Exécuter les microservices utilisant des conteneurs sur AWS

Novembre 2017

Mentions légales

Ce document est fourni à titre informatif uniquement. Il présente l'offre de produits et les pratiques actuelles d'AWS à la date de publication de ce document, des informations qui sont susceptibles d'être modifiées sans préavis. Il incombe aux clients de procéder à leur propre évaluation indépendante des informations contenues dans ce document et chaque client est responsable de son utilisation des produits ou services AWS, chacun étant fourni « en l'état », sans garantie d'aucune sorte, qu'elle soit explicite ou implicite. Ce document n'offre pas de garantie, représentation, engagement contractuel, condition ou assurance de la part d'AWS, de ses sociétés apparentées, fournisseurs ou concédants de licence. Les responsabilités et obligations d'AWS vis-à-vis de ses clients sont régies par les contrats AWS. Le présent document ne fait partie d'aucun contrat et ne modifie aucun contrat entre AWS et ses clients.

Table des matières

Résumé

Ce livre blanc est destiné aux architectes et aux développeurs qui souhaitent exécuter des applications conteneurisées à l'échelle en production sur Amazon Web Services (AWS). Ce document fournit des conseils pour la gestion du cycle de vie, la sécurité et l'architecture des applications logicielles et des modèles de conception pour les applications basées sur les conteneurs sur AWS.

Nous avons également discuter des meilleures pratiques architecturales pour l'adoption des conteneurs sur AWS, et comment les modèles de conception de logiciels traditionnels évoluent dans le contexte des conteneurs. Nous exploitons les principes de Martin Fowler de microservices et les accordons au facteur de douze modèles d'application et des considérations en situation réelle. Après avoir lu ce livre blanc, vous aurez un point de départ pour créer des microservices à l'aide de bonnes pratiques et de modèles de conception de logiciels.

Introduction

Au fur et à mesure que les applications basées sur les microservices modernes gagnent en popularité, les conteneurs sont une attrayante composante de base pour la création agile, évolutive et efficace des architectures de microservices. Que vous envisagiez un système hérité ou une application vierge pour les conteneurs, il des modèles de conception de logiciels éprouvées bien connus, que vous pouvez appliquer.

Les microservices sont une approche organisationnelle et architecturale du développement logiciel dans laquelle le logiciel se compose de petits services indépendants qui communiquent sur les API bien définis. Ces services sont détenues par de petites équipes autonomes en termes de contenus. Les architectures de microservices rendent plus facile la mise à l'échelle et le développement des applications. Cela permet l'innovation et accélère le temps de mise sur le marché de nouvelles fonctions. Conteneurs d'isolation et de conditionnement pour les logiciels. Envisagez d'utiliser des conteneurs pour obtenir plus de vitesse de déploiement et de densité de ressources.

Comme proposées par Martin Fowler,[1] les caractéristiques d'une architecture de microservices incluent les éléments suivants :

- Structure de composants via des services
- Des capacités concernant les affaires bien organisées
- Des produits pas des projets
- Des points de terminaison et des canaux de vidage astucieux
- Gouvernance décentralisée
- Décentralisation de gestion des données
- Automatisation de l'infrastructure
- Conçu pour la tolérance aux pannes
- Conception évolutionniste

Ces caractéristiques nous indiquent comme une architecture de microservices est supposée se comporter. Pour vous aider à atteindre ces caractéristiques, de nombreuses équipes de développement ont adopté le modèle méthodologique de l'app douze facteurs.[2] Les douze facteurs sont un ensemble de bonnes pratiques

pour créer des applications modernes optimisés pour le cloud computing. Les douze facteurs couvrent quatre domaines clés : le déploiement, le dimensionnement, la portabilité et l'architecture :

1. Codebase - Une base de code tracée par contrôle de révision, de nombreux déploiements
2. Dépendances - Déclarer explicitement et isoler les dépendances.
3. Configuration - Stockez des configurations dans l'environnement
4. Services de soutien - Traiter des services de sauvegarde en tant que ressources attachées
5. Construction, décharge, exécution - Séparer strictement la création et l'exécution des étapes
6. Processus - Exécutez l'application comme un ou plusieurs processus sans état
7. Port de liaison - Exporter les services via le port de liaison
8. Simultanéité - Mise à l'échelle par le modèle de processus
9. Disponibilité - Maximiser la robustesse avec démarrage rapide et arrêt élégant
10. Parité développer/produire - Faire en sorte que le développement, la mise en place de tests et la production soient le plus similaires que possible
11. Registres - Traiter les registres en tant que flux
12. Processus administratif - Exécuter, administrer/gérer les tâches comme des processus uniques

Après avoir lu ce livre blanc, vous saurez comment adapter les caractéristiques de conception de microservices aux applications en douze facteurs, en aval du le modèle de conception à implanter

Structure de composants via des services

Dans une architecture de microservices, les logiciels se composent de petits services indépendants qui communiquent sur des API bien définis. Ces petits composants sont divisés de façon à ce que chacun d'entre eux fasse une seule chose et la fasse bien, tout en collaborant pour fournir une application complète en matière de fonctions. Une analogie peut être faite avec le Walkman, lecteur portable de cassettes audio si populaires dans les années 80 : les piles apportent l'énergie, les bandes audio constituent le moyen, les écouteurs représentent la

sortie, tandis que le lecteur principal de bande fonctionne en appuyant sur des touches. En utilisant ces différents composants ensemble, la musique sonne De même, les microservices doivent être découplés et chacun doit se concentrer sur une fonctionnalité. De plus, une architecture de microservices permet les remplacements ou la mise à niveau. En utilisant l'analogie du Walkman, si les écouteurs sont en mauvais état, nous pouvons les remplacer sans remplacer le lecteur cassettes. Si une commande de gestion dans notre application de maintien de stockage prend du retard ou exécute trop lentement, nous pouvons la remplaçer par un composant plus performant et plus simple. Une telle permutation n'affecterait ni interromprait en aucun cas d'autres microservices du système.

Grâce à modularisation, les microservices offrent la liberté de conception de chaque fonction comme une boîte noire. Autrement dit, les microservices masquent les détails de leur complexité à partir d'autres composants. Toutes les communications entre services se produisent à l'aide des API définis pour empêcher les dépendances implicites et masquées.

Le découplage augmente la flexibilité, en supprimant la contrainte pour une équipe de développement d'avoir à attendre qu'une autre équipe finisse le travail dont elle a besoin. Lorsque les conteneurs sont utilisés, les images de conteneur peuvent être échangées pour d'autres images de conteneur. Cela peut être différentes versions de la même image ou une somme de différentes images - aussi longtemps que les fonctionnalités et les limites sont conservés.

Conteneurisation est un système d'exploitation au niveau de la méthode de virtualisation pour déployer et exécuter des applications distribuées sans lancer toute une machine virtuelle (VM) pour chaque application. Conteneur d'images pour modularité dans les services. Ils sont élaborés en construisant des fonctionnalités sur une image de base. Les développeurs, les équipes d'exploitation et les responsables informatiques doivent se mettre d'accord sur les images de base qui ont le profil d'outils et de sécurité qu'ils souhaitent. Ces images peuvent ensuite être partagées au sein de l'organisation comme composante initiale. Le remplacement ou la mise à niveau de ces images de base est aussi simple que la mise à jour du champ FROM dans un Dockerfile et la reconstruction, généralement par le biais d'une intégration continue/livraison continue (CI/CD).

Voici les facteurs clés de la méthodologie de modèles d'application de douze facteurs qui jouent un rôle dans structure de composants :

- **Dépendances** (déclarer et isoler explicitement les dépendances) - Les dépendances sont autonome dans le conteneur et non partagées avec d'autres services.
- **Disponibilité** (maximiser la robustesse avec rapidité de démarrage et arrêt élégant) - La disponibilité est exploité et satisfaite par les conteneurs qui sont facilement extraits à partir d'un référentiel et supprimés lorsqu'ils ne s'exécutent plus.
- **Simultanéité** (augmentation de la taille des instances via le processus de modèle) - La simultanéité se compose de tâches ou de capsules (faites par des conteneurs travaillant ensemble) pouvant être mis à l'échelle de façon automatique dans une mémoire et de CPU de manière efficace.

Comme chaque fonction de travail est mise en œuvre en tant que son propre service, le nombre de services conteneurisés augmente. Chaque service doit avoir sa propre intégration et son propre canal de déploiement. Ceci augmente l'agilité. Étant donné que les services conteneurisées sont soumis à des déploiements fréquentes, vous avez besoin d'introduire un niveau de coordination capable de détecter quels conteneurs sont en cours d'exécution sur quels hôtes. Finalement, vous souhaiterez avoir un système qui fournit l'état des conteneurs, les ressources disponibles dans un cluster, etc.

Les systèmes d'orchestration et de planification de conteneurs vous permettent de définir des applications, par l'assemblage d'un ensemble de conteneurs qui fonctionnent ensemble. Vous pouvez penser à la définition en tant que projet pour vos applications. Vous pouvez spécifier différents paramètres, tels que les conteneurs à utiliser et les référentiels auxquels ils appartiennent, les ports qui doivent être ouverts sur l'instance de conteneur de l'application et les volumes de données qui doivent être montées.

Les systèmes de gestion de conteneurs vous permettent d'exécuter et de maintenir un nombre d'instances spécifié d'un ensemble de conteneurs - conteneurs qui sont instanciées ensemble et collaborent à l'aide des liens ou des volumes. (Amazon ECS se réfère à eux comme tâches (*Tasks)*, Kubernetes les nomme capsules (*Pods*). Les planificateurs maintiennent le nombre souhaité de conteneurs définis pour le service. En outre, le service d'infrastructure peut être exécutée derrière un programme d'équilibrage de charge afin de répartir le trafic entre les groupes de conteneurs associés au service.

Des capacités concernant les affaires bien organisées

Il est très important que les équipes de développement se mettent d'accord pour définir exactement ce qui constitue un microservice. Quels sont ses limites ? Un microservice est-il une application ? Un microservice est-il une bibliothèque partagée ?

Avant les microservices, l'architecture du système serait organisée autour de capacités technologiques telles que l'interface utilisateur, des bases de données et une logique axée sur le serveur. Dans une approche basée sur les microservices, en tant que bonne pratique, chaque équipe de développement s'approprie le cycle de vie de son service tout au long du processus jusqu'au client. Par exemple, une équipe de recommandations peut posséder le développement, le déploiement, le support, de production et collecter des commentaires des clients.

Dans une organisation axées sur les microservices, les petites équipes agissent de manière autonome pour développer, déployer et gérer le code en production. Cela permet aux équipes de travailler à leur propre rythme pour fournir des fonctionnalités. La responsabilité et la promotion d'une culture de la propriété, permet aux équipes de mieux s'adapter aux objectifs de leur organisation et d'être plus productives.

Les microservices sont autant une attitude d'organisation qu'une approche technologique. Ce principe est connu sous le nom de Conway Law :

> « Les organisations qui conçoivent des systèmes... sont contraintes de produire des conceptions qui soient des copies des structures de communication de ces organisations. » M. Conway[3]

Lorsque l'architecture et les capacités sont organisées autour des fonctions professionnelles atomiques, les dépendances entre les composants sont largement couplées. Tant qu'il y a un contrat de communication entre les services et les équipes, chaque équipe peut fonctionner à sa propre vitesse. Avec cette approche, la pile peut être polyglotte, ce qui signifie que les développeurs sont libres d'utiliser les langages de programmation optimaux pour leur composant. Par exemple, l'interface utilisateur peut être écrite en JavaScript ou HTML5, le serveur principal dans Java et le traitement de données peut être effectuée dans Python.

Cela signifie que les fonctions professionnelles peuvent donner lieu à des décisions de développement. S'organiser autour de capacités signifie que chaque équipe d'API se charge complètement de la fonction, des données et de la performance.

Voici les facteurs clés de la méthodologie de modèles d'application de douze facteurs qui jouent un rôle dans l'organisation autour de capacités :

- **Codebase** (Une base de code tracée par contrôle de révision, de nombreux déploiements) - Chaque microservice possède sa propre base de code dans un référentiel distinct et tout au long du cycle de vie de la modification de code.
- **Construction, décharge, exécution** (strictement séparer la création et l'exécution des étapes) - Chaque microservice a son propre canal de déploiement et sa fréquence de déploiement. Cela permet aux équipes de développement en charge des microservices de varier les vitesses afin de répondre aux besoins des clients.
- **Processus** (exécuter l'application comme un ou plusieurs processus statiques) - Chaque microservice ne fait qu'une seule chose et la fait vraiment bien. Les microservices sont conçus pour résoudre le problème de manière optimale.
- **Processus administratifs** (exécuter, administrer/gérer les tâches comme des processus uniques) : chaque microservice a ses propres tâches administratives ou de gestion afin que sa fonction soit définie.

Pour obtenir une architecture de microservices organisée autour de capacités professionnelles, utilisez des modèles de conception courants:

- **Commandement** - Ce modèle permet d'encapsuler une demande en tant qu'objet, ce qui vous permet de paramétrer des clients avec différentes demandes, file d'attente ou registrer les demandes ou les mettre en attente et prendre en charge des opérations infaisables.
- **Adapter** - Ce modèle permet de respecter les impédances d'un ancien composant à un nouveau système.
- **Exemplaire unique** - Ce modèle est conçu pour une application qui nécessite uniquement une seule instance d'un objet.

- **Chaîne de responsabilité** - Ce modèle permet d'éviter l'association de l'expéditeur d'une requête à son destinataire en donnant à plus d'un objet la possibilité de gérer la demande.
- **Composé** - Ce modèle permet à une application de manipuler une collection hiérarchique d'objets « primitif s» et « composites ». Un service peut être composée d'autres fonctions plus petites.

Des produits pas des projets

Les entreprises qui possèdent des applications matures et ont adopté avec succès un logiciel, souhaitant maintenir et étendre leur base d'utilisateurs seront probablement plus efficaces si elles se centrent sur l'expérience de leurs clients et utilisateurs finaux.

Pour rester saine, simplifier les opérations et améliorer l'efficacité, votre organisation d'ingénierie doit traiter des composants logiciels des produits qui peuvent être continuellement améliorés et qui sont en constante évolution. Ceci est à l'opposé de la stratégie qui consiste à traiter le logiciel comme un projet, qui est effectuée par une équipe d'ingénieurs et ensuite remise en mains à une équipe d'exploitation qui est responsable de son exécution. Lorsque l'architecture logicielle est divisée en plusieurs petites microservices, il devient possible pour chaque microservice de devenir un produit individuel. Pour les microservices internes, l'utilisateur final du produit est un autre équipe ou service. Pour un microservice externe, l'utilisateur final est le client.

L'avantage de base de traiter le logiciel en tant que produit est l'amélioration de l'expérience de l'utilisateur final. Le fait que votre organisation traite ses logiciels comme une amélioration constante de produit plutôt qu'un projet ponctuel, génère un code mieux conçu pour de futurs travaux. Plutôt que de prendre des raccourcis pouvant entraîner des problèmes à l'avenir, les ingénieurs vont envisager un logiciel de façon à ce qu'ils puissent continuer à le maintenir sur le long terme. Les logiciels conçus de cette façon sont plus faciles à utiliser, maintenir et étendre. Vos clients apprécient un tel logiciel fiable, car ils peuvent compter sur lui.

En outre, lorsque les ingénieurs sont responsables de la création, la livraison et l'exécution des logiciels, ils ont une bien meilleure visibilité sur la manière dont leur logiciel s'exécute dans des scénarios réels, ce qui accélère la boucle de rétroaction. Il est ainsi plus facile d'améliorer le logiciel ou de résoudre les problèmes.

Voici les facteurs clés de la méthodologie de modèles d'application de douze facteurs qui jouent un rôle dans l'adoption d'un état d'esprit de produits pour la livraison de logiciels:

- **Construction, décharge, exécution** - Les ingenieurs adoptent une culture « devops » qui leur permet d'optimiser les trois étapes.
- **Configuration** - Les ingénieurs peuvent construire une bien meilleure configuration de gestion de logiciel en raison de leur implication avec la façon dont ce logiciel est utilisé par le client.
- **Parité développer/produire** - Le logiciel traité comme un produit, peut être développé de façon itérative en parties plus petites qui prennent moins de temps à terminer et à déployer que les projets de longue durée, ce qui permet au développement et à la production d'être plus proches dans la parité.

L'adoption d'un état d'esprit de produits repose sur la culture et le processus - deux facteurs qui conduisent au changement. L'objectif de l'équipe d'ingénierie de votre organisation doit être de briser les murs entre les ingénieurs qui construisent le code et les ingénieurs qui exécutent le code en production. Les étapes suivantes sont cruciales :

- **Automatisation de la mise en service** - Les opérations doivent être exécutées automatiquement plutôt que manuellement. Ceci augmente la vitesse ainsi que l'intégration d'ingénierie et les opérations.
- **Libre-service** - Les ingénieurs doivent être en mesure de configurer et de dimensionner leurs propres dépendances. Ceci est rendu possible par des environnements conteneurisées qui permettent aux ingénieurs de concevoir leur propre conteneur avec tout ce dont ils ont besoin.
- **Intégration continue** - Les ingénieurs doivent vérifier fréquemment dans le code, afin que des améliorations supplémentaires soient disponibles pour passer en revue et tester le plus rapidement possible.
- **Création et messages délivrés en continue** - Le processus de création du code qui a été enregistré et sa diffusion à la production doivent être automatisés afin que les ingénieurs puissent libérer du code sans intervention manuelle.

Les microservices utilisant des conteneurs aident les organisations d'ingénieurs à implémenter ces bonnes pratiques en créant un format normalisé pour la livraison de logiciels qui permettent que l'automatisation se génère facilement et

soit utilisée dans différents environnements, y compris les locaux, la garantie de qualité, et la production.

Points de terminaison astucieux et canaux simplistes

Au fur et à mesure que votre organisation d'ingénierie passe de la création des architectures monolithiques à la création des architectures de microservices, elle aura besoin de comprendre comment activer les communications entre les microservices. Dans un monolithe, les différents composants sont tous dans le même processus. Dans un environnement de microservices, les composants sont séparés par des limites strictes. À l'échelle, un environnement de microservices aura souvent les différents composants distribués sur un cluster de serveurs afin qu'ils ne soient pas encore forcément coimplantées sur le même serveur.

Autrement dit, il existe deux formes principales de communication entre services :

- **Demande/Réponse -** Un service invoque explicitement un autre service en effectuant une demande soit pour y stocker les données soit pour récupérer des données à partir de celui-ci. Par exemple, lorsqu'un nouvel utilisateur crée un compte, le service utilisateur fait une demande au service de facturation pour transmettre l'adresse de facturation depuis le profil utilisateur afin que le service de facturation puisse la garder.
- **Publier/s'abonner** - Système d'architecture basé sur évènement dans lequel un service invoque implicitement un autre service qui a été surveillé pour un événement. Par exemple, lorsqu'un nouvel utilisateur crée un compte, le service utilisateur publie l'événement d'abonnement de ce nouvel utilisateur et le service e-mail qui le surveillait est déclenché pour envoyer un e-mail à l'utilisateur en lui demandant de vérifier son e-mail.

Un piège d'architecture qui entraîne généralement des problèmes plus tard est de tenter de résoudre les exigences de communication en créant votre propre bus de services d'entreprise complexes pour le routage des messages entre les microservices. Il est préférable d'utiliser un agent de messages tel que Kafka ou Amazon Simple Notification Service (Amazon SNS) et Amazon Simple Queue Service (Amazon SQS). Les architectures de microservices favorisent ces outils, car ils permettent une approche décentralisée dans laquelle les points de terminaison qui produisent et consomment des messages sont intelligents, tandis que le canal entre les points de terminaison est simpliste. En d'autres termes,

concentrer la logique dans les conteneurs et s'abstenir d'exploiter (et aussi de coupler) des bus et des services de messagerie sophistiqués.

L'avantage de base à la création de points de terminaison intelligents et de canaux simplistes est la capacité à décentraliser l'architecture, notamment lorsqu'il s'agit de la façon dont les points de terminaison sont conservés, mis à jour et étendus. Un objectif des microservices est de permettre de travailler en parallèle sur différentes bords de l'architecture qui ne sera pas en conflit entre eux. La génération de canaux simplistes permet à chaque microservice d'encapsuler sa propre logique pour formater ses réponses sortantes ou compléter ses demandes entrantes.

Voici les facteurs clés de la méthodologie de modèles d'application de douze facteurs qui jouent un rôle dans la création de terminaison astucieux et canaux simplistes :

- **Port Liant** - Les services se lient à un port pour surveiller les demandes entrantes et envoyer des demandes au port d'un autre service. Le canal entre les deux n'est qu'un protocole de réseau simpliste comme HTTP.
- **Services de soutien** - Les canaux simplistes permettent en arrière-plan que des microservices soient attachés à un autre microservice de la même manière que vous attachez une base de données.
- **Simultanéité** - Un canal de communication entre microservices correctement conçu, rend possible le travail simultané de plusieurs microservices. Par exemple, plusieurs observateur microservices peuvent répondre et commencer à travailler en parallèle en réponse à un seul événement généré par un autre microservice.

Gouvernance décentralisée

Au fur et à mesure que votre entreprise grandit et établit une plus grande quantité de code guidé par les processus de travail, un défi auquel elle pourrait faire face est la nécessité de mettre à l'échelle l'équipe d'ingénierie et de lui permettre de travailler en parallèle de manière efficace sur une vaste base de code. De plus, votre organisation d'ingénierie voudra résoudre les problèmes en utilisant les meilleurs outils disponibles.

La gouvernance décentralisée est une approche qui fonctionne bien au côté des microservices pour permettre aux organisations d'ingénierie de s'attaquer à ce

défi. Les feux de signalisation sont un bon exemple de gouvernance décentralisée. Les feux tricolores d'une ville peuvent être chronométrés de façon individuelle ou en petits groupes, ou ils peuvent réagir à des capteurs installés dans le sol. Toutefois, pour la ville dans son ensemble, nul besoin d'un centre « spécialiste » du contrôle de la circulation pour faire circuler les voitures. Les optimisations locales implantées séparément travaillent ensemble pour fournir une solution adaptée à la ville. Décentraliser la gouvernance élimine les goulots d'étranglement potentiels qui empêchaient les ingénieurs d'être en mesure de développer le meilleur code de résolution de problèmes professionnels.

Lorsqu'une équipe lance son premier projet vierge il s'agit généralement d'une petite équipe de quelques personnes travaillant ensemble sur une même base de code. Une fois le projet vierge terminé, l'entreprise va rapidement détecter les possibilités d'élargir leur première version. Les commentaires des clients génèrent des idées concernant les nouvelles fonctions à ajouter et les moyens d'étendre la fonctionnalité des fonctions existantes. Au cours de cette phase, les ingénieurs commenceront à agrandir la base de code et votre organisation commencera à diviser l'organisation d'ingénierie en équipes focalisées sur le service.

La gouvernance décentralisée signifie que chaque équipe peut utiliser son expertise pour choisir les meilleurs outils pour résoudre leurs problèmes spécifiques. Forcer toutes les équipes à utiliser la même base de données, ou le même langage d'exécution, n'est pas raisonnable, car les problèmes qu'elles ont à résoudre ne sont pas uniformes. Toutefois, la gouvernance décentralisée n'est pas sans limites. Il est utile d'utiliser des normes d'un bout à l'autre de l'organisation, par exemple un code de génération standard ainsi qu'un processus d'examen, car cela permet à chaque équipe de continuer à fonctionner ensemble.

Voici les facteurs clés de la méthodologie de modèles d'application de douze facteurs qui jouent un rôle dans l'établissement de la gouvernance décentralisée :

- **Dépendances** - La gouvernance décentralisée permet aux équipes de choisir leurs propres dépendances et, par conséquent, l'isolation de dépendance est essentielle pour que cela fonctionne correctement.
- **Construction, décharge, exécution** - La gouvernance décentralisée doit permettre aux équipes ayant différents processus de génération d'utiliser leur propre ensemble d'outils, cependant elle doit permettre la sortie et l'exécution du code pour qu'il soit homogène, même avec les différents outils de construction sous-jacents.

- **Services de soutien** - Si chaque ressource consomméeest traitée comme s'il s'agissait d'un service externe, puis la gouvernance décentralisée permet au microservice de ressources d'être refactorisé ou développé de différentes façons, tant qu'ils respectent un contrat externe pour la communication avec d'autres services.

Dans le passé, on privilégiait une gouvernance centralisée car il était difficile de déployer efficacement une application polyglotte. Les applications polyglottes nécessitent des mécanismes de construction différents pour chaque langue et une infrastructure sous-jacente qui puisse exécuter plusieurs langues et structures. Les architectures polyglottes avaient différentes dépendances, ce qui pouvais parfois engendrer des conflits.

Les conteneurs résolvent ces problèmes en permettant la délivrabilité pour de chaque équipe d'un format commun : une image Docker qui contient son composant. Les contenus du conteneur peuvent être n'importe quel type d'exécution écrite dans n'importe quel langage. Cependant, le processus de construction sera uniforme, car tous les conteneurs sont générés à l'aide du format de Dockerfile commun. En outre, tous les conteneurs peuvent être déployés de la même manière et lancés sur n'importe quelle instance, car ils portent en eux leur propre temps d'exécution et leurs propres dépendances.

Une organisation d'ingénierie qui choisit d'employer une gouvernance décentralisée et d'utiliser les conteneurs pour expédier et déployer cette architecture polyglotte se rendra compte que son équipe d'ingénierie est en mesure de créer le code performant et itérer plus rapidement.

Décentralisation de gestion des données

Les architectures monolithiques utilisent souvent une base de données partagée, qui peut être un magasin de données unique pour l'ensemble de l'application ou de nombreuses applications. Cela entraîne des complexités en changeant les schémas, les mises à jour, les temps d'arrêt, la gérance de la rétrocompatibilité des risques. Une approche de service exige que chaque service possède son propre stockage de données et qu'il ne partage pas ces données directement avec personne d'autre.

Toutes les données destinées à la communication doivent être activés via des services qui englobe les données. Par conséquent, chaque service équipe choisit pour son application le type de stockage de données le plus optimal et le meilleur

schéma. Le choix du type de base de données est de la responsabilité des équipes de service. Il s'agit d'un exemple de décision de décentralisation - travailler sans groupe central de normes, en dehors de quelques conseils minimum sur la connectivité. AWS propose de nombreux services de stockage entièrement gérés, tels qu'un magasin d'objets, un magasin clé-valeur, un stockage de fichiers, un stockage bloc ou une base de donnée traditionnelle. Vous disposez de nombreuses options, y compris Amazon Simple Storage Service (Amazon S3), Amazon DynamoDB, Amazon Relational Database Service (Amazon RDS), et Amazon Elastic Block Store (Amazon EBS).

Décentraliser la gestion des données améliore la conception d'applications en permettant le meilleur stockage des données pour la tâche à utiliser. Cela supprime également la lourde tâche de mise à jour d'une base de données partagée, qui pourraient valoir des week-ends de temps d'arrêt et de fonctionnement, si tout va bien. Etant donné que chaque équipe de service possède ses propres données, sa prise de décision devient plus indépendante. Les équipes peuvent se composer elles-mêmes et suivre leur propre paradigme de développement.

Un autre avantage de la gestion des données décentralisée est la disponibilité et la tolérance aux pannes de la pile. Si un magasin de données particulier n'est pas disponible, l'ensemble complet des piles d'applications ne cesse pas de réagir. Au lieu de cela, l'application entre dans un état dégradé, perdant certaines capacités tout en continuant à servir les demandes. Cela permet à l'application d'être tolérante aux pannes par définition.

Voici les facteurs clés de la méthodologie de modèles d'application de douze facteurs qui jouent un rôle dans l'organisation autour de capacités :

- **Disponibilité** (maximiser la robustesse avec démarrage rapide et arrêt élégant) - Les services doivent être robustes et ne pas dépendre de capacités externes. Ce principe permet d'avantage aux services de fonctionner avec une capacité limitée si un ou plusieurs composants échouent.
- **Services de soutien** (traiter les services de sauvegarde en tant que ressources attachées) - Un service de soutien est un service que l'application consomme sur le réseau tels que les magasins de données, les systèmes de messagerie, etc. En règle générale, les services de soutien sont gérés par les opérations. L'application ne doit faire aucune distinction entre un service local et un service externe.

- **Processus administratif** (exécuter, administrer/gérer les tâches comme des processus uniques) - Les processus nécessaires pour effectuer le travail régulier de l'application, par exemple, l'exécution de migrations de bases de données. Les processus administratifs doivent être exécutés de manière similaire, indépendamment des environnements.

Pour obtenir une architecture de microservices avec des données de gestion découplés, certains modèles de conception courants peuvent être utilisés :

- **Contrôleur** - Permet de diriger la demande vers le stockage de données approprié à l'aide du mécanisme approprié.
- **Mandataire** - Permet de fournir un substitut ou un remplaçant pour un autre objet afin de contrôler l'accès à celui-ci.
- **Visiteur** - Permet de représenter une opération à effectuer sur les éléments d'un objet structure.
- **Interprète** - Permet de mapper un service pour stocker des données sémantique.
- **Observateur** - Permet de définir la dépendance entre les objets de sorte que lorsqu'un objet change d'état, toutes ses charges soient notifiées et mises à jour automatiquement.
- **Décorateur** - Permet d'attacher des responsabilités supplémentaires à un objet de manière dynamique. Les décorateurs offrent une alternative flexible de sous-classement pour étendre les fonctionnalités.
- **Souvenir** - Permet de capturer et externaliser l'état interne d'un objet afin que l'objet puisse être ramené à cet état ultérieurement.

Automatisation de l'infrastructure

Les architectures actuelles, quelles soient monolithiques ou basées sur microservices, bénéficient de grands avantages de l'automatisation au niveau de l'infrastructure. Avec l'introduction de machines virtuelles, les équipes informatiques ont pu facilement répliquer des environnements et créer les modèles de système d'exploitation qu'elles voulaient. Le système d'exploitation hôte devient disponible et inaltérable. Avec la technologie du cloud, l'idée fleuri et l'échelle a été ajoutée à cette combinaison. Il n'est pas nécessaire de prédire le futur lorsque vous pouvez simplement mettre en service à la demande les ressources dont vous avez besoin et payez uniquement pour ce que vous utilisez. Si un environnement n'est plus utile, vous pouvez interrompre les ressources.

Une image mentale utile pour l'infrastructure en tant que code est une image d'un dessin d'architecte pour y vivre. Tout comme un plan avec les murs, les fenêtres et les portes peut être transformé en un immeuble réel, ainsi. vos programmes d'équilibrage de charge, vos bases de données, ou équipements du réseau peuvent être écrits dans le code source, puis exemplifiés.

Les Microservices n'ont pas seulement besoin d'une 'infrastructure en tant que code disponible, ils doivent également être générés, testés et déployés automatiquement. L'intégration continue et la livraison continue sont importantes pour les monolithiques, mais indispensables pour les microservices. Chaque service a besoin de son propre canal, un canal qui permette d'accueillir plusieurs choix technologiques effectués par l'équipe, aussi variés soient-ils.

Une infrastructure automatisée fournit une bonne répétabilité pour rapidement mettre en place la configuration des environnements. Chacun de ces environnements peut se dédier à un seul objectif : au développement, à l'intégration, à la production, au test de la validation utilisateur (UAT) ou à celui des performances. Une infrastructure qui est décrite en tant que code, puis instanciée peut facilement être annulée. Cela réduit considérablement le risque de changement et, à son tour, favorise l'innovation et les expériences.

Voici les facteurs clés de la méthodologie de modèles d'application de douze facteurs qui jouent un rôle dans l'automatisation de l'infrastructure :

- **Codebase** (une base de code tracée par contrôle de révision, de nombreux déploiements) - Etant donné que l'infrastructure peut être décrite en tant que code, traitez tous les codes de la même manière et conservez-les dans le référentiel de service.
- **Configuration** (stockez les configurations dans l'environnement) - L'environnement doit conserver et partager ses propres spécificités.
- **Construction, décharge, exécution** (séparer strictement la création et l'exécution des étapes) - Un environnement pour chaque objectif.
- **Disponibilité** (maximiser la robustesse avec démarrage rapide et arrêt élégant) - Ce facteur dépasse le processus de couche et se transmet à des couches en aval telles que conteneurs, machines virtuelles et serveur cloud privé virtuel.
- **Parité développer/produire** - Faire en sorte que le développement, la mise en place de tests et la production soient le plus similaires possible.

Les applications brillantes utilisent certaines formes d'infrastructure en tant que code. Les ressources telles que les bases de données, les clusters de conteneur et les programmes d'équilibrage de charge peuvent être instanciées à partir de la description.

Pour emballer l'application avec un canal CI/CD, vous devez choisir un référentiel de code, un canal d'intégration, un artefact de construction de solutions, et un mécanisme de déploiement de ces artefacts. Un microservice doit faire une seule chose et bien la faire. Cela signifie que lorsque vous générez une application complète, potentiellement il peut y avoir un grand nombre de services. Chacun d¡entre eux a besoin de son propre canal d'intégration et de déploiement. En maintenant l'automatisation de l'infrastructure à l'esprit, les architectes confrontés à ce défi qu'est la prolifération de services, seront en mesure de trouver des solutions courantes et répliquer les canaux qui ont effectué avec succès un service particulier.

Finalement, l'objectif est de permettre aux développeurs de faire des mises à jour de code et d'envoyer l'application de mise à jour à plusieurs environnements en quelques minutes. Il existe plusieurs façons de faire des déploiements de phases, y compris les méthodes bleu/vert et canari. Avec le déploiement bleu/vert, deux environnements cohabitent et l'un d'entre eux exécute une version plus récente de l'application. Le trafic est envoyé à la version plus ancienne jusqu'à l'apparition d'un commutateur qui achemine l'ensemble du trafic vers le nouvel environnement. Vous pouvez voir un exemple de ce qui se passe dans cette référence architecture :[4]

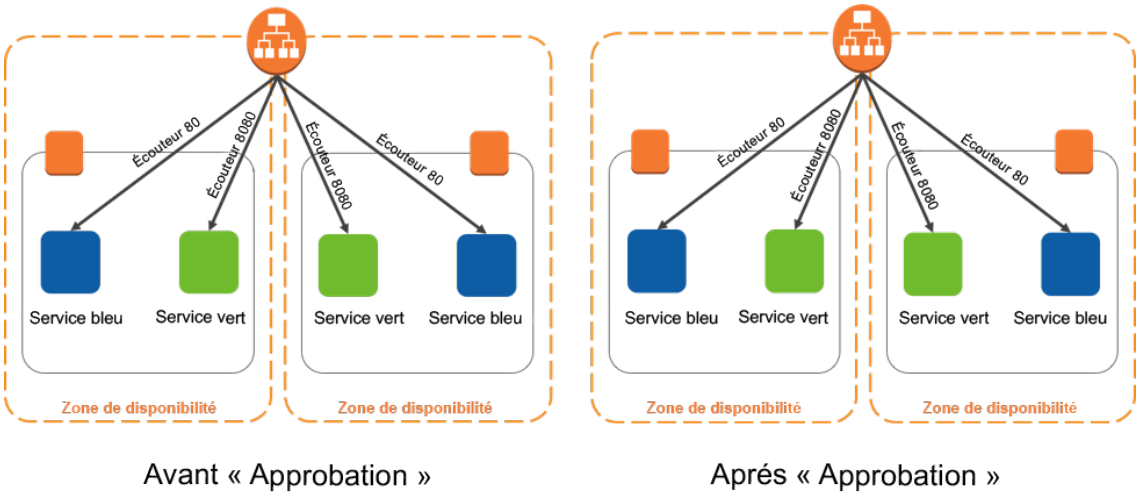

Figure 1 : Déploiement bleu/vert

Dans ce cas, nous utilisons un commutateur de groupes cibles derrière un équilibreur de charge afin de rediriger le trafic de l'ancienne ressource à la nouvelle. Un autre moyen d'y parvenir consiste à utiliser des services affrontés par deux équilibreurs de charge et mettre en marche le commutateur au niveau DNS.

Conçu pour la tolérance aux pannes

> «Tout échoue constamment.» - Werner Vogels

Cet adage n'est pas moins vrai dans le conteneur des économies qu'il ne l'est pour le cloud. Atteindre un haut niveau de disponibilité est une priorité absolue pour les charges de travail, mais reste une entreprise ardue pour les équipes de développement. Les applications modernes exécutées dans des conteneurs ne doivent pas être chargées de la gestion des couches sous-jacentes, à partir d'infrastructures physiques comme des sources d'électricité ou des contrôles environnementaux pour la stabilité du système d'exploitation sous-jacent. Si un ensemble de conteneurs échoue pendant sa mission de fournir un service, ces conteneurs doivent être instanciés de nouveau de façon automatique et sans délai. De la même façon, comme les microservices interagissent les uns avec les autres sur le réseau bien plus qu'ils ne le font au niveau local et synchrone, les connexions doivent être surveillées et gérées. La latence et les délais d'expiration doivent être tenus en compte et correctement traités. De façon plus générale, les microservices doivent appliquer les mêmes erreurs d'essai et le backoff exponentiel comme indiqué avec les applications qui s'exécutent dans un réseau environnement.[5]

Concevoir en pour la tolérance aux pannes signifie également tester la conception et que les services de surveillance gèrent les conditions de détérioration. Il n'est pas nécessaire que tous les services technologiques appliquent ce principe au même niveau que le fait Netflix[6, 7] mais nous vous recommandons de tester ces mécanismes souvent.

La conception pour la tolérance aux pannes génère une infrastructure de réparation automatique qui agit avec la maturité attendue pour les charges de travail récentes. Les appels de prévention urgents garantissent un niveau de base de satisfaction pour l'équipe maître du service. Cela sert également à baisser le niveau de stress qui d'une autre façon pourrait grandir vite à cause de l’usure. La conception pour la tolérance aux pannes fournira une disponibilité accrue pour vos produits. Elle peut protéger une entreprise des interruptions susceptibles d'entamer la confiance des clients.

Voici les facteurs clés de la méthodologie de modèles d'application de douze facteurs qui jouent un rôle dans la conception pour la tolérance aux pannes :

- **Disponibilité** (maximiser la robustesse avec démarrage rapide et arrêt élégant) - La production penche vers les images de conteneur et s'efforce pour obtenir des processus qui peuvent démarrer et s'arrêter en quelques secondes.
- **Registres** (traiter les registres en tant que flux d'événements) - Si une partie d'un système échoue, le dépannage est nécessaire. Assurez-vous qu'il existe du matériel pour l'analyse.
- **Parité développer/produire** - Faire en sorte que le développement, la mise en place de tests et la production soient le plus similaires que possible.

Nous recommandons que les hôtes de conteneur fassent partie d'un groupe de réparation automatique. Idéalement, les systèmes de gestion de conteneur surveillent les différents centres de données et les microservices qui les couvrent, atténuant les événements possibles à niveau physique.

Les conteneurs offrent une abstraction de la gestion du système d'exploitation. Vous pouvez traiter des instances de conteneur en tant que serveurs immuables. Les conteneurs se comportent de manière identique sur un ordinateur de développement ou sur une flotte de machines virtuelles dans le cloud.

Un modèle de conteneur très utile pour le renforcement de résilience d'une application est le disjoncteur. Dans cette approche, une application de conteneur est mandatée par un conteneur en charge de la surveillance des tentatives de connexion de l'application du conteneur. Si les connexions réussissent, le disjoncteur conteneur reste dans l'état Fermé, laissant que la communication passe. Lorsque les connexions commencent à tomber en panne, le disjoncteur logique se déclenche. Si un seuil prédéfini pour le ratio de réussite/échec est dépassé, le conteneur entre dans un état qui empêche d'autres connexions. Ce mécanisme offre un point d'inflexion propre et prévisible, un point de départ pour les situations de défaillance partielles qui peuvent rendre la récupération difficile. L'application de conteneur peut bouger et passer à un service de sauvegarde ou entrer dans un état endommagé.

Les services de gestion de conteneurs modernes permettent aux développeurs de récupérer en temps quasi réel les mises à jour (basée sur les événements) sur l'état des conteneurs. Docker prend en charge plusieurs moteurs de registres (liste de Docker v17.06) :[8]

Driver	Description
Aucun	Aucun registre ne sera disponible pour le conteneur et les registres Docker n'enverrons aucune donnée de sortie.
json de fichier	Les registres sont formatés au format JSON. Le moteur de registre par défaut pour Docker.
syslog	Ecrit les messages de registres dans la fonction syslog. Le démon syslog doit être en cours d'exécution sur la machine hôte.
journald	Ecrit les messages de registres à journald. Le démon journald doit être en cours d'exécution sur la machine hôte.
gelf	Ecrit les messages de registres à un Graylog Extended Log Format (GELF), point de terminaison comme Graylog ou Logstash.
fluentd	Ecrit les messages de registres à fluentd (donnée à transmettre). Le démon fluentd doit être en cours d'exécution sur la machine hôte.
awslogs	Ecrit les messages de registres à Amazon CloudWatch Logs.
splunk	Ecrit les messages de registres à splunk à l'aide de HTTP Event Collector.
etwlogs	Ecrit les messages de registres comme des évènements de Event Tracing for Windows (ETW). Disponible uniquement sur les plateformes Windows.
gcplogs	Ecrit les messages de registres à Google Cloud Platform (GCP) Logging.

Envoyer ces registres à la destination appropriée devient aussi simple que de les spécifier d'une manière clé/valeur. Vous pouvez ensuite définir des mesures et alarmes appropriées dans votre solution de surveillance. Un autre moyen pour collecter les matériaux de télémétrie et de dépannage des conteneurs est de lier un conteneur de registres pour l'application de conteneurs dans un modèle générique appelée *sidecar*. Plus spécifiquement, dans le cas d'un conteneur dont la fonction est de normaliser et standardiser les donnés de sortie, le modèle est connu sous le nom d'*adaptateur*.

Les conteneurs peuvent également être exploités pour garantir que les différents environnements soient aussi semblables que possible. L'infrastructure en tant que code peut être utilisée pour convertir l'infrastructure en modèles et répliquer facilement une empreinte.

Conception évolutionniste

Dans les systèmes modernes de conception architecturale, vous devez assumer que vous n'avez pas toutes les conditions nécessaires par avance. Par conséquent, il devient impossible d'avoir une étape de conception détaillée au début. Les

services doivent évoluer via différentes versions du logiciel. Comme les services sont consommés, il y a des apprentissages à faire de l'utilisation réelle qui aident à faire évoluer leurs fonctionnalités.

Un exemple de ceci peut être une mise à jour silencieuse d'un logiciel sur place d'un appareil. Même si la fonctionnalité est déployée, une stratégie de test alpha/beta peut être utilisée pour comprendre le comportement en temps réel. La fonctionnalité peut être ensuite déployée plus largement ou modifiée et travaillée à partir du retour reçu.

En utilisant des techniques de déploiement, telles qu'une sortie canari,[9] une nouvelle fonctionnalité peut être testée à la mode rapide par rapport à son groupe cible. Cela fournit à l'équipe de développement des commentaires dès le début.

Résultant du principe de conception évolutive, une équipe de service peut créer un ensemble de fonctions viables minimales nécessaires à la pile et aux utilisateurs. L'équipe de développement n'a pas besoin de couvrir les cas limite pour déployer des fonctionnalités. Au lieu de cela, l'équipe peut se concentrer sur les éléments nécessaires et faire évoluer la conception au fur et à mesure qu'elle reçoit les commentaires des clients. Dans une étape postérieure, l'équipe peut décider de refactoriser si elle estime qu'elle a suffisamment de commentaires.

Voici les facteurs clés de la méthodologie de modèles d'application de douze facteurs qui jouent un rôle dans l'automatisation de l'infrastructure :

- **Codebase** (une base de code tracée par contrôle de révision, de nombreux déploiements) - Permet de développer des fonctions plus rapidement car de nouveaux commentaires peuvent être rapidement intégrés.
- **Dépendances** (explicitement déclarer et isoler les dépendances) - Permet des itérations rapides de la conception car les fonctionnalités sont étroitement liées aux externalités.
- **Configuration** (stocker des configurations dans l'environnement) - Tout ce qui est susceptible de varier entre les déploiements (mise en place, production, développeur d'environnements, etc.). La configuration varie considérablement à travers le déploiement, tandis que le code ne varie pas. Avec les configurations stockées en dehors de code, la conception peut évoluer quel que soit l'environnement.
- **Construction, décharge, exécution** (séparer strictement la création et l'exécution des étapes) - Contribuez à déployer de nouvelles fonctions à

l'aide de différentes techniques de déploiement. Chaque version possède un ID spécifique et peut être utilisée pour gagner en efficacité de conception et en commentaires des utilisateurs.

Les modèles suivants de conception de logiciels peuvent être utilisés pour réussir une conception évolutive :

- **Sidecar** étend et améliore le service principal.
- **Ambassador** crée des services d'assistance qui envoient des demandes réseau pour le compte d'un client de service ou application.
- **Chain** fournit un ordre défini de démarrage et d'arrêt de conteneurs.
- **Proxy** fournit un substitut ou un remplaçant pour un autre objet afin de contrôler l'accès à celui-ci.
- **Strategy** définit une famille d'algorithmes, encapsule chacun d'entre eux et les rend interchangeables. La stratégie permet à l'algorithme de varier indépendamment des clients qui l'utilisent.
- **Iterator** fournit un moyen d'accéder aux éléments d'un objet d'agrégation de manière séquentielle sans exposer sa représentation sous-jacente.

Les conteneurs fournissent des outils supplémentaires pour développer la conception de façon plus rapide avec les couches d'images.

Au fur et à mesure que la conception évolue, chaque couche d'image peut être ajoutée, tout en maintenant l'intégrité des couches non affectées. En utilisant Docker, une couche d'image est une modification à une image ou une image intermédiaire. Chaque commande (FROM, RUN, COPY, etc.) dans le Dockerfile entraîne la modification de l'image précédente, créant ainsi une nouvelle couche. Docker va générer uniquement la couche qui a été modifiée et celles qui viennent après. Cette opération est appelée mise en cache *multicouche*. En utilisant la mise en cache multicouche les temps de déploiement peuvent être réduits.

Les stratégies de déploiement telles que Canari font augmenter l'agilité de conception en partant des commentaires des utilisateurs. *Canary release* est une technique utilisée pour réduire les risques inhérents à une nouvelle version du logiciel publié. Dans une version canari, le nouveau logiciel est lentement déployé dans un petit sous-ensemble d'utilisateurs avant qu'il ne soit déployé dans l'ensemble de l'infrastructure et mis à la disposition de tout le monde. Dans le schéma suivant, une version canari peut facilement être mise en œuvre avec les conteneurs à l'aide de AWS primitives. Lorsqu'un conteneur annonce son état via

une vérification de l'état API, canari dirige plus de trafic vers lui. L'état du canari et l'exécution est maintenue - en utilisant d'Amazon DynamoDB, Amazon Route 53, Amazon CloudWatch, Amazon Elastic Container Service (Amazon ECS) et AWS Step Functions.

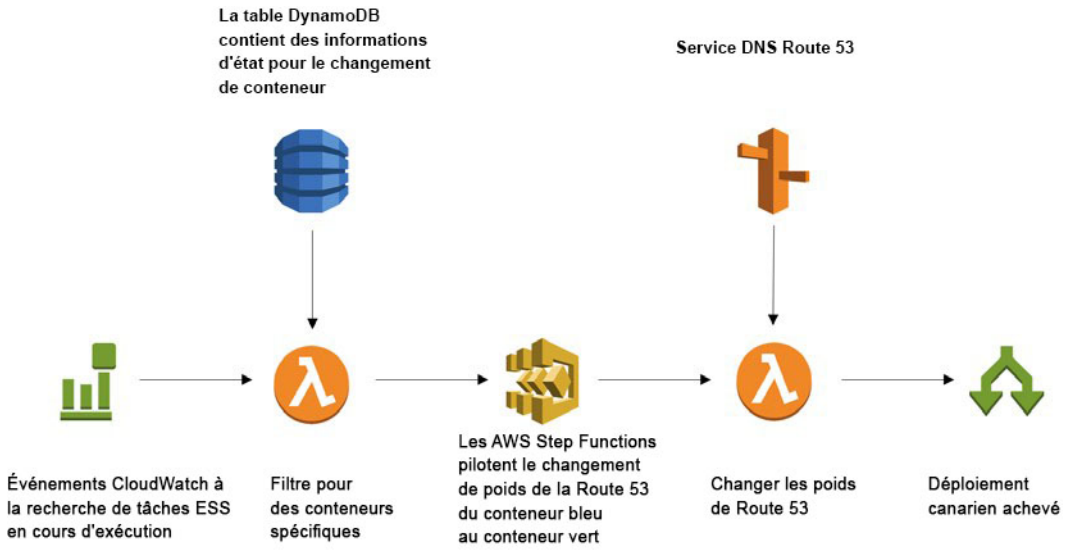

Figure 2 : Déploiement de Canari avec des conteneurs

Enfin, l'utilisation de mécanismes de surveillance permet de s'assurer que les équipes de développement puissent développer la conception tout comme les habitudes d'utilisation changent avec les variables.

Conclusion

Microservices peut être conçu à l'aide de la méthodologie du modèle d'application en douze facteurs. Les modèles de conception de logiciels vous permettent d'y parvenir facilement. Ces modèles de conception logicielle sont bien connus. Lorsqu'ils sont appliqués dans le bon contexte, ils apportent des avantages de conception de microservices. AWS propose une large gamme d'objets primitifs qui peut être utilisés pour activer les microservices conteneurisés.

Participants

Les personnes suivantes ont participé à l'élaboration de ce document :

- Asif Khan, Technical Business Development Manager, Amazon Web Services
- Pierre Steckmeyer, Solutions Architect, Amazon Web Service
- Nathan Peck, Developer Advocate, Amazon Web Services

Remarques

[1] https://martinfowler.com/articles/microservices.html

[2] https://12factor.net/

[3] https://en.wikipedia.org/wiki/Conway's_law

[4] https://github.com/awslabs/ecs-blue-green-deployment

[5] https://docs.aws.amazon.com/general/latest/gr/api-retries.html

[6] https://github.com/netflix/chaosmonkey

[7] https://github.com/Netflix/SimianArmy

[8] https://docs.docker.com/engine/admin/logging/overview/

[9] Le déploiement Canari est une technique visant à réduire le risque d'introduire une nouvelle version du logiciel en production, en déployant lentement la modification à un petit sous-ensemble d'utilisateurs avant de la déployer à l'ensemble de l'infrastructure et de la rendre disponible pour tout le monde. Consultez https://martinfowler.com/bliki/CanaryRelease.html