Exécuter des annulations sûres pendant les déploiements

Sandeep Pokkunuri

Copyright © 2019 Amazon Web Services, Inc. and/or its affiliates. Tous droits réservés.

Éviter de passer par des portes à sens unique est chez Amazon l'un des principes directeurs lorsque nous élaborons des solutions. Cela signifie que nous évitons les choix difficiles à inverser ou à étendre. Nous appliquons ce principe à toutes les phases du développement logiciel, de la conception des produits, des fonctionnalités et des API aux systèmes de gestion et aux déploiements. Dans cet article, je vais décrire comment nous appliquons ce principe aux déploiements de logiciels.

Un déploiement fait passer un environnement logiciel d'un état (version) à un autre. Le logiciel peut parfaitement fonctionner dans l'un ou l'autre de ces états. Cependant, le logiciel peut ne pas fonctionner correctement pendant ou après la transition aval (mise à niveau ou récupération en aval) ou la transition amont (rétrogradation ou restauration). Lorsque le logiciel ne fonctionne pas bien, cela entraîne une interruption de service qui le rend peu fiable pour les clients. Dans cet article, je suppose que les deux versions du logiciel fonctionnent correctement. Mon objectif est de m'assurer que la reconduction ou la rétrogradation pendant le déploiement n'entraîne pas d'erreurs.

Avant de lancer une nouvelle version du logiciel, nous la testons dans un environnement de test bêta ou gamma selon de multiples dimensions telles que la fonctionnalité, la simultanéité, la performance, l'échelle et le traitement des pannes en aval. Ces tests nous aident à découvrir les problèmes éventuels dans la nouvelle version et à les résoudre. Cependant, il se peut que cela ne soit pas toujours suffisant pour assurer un déploiement réussi. Nous pouvons rencontrer des circonstances inattendues ou un comportement logiciel non optimal dans les environnements de production. Chez Amazon, nous voulons éviter de nous mettre dans une situation où l'annulation du déploiement pourrait entraîner des erreurs pour nos clients. Pour éviter cette situation, nous nous préparons pleinement à une restauration avant chaque déploiement. Une version du logiciel pouvant être restaurée sans erreur ni interruption des fonctionnalités disponibles dans la version précédente s'appelle une version rétrocompatible. Nous planifions et vérifions que notre logiciel est rétrocompatible à chaque révision.

Avant d'entrer dans les détails sur la façon dont Amazon aborde les mises à jour logicielles, discutons de certaines des différences entre les déploiements de logiciels autonomes et distribués.

Déploiements de logiciels autonomes ou distribués

Pour les logiciels autonomes qui s'exécutent en tant que processus unique sur un périphérique, les déploiements sont atomisés. Deux versions du logiciel ne fonctionnent jamais simultanément. Si le logiciel autonome maintient l'état alors la nouvelle version doit lire (à savoir désérialiser) les données écrites (à savoir sérialisées) par l'ancienne version et vice versa. Si cette condition est remplie, le déploiement peut se faire en toute sécurité pour la récupération en aval et l'annulation.

Dans un système distribué, les déploiements deviennent plus complexes. Ils sont réalisés par le biais de mises à jour régulières, de sorte que la disponibilité n'est pas affectée. La nouvelle version est déployée sur un sous-ensemble d'hôtes à la fois afin que les autres hôtes puissent continuer à traiter les demandes. Généralement, ces hôtes communiquent entre eux par le biais d'un appel de procédure à distance (RPC) ou d'un état persistant partagé (par exemple, métadonnées ou points de contrôle). Une telle communication ou un état partagé peut présenter des défis supplémentaires. L'enregistreur et le lecteur peuvent utiliser des versions différentes du logiciel. Par conséquent, ils pourraient interpréter les données différemment. Il se peut même que le lecteur ne lise pas complètement les données, ce qui peut entraîner une panne.

Problèmes liés aux modifications de protocole

Nous avons constaté que la cause la plus courante de l'impossibilité d'exécuter une annulation était une modification de protocole. Supposons une modification de code qui compresse les données tout en les conservant sur le disque. Une fois que la nouvelle version a écrit certaines données compressées, la restauration n'est plus possible. L'ancienne version ne sait pas qu'elle doit décompresser les données après la lecture sur le disque. Si les données sont stockées dans un blob ou un magasin de documents, leur lecture par les autres serveurs échoue, alors même que le déploiement est en cours. Si ces données sont transmises entre deux processus ou serveurs, le destinataire ne pourra pas les lire.

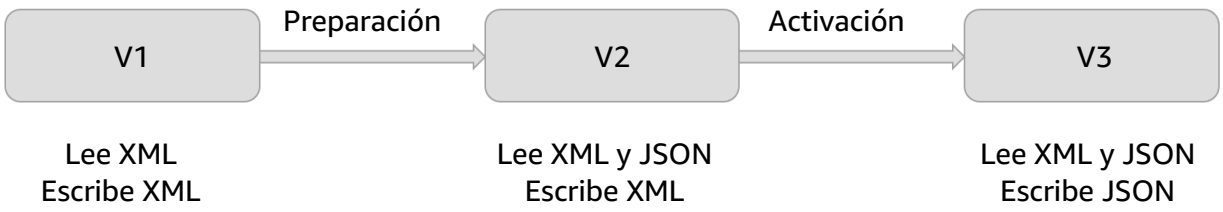
Parfois, les changements de protocole peuvent être très subtils. Supposons deux serveurs qui communiquent de manière asynchrone via une connexion. Pour indiquer mutuellement qu'ils sont actifs, ils acceptent de s'envoyer une pulsation toutes les 5 secondes. Si un serveur ne reçoit de pulsation dans le délai, il suppose que l'autre serveur est en panne et ferme la connexion.

Maintenant, supposons un déploiement qui augmente la période de pulsation à 10 secondes. La validation de code semble mineure, puisque seul un nombre change. Cependant, il n'est pas indiqué d'effectuer une récupération en aval et une annulation. Pendant le déploiement, le serveur exécutant la nouvelle version envoie une pulsation toutes les 10 secondes. Par conséquent, le serveur exécutant l'ancienne version ne reçoit aucune pulsation pendant plus de 5 secondes et met fin à la connexion avec le serveur exécutant la nouvelle version. Dans une grande flotte, cette situation peut se produire avec plusieurs connexions, entraînant une baisse de disponibilité.

Ces changements subtils sont difficiles à analyser en lisant du code ou des documents de conception. Par conséquent, nous vérifions explicitement que chaque déploiement est sûr pour la récupération en aval et la restauration.

Technique de déploiement en deux phases

Une façon de nous assurer que nous pouvons effectuer une restauration en toute sécurité est d'utiliser une technique communément appelée déploiement en deux phases. Supposons le scénario hypothétique suivant avec un service qui gère les données (écritures sur, lecture depuis) sur Amazon Simple Storage Service (Amazon S3). Le service s'exécute sur une flotte de serveurs répartis dans plusieurs zones de disponibilité pour la mise à l'échelle et la disponibilité.
Actuellement, le service utilise le format XML pour conserver les données. Comme indiqué dans le diagramme suivant dans la version V1, tous les serveurs écrivent et lisent le format XML. Pour des raisons professionnelles, nous souhaitons conserver les données au format JSON. Si nous apportons cette modification dans un déploiement, les serveurs qui ont récupéré la modification écriront en JSON. Mais les autres serveurs ne savent pas encore lire JSON. Cette situation provoque des erreurs. Par conséquent, nous divisons ce changement en deux parties et effectuons un déploiement en deux phases.



Comme indiqué dans le schéma précédent, nous appelons la première phase la préparation. Au cours de cette phase, nous préparons tous les serveurs à lire JSON (en plus de XML), mais ils continuent à écrire du code XML en déployant la version V2. Cette modification ne change rien du point de vue opérationnel. Tous les serveurs peuvent toujours lire le format XML et toutes les données sont encore écrites en XML. Si nous décidons d'annuler cette modification, les serveurs reviendront à une condition dans laquelle ils ne pourront pas lire JSON. Ce n'est pas un problème, car aucune des données n'a encore été écrite en JSON.

Comme indiqué dans le schéma précédent, nous appelons la seconde étape l'activation. Dans cette étape, nous activons les serveurs pour qu'ils utilisent le format JSON pour l'écriture en déployant la version V3. Lorsque chaque serveur détecte cette modification, il commence à écrire en JSON. Les serveurs qui n'ont pas encore pris en compte cette modification peuvent toujours lire JSON, car ils ont été préparés lors de la première étape. Si nous décidons d'annuler cette modification, toutes les données écrites par les serveurs qui étaient temporairement dans la phase d'activation sont en JSON. Les données écrites par des serveurs qui n'étaient pas dans la phase d'activation sont en XML. Cette situation est satisfaisante car, comme indiqué dans la V2, les serveurs peuvent toujours lire XML et JSON après l'annulation.
Bien que le diagramme précédent montre le remplacement du format de sérialisation XML par JSON, la technique générale est applicable à toutes les situations décrites précédemment dans la section Modifications de protocole. Par exemple, rappelez-vous le scénario précédent dans lequel la fréquence de pulsation entre les serveurs devait passer de cinq à 10 secondes. Dans l'étape de préparation, nous pouvons obliger tous les serveurs à assouplir la fréquence de pulsation attendue à 10 secondes bien que tous les serveurs continuent à envoyer une pulsation toutes les 5 secondes. Dans la phase d'activation, nous changeons la fréquence de pulsation et la fixons à une fois toutes les 10 secondes.

Précautions à prendre avec les déploiements en deux phases

Je vais maintenant décrire les précautions que nous prenons lorsque nous suivons la technique du déploiement en deux phases. Bien que je fasse référence à l'exemple de scénario décrit dans la section précédente, ces précautions s'appliquent à la plupart des déploiements en deux phases.

De nombreux outils de déploiement permettent aux utilisateurs de considérer un déploiement comme réussi si un nombre minimum d'hôtes capturent le changement et se déclarent en sains. Par exemple, AWS CodeDeploy a une configuration de déploiement appelée minimumHealthyHosts.

Une hypothèse essentielle dans l'exemple de déploiement en deux phases est qu'à la fin de la première phase, tous les serveurs sont mis à niveau pour lire XML et JSON. Si un ou plusieurs serveurs ne parviennent pas à se mettre à niveau pendant la première phase, ils ne pourront pas lire les données pendant et après la deuxième phase. Par conséquent, nous vérifions explicitement que tous les serveurs ont pris en compte les modifications dans la phase de préparation.

Lorsque je travaillais sur Amazon DynamoDB, nous avons décidé de modifier le protocole de communication entre un grand nombre de serveurs couvrant plusieurs microservices. J'ai coordonné les déploiements entre tous les microservices afin que tous les serveurs atteignent la phase de préparation, puis passent à la phase d'activation. Par précaution, j'ai explicitement vérifié que le déploiement avait réussi sur chaque serveur à la fin de chaque phase.

Bien que chacune des deux phases puisse être annulée en toute sécurité, nous ne pouvons pas annuler les deux modifications. Dans l'exemple précédent, à la fin de la phase d'activation, les serveurs écrivent des données en JSON. La version du logiciel utilisée avant les modifications de préparation et d'activation ne sait pas comment lire JSON. Par conséquent, par précaution, nous laissons un long délai entre la phase de préparation et la phase d'activation. Nous appelons cette période la période confirmation, et sa durée est généralement de quelques jours. Nous attendons pour nous assurer de ne pas revenir à une version antérieure.

Après la phase d'activation, nous ne pouvons pas supprimer en toute sécurité la possibilité du logiciel à lire XML. Il n'est pas prudent de la supprimer, car toutes les données écrites avant la phase de préparation sont en XML. Nous ne pouvons l'empêcher de lire XML qu'après avoir vérifié que chaque objet a été réécrit en JSON. Ce processus s'appelle le renvoi. Cela peut nécessiter des outils supplémentaires pouvant être exécutés simultanément pendant que le service écrit et lit des données.

Meilleures pratiques pour la sérialisation

La plupart des logiciels impliquent de sérialiser les données, que ce soit pour la persistance ou le transfert sur un réseau. Au fur et à mesure de son évolution, il est courant que la logique de sérialisation change. Les modifications peuvent aller de l'ajout d'un nouveau champ à la modification complète du format. Au fil des ans, nous sommes parvenus à certaines pratiques recommandées pour la sérialisation:

- Nous évitons généralement de développer des formats de sérialisation personnalisés.

  La logique initiale de la sérialisation personnalisée peut sembler insignifiante et même offrir de meilleures performances. Cependant, les itérations suivantes du format présentent des problèmes qui ont déjà été résolus par des cadres bien établis tels que JSON, Protocol Buffers, Cap'n Proto et FlatBuffers. Lorsqu'ils sont utilisés de manière appropriée, ces cadres fournissent des fonctionnalités de sécurité telles que l'échappement, la compatibilité ascendante et le suivi de l'existence des attributs (c'est-à-dire si un champ a été défini explicitement ou si une valeur par défaut a été affectée de manière implicite).

- À chaque modification, nous attribuons explicitement une version distincte aux sérialiseurs.

  Nous le faisons indépendamment du code source ou de la gestion des versions. Nous enregistrons également la version du sérialiseur avec les données sérialisées ou dans les métadonnées. Les anciennes versions du sérialiseur continuent de fonctionner dans le nouveau logiciel. Il est généralement utile d'émettre une métrique pour la version des données écrites ou lues. Ainsi, les opérateurs ont une visibilité et disposent d'informations de dépannage en cas d'erreur. Tout cela s'applique également aux versions RPC et API.

- Nous évitons la sérialisation des structures de données que nous ne pouvons pas contrôler.

  Par exemple, nous pourrions sérialiser les objets de collection de Java en utilisant la réflexion. Cependant, lorsque nous essayons de mettre à niveau le JDK, l'implémentation sous-jacente de telles classes peut changer, ce qui entraîne l'échec de la désérialisation. Ce risque s'applique également aux classes de bibliothèques partagées entre des équipes.

- En règle générale, nous concevons des sérialiseurs pour permettre la présence d'attributs inconnus.

  Lorsque cela est possible, nos sérialiseurs conservent des attributs inconnus lors de l'écriture des données. Avec cette possibilité, même si un serveur exécutant la nouvelle version du logiciel inclut de nouveaux attributs dans les données lors de la sérialisation, les serveurs exécutant l'ancienne version n'effaceront pas les attributs lors de la mise à jour des mêmes données. Ainsi, un déploiement en deux phases n’est pas nécessaire.

Comme beaucoup de nos meilleures pratiques, nous les partageons avec la prudence ; nos directives ne s'appliquent pas à toutes les applications et tous les scénarios.

Vérification de la possibilité d'annuler en toute sécurité une modification

En règle générale, nous vérifions explicitement qu’une modification logicielle peut être récupérée en aval et annulée dans ce qu'on appelle un test upgrade-downgrade. Pour ce processus, nous avons mis en place un environnement de test représentatif des environnements de production. Au fil des ans, nous avons identifié quelques modèles que nous évitons lors de la configuration d’environnements de test.

J'ai connu des situations dans lesquelles le déploiement d'un changement d'environnement production provoquait des erreurs, alors que le changement avait réussi tous les tests dans l'environnement de test. Dans un cas, les services de l’environnement de test n’avaient chacun qu’un serveur. Ainsi, tous les déploiements étaient atomisés, ce qui excluait la possibilité d’exécuter simultanément différentes versions du logiciel. Maintenant, même si les environnements de test ne voient pas autant de trafic que les environnements de production, nous utilisons plusieurs serveurs de différentes zones de disponibilité derrière chaque service, comme ce serait le cas dans un environnement de production. Nous aimons la frugalité chez Amazon, mais pas quand il s’agit d’assurer la qualité.

Dans une autre occasion, l'environnement de test avait plusieurs serveurs. Cependant, le déploiement a été effectué sur tous les serveurs à la fois pour accélérer les tests.. Cette approche a également empêché les anciennes et les nouvelles versions du logiciel de s'exécuter en même temps. Le problème lié à la récupération en aval n'a pas été détecté. Nous utilisons maintenant la même configuration de déploiement dans tous les environnements de test et de production.

Pour les modifications impliquant une coordination entre microservices, nous maintenons le même ordre de déploiement sur tous les microservices dans des environnements de test et de production. Cependant, l'ordre de la récupération en aval et l'annulation peut être différent. Par exemple, nous suivons généralement un ordre spécifique dans le contexte de la sérialisation. Autrement dit, les lecteurs passent avant les enregistreurs lors de la récupération en aval, et inversement lors de l'annulation. Un ordre approprié est généralement suivi dans les environnements de test et de production.

Lorsque la configuration d'un environnement de test est similaire à celle des environnements de production, nous simulons le plus fidèlement possible le trafic de production. Par exemple, nous créons et lisons plusieurs enregistrements (ou messages) successivement rapidement. Toutes les API

sont utilisées en continu. Ensuite, nous faisons passer l'environnement par trois phases, chacune d'une durée raisonnable pour identifier les bogues potentiels. La durée est suffisamment longue pour que tous les API, flux de travail et tâches par lots soient exécutés au moins une fois.
Tout d'abord, nous déployons le changement sur environ la moitié de la flotte pour assurer la coexistence des versions logicielles. Ensuite, nous terminons le déploiement. Enfin, nous initions le déploiement de l'annulation et suivons les mêmes phases jusqu'à ce que tous les serveurs exécutent l'ancien logiciel. S'il n' existe pas d'erreur ou de comportement inattendu lors de ces phases, nous considérons que le test a abouti.

Conclusion

Pour rendre un service fiable, il est essentiel de pouvoir annuler un déploiement sans perturbation pour nos clients. Le test explicite de l'exécution des annulations en toute sécurité évite de recourir à une analyse manuelle, qui peut être sujette à des erreurs. Lorsque nous déterminons que l'annulation d'un changement n'est pas sûre, il est généralement possible de le diviser en deux, chacun pouvant être récupéré en aval ou annulé.