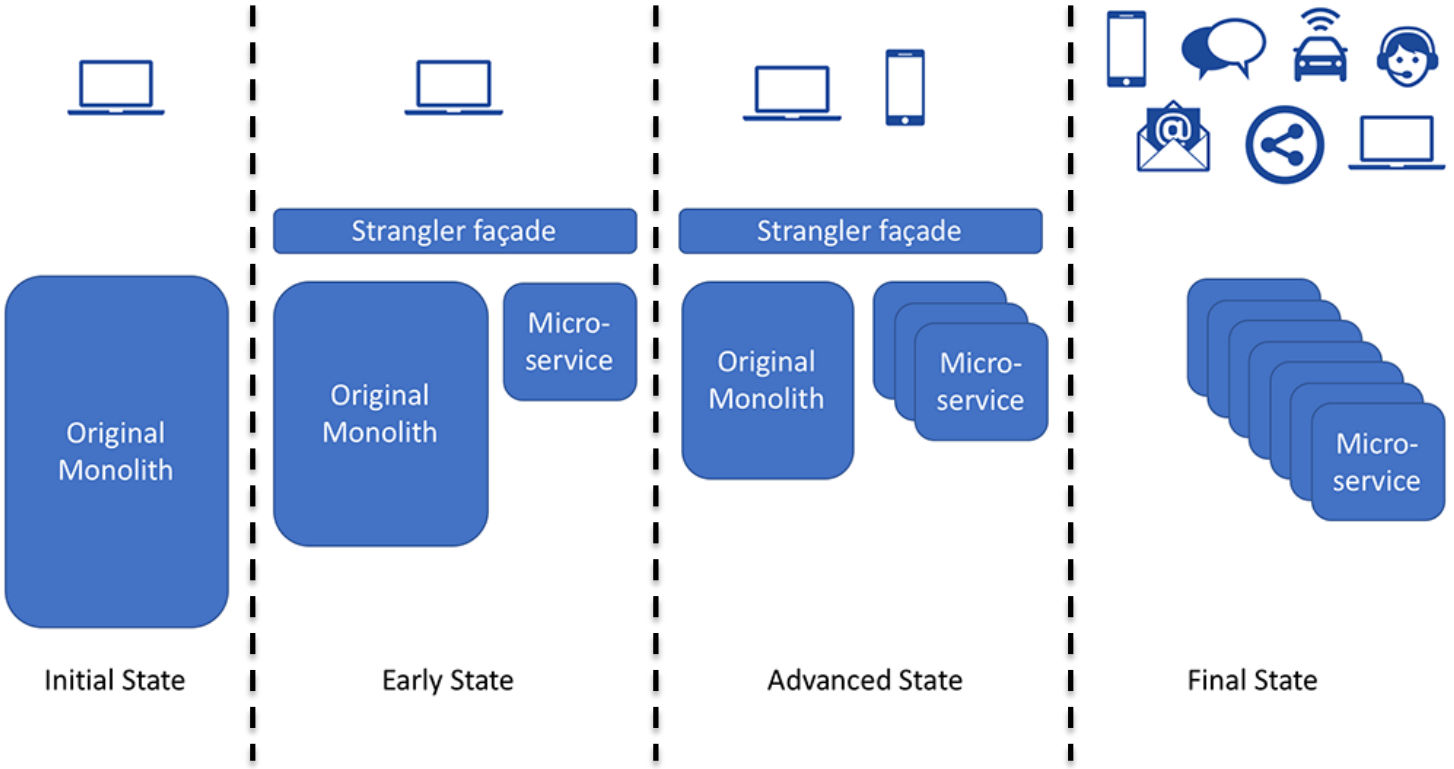
- «Nous avons traversé une période d'introspection sérieuse et avons conclu qu'**une architecture orientée services nous donnerait le niveau d'isolement qui nous permettrait de construire de nombreux composants logiciels rapidement et indépendamment**. » -- Werner Vogel
- Les **leçons tirées de l'expérience** chez Amazon :
  - **Leçon 1**: Lorsqu'elle est **appliquée de manière rigoureuse, une orientation de service stricte est une excellente technique pour réaliser l'isolation**;
    - Permet d'atteindre un niveau d'indépendance et de contrôle inégalé.
  - **Leçon 2**: **Interdire aux clients l'accès direct aux bases de données permet d'améliorer la scalabilité et la fiabilité de l'état de vos services** sans impliquer vos clients.
  - **Leçon 3**: **Les processus de développement et d'exploitation tirent un avantage considérable du passage à l'orientation service**.
    - **Facteur clé** dans la création d'équipes capables d'innover rapidement avec une forte orientation client.
    - **Chaque service est associé à une équipe**. Cette équipe est entièrement responsable du service.
- Résultat :
  - En 2011, Amazon effectuait environ 15 000 déploiements par jour.
  - En 2015, ils effectuaient près de 136 000 déploiements par jour.

- Introduction
- Architecture à faible couplage
- Monolithes vs microservices
- **Strangler Application Pattern**
- Conclusion

"Strangler Application Pattern"

- "**Strangler Application Pattern**"
  - Inventée par Martin Fowler en 2004
  - Inspiré par la visite de vignes d'étrangleur lors d'un voyage en Australie
  - Référence: StranglerFigApplication – Martin Fowler
- Utilisez le "Strangler Application Pattern" pour faire évoluer en toute sécurité notre architecture d'entreprise.
  - Après avoir décidé que notre architecture actuelle est trop étroitement couplée, nous pouvons commencer à découpler en toute sécurité des parties des fonctionnalités de notre architecture existante.
  - Ce faisant, nous permettons aux équipes prenant en charge la fonctionnalité découplée de développer, tester et déployer de manière indépendante leur code en production avec autonomie et sécurité, et de réduire l'entropie architecturale.

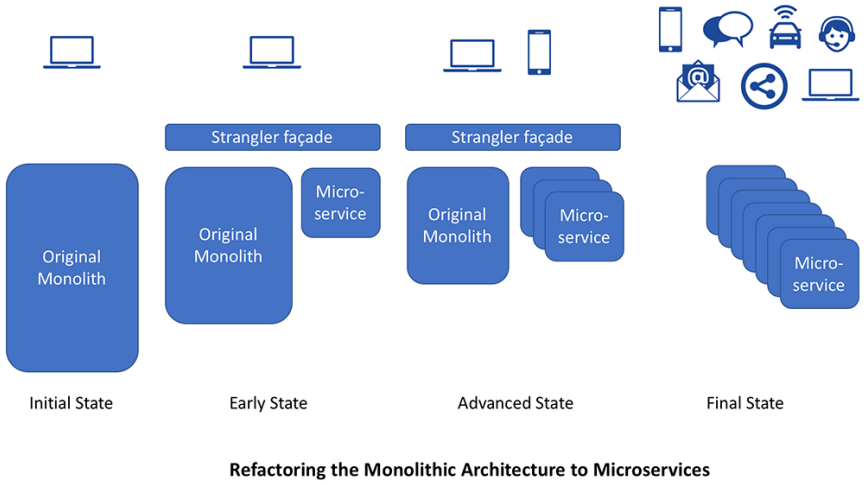
"Strangler Application Pattern"



Refactoring the Monolithic Architecture to Microservices

"Strangler Application Pattern"

- "Strangler Application Pattern"
  1. Placer la fonctionnalité existante derrière une API, où elle reste inchangée – Early State
  2. Implémenter toute nouvelle fonctionnalité en utilisant la nouvelle architecture souhaitée, en appelant l'ancien système si nécessaire – Advanced State



Refactoring the Monolithic Architecture to Microservices

- Lorsque nous implémentons le "Strangler Application Pattern", nous cherchons à accéder à tous les services via des API (services versionnés ou services immuables).
- L'utilisation d'API nous permet de modifier un service sans impacter les appelants, ce qui permet au système d'être couplé de manière plus souple.
  - Si nous devons modifier les arguments, nous créons une nouvelle version d'API et faisons migrer les équipes qui dépendent de notre service vers la nouvelle version.
  - Si les services que nous appelons ne possèdent pas d'API clairement définies, nous devons les définir.

"Strangler Application Pattern"

- En dissociant de manière répétée les fonctionnalités de notre système existant à couplage étroit, nous transférons notre travail dans un écosystème sûr et dynamique dans lequel les développeurs peuvent être beaucoup plus productifs.
  - Entraîne une réduction progressive des fonctionnalités héritées de l'application initiale.
  - L'application initiale pourrait même disparaître complètement à mesure que toutes les fonctionnalités nécessaires migreront vers notre nouvelle architecture.
- En utilisant le "Strangler Application Pattern", nous évitons de simplement recréer les fonctionnalités existantes dans une nouvelle architecture ou en utilisant de nouvelles technologies.
- Comme pour toute transformation, nous cherchons à créer des gains rapides et à fournir une valeur incrémentale rapide avant de continuer à itérer.
  - L'analyse initiale nous aide à identifier le travail le plus simple possible qui permettra d'atteindre utilement un résultat commercial en utilisant la nouvelle architecture.

Étude de cas : Blackboard Learn (2011)

Étude de cas : Blackboard Learn

- DOES14 - David Ashman - Blackboard Learn - Keep Your Head in the Clouds
- Blackboard Inc.
  - Un des pionniers dans la fourniture de technologie aux établissements d'enseignement.
  - Chiffre d'affaires annuel: environ 650 millions de dollars en 2011.
- Contexte
  - En 2011, le logiciel Learn, était "packagé" pour être installée et exécutée sur site ("on-premise") chez les clients.
  - Problèmes associés à une base de code J2EE héritée datant de 1997.

- En 2010, Ashman s'est concentré sur la complexité et les délais de plus en plus longs associés à l'ancien système
  - «Nos processus de construction, d'intégration et de test devenaient de plus en plus complexes et sujets aux erreurs.

    Et plus le produit est volumineux, plus nos délais de production sont longs et plus les résultats pour nos clients se détériorent.

    Même obtenir le retour d'information de notre processus d'intégration nécessitait 24 à 36 heures. "

Étude de cas : Blackboard Learn

Les répercussions de cette situation sur la productivité des développeurs ont été rendues visibles par Ashman dans les graphiques générés à partir de leur référentiel de code source remontant à 2005.

- Le graphique supérieur représente le nombre de lignes de code dans le référentiel de code monolithique Blackboard Learn.
- Le graphique du bas représente le nombre de commits de code.
- Le **problème** qui est **devenu évident** pour Ashman était que **le nombre de commits a commencé à diminuer**, ce qui montre objectivement la difficulté croissante d'introduire des modifications de code, alors que le nombre de lignes de code a continué d'augmenter.

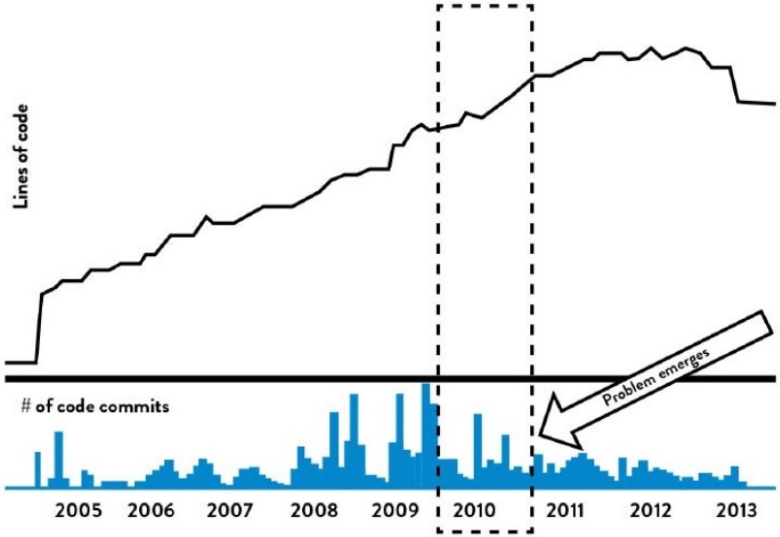


Figure 23: *Blackboard Learn code repository: before Building Blocks (Source: "DOES14 - David Ashman - Blackboard Learn - Keep Your Head in the Clouds," YouTube video, 30:43, posted by DevOps Enterprise Summit 2014, October 28, 2014, https://www.youtube.com/watch?v=SSmixnMps14.)*

Étude de cas : Blackboard Learn

- En 2012, Ashman s'est donc concentré sur la mise en œuvre d'un projet de nouvelle architecture de code utilisant "Strangler Pattern".
  - Basée sur l'utilisation de "building blocks"
  - Permettaient aux développeurs de travailler dans des modules séparés, découplés de la base de code monolithique et accessibles via des API fixes.
  - A permis de travailler avec beaucoup plus d'autonomie, sans avoir à communiquer et à se coordonner en permanence avec les autres équipes de développement.
- Lorsque les building blocs ont été mis à la disposition des développeurs, la taille du référentiel de code source monolithique a commencé à diminuer (mesurée en nombre de lignes de code).
  - Ashman a expliqué que cela était dû au fait que les développeurs déplaçaient leur code dans le référentiel de code source des modules des building blocks.

    « En fait, chaque développeur qui avait le choix travaillait dans la base de code des building blocks, où il pouvait travailler avec plus d'autonomie, de liberté et de sécurité. »

Étude de cas : Blackboard Learn

- Lien entre la croissance exponentielle du nombre de lignes de code et la croissance exponentielle du nombre de commits de code pour les référentiels de codes des building blocks.

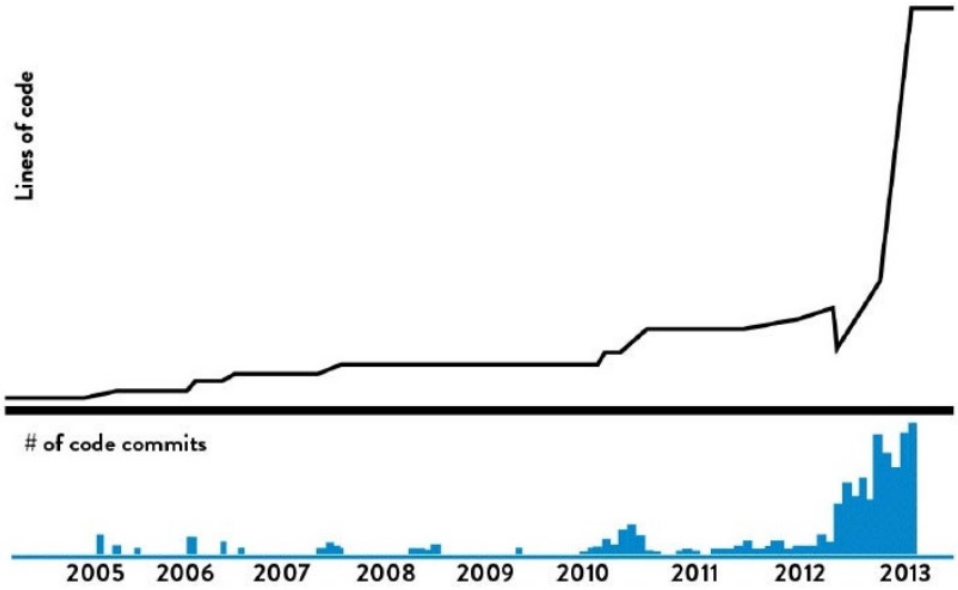


Figure 24: *Blackboard Learn code repository: after Building Blocks (Source: "DOES14 - David Ashman - Blackboard Learn - Keep Your Head in the Clouds." YouTube video, 30:43, posted by DevOps Enterprise Summit 2014, October 28, 2014, https://www.youtube.com/watch?v=SSmixnMpsI4.)*

Étude de cas : Blackboard Learn

- La nouvelle base de code de building blocks a permis aux développeurs d'être plus productifs et de rendre le travail plus sûr, car les erreurs entraînaient de petites défaillances locales plutôt que des catastrophes majeures ayant eu un impact sur le système mondial.
- Ashman a conclu:

  «Le fait de faire travailler les développeurs dans l'architecture des building blocs a permis d'améliorer considérablement la modularité du code, leur permettant de travailler avec plus d'indépendance et de liberté.

  En plus des mises à jour de notre processus de création, ils ont également obtenu plus rapidement des commentaires plus précis sur leur travail, ce qui se traduisait par une meilleure qualité. »

- Introduction
- Architecture à faible couplage
- Monolithes vs microservices
- Strangler Application Pattern
- **Conclusion**

Conclusion

- Dans une large mesure, l'architecture dans laquelle nos services opèrent dicte la manière dont nous testons et déployons notre code.
  - Ceci a été validé dans le *Puppet Labs' 2015 State of DevOps Report*, montrant que **l'architecture est l'un des meilleurs prédicteurs de la productivité des ingénieurs** qui y travaillent et de la manière dont les changements peuvent être effectués rapidement et en toute sécurité.

- Parce que nous sommes souvent coincés avec des architectures optimisées pour un ensemble d'objectifs organisationnels différents ou pour une époque révolue, nous devons pouvoir migrer en toute sécurité d'une architecture à une autre.
  - Les études de cas présentées dans ce chapitre, ainsi que l'étude de cas Amazon présentée précédemment, décrivent des techniques telles que le "Strangler Application Pattern" qui peuvent nous aider à migrer de manière progressive entre architectures, ce qui nous permet de nous adapter aux besoins de l'organisation.

Conclusion – Part III

- Dans les chapitres précédents de la partie III, nous avons mis en place une architecture et des pratiques techniques permettant un flux de travail rapide de Dev à Ops, de sorte que la valeur puisse être livrée rapidement et en toute sécurité aux clients.

- Dans la Partie IV: The Second Way, Les pratiques techniques du retour d'information, nous allons créer l'architecture et les mécanismes permettant le flux rapide et réciproque des retours d'information de droite à gauche, afin de rechercher et de résoudre les problèmes plus rapidement, de générer des retours d'information et d'assurer de meilleurs résultats pour nos clients. Cela permettra à notre organisation d'accroître encore le rythme auquel elle peut s'adapter.