Automatisation de déploiements sécurisés sans intervention

Clare Liguori

Copyright © 2020, Amazon Web Services, Inc. et/ou ses affiliés. Tous droits réservés

Lorsque j'ai passé mon entretien d'embauche chez Amazon, j'ai veillé à demander à l'un des recruteurs : « À quelle fréquence déployez-vous en production ? » À l'époque, je travaillais sur un produit pour lequel sortaient des mises à jour une ou deux fois par an, même s'il m'arrivait parfois de devoir publier un petit correctif entre deux mises à jour majeures. Pour chaque correctif, je passais des heures à assurer un déploiement correct. Ensuite, je vérifiais frénétiquement les journaux et les métriques pour voir si quelque chose avait été cassé après le déploiement et si je devais revenir à la version antérieure.

J'avais lu qu'Amazon pratiquait le déploiement continu. Donc, lors de mon entretien, je tenais à savoir combien de temps je devrais passer à gérer et à surveiller les déploiements en tant que développeur chez Amazon. Le recruteur m'a répondu que les modifications étaient déployées automatiquement en production plusieurs fois par jour par des pipelines de déploiement continu. Je lui ai alors demandé combien de temps il consacrait chaque jour à surveiller les différents déploiements et à vérifier l'impact éventuel dans les journaux et métriques comme je le faisais moi. Il m'a simplement répondu qu'il ne le faisait généralement pas. Comme les pipelines assuraient ce travail pour son équipe, la plupart des déploiements n'étaient activement surveillés par personne. « Waouw ! », me suis-je exclamée. Lorsque j'ai rejoint Amazon, j'étais curieuse de voir comment fonctionnaient exactement ces déploiements automatisés sans intervention.

Comment le déploiement continu sécurisé permet de libérer du temps de travail pour les développeurs

Depuis, j'ai vu de mes propres yeux comment Amazon met en place des pipelines de déploiement continu pour nous aider à déployer rapidement et en toute sécurité. J'ai pu apprécier comment nos pratiques de sécurité en matière de déploiement continu affranchissent les développeurs du temps passé à travailler sur les déploiements. Dès que je pousse du code de production dans la branche principale du référentiel de code source de mon service, je l'oublie pour passer à ma tâche suivante. Le pipeline de mon équipe prend le relais pour mettre cette modification en production. La publication de ma modification code dans un service de production est entièrement automatisée par le pipeline. Ainsi, la dernière fois que moi ou un autre développeur touchons ou examinons un fragment de code, c'est quand il est fusionné au référentiel de code source.

Mon équipe a mis en place ce pipeline avec des étapes automatisées qui déploient nos modifications en toute sécurité en production pour que nous n'ayons pas à surveiller chaque déploiement. Le pipeline fait passer aux dernières modifications une batterie de tests et de contrôles de sécurité du déploiement. Ces étapes automatisées évitent que des défauts avec un impact sur les clients arrivent en production et en limitent l'incidence s'ils atteignent malgré tout la production. En tant que développeur, je suis certaine que le pipeline déploiera en production, avec précaution et en toute sécurité, les modifications que j'ai apportées, sans que je doive assurer une surveillance active.

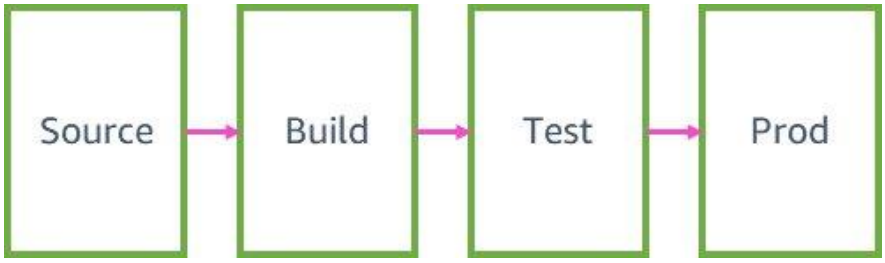
Le cheminement vers une la distribution continue

Amazon n'a pas toujours pratiqué la distribution continue. Les développeurs passaient des heures et des jours à gérer les déploiements de leur code en production. Nous avons adopté la distribution continue à l'échelle de l'entreprise afin d'automatiser et de standardiser le déploiement des logiciels et de réduire le temps nécessaire pour que les changements arrivent en production. Les améliorations

apportées à notre processus de publication se sont accumulées progressivement dans le temps. Nous avons identifié les risques liés au déploiement et trouvé des solutions pour les réduire grâce à une nouvelle automatisation de la sécurité dans les pipelines. Nous continuons à répéter le processus de mise en production en identifiant de nouveaux risques et de nouveaux moyens d'améliorer la sécurité du déploiement. Pour en savoir plus sur notre cheminement vers la distribution continue et sur la façon dont nous continuons à nous améliorer, consultez l'article de la Builders' Library Aller plus vite avec la distribution continue.

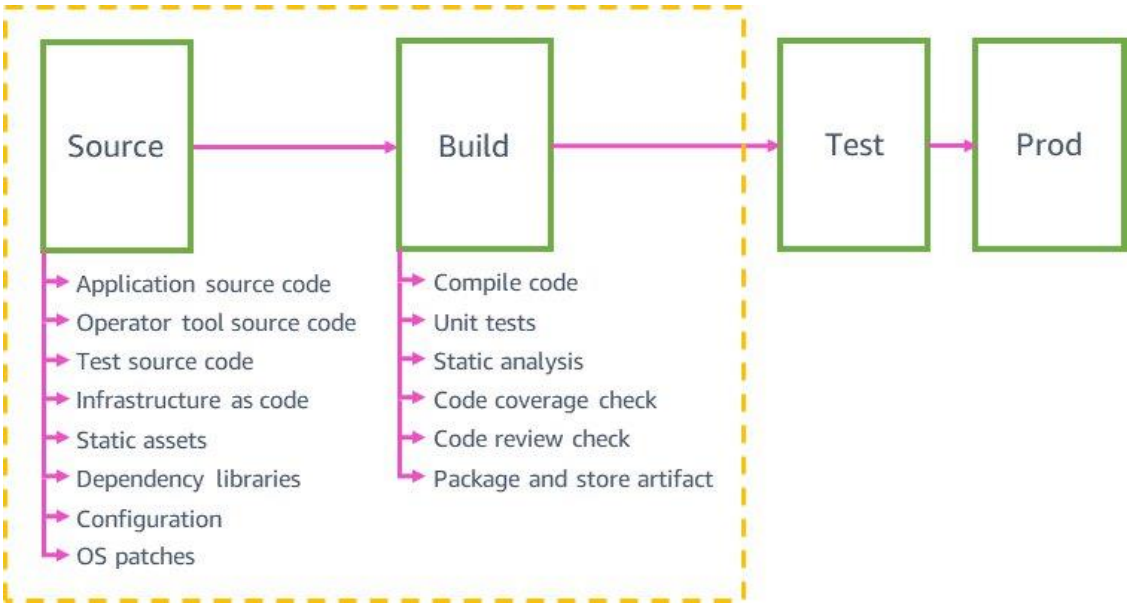
Les quatre phases d'un pipeline

Dans cet article, nous passons en revue les étapes que traverse une modification de code dans un pipeline vers la production chez Amazon. En général, un pipeline de distribution continue comporte quatre phases principales : source, création, tests et production. Nous verrons en détail ce qui se passe dans chacune de ces phases pour un service AWS type, et nous vous donnerons un exemple de la manière dont une équipe classique des services AWS pourrait mettre en place un de ses pipelines.



Source et création

Le diagramme suivant vous donne un aperçu étapes source et création que vous pourriez trouver dans les pipelines types des équipes de services AWS.



Sources des pipelines

Chez Amazon, les pipelines valident automatiquement et déploient en toute sécurité tout type de modification du source en production, et pas uniquement les changements du code d'application. Ils peuvent valider et déployer des modifications de sources, telles que les ressources statiques d'un site web, des outils, des tests, l'infrastructure, la configuration et le système d'exploitation sous-jacent de l'application. Toutes ces modifications font l'objet d'un contrôle des versions dans les différents référentiels de code source. Les dépendances de code source, telles que les bibliothèques, les langages de programmation et des paramètres comme les ID d'AMI, sont automatiquement mises à jour vers la dernière version au moins une fois par semaine.

Ces sources sont déployés dans différents pipelines avec les mêmes mécanismes de sécurité (comme la restauration automatique) que ceux que nous employons pour le déploiement de code d'application. Par exemple, les valeurs de configuration d'un service qui peuvent changer au moment de l'exécution (comme les augmentations des limites de taux d'API et les indicateurs de fonctionnalités) sont déployées automatiquement dans un pipeline de configuration dédié. Les modifications de source sont automatiquement annulées si elles génèrent des problèmes en production pour le service (par exemple, l'échec de l'analyse d'un fichier de configuration).

Un micro-service type peut avoir un pipeline de code d'application, un pipeline d'infrastructure, un pipeline de correctifs de système d'exploitation, un pipeline de configuration/indicateurs de fonctionnalités et un pipeline d'outils opérateur. Le recours à avoir plusieurs pipelines pour le même micro-service nous aide à déployer les changements en production plus rapidement. Les modifications de code d'application qui ne passent pas les tests d'intégration et bloquent le pipeline d'application n'affectent pas les autres pipelines. Ainsi, ils n'empêchent pas les modifications du code d'infrastructure d'arriver en production via le pipeline d'infrastructure. Tous les pipelines d'un même micro-service ont tendance à se ressembler. Par exemple, un pipeline d'indicateurs de fonctionnalités utilise les mêmes techniques de déploiement sécurisé que le pipeline de code d'application, car un

changement incorrect de configuration d'indicateurs de fonctionnalités peut avoir un impact sur la production aussi négatif qu'une mauvaise modification de code d'application.

Vérification du code

Toutes les modifications apportées en production commencent par un examen du code et doivent être approuvées par un membre de l'équipe avant d'être intégrées à la branche *principale* (notre version de « master » ou « trunk »), qui démarre automatiquement le pipeline. Le pipeline respecte l'exigence qui impose que toutes les validations sur la branche principale doivent être vérifiées et approuvées par un membre de l'équipe des services de ce pipeline. Le pipeline empêchera le déploiement de toute validation non vérifiée.

Dans le cas des pipelines entièrement automatisés, la vérification du code est la dernière révision et approbation manuelle d'un ingénieur avant qu'une modification de code soit déployée en production. Il s'agit donc d'une étape critique. Les vérificateurs de code évaluent l'exactitude du code et déterminent également si la modification peut être déployée en toute sécurité en production. Ils évaluent si le code a passé suffisamment de tests (tests unitaires, tests d'intégration et tests Canary), s'il est suffisamment instrumenté pour le suivi du déploiement et s'il peut être restauré en toute sécurité. Certaines équipes utilisent une liste de contrôle personnalisée comme celle de l'exemple suivant, qui est automatiquement ajoutée à chaque vérification de code de l'équipe pour rechercher explicitement les problèmes de sécurité de déploiement.

Exemple de liste de contrôle pour la vérification de un code

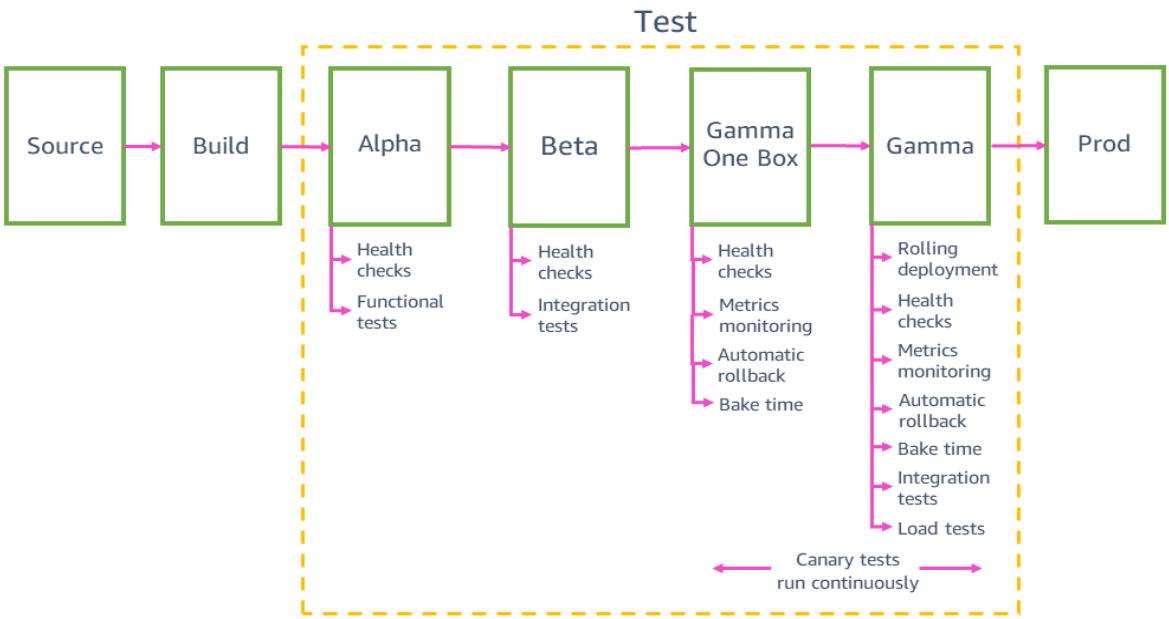
```
## Testing
[ ] Did you write new unit tests for this change?
[ ] Did you write new integration tests for this change?

Include the test commands you ran locally to test this change:
```
mvn test && mvn verify
```


## Monitoring
[ ] Will this change be covered by our existing monitoring?
 (no new canaries/metrics/dashboards/alarms are required)
[ ] Will this change have no (or positive) effect on resources and/or
limits?
 (including CPU, memory, AWS resources, calls to other services)
[ ] Can this change be deployed to Prod without triggering any alarms?

## Rollout
[ ] Can this change be merged immediately into the pipeline upon approval?
[ ] Are all dependent changes already deployed to Prod?
[ ] Can this change be rolled back without any issues after deployment to
Prod?
```

Création et tests unitaires

La phase de création comprend la compilation et le test unitaire du code. Les outils et la logique de création peuvent varier d'un langage à l'autre, voire d'une équipe à l'autre. Par exemple, les équipes peuvent sélectionner les frameworks de test unitaire, les linters et les outils d'analyse statique qui leur conviennent le mieux. En outre, les équipes peuvent choisir la configuration de ces outils, comme la couverture de code minimale acceptable dans leur framework de test unitaire. Les outils et les types de tests qui seront exécutés varieront également selon le type de code déployé par le pipeline. Ainsi, les tests unitaires sont utilisés pour le code d'application, et les linters pour les modèles d'infrastructure comme code (IAC). Toutes les créations s'exécutent sans accès réseau afin de les isoler et d'encourager leur reproductibilité. En général, les tests unitaires simulent tous les appels d'API aux dépendances, comme d'autres services AWS. Les interactions avec les dépendances non simulées « en direct » sont testées plus tard dans le pipeline lors des tests d'intégration. Par rapport aux tests d'intégration, les tests unitaires avec dépendances simulées peuvent tester des incidents en périphérie, tels que des erreurs inattendues renvoyées par des appels d'API, et veiller à un traitement approprié des erreurs dans le code. Lorsque la création est terminée, le code compilé est empaqueté et signé.

Déploiements tests en environnements de pré-production

Avant de déployer en production, le pipeline déploie et valide les modifications dans plusieurs environnements de pré-production, par exemple, alpha, bêta et gamma. Alpha et bêta permettent de vérifier que le dernier code fonctionne comme prévu en effectuant des tests d'API fonctionnels et des tests d'intégration de bout en bout. L'environnement gamma s'assurer que le code est fonctionnel et peut être déployé en toute sécurité en production. Il est aussi proche que possible de l'environnement de production, avec la même configuration de déploiement, un surveillance et des alarmes identiques, et les mêmes tests canary en continu qu'en production. L'environnement gamma est également déployé dans plusieurs régions AWS afin de tenir compte de l'impact potentiel des différences régionales.



Tests d'intégration

Les tests d'intégration nous aident à utiliser automatiquement un service de la manière que les clients dans le cadre du pipeline. Ces tests mettent à l'épreuve toute la pile de bout en bout en appelant de véritables API qui s'exécutent sur une infrastructure réelle à chaque étape de la préproduction et pour tous les scénarios significatifs des clients. L'objectif des tests d'intégration est de détecter tout comportement inattendu ou incorrect du service avant son déploiement en production.

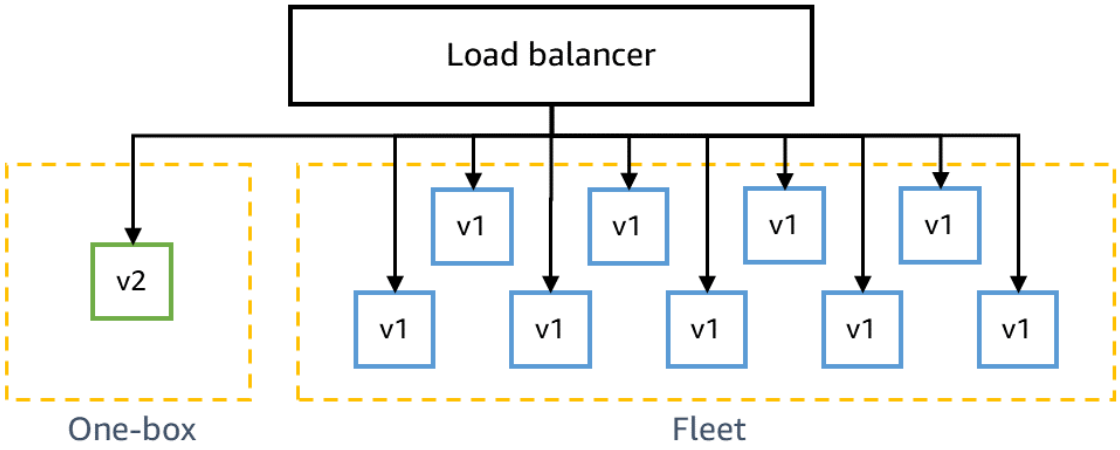
Alors que les tests unitaires sont effectués sur des dépendances simulées, les tests d'intégration le sont sur un système de pré-production qui appelle des dépendances réelles, validant ainsi les hypothèses des simulateurs sur le comportement de ces dépendances. Les tests d'intégration valident le comportement des API individuelles sur différentes entrées. En outre, ils valident les flux de travail complets qui relient plusieurs API, comme la création d'une ressource, sa description jusqu'à ce qu'elle soit prête, puis son utilisation.

Les tests d'intégration exécutent des cas de tests positifs et négatifs, par exemple en fournissant une entrée non valide à une API et en vérifiant qu'une erreur « entrée non valide » est renvoyée comme prévu. Certains pipelines effectuent un fuzzing pour générer de nombreuses entrées d'API possibles et vérifier qu'elles ne génèrent aucune défaillance interne du service. Certains pipelines effectuent également un court test de charge dans une phase de pré-production pour s'assurer que les dernières modifications n'entraînent pas de latence ou de régressions du débit à des niveaux de charge réels.

Rétrocompatibilité et test « one-box »

Avant de passer en production, nous devons nous assurer que le dernier code est rétrocompatible et peut être déployé en toute sécurité parallèlement au code actuel. Par exemple, nous devons déterminer si le dernier code écrit des données dans un format que le code actuel ne peut pas analyser. Le stade « *one-box* » dans l'environnement gamma déploie le dernier code dans la plus petite unité de déploiement, par exemple une seule machine virtuelle ou un seul conteneur, ou un petit pourcentage d'invocations AWS Lambda. Ce déploiement « one-box » laisse le reste de l'environnement gamma déployé avec le code actuel pendant un certain temps, par exemple 30 minutes ou une heure. Il n'est pas nécessaire de diriger le trafic vers l'unité « one-box ». Il peut être ajouté au même équilibreur de charge ou interroger la même file d'attente que le reste de l'environnement gamma. Par exemple, dans un environnement gamma de dix conteneurs derrière un équilibreur de charge, l'unité « one-box » reçoit 10 % du trafic gamma généré par les tests canary en continu. Le déploiement « one-box » surveille les taux de réussite des tests canary et les métriques des services pour détecter tout impact du déploiement ou de la coexistence d'un parc déployé « mixte ».

Le diagramme suivant montre l'état d'un environnement gamma après le déploiement du nouveau code à l'étape « one-box », mais sans qu'il ait été déployé pour le reste du parc de l'environnement :



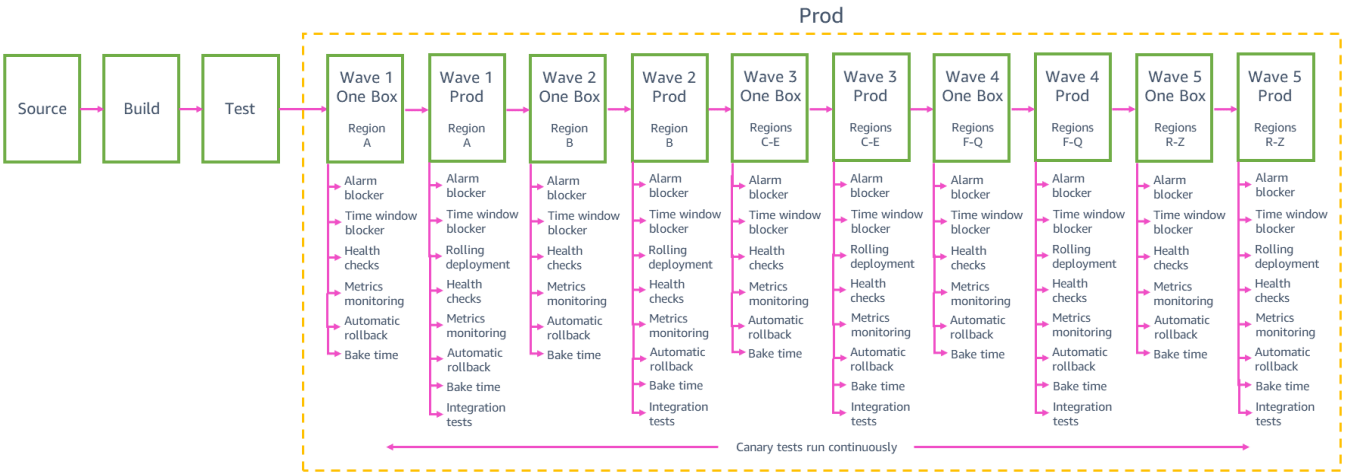
Nous devons également veiller à ce que le dernier code soit rétrocompatible avec nos dépendances, par exemple si une modification doit être apportée aux micro-services selon un ordre spécifique. Les micro-services dans les environnements de pré-production appellent généralement le point de terminaison de production des services appartenant à une autre équipe, comme Amazon Simple Storage Service (S3) ou Amazon DynamoDB, mais ils appellent le point de terminaison de pré-production des autres micro-services de l'équipe des services dans la même étape. Par exemple, le micro-service A de l'environnement gamma d'une équipe appelle le micro-service B de l'environnement gamma de la même équipe, mais appelle le point de terminaison de l'environnement de production pour Amazon S3.

Certains pipelines effectuent également des tests d'intégration dans une étape de rétrocompatibilité distincte que nous appelons *zêta*, qui est un environnement distinct où chaque micro-service n'appelle que les points de terminaison de production, en testant que les changements allant en production sont compatibles avec le code actuellement déployé en production dans plusieurs micro-services. Par exemple, le micro-service A dans zêta appelle le point de terminaison de production du micro-service B et le point de terminaison de production d'Amazon S3.

Pour une description des stratégies d'écriture et de déploiement de modifications rétrocompatibles, consultez l'article de la Builders' Library Exécuter des restaurations sûres pendant les déploiements.

Déploiements en production

Notre objectif N° 1 pour les déploiements en production chez AWS est de prévenir les impacts négatifs sur plusieurs régions en même temps et sur plusieurs zones de disponibilité de la même région. La limitation de la portée de chaque déploiement individuel diminue l'impact potentiel sur les clients de l'échec de déploiements en production et évite un impact sur plusieurs zones de disponibilité ou régions. Pour limiter la portée des déploiements automatiques, nous divisons la phase de production du pipeline en plusieurs étapes et déploiements dans chaque région. Les équipes divisent les déploiements régionaux en déploiements encore plus petits en les déployant dans des zones de disponibilité individuelles ou dans des partitions internes de leur service (appelées *cellules*) de leur pipeline afin de limiter encore plus la portée de l'impact potentiel d'un échec de déploiement en production.



Déploiements échelonnés

Chaque équipe doit trouver un équilibre entre la sécurité des déploiements de petite envergure et la vitesse à laquelle nous pouvons distribuer les modifications aux clients de toutes les régions. Le déploiement de modifications dans 24 régions ou 76 zones de disponibilité par le biais du pipeline, l'une après l'autre, présente le risque le plus faible d'un impact de grande envergure, mais le pipeline pourrait mettre des semaines à diffuser une modification aux clients du monde entier. Nous avons constaté que le regroupement de déploiements en « vagues » de taille croissante, comme nous l'avons vu dans l'exemple de pipeline de production précédent, nous aide à atteindre un bon équilibre entre risque et vitesse de déploiement. L'étape de chaque vague du pipeline orchestre des déploiements dans un groupe de régions, les modifications étant promues d'une vague à l'autre. Les nouvelles modifications peuvent entrer dans la phase de production du pipeline à tout moment. Dès qu'un ensemble de modifications passe de la première à la deuxième étape de la vague 1, l'ensemble suivant de modifications de l'environnement gamma passe à la première étape de la vague 1 et ce, afin d'éviter d'avoir de grands lots de changements en attente de déploiement en production.

Ce sont les deux premières vagues du pipeline qui déterminent le plus la confiance en la modification : la première vague déploie dans une région à faible nombre de requêtes afin de limiter l'impact éventuel du premier déploiement en production de la nouvelle modification. La vague ne déploie que dans une seule zone de disponibilité (ou cellule) à la fois dans cette région afin de déployer prudemment la modification dans toute la région. La deuxième vague déploie ensuite dans une zone de disponibilité (ou cellule) à la fois dans une région au nombre de requêtes élevé, où il est très probable que les clients utiliseront tous les nouveaux chemins de code et où nous obtenons une bonne validation des modifications.

Dès que nous avons une plus grande confiance dans la sécurité de la modification grâce aux déploiements des vagues initiales du pipeline, nous pouvons déployer dans de plus en plus de régions en parallèle dans la même vague. Par exemple, l'exemple précédent de pipeline de production déploie dans trois régions lors de la vague 3, puis dans 12 régions lors de la vague 4, puis dans les autres régions lors de la vague 5. Le nombre exact et le choix des régions dans chacune de ces vagues ainsi que le nombre de vagues dans le pipeline d'une équipe des services dépendent des modèles et de l'échelle d'utilisation d'un service individuel. Les dernières vagues du pipeline nous aident toujours à atteindre

notre objectif visant à prévenir des impacts négatifs sur plusieurs zones de disponibilité d'une même région. Lorsqu'une vague déploie dans plusieurs régions en parallèle, elle adopte le même comportement prudent de déploiement pour chaque région que lors des vagues initiales. Chaque étape de la vague ne déploie que dans une seule zone de disponibilité ou cellule de chaque région de la vague.

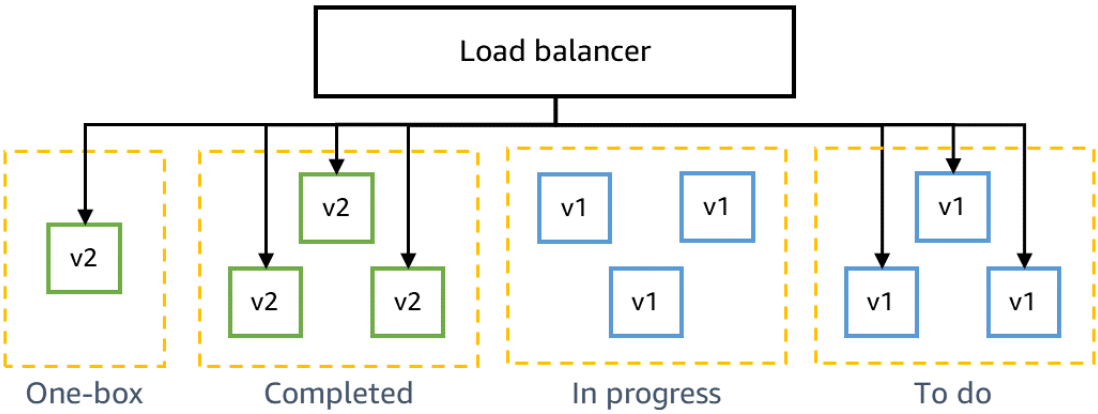
Déploiements « one-box » et continus

Les déploiements à chaque vague de production commencent par une étape « one-box ». Comme pour l'étape « one-box » de la phase gamma, chaque étape « one-box » de production déploie le dernier code dans une seule *unité* (une seule machine virtuelle, un seul conteneur ou un petit pourcentage d'invocations Lambda) dans chacune des régions ou zones de disponibilité de la vague. Le déploiement « one-box » en production réduit l'impact potentiel des modifications sur la vague en limitant à l'origine les requêtes qui sont traitées par le nouveau code de cette vague. En règle générale, l'unité « one-box » sert au maximum 10 % des requêtes globales de la région ou zone de disponibilité. Si la modification a un impact négatif sur l'unité « one-box », le pipeline l'annule automatiquement et ne l'applique pas aux autres étapes de production.

Après l'étape « one-box », la plupart des équipes utilisent des déploiements continus pour déployer dans le parc de production principal de la vague. Un déploiement continu garantit que le service dispose d'une capacité suffisante pour répondre à la charge de production tout au long du déploiement. Il contrôle la vitesse de *mise en service* du nouveau code (c'est-à-dire le moment où il commence à acheminer le trafic de production) afin de limiter l'impact des modifications. Dans un déploiement continu classique dans une région, au maximum 33 % des unités du service dans cette région (conteneurs, invocations Lambda ou logiciels s'exécutant sur des machines virtuelles) sont remplacés par le nouveau code.

Lors d'un déploiement, le système de déploiement choisit d'abord un lot initial d'un maximum de 33 % des unités à remplacer par le nouveau code. Pendant le remplacement, au moins 66 % de la capacité globale est saine et répond aux requêtes. Tous les services sont dimensionnés pour résister à la perte d'une zone de disponibilité dans la région, de sorte que nous savons que le service peut encore assurer la charge de production à cette capacité. Une fois que le système de déploiement a déterminé qu'une unité du lot initial d'unités a passé les vérifications de l'état, une unité du parc restant peut être remplacée par le nouveau code, et ainsi de suite. Dans l'entrefaite, nous maintenons toujours un minimum de 66 % de la capacité pour répondre aux requêtes. Pour limiter encore l'impact des modifications, les pipelines de certaines équipes ne déploient que 5 % de leurs unités à la fois. En revanche, elles procèdent ensuite à des *restaurations rapides*, le système remplaçant 33 % des unités à la fois par le code précédent pour accélérer la restauration.

Le diagramme suivant illustre l'état d'un environnement de production au milieu d'un déploiement continu. Le nouveau code a été déployé au stade « one-box » et sur le premier lot du parc principale de production. Un autre lot a été retiré de l'équilibreur de charge et est arrêté pour être remplacé.



Surveillance des métriques et restauration automatique

Les déploiements automatisés en production en cours dans le pipeline ne sont généralement pas surveillés activement par un développeur pour vérifier les métriques et lancer manuellement une restauration s'il constate des problèmes. Ces déploiements s'effectuent sans intervention. Le système de déploiement surveille activement une alarme pour déterminer s'il doit automatiquement annuler un déploiement. Une restauration rétablit l'environnement à l'image de conteneur, au package de déploiement de fonction AWS Lambda ou au package de déploiement interne qui était déployé précédemment. Nos packages de déploiement interne sont similaires à des images de conteneurs. Ils sont immuables et utilisent un total de contrôle pour vérifier leur intégrité.

Chaque micro-service dans chaque région a généralement une alarme de niveau de gravité élevé qui se déclenche en fonction de seuils de métriques qui ont un impact sur les clients du service (comme les taux d'erreur et la latence élevée) et de métriques d'état du système (comme l'utilisation de l'UC), comme l'illustre l'exemple suivant. Cette alarme de niveau de gravité élevé est utilisée pour appeler l'ingénieur d'astreinte et pour restaurer automatiquement le service si un déploiement est en cours. Souvent, la restauration est déjà en cours avant que l'ingénieur d'astreinte soit contacté et commence son intervention.

Exemple d'alarme de micro-service de niveau de gravité élevé

```
ALARM("FrontEndApiService_High_Fault_Rate") OR
ALARM("FrontEndApiService_High_P50_Latency") OR
ALARM("FrontEndApiService_High_P90_Latency") OR
ALARM("FrontEndApiService_High_P99_Latency") OR
ALARM("FrontEndApiService_High_Cpu_Usage") OR
ALARM("FrontEndApiService_High_Memory_Usage") OR
ALARM("FrontEndApiService_High_Disk_Usage") OR
ALARM("FrontEndApiService_High_Errors_In_Logs") OR
ALARM("FrontEndApiService_High_Failing_Health_Checks")
```

Les changements introduits par un déploiement peuvent avoir un impact sur les micro-services en amont et en aval. Le système de déploiement doit donc surveiller l'alarme de niveau de gravité élevé

du micro-service en cours de déploiement *et* les alarmes de niveau de gravité élevé des autres micro-services de l'équipe afin de déterminer quand lancer une restauration. Les modifications du déploiement peuvent également affecter les métriques des tests canary continus. Le système de déploiement doit donc également surveiller les échecs éventuels de tests canary. Pour restaurer automatiquement toutes ces zones pouvant être impactées, les équipes créent des alarmes globales de niveau de gravité élevé que le système de déploiement doit surveiller. Les alarmes globales de niveau de gravité élevé recensent l'état de toutes les alarmes individuelles de niveau de gravité élevé des micro-services de l'équipe et l'état des alarmes des tests canary pour constituer un seul état global, comme dans l'exemple suivant. Si l'une des alarmes de niveau de gravité élevé des micro-services de l'équipe est activée, tous ses déploiements en cours sur l'ensemble de ses micro-services dans cette région sont automatiquement annulés.

Exemple d'alarme globale de restauration de niveau de gravité élevé

```
ALARM("FrontEndApiService_High_Severity") OR
ALARM("BackendApiService_High_Severity") OR
ALARM("BackendWorkflows_High_Severity") OR
ALARM("Canaries_High_Severity")
```

Une étape « one-box » ne prend en charge qu'un petit pourcentage du trafic global. Ainsi, les problèmes introduits par un déploiement « one-box » ne risquent pas déclencher l'alarme de restauration pour niveau de gravité élevé du service. Afin de détecter et d'annuler les modifications qui génèrent des problèmes à la phase « one-box » avant qu'ils atteignent les autres phases de production, les phases « one-box » restaurent en outre sur la base des mesures limitées à une seule unité. Par exemple, elles reviennent au taux d'erreur des requêtes qui étaient traitées spécifiquement par l'unité « one-box », et qui représente un petit pourcentage du nombre total de requêtes.

Exemple d'alarme de restauration « one-box »

```
ALARM("High_Severity_Aggregate_Rollback_Alarm") OR
ALARM("FrontEndApiService_OneBox_High_Fault_Rate") OR
ALARM("FrontEndApiService_OneBox_High_P50_Latency") OR
ALARM("FrontEndApiService_OneBox_High_P90_Latency") OR
ALARM("FrontEndApiService_OneBox_High_P99_Latency") OR
ALARM("FrontEndApiService_OneBox_High_Cpu_Usage") OR
ALARM("FrontEndApiService_OneBox_High_Memory_Usage") OR
ALARM("FrontEndApiService_OneBox_High_Disk_Usage") OR
ALARM("FrontEndApiService_OneBox_High_Errors_In_Logs") OR
ALARM("FrontEndApiService_OneBox_Failing_Health_Checks")
```

En plus de restaurer en cas d'alarmes définies par l'équipe des services, notre système de déploiement peut également détecter automatiquement des anomalies dans les métriques communes émises par notre framework de service Web interne et lancer une restauration. La plupart de nos micro-services émettent des métriques, tels que le nombre de requêtes, la latence des requêtes et le nombre d'erreurs, dans un format standard. Grâce à ces métriques standard, le système de déploiement peut revenir automatiquement en arrière s'il détecte des anomalies dans les métriques pendant un déploiement. Par exemple, si le nombre de requêtes tombe soudainement à zéro, ou si le temps de latence ou le nombre d'erreurs devient beaucoup plus élevé que la normale.

Temps d’attente

Parfois, l'impact négatif d'un déploiement n'est pas immédiatement apparent. Dans ce cas, on le dit *latent*. Cela signifie qu'il n'apparaît pas immédiatement pendant le déploiement, surtout si le service est alors peu sollicité. Valider le passage à l'étape suivante du pipeline immédiatement après la fin du déploiement peut finalement impacter plusieurs régions au moment où l'impact se produit dans la première région. Avant de valider le passage à l'étape de production suivante, chaque étape de production du pipeline a un *temps d'attente*. Le pipeline continue à surveiller l'alarme de l'équipe pour tout impact latent après la fin d'un déploiement et avant de passer à l'étape suivante.

Pour calculer le temps d'attente d'un déploiement, nous devons mettre en balance le risque de provoquer un impact plus important si nous validons trop rapidement des modifications dans plusieurs régions et la vitesse à laquelle nous distribuons ces modifications aux clients à l'échelle mondiale. Nous avons constaté qu'un bon moyen d'équilibrer ces risques consiste à allonger le temps d’attente des premières vagues du pipeline pendant que nous gagnons en confiance dans la sécurité de la modification, puis à raccourcir le temps d’attente des vagues suivantes. Notre objectif reste de réduire le risque d'impact qui toucherait plusieurs régions. Comme la plupart des déploiements ne sont pas activement surveillés par un membre de l'équipe, les temps d’attente par défaut du pipeline sont prudentes et la modification sera déployée dans toutes les régions en quatre ou cinq jours ouvrés environ. Les services plus volumineux ou très critiques ont des temps d’attente encore plus prudents avant que leurs pipelines ne déploient un changement à l'échelle mondiale.

Un pipeline classique attend au moins une heure après chaque étape « one-box », au moins 12 heures après la première vague régionale et au moins deux à quatre heures après chacune des autres vagues régionales, avec un temps d’attente supplémentaire pour les différentes régions, zones de disponibilité et cellules de chaque vague. Le temps d’attente englobe l'obligation d'attendre un nombre spécifique de points de données dans les métriques de l'équipe (par exemple, « attendre au moins 100 requêtes à l'API de création ») pour s'assurer que le nombre de demandes a été suffisant pour qu’il soit probable que le nouveau code ait été pleinement testé. Pendant tout le temps d’attente, le déploiement est automatiquement annulé si l'alarme globale de l'équipe passe en état d'alerte.

Bien que cela soit extrêmement rare, dans certains cas, une modification urgente (par exemple un correctif de sécurité ou l’atténuation d'un événement de grande envergure qui affecte la disponibilité du service) peut devoir être distribuée aux clients plus rapidement que le temps habituellement nécessaire au pipeline pour tester et déployer les modifications. Dans ces situations, nous pouvons réduire le temps d’attente du pipeline pour accélérer le déploiement, mais nous avons besoin d'un niveau élevé de contrôle sur la modification pour le faire. Nous avons alors besoin de donc faire appel aux ingénieurs principaux de l'entreprise. L'équipe doit analyser la modification de code, ainsi que son urgence et son risque d'impact, avec des développeurs très expérimentés experts en matière de sécurité opérationnelle. La modification passe toujours par les mêmes étapes de pipeline, mais avance plus rapidement à l'étape suivante. Nous gérons le risque d'un déploiement plus rapide en limitant les modifications de code à la volée dans le pipeline afin de ne permettre que les plus minimes nécessaires pour résoudre le problème actuel et en surveillant activement les déploiements.

Alarmes et bloqueurs de créneaux

Le pipeline empêche les déploiements automatiques en production lorsqu'il existe un risque plus élevé d'impact négatif. Le pipeline utilise un ensemble de « bloqueurs » qui évaluent le risque de déploiement. Par exemple, le déploiement automatique d'une nouvelle modification en production lorsqu'un problème est en cours dans l'environnement pourrait aggraver ou prolonger l'impact. Avant de lancer un nouveau déploiement à n'importe quelle étape de production, le pipeline vérifie l'alarme globale de niveau de gravité élevé de l'équipe pour déterminer la présence de problèmes actifs. Si l'alarme est en état d'alerte, le pipeline empêche le changement d'avancer. Les pipelines peuvent également vérifier les alarmes à l'échelle de l'organisation, comme une alarme d'événement à grande échelle qui indique si un impact important a été détecté dans les systèmes d'une autre équipe et empêche de commencer un nouveau déploiement qui pourrait ajouter à l'impact global. Les développeurs peuvent ignorer ces bloqueurs de déploiement lorsqu'une modification doit être déployée en production pour récupérer d'un problème de niveau de gravité élevé.

Le pipeline est également configuré avec un ensemble de créneaux qui définissent quand un déploiement peut commencer. Lorsque nous configurons des créneaux, nous devons équilibrer deux causes de risque de déploiement. D'une part, de très petits créneaux peuvent entraîner l'accumulation de modifications dans le pipeline alors que le créneau est fermé, ce qui augmente la probabilité que l'une de ces modifications ait un impact sur le déploiement suivant lorsque le créneau s'ouvrira. D'autre part, de très larges créneaux qui s'étendent en dehors des heures de travail normales augmentent le risque de prolonger l'impact d'un échec de déploiement. En dehors des heures de travail, il faut plus de temps pour faire intervenir l'ingénieur d'astreinte que pendant la journée, lorsque ce dernier et les autres membres de l'équipe travaillent. Pendant les heures de travail normales, l'équipe peut intervenir plus rapidement après un échec de déploiement si des mesures de restauration manuelles sont nécessaires.

La plupart des déploiements ne sont pas surveillés activement par un membre de l'équipe. Nous optimisons donc le calendrier des déploiements pour réduire le temps nécessaire à l'intervention d'un ingénieur d'astreinte, au cas où une action manuelle serait requise pour la reprise après une restauration automatique. Les ingénieurs d'astreinte mettent généralement plus de temps à intervenir la nuit, les jours fériés et les week-ends, ces périodes sont donc exclues des créneaux. En fonction des habitudes d'utilisation du service, certains problèmes peuvent ne pas apparaître avant plusieurs heures après le déploiement. C'est la raison pour laquelle de nombreuses équipes excluent également les vendredis et les fins d'après-midi de leurs créneaux de déploiement afin de réduire le risque de devoir faire intervenir l'ingénieur d'astreinte la nuit ou le week-end après un déploiement. Nous avons constaté que cet ensemble de créneaux permet un rétablissement rapide même lorsqu'une action manuelle est nécessaire, garantit une intervention moindre avec les ingénieurs d'astreinte en dehors des horaires habituels de travail, et permet de veiller à ce qu'un petit nombre de modifications s'accumulent la fermeture des créneaux.

Les pipelines en tant que code

L'équipe des services AWS type dispose de nombreux pipelines pour déployer ses multiples micro-services et types de sources (code d'application, code d'infrastructure, correctifs du système d'exploitation, etc.). Chaque pipeline comporte de nombreuses étapes de déploiement pour un nombre toujours croissant de régions et de zones de disponibilité. Cela se traduit par un grand nombre de

configurations que l'équipe doit gérer dans le systèmes de pipeline, de déploiement et d'alarme, et par beaucoup d'efforts pour se tenir au courant des dernières bonnes pratiques et des nouvelles régions et zones de disponibilité. Ces dernières années, nous avons adopté la pratique des « pipelines en tant que code » comme moyen de configurer plus facilement et de manière cohérente des pipelines sécurisés et actualisés en modélisant cette configuration en code. Notre outil interne « pipelines en tant que code » s'appuie sur une liste centralisée de régions et de zones de disponibilité pour ajouter facilement de nouvelles régions et zones de disponibilité aux pipelines chez AWS. L'outil permet également aux équipes de modéliser les pipelines en utilisant l'historique, en définissant la configuration commune aux pipelines d'une équipe dans une classe parente (par exemple, quelles régions vont dans chaque vague et quelle temps d'attente devrait être appliqué pour chaque vague) et en définissant toute la configuration des pipelines de micro-services comme une sous-classe qui hérite de toute la configuration commune.

Conclusion

Chez Amazon, nous avons élaboré nos pratiques de déploiement automatisé au fil du temps en nous basant sur ce qui nous aide à équilibrer sécurité et vitesse de déploiement. En même temps, nous cherchons à réduire au maximum le temps que les développeurs doivent passer à se préoccuper des déploiements. L'intégration de la sécurité des déploiements automatisés dans le processus de publication en utilisant des tests de pré-production approfondis, des restaurations automatiques et des déploiements échelonnés en production nous permet de réduire l'impact potentiel sur la production causé par les déploiements. Cela signifie que les développeurs ne doivent pas surveiller activement les déploiements en production.

Grâce à des pipelines entièrement automatisés, les développeurs utilisent les vérifications de code pour contrôler leur code mais aussi pour s'assurer que la modification est prête à passer en production. Une fois la modification fusionnée au référentiel de code source, le développeur peut passer à la tâche suivante et ne plus se préoccuper du déploiement, en faisant confiance au pipeline pour mettre sa modification en production en toute sécurité et avec prudence. Le pipeline automatisé se charge du déploiement en continu en production plusieurs fois par jour, tout en maintenant un équilibre entre sécurité et rapidité. En modélisant en code notre pratique de distribution continue, il est plus facile que jamais pour les équipes des services AWS de mettre en place leurs pipelines pour déployer leurs modifications de code automatiquement et en toute sécurité.