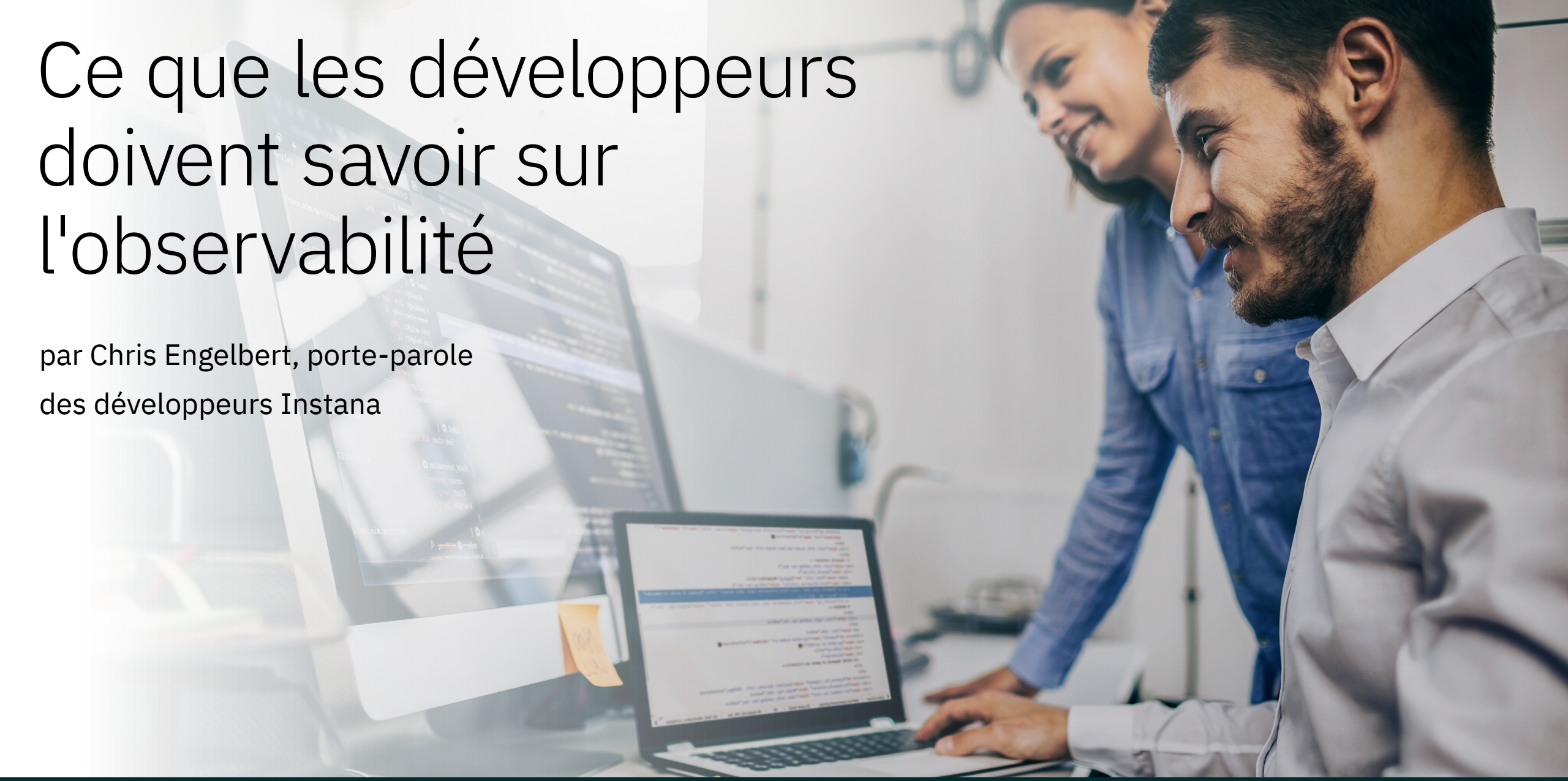
Ce que les développeurs doivent savoir sur l'observabilité

par Chris Engelbert, porte-parole
des développeurs Instana



Sommaire

01

Introduction

02

Terminologie de l'observabilité

03

Traçage distribué

04

Une ingénierie performante

05

Développement basé sur l'observabilité

06

Être ou ne pas être... open source

07

Se simplifier la vie

01 Introduction

Nous les développeurs, nous nous souvenons de l'époque où Nagios était une technologie de pointe. Depuis ce temps, le terme "surveillance" a mauvaise presse. Nous avions des tableaux de bord qui débordaient de graphiques et de chiffres et qui ne nous servaient en rien à résoudre les problèmes, et pourtant il nous était demandé de les surveiller. Bien évidemment, ces tableaux de bord nous montraient que quelque chose ne tournait pas rond, mais ce n'était pas notre rôle de gérer le système. Nous finissions par nous déconcentrer très vite face à toutes ces métriques.

Pourquoi Nagios n'a-t-il pas donné de meilleurs résultats pour nous, les développeurs ? En dehors des métriques de performance qui étaient collectées directement à partir de compteurs intégrés au flux de code des applications, tout se résumait à une surveillance de type boîte noire : principalement des envois de commandes ping à des hôtes, des connexions à des services, le tout en général accompagné d'une analyse de journaux à l'aide d'expressions régulières.

Pour aggraver le tout, à l'époque ancienne des opérations de service, le développement et les opérations étaient des univers complètement séparés. Nous avions des administrateurs de base de données qui empêchaient tout le monde d'exécuter ne serait-ce qu'une instruction SELECT de leur propre chef.

L'équipe des opérations - celle qui exploitait le système en production, gérait les pannes système et relayait les informations sur les incidents aux ingénieurs - faisait tout son possible pour écarter les développeurs des systèmes de production. Dans le meilleur des cas, elle ne permettait qu'à un nombre très limité de développeurs d'y accéder.

Enfin, il y avait le service d'ingénierie. C'était notre travail de créer des fonctions, de les faire tester, et ça s'arrêtait là. Là où nous nous impliquions, c'était pour corriger les bogues. Pour nous, le code source était un art en soi. Et bien évidemment, c'était aussi notre chasse gardée, que nous tenions nous aussi jalousement à l'écart des non-développeurs.

L'exploitation et la surveillance du système ne faisaient tout simplement pas partie de notre mission. À l'exception d'une tâche bien particulière : l'ajout d'éléments de surveillance dans notre superbe code source - un peu comme si on avait posé un revêtement en aluminium sur la chapelle Sixtine. Nous détestions ça pour toutes les bonnes raisons. Ça se répandait comme une traînée de poudre.

Les responsabilités étaient fractionnées, et pendant longtemps, les ingénieurs logiciels écrivaient du code, puis rentraient chez eux.

L'équipe des opérations prenait le relais à partir de là et s'occupait de gérer tout ce qui n'allait pas.

Mais les temps ont changé. Aujourd'hui, les systèmes sont différents, les déploiements se déroulent d'une autre façon et les frontières entre les équipes sont floues, voire complètement inexistantes. Cela dit, en tant que développeurs, nous sommes plus proches que jamais des opérations et nous jouons un rôle indirect dans l'exploitation des systèmes.

Face à la complexité des systèmes modernes qui exécutent des microservices, la notion de "surveillance riche de sens" joue désormais un rôle important dans notre quotidien. Pour éviter la surenchère de métriques dont nous avons pâti dans le passé, nous devons nous pencher sur les avantages de l'observabilité.

Ce livre électronique est un état des lieux de cet univers nouveau. Nous allons laisser derrière nous toutes nos expériences désagréables de la surveillance. Nous allons faire nos premiers pas dans le monde de l'observabilité et comprendre son importance de plus en plus grande pour les développeurs.

02

Terminologie de l'observabilité

Tout d'abord, clarifions la terminologie de base de l'observabilité. Pour se plonger dans l'univers de l'observabilité, il est important de comprendre les différences entre la surveillance et l'observabilité, ainsi que la manière dont les données sont représentées pour l'utilisateur.

Surveillance

Traditionnellement, la surveillance s'est intéressée en priorité aux métriques de série temporelle. Le processus était toujours le même : collecter un ensemble de métriques, les insérer dans des graphiques dans des tableaux de bord, déterminer les métriques pour lesquelles des alertes devaient être définies, et sélectionner des seuils d'alerte. C'était mieux que rien, mais cette approche était loin d'être idéale, car elle se traduisait par un nombre trop élevé ou trop faible d'alertes. Elle inspirait de plus un faux sentiment de sécurité et le traitement des incidents demandait beaucoup de temps.

Même si nous étendons la surveillance à la santé et à la performance des services, la surveillance traditionnelle a une approche des alertes qui est basée sur les symptômes. Les problèmes sont les symptômes auxquels elle réagit. Si un service externe devient inaccessible, alors c'est un symptôme. Au lieu de réagir aux symptômes, il serait préférable de collecter des données pertinentes fournies avec un contexte complet.

Observabilité

Même si l'observabilité n'a rien de nouveau, elle est le complément parfait de la surveillance. Elle peut être considérée comme un sur-ensemble.

Le terme même, observabilité, implique que l'on observe quelque chose. L'élément important ici est d'observer à un niveau plus granulaire que la surveillance. Tout en continuant d'inclure tous les chiffres et les graphiques de la surveillance, l'observabilité apporte en plus des indications sur ce qu'il est important de surveiller, et fournit ces indications à toutes les différentes équipes qui étaient auparavant séparées les unes des autres.

Qui a besoin de centaines de graphiques et de tableaux, surtout s'ils ne n'aident jamais à résoudre les problèmes ? Personne. En revanche, il est bien possible que les données indispensables pour aider à résoudre un problème coriace n'aient même pas été collectées.

De plus, l'observabilité ajoute le traçage réparti, qui est à la base une trace de pile de microservices. Nous expliquerons plus loin de quoi il s'agit. Et n'oubliez pas : une analyse de journal significative va bien plus loin qu'une simple recherche d'erreurs basée sur une expression régulière (regex).

02 Terminologie de l'observabilité

Ces deux derniers éléments sont conçus pour mettre en œuvre la surveillance depuis l'intérieur des applications et des services. Contrairement à une surveillance pure et simple, nous collectons des informations provenant directement de l'intérieur du service et les réunissons avec tous les autres éléments que nous connaissons sur le système.

L'observabilité intègre dès le départ les domaines de défaillance connus dans notre système. Par exemple, un service qui se connecte à un autre service via le protocole HTTP peut ne pas réussir à se connecter pour diverses raisons : c'est un domaine de défaillance connu. Les domaines de défaillance peuvent varier selon la manière dont les utilisateurs envisagent le service ou le système dans son ensemble. Pour cette raison, il est important de prendre en compte différentes perspectives lors de la conception des domaines de défaillance et du contexte qui leur est associé : par exemple un débogage facile, selon le public.

Lorsque nous examinons l'observabilité, nous discernons trois piliers qui apportent des informations sur notre problème et permettent de le comprendre : les métriques de santé et de performance, les traces réparties et les journaux.

Observabilité d'entreprise

Dans l'entreprise, l'observabilité doit être hautement évolutive pour pouvoir gérer les complexités et l'évolutivité de systèmes transitoires qui exécutent des microservices éphémères ou des applications sans serveur. Les systèmes conçus pour l'observabilité d'entreprise effectuent un suivi automatique de l'infrastructure ou de l'environnement de service, qui se modifient constamment. Une partie de ce processus consiste à découvrir des instances ou des services nouveaux ou arrêtés sans intervention manuelle. Ils fournissent normalement des bibliothèques d'encapsuleur pour les modifications non intrusives et minimales du code source. Ils vont parfois même jusqu'à une instrumentation du code entièrement automatique qui permet d'ajouter des mesures et des sondes de santé dans une application déjà en cours d'exécution.

L'instrumentation du code permet d'ajouter les points de départ et de fin des opérations dans la base de code en cours d'exécution, sans avoir à les ajouter manuellement dans le code source. Cela dit, les développeurs se trouvent libérés pour la majeure partie de la tâche fastidieuse consistant à ajouter des éléments de surveillance et de traçage dans leur base de code. Ils disposent ainsi de plus de temps pour travailler sur les cas d'utilisation de l'entreprise.

En outre, le profilage de code des systèmes de développement, de transfert et de production, toujours activé mais mobilisant peu de temps système, permet de mieux connaître les performances des services dans des conditions opérationnelles, aspect que les développeurs attendaient depuis longtemps. Les systèmes ont toujours eu des comportements différents de ce que nous attendions dans les environnements de production.

Toutefois, après avoir recueilli toutes ces données, il est important de ne pas se contenter de les afficher selon la vieille méthode, avec des chiffres, des tableaux et des diagrammes. Il convient de prendre du recul et de percevoir que c'est le contexte des données qui est important pour résoudre les problèmes, surtout pour les développeurs.

Pour rendre utiles les quantités massives de données, les solutions d'observabilité d'entreprise doivent fournir une corrélation automatique, qui facilite la compréhension et crée des informations exploitables. Ces solutions facilitent aussi souvent l'analyse de la cause première, en fournissant le contexte nécessaire des événements et des informations corrélés.

03 Traçage réparti

Les traces réparties sont aux systèmes répartis ce que les traces de pile sont aux applications et aux exceptions ou aux paniques. Il s'agit d'une technique permettant de capturer et de chronométrer les gestionnaires de services et les appels internes pendant qu'une requête progresse dans l'environnement d'un système pour générer sa réponse. Pour cette raison, on parle aussi parfois de traçage de requête répartie.

Le traçage réparti aide les développeurs à analyser les flux de requêtes et à déterminer la cause première des problèmes ou des goulets d'étranglement des performances.

Imaginez qu'un utilisateur appelle un service utilisateur pour lui demander les détails de son propre compte. Le service lui-même appelle plusieurs autres services plus bas dans la pile afin d'extraire différents types d'informations. La Figure 1 montre le flux de base de l'appel. Les informations indiquant par exemple l'auteur de l'appel et son destinataire sont visibles, mais des données importantes telles que les informations de temps sont absentes de ce diagramme.

Ce scénario montre à quel moment le traçage réparti entre en jeu. Les Figures 2 et 3 affichent le flux de l'appel de deux manières différentes, sous la forme d'une barre de temps et d'une arborescence de type trace de pile.

Figure 1. Une requête envoyée à un service utilisateur progressant dans les services internes

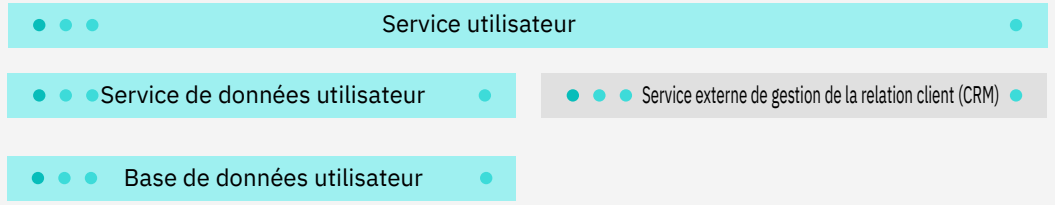


Figure 2. Flux d'appels représenté sous la forme d'un diagramme de temps et de hiérarchie

Figure 3. Diagramme hiérarchique du flux de code

03

Traçage réparti

Ces deux vues fournissent les informations nécessaires permettant aux développeurs de rechercher les appels lents, les opérations mettant longtemps à s'exécuter, les services défaillants et l'endroit où ces défaillances se produisent.

Pour représenter ces informations, la trace est décomposée en éléments, tels que les séquences et les appels.

Une séquence, parfois appelée séquence de temps, représente l'exécution d'une seule opération à l'intérieur du flux de la trace répartie. Dans ce contexte, une opération est une exécution de code qui possède une heure de début et une heure de fin, ainsi que des séquences parents et enfants le cas échéant.

En outre, une séquence contient des métadonnées qui fournissent du contexte sur la séquence elle-même. Pour connecter plusieurs séquences, des identificateurs (id-séquence et id-trace) sont utilisés pour créer les hiérarchies parents/enfants.

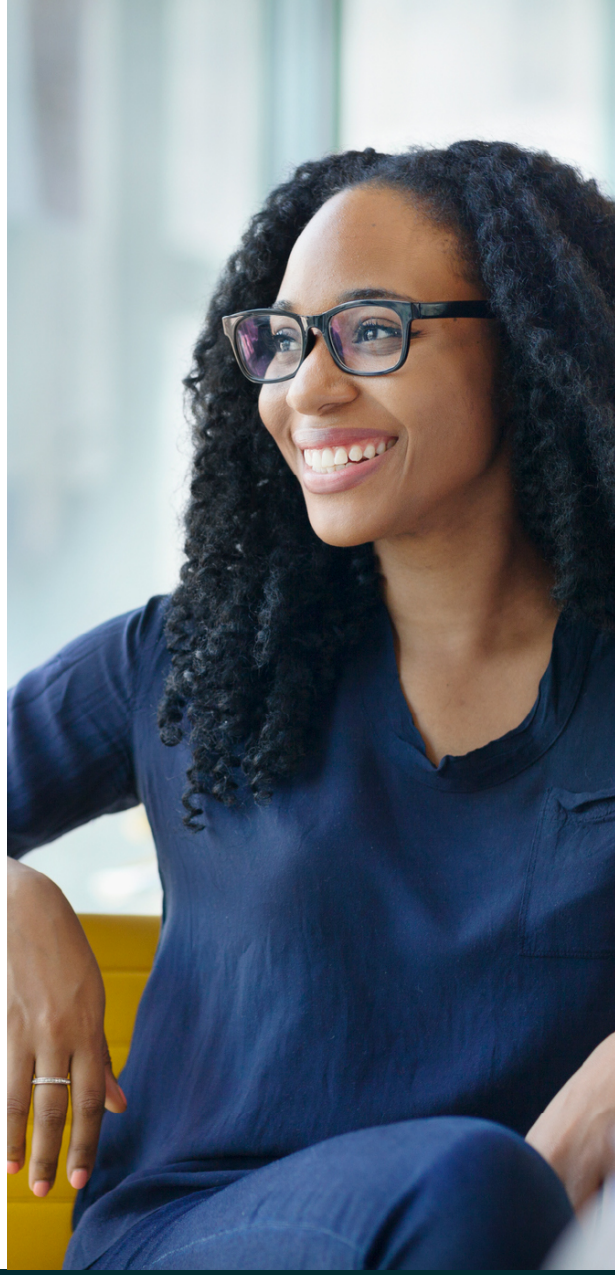
La communication entre deux séquences est représentée par des appels. Les appels contiennent des informations sur le type de connexion qui a été utilisé, des informations d'en-tête et des données de réponse, telles que les codes d'état du protocole HTTP si un appel était erroné. Des informations contextuelles personnalisées peuvent également être ajoutées.

L'importance de l'observabilité

Lorsque l'on travaille avec des systèmes répartis, les enjeux sont très différents de ceux des applications monolithiques que nous construisions dans le passé. Même si nous apprécions d'avoir une vision simple du monde, la réalité est très différente, parfois sans que nous le sachions.

De nombreux systèmes et services différents s'exécutent de façon indépendante et simultanée. La majorité des services comprennent toujours un ou plusieurs sous-appels à d'autres services ou bases de données. Comprendre l'interaction entre ces services est l'un des éléments les plus importants pour un développeur lorsqu'il tente de corriger un bogue ou d'atténuer un problème de performance.

Il est également important de ne pas faire figurer trop de diagrammes de métriques dans les tableaux de bord, sous peine de revenir des années en arrière. Ce que nous recherchons, ce sont des informations pertinentes qui nous mènent à la cause d'un problème le plus rapidement possible. Nous avons besoin que le système nous indique quelles métriques sont importantes pour le domaine de défaillance de chaque service. Il est important ici de nous concentrer uniquement sur ce qui compte.



03 Traçage réparti

Il est logique d'adopter cette approche, car rien ne garantit que nous obtiendrons un système exempt de défaillance, quel que soit le nombre de métriques que nous ajoutons. C'est plutôt l'inverse : les défaillances de notre système ne peuvent être évitées. J'ai abordé ce problème dans un article de blog intitulé Building Resilient Applications-Embrace the Failure (Créer des applications résilientes : accepter la défaillance), qui étudie ce sujet en détail. Le mieux que nous puissions faire est de nous préparer à ce scénario en comprenant le domaine de défaillance.

En définitive, l'élément important pour nous, les développeurs, est de retrouver notre créativité et notre motivation. Nous voulons travailler sur les éléments qui font avancer l'entreprise, à savoir nos cas d'utilisation métier. Nous devons éviter d'encombrer notre code source avec des millions de métriques ou de points de traçage. Et surtout, nous ne voulons pas passer notre temps à éteindre des incendies ou à corriger des bogues.

Sur ce dernier point cependant, un système d'observabilité clair, pratique et capable de fournir des informations peut être utile pour toutes ces tâches. En fournissant des bibliothèques de type encapsuleur léger, ou, mieux encore, une instrumentation automatique, nous pouvons conserver une base source propre et fidèle au cas d'utilisation métier. Cela nous permet également de récupérer beaucoup de temps : le temps consacré à analyser des problèmes complexes dans les systèmes répartis, le temps passé à ajouter des points de métrique, le temps consacré

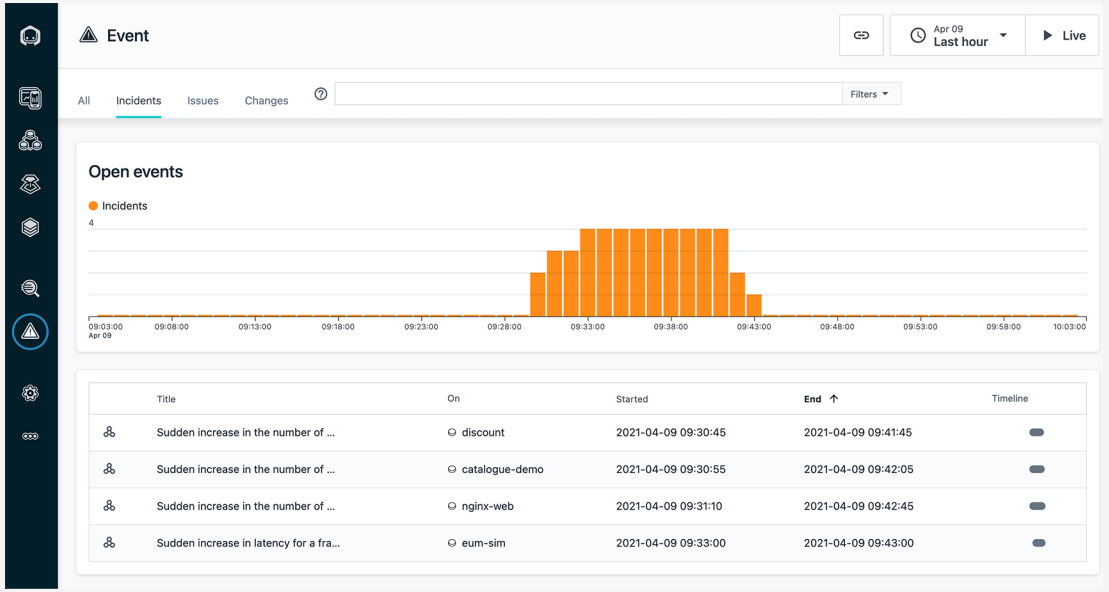


Figure 4. Visibilité rapide de l'état de santé d'un service avec marqueurs d'édition. Source : Instana.

03
Traçage réparti

à comprendre l'interaction et la communication entre les systèmes et, non des moindres, le temps perdu à rechercher les goulets d'étranglement. Aujourd'hui, une trace répartie peut nous aider à comprendre les faiblesses et les changements du système actuel avant de tomber sur des goulots d'étranglement ou des problèmes d'évolutivité.

Le délai de résolution

Lorsqu'un déploiement échoue ou que notre nouvelle version se comporte de manière erratique, nous devons être rapides. Soit nous revenons à une version antérieure, soit nous analysons le problème, recherchons un correctif et l'implémentons, et l'affaire est alors réglée.

Parfois, le problème est une simple négligence, reproductible localement et rapidement réparable. L'affaire se corse, cependant, lorsque nous ne pouvons pas reproduire le problème directement et localement. Dans ce cas, les outils d'observabilité peuvent être d'une grande utilité. La réalité est qu'il n'existe aucun moyen de prévoir le degré de complexité des problèmes qui se poseront dans nos environnements de production. Cette impossibilité à prévoir fait que nous devons collecter les données requises en production, pendant l'exécution.

Le traçage réparti, qui offre un moyen simple d'examiner le flux de requêtes, permet de collecter rapidement des informations sur l'endroit où la requête a échoué. Nous pouvons également voir si l'erreur a remonté la pile ou si elle a été traitée quelque part entre les deux, et si un utilisateur a été affecté.

Les traces réparties fournissent également des informations sur les auteurs des appels et leurs destinataires, avec leurs en-têtes respectifs et des informations sur le temps ou de nouvelles tentatives.

Cette information constitue une première étape rapide de l'analyse de la cause première et permet éventuellement d'obtenir l'aide de collègues en charge du domaine responsable pour résoudre le problème. Tout cela peut être accompli sans tâtonner au hasard, mais en s'appuyant sur des conclusions fondées sur des données.

Cette approche d'analyse descendante n'est pas seulement beaucoup plus rapide, mais elle permet aussi de localiser rapidement la cause première réelle du problème. Nous obtenons une vue d'ensemble de tous les aspects du problème et identifions ceux que nous devons inclure dans l'enquête, en fonction des dépendances réelles en amont ou en aval.

En résumé, lors de la recherche du problème, l'observabilité fournit les connaissances dont nous avons besoin pour agir rapidement et, dans le meilleur des cas, en n'introduisant qu'une dépendance minimale vis-à-vis des autres équipes ou des partenaires externes tels que les hébergeurs. Ces derniers peuvent être particulièrement lents à répondre lors de l'examen d'un problème demandant à être résolu dans des délais très courts.

Un déploiement qui a mal tourné

Historiquement, en tant que développeur, je pourrais sauter cette section. Les déploiements ne relevaient pas de ma responsabilité.

03 Traçage réparti

Aujourd'hui, cependant, les développeurs écrivent des configurations Dockerfile pour créer des images Docker. Ils fournissent aussi souvent des configurations de déploiement sous la forme de descripteurs YAML Kubernetes (K8s), décrivant les conteneurs, les services, et ainsi de suite - ou du moins partiellement - pour de bonnes raisons.

DevOps n'est pas le seul chargé de fournir ces descripteurs. En effet, un déploiement complet est le plus souvent une combinaison de descripteurs de déploiement K8s d'ingénierie et de DevOps.

Pour reprendre, voilà que notre dernier déploiement a échoué. Il n'y a pas de problème apparent concernant le processus de déploiement lui-même, mais toujours est-il que le service tombe en panne juste après le déploiement. C'est maintenant à notre tour d'entrer en scène et de briller.

La première étape est de comprendre la situation. Est-ce que c'était vraiment nous ? L'examen d'une trace répartie permet souvent de répondre rapidement à cette question. Est-ce notre service qui provoque une erreur, par exemple un code de statut HTTP 500, ou le problème provient-il de plus loin dans la pile ? Le dépendance du service en aval renvoie-t-elle la défaillance ? Peut-être envoyons-nous les mauvaises données. Ici encore, une recherche rapide permet de répondre à cette question.

Les solutions d'observabilité fournissent les preuves nécessaire, signalées à l'origine par les systèmes et basées sur notre analyse du domaine de défaillance, sous la forme d'observations précises et contextuelles. Les traces réparties sont le témoin privilégié lorsque vous cherchez à comprendre rapidement des situations encore inconnues. Le temps moyen de réparation et le temps moyen de restauration du service sont ici les métriques fondamentales.

Vous vous souvenez de la règle du "Ne jamais déployer un vendredi" ? Moi aussi, et je parie que c'est le cas pour tout le monde. De nos jours, je dirais qu'avec une bonne observabilité et la possibilité de détecter immédiatement de petits changements dans le comportement, le temps d'attente ou le taux d'erreur, nous pouvons et devrions effectivement déployer les vendredis. En commençant par contre peut-être tôt le matin. Dans ce cas, il nous reste encore un bon nombre d'heures pour corriger un problème ou effectuer une annulation en cas de difficulté.

Si nous ne pouvons pas corriger un problème immédiatement parce que nous sommes allés trop loin, notre seule solution est d'annuler pour revenir à une version précédente. Avec un flux de déploiement automatisé, ce processus est assez facile. Nous lançons notre pipeline d'intégration continue et de distribution continue (CI/CD) avec une étiquette ou une version d'édition différente, et c'est parti. Nous venons de gagner beaucoup de temps pour étudier plus en profondeur, corriger plus soigneusement et éliminer la pression supplémentaire de devoir agir à toute vitesse.

04

Une ingénierie performante

Après avoir beaucoup parlé d'accélérer l'ingénierie et de réserver notre temps aux missions générant le plus de valeur, il est maintenant temps de parler de l'ingénierie elle-même et de la façon dont elle est en train de changer, ou dont elle a changé ces dernières années.

La page 18 du rapport Accelerate State of DevOps 2019 décrivait 4 points indispensables aux équipes d'ingénieurs de demain pour être performantes. Ces points faisaient ressortir le fait que les ingénieurs, les équipes des opérations et les administrateurs de base de données n'agissaient désormais plus dans des domaines totalement séparés.

Mais que signifie réellement cette affirmation ? Allons-nous nous être incorporés à l'équipe des opérations ? Certainement pas. Je ne me considèrerais jamais comme un responsable des opérations, même si une bonne partie de mes tâches au quotidien relèvent de ce domaine. Mais notre différence reste bien là. Je ne suis en aucun cas un spécialiste de l'ingénierie de déploiement, même si d'autres le sont. Et ceci vaut pour tout autre sujet, même en ingénierie. Tout le monde n'est pas ingénieur spécialisé en performances. Il reste nécessaire de pouvoir faire appel à ces spécialistes très pointus, mais il est plus que jamais important pour tous d'avoir une vue d'ensemble.

04

Une ingénierie performante

Les 4 questions auxquelles nous voulons répondre sont les suivantes :

- Quelle est notre fréquence de déploiement ?
- Quel est notre délai d'introduction de changements ?
- Quel est notre délai de restauration du service ?
- Quel est notre taux d'échec suite à des changements ?

Les réponses à ces questions nous indiquent à quelle vitesse nous pouvons itérer. Nos réponses constituent une bonne base de référence, mais elles doivent être honnêtes. Il en va de même pour notre base de code : nous ne pouvons optimiser que si nous comprenons le problème.

Quelle est notre fréquence de déploiement ?

Est-ce que nous déployons une fois par an, une fois par mois, une fois par jour ou plusieurs fois par jour ? Si nous déployons plus souvent, cela signifie que nous devons comprendre mieux et plus rapidement le comportement des nouvelles versions.

Combien de temps faut-il pour lancer le code en production ?

De combien de temps avons-nous réellement besoin pour un déploiement ? S'agit-il d'une opération majeure, éventuellement d'un train d'édition entre toutes les équipes ? Peut-être pouvons-nous déployer indépendamment, ou déployer automatiquement une fois que tous les tests sont validés.

Dans quel délai pouvons-nous effectuer la reprise suite à une panne ?

En cas de dysfonctionnement, qui peut analyser le problème ? Ma question favorite est la suivante : "Qui peut résoudre le problème de mon service si je suis en congés, sans téléphone ni connexion à Internet, et combien de temps faudra-t-il à la personne pour comprendre le problème ?". Combien de déploiements échouent-ils ?

Dernière question, la plus courante mais non des moindres, celle à laquelle la plupart des gens peuvent probablement répondre sans trop réfléchir : les déploiements échouent-ils et, si oui, combien et pourquoi ?

Cette liste est un excellent point de départ pour déterminer et comprendre la vitesse des itérations dans notre entreprise, notre équipe, et dans notre service ou notre application.

J'irais même plus loin en disant qu'il manque ici une question fondamentale : celle des différences entre les éditions. Il y a de quoi se féliciter d'avoir un faible nombre d'échecs de déploiements, et peut-être sommes-nous déjà capables de déployer plusieurs fois par jour. Cependant, il reste une question qui n'a pas été posée : quelle est la différence en termes d'erreurs et de performances entre les éditions ?

Les meilleurs déploiements et l'itération la plus rapide ne valent rien si chaque édition est moins performante que la précédente ou si elle augmente le taux d'erreurs. L'essentiel, c'est que nous ne voulons pas continuer à jouer les pompiers, pas vrai ?

Le développement est en train de changer

La manière dont nous développons les applications est manifestement en train de changer. Ces dernières années, la nécessité d'ajouter des systèmes en scale-out n'a jamais été aussi importante. Si l'on met de côté les solutions de l'Internet des objets (IoT), le nombre d'utilisateurs, de clients ou de personnes que nous voulons voir satisfaits de notre travail ne cesse d'augmenter.

De nombreuses entreprises sont encore en phase d'adoption, mais il est presque certain que les nouveaux systèmes ne sont plus construits sur un mode monolithique. L'évolutivité est devenue trop importante, tandis qu'une mise à l'échelle verticale est trop coûteuse et, dans certains cas, irréalisable.

Les microservices, même s'ils ne sont pas l'arme infaillible comme beaucoup l'affirment, sont clairement là pour rester. Le fait de diviser de grandes quantités de travail en tous petits fragments, puis de les déployer indépendamment les uns des autres semble une excellente solution et un principe formidablement bien adapté à l'évolutivité. La possibilité de pouvoir mettre à l'échelle certaines parties du système de manière indépendante au gré des besoins est fabuleuse. L'inconvénient est que cette évolutivité a un coût, car elle oblige à ajouter de nombreuses opérations de réseau entre les services. Les opérations peuvent dépasser leur délai d'attente, échouer ou renvoyer des résultats inexploitables. Nous luttons désormais contre un ennemi totalement inédit.

04 Une ingénierie performante

Par ailleurs, je me souviens de l'époque où les données étaient stockées dans des bases de données relationnelles. Personne ne se posait de questions, car il n'y avait pas d'autres solutions. Même si les bases de données relationnelles ont encore de beaux jours devant elles, nous avons aujourd'hui le choix entre de nombreux systèmes différents : simples systèmes de type clé-valeur ou magasins de documents, bases de données orientées graphes ou bases de données de séries chronologiques, ou magasins de colonnes pour les agrégations extrêmement rapides.

Tous ces systèmes sont différents, et tous se comportent différemment. Personne ne peut les maîtriser tous à fond. Cependant, l'avantage que nous avons en tant que développeurs est que nous avons la possibilité de sélectionner le meilleur outil pour la tâche à accomplir, tout en continuant de découvrir des nouveautés.

Dernier point, mais pas le moindre, nous déployons les systèmes et les services d'une façon qui a changé par rapport aux années précédentes. Le déploiement sur des systèmes dédiés est devenu l'exception parmi les nouvelles installations. Aujourd'hui, nous déployons presque tous sur des machines virtuelles. Beaucoup d'entre nous déploient déjà dans le cloud ou dans des environnements d'auto-hébergement avec Kubernetes ou Cloud Foundry. Docker et d'autres environnements d'exécution conteneurisés tels que CRI-O sont monnaie courante sur les machines des développeurs.

Toutes ces technologies sont excellentes, et rapprochent le développement sur ma machine du futur environnement d'exécution. J'ai toujours détesté la phrase "Ça marche sur ma machine", même si elle était vraie. La raison pour laquelle je ne l'aimais pas est simple. Cette phrase voulait dire qu'il était en train de se passer quelque chose que je ne pouvais pas comprendre ou expliquer immédiatement.

Pipelines de développement

Si nous voulons itérer plus rapidement, nous devons mettre en place un bon support. Bien que les tests unitaires ne soient, on l'espère, pas nouveaux, et que les tests d'intégration de base soient couramment utilisés, tout ce qui les suit est souvent considéré comme trop complexe, trop coûteux ou pas assez orienté développeur.

Cependant, avec des cycles d'itération courts et des systèmes répartis, les tests d'intégration sont devenus plus importants que jamais. L'intégration continue sur un système plus proche de la préproduction et de la production est une nécessité pour avoir des cycles de feedback rapides pendant le développement. Ils sont aussi une étape bien adaptée pour collecter les premières informations sur les performances et le taux d'erreurs. La détection de problèmes potentiels dès le début du développement d'une fonction permet d'éviter les longs cycles de reconstruction et de valider rapidement les attentes, ou de prouver au contraire qu'elles sont erronées.

Un autre élément du puzzle est qu'il est préférable d'automatiser les tests de charge ordinaires chaque fois que possible. Les raisons sont les mêmes qu'auparavant : des boucles de feedback courtes et une validation précoce des modèles, des attentes et des questions liées aux performances.

La dernière étape pour qu'une équipe d'ingénierie soit vraiment performante est la distribution continue, qui nécessite que tous les éléments que l'on vient de mentionner soient mis en place. Il faut aussi avoir le courage d'accepter l'échec. Lorsque l'on sait qu'une bonne compréhension de la défaillance, l'analyse rapide de la cause première et le déploiement d'un correctif du bogue sont non seulement possibles, mais gérés par tous les membres de l'équipe et par tous les outils tout au long du processus de production, il est automatiquement beaucoup plus facile de trouver ce courage. Nous nous sentons beaucoup plus protégés pour continuer.

Cela dit, les situations de défaillance doivent être reconnues rapidement. Les outils d'observabilité enrichis par le contexte doivent prendre en charge le processus d'analyse du problème et aider à rechercher la cause première aussi rapidement que possible. En outre, tous les services doivent être aussi résilients que possible en cas d'échec des dépendances.

Une fois le correctif créé et validé, le pipeline se déclenche, construit et teste la nouvelle version et la déploie en production. Les petits services

04
Une ingénierie performante

indépendants sont plus faciles à gérer avec un tel processus que les grands services monolithiques.

Enfin, on soulignera de nouveau que toutes les étapes doivent être entièrement automatisées pour réduire l'impact de l'échec d'un déploiement ou d'une version inutilisable.

En production

Juste après le lancement en production d'un déploiement, la question la plus importante hormis "Est-ce que ça marche ?" était "Comment ça se passe côté performances ?"

Comme on vient de le voir, un cycle de traitement rapide n'a de sens que si les nouvelles versions ne sont pas moins performantes dans l'ensemble que les anciennes. Il existe des exceptions à cette règle, mais il ne doit s'agir que d'un impact connu, attendu et dont on savait à l'avance qu'il se solderait par une détérioration des performances, un impact anticipé pour une éventuelle mise à l'échelle de l'infrastructure.

Les métriques clés importantes à surveiller immédiatement après un déploiement sont les changements inattendus du temps d'attente moyen, du taux d'erreurs ou des appels en aval, par exemple, le nombre d'appels de base de données ou d'autres éléments similaires. Les très bons outils d'observabilité offrent des comparaisons directes avant/après déploiement qui permettent d'accéder facilement à ces métriques.

En tout cas, partons de cette hypothèse : notre déploiement s'est bien déroulé, les chiffres immédiats

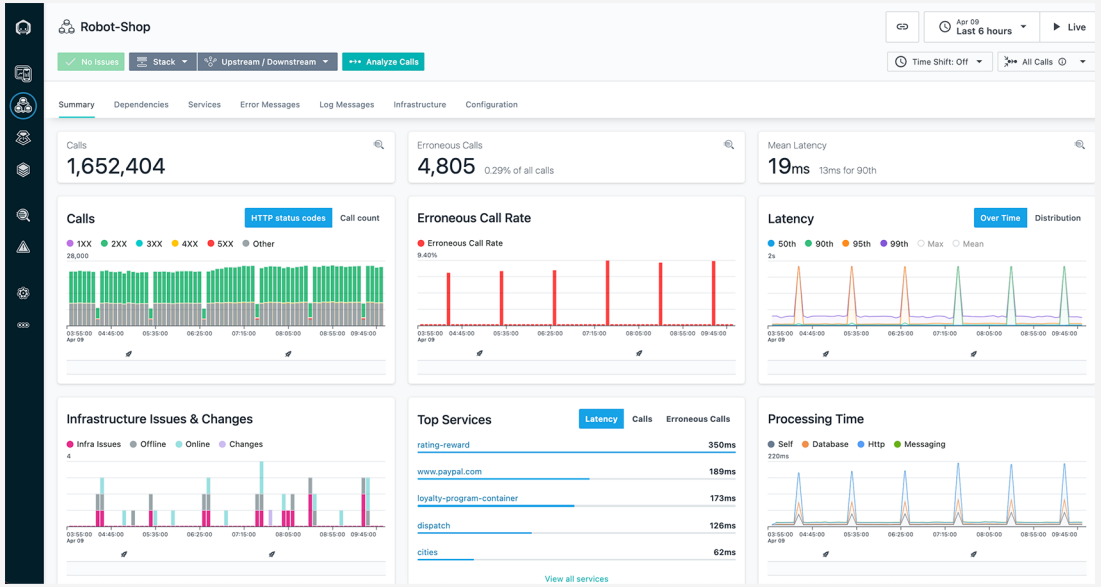


Figure 5. Repères correspondant aux éditions dans le tableau de bord des services, fournissant des informations immédiates après et entre les éditions. Source : Instana

04
Une ingénierie performante

sont bons, et le système a un comportement conforme aux plages attendues. Est-ce la fin de l'histoire ? Un déploiement qui ne rencontrera plus jamais aucune embûche ?

Malheureusement, nous savons que c'est rarement ce qui arrive dans les environnements de production. Il y a toujours quelque chose sur le point d'arriver, tout le temps. Soyez prêt. Même si nous sommes infiniment moins souvent obligés de jouer les pompiers pour éteindre les incendies, nous serons appelés à le refaire. N'oubliez pas qu'il est tout bonnement impossible de créer un système sans failles.

Il est vrai qu'à ce stade du processus, l'équipe d'ingénierie est moins impliquée directement en cas d'incident. Mais, pour le souligner une fois de plus, lorsque l'on nous demande d'intervenir, une présentation d'ensemble intelligente de tous les composants, réseaux, machines et applications concernés et de leurs interconnexions nous aide à nous plonger rapidement dans le vif du sujet. C'est une tâche qui devient de plus en plus complexe à chaque ajout d'abstraction ou de complexité architecturale au système. Ce qui nous simplifie et nous facilite la vie pendant le développement accentue la complexité lors de l'analyse partielle de systèmes inconnus.

Il n'y a qu'une seule métrique en fonction de laquelle nous devons nous évaluer, surtout lorsque nous sommes d'astreinte : le TMPRSC - le temps moyen pour retourner se coucher. Merci à Karthik Kumar qui nous a communiqué cette métrique très importante.

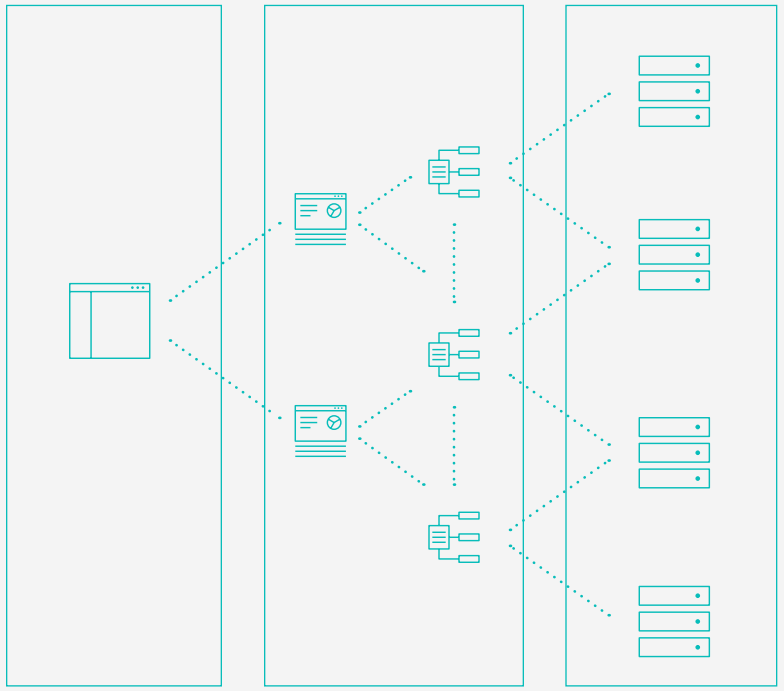


Figure 6 La complexité architecturale empêche parfois de raisonner rapidement

05 Développement basé sur l'observabilité

Maintenant que nous avons compris l'importance de l'observabilité, penchons-nous sur les atouts du développement axé sur l'observabilité (ODD). Récapitulons rapidement ses éléments les plus importants.

Le développement axé sur l'observabilité est une extension directe du développement axé sur le comportement. Il étend les tests au comportement d'un composant, en fixant des critères de performance et de santé. Les tests doivent être mis en œuvre tôt et exécutés pendant tout le cycle de développement. Ces tests permettent de connaître les performances avant, pendant et après le développement d'une fonction. Ils permettent de comparer les attentes initiales avec la réalité et fournissent des informations si le système de production nécessite une mise à l'échelle avant le déploiement final de la fonction.

Construisez le cycle de développement autour de l'idée de boucles de feedback courtes. Le véritable secret est de placer l'observabilité au cœur du processus de développement. Faites-en une approche proactive et travaillez en collaboration avec les autres équipes. Ne constituez pas de silos.

Au cours de ces interactions, proposez des métriques de santé et commencez à les implémenter dès le début de la mise en œuvre de la fonctionnalité. N'oubliez pas de tenir compte du nombre de métriques. L'idéal est de sélectionner 3 à 5 métriques, et au maximum 10.

Si la solution d'observabilité nécessite une intégration manuelle à l'aide d'encapsuleurs, ajoutez-les également dès le départ. Si vous pouvez mettre en œuvre une instrumentation automatique de notre service, c'est encore mieux. Notre code - notre "art" - reste intact.

En ce qui concerne les optimisations, n'essayez pas de jouer aux devinettes. Pour sélectionner les points d'optimisation, prenez en compte toutes les données fournies par la solution d'observabilité, par les métriques, par les traces réparties et par les informations sur l'infrastructure. Trouvez les facteurs qui contribuent le plus au temps d'attente et aux taux d'erreur et corrigez-les en premier. Commencez par ce qui est le plus facile, car cela permet d'obtenir rapidement de bons résultats et de se motiver.

Jetez également un œil sur les données de profilage du code. Un profileur toujours actif, conçu pour la production, offre un niveau de compréhension presque inimaginable auparavant. Jusqu'à il n'y a pas si longtemps, je n'ai jamais eu l'occasion d'avoir sur mon ordinateur un profileur qui donne une image proche de la production. Attention, cependant : n'utilisez pas n'importe quel profileur. Les profileurs prêts pour la production sont conçus sur mesure pour leur scénario d'utilisation respectif, avec un temps système extrêmement bas.

Le dernier conseil risque de vous faire mal rien qu'à le lire. Testez... suspense, suspense... en production. Oui, vous avez bien lu. Testez en production, mais avec des indicateurs de fonction. Activez le nouveau comportement, la nouvelle fonction, sur la base d'un ensemble d'ID utilisateurs, de paramètres spéciaux, et soyez créatif.

Le raisonnement qui sous-tend cette étape, sans compter l'abandon total de la vieille règle qui était de ne jamais tester en production, est que chaque environnement est unique. L'état de l'infrastructure, les dépendances, la date et l'heure, le déploiement, l'environnement lui-même, voire la phase de la lune... tout cela peut affecter notre code.

Alors, après toute cette discussion sur le développement et les raisons pour lesquelles l'observabilité facilite la vie du développeur, comment pouvons-nous mettre en application l'observabilité ? Tout d'abord, demandez à l'équipe DevOps si elle dispose déjà d'une solution utilisée par une autre équipe.

06
Être ou ne pas être... open source

Comme c'est le cas pour quasiment tout ce qu'elle fait, le communauté open source a fait un superbe travail en proposant de nombreuses options. Et les options sont une bonne chose, pas vrai ?

Oui et non. L'un des principaux obstacles posés par la plupart des solutions est qu'elles effectuent soit une surveillance des métriques, soit un traçage réparti, mais pas les deux. Le principal problème est que nous perdons la corrélation entre les informations. Les outils logiciels open source comportent des tableaux de bord indépendants, des silos de données indépendants et, le plus souvent, ne permettent pas d'alterner entre les métriques, les traces et l'infrastructure pour traiter une requête individuelle.

Créer l'observabilité manuellement

Pour créer l'observabilité, il est courant d'utiliser deux outils distincts : comme on l'a vu, un outil pour les métriques et la partie surveillance (en général Prometheus), et un outil pour le traçage réparti (en général Jaeger ou Zipkin).

Ces solutions open source sont "gratuites" - je vous expliquerai plus loin pourquoi j'ai mis des guillemets - et peuvent être intégrées à un service d'application. L'intégration, dans ce cas, signifie que nous devons ajouter le traceur et le collecteur de métriques à notre code source. Il existe des encapsuleurs pour les bibliothèques les plus courantes dans de nombreux langages de programmation. Ils peuvent être utilisés directement.

Un exemple simple d'une métrique Prometheus mesurant le nombre de requêtes par seconde se présente comme le fragment suivant, à l'aide d'une instance de compteur qui compte le nombre de requêtes. Voir la Figure 7.

Pas trop mal pour une utilisation directe. Heureusement, de nombreuses structures fournissent ce type de métriques en utilisant directement Prometheus.

L'autre solution consiste à utiliser Prometheus pour mesurer l'utilisation des ressources, telles que les connexions de la base de données ou les environnements d'exécution de requêtes. Il existe de nombreuses intégrations proposées par la communauté Prometheus. Dans l'univers Java, la célèbre solution Hibernate de mappage objet/relationnel (ORM) peut être utilisée avec juste un peu de code. Voir la Figure 8.

L'intégration de Prometheus gère à notre place tous les aspects ingrats de l'implémentation.

Côté traçage réparti, nous avons Zipkin ou Jaeger. Avec ces deux solutions, nous pouvons ajouter le traçage réparti à notre application. Ici encore, l'intégration n'est pas trop compliquée,

```
public class FooHandler {
  Counter counter = Counter
      .build()
      .namespace("my-app")
      .name("foo-handler")
      .help("number of requests")
      .register();

public ResponseEntity handler(Request request) {
  counter.inc(1);
  // do some business thing here
  }
}
```

Figure 7 Exemple d'une métrique Prometheus

```
new HibernateStatisticsCollector()
   .add(sessionFactory, "my-app")
   .register();
```

Figure 8 Exemple d'Hibernate avec Prometheus

06
Être ou ne pas être… open source

comme le montre le fragment suivant basé sur notre FooHandler existant. Voir la Figure 9.

Ce n'est pas très compliqué, mais nous avons un peu l'impression d'être revenus en arrière dans le temps. Le pire est que nous avons encombré notre code en faisant la part plus belle à la surveillance et au traçage qu'à la logique applicative elle-même.

Interprétation des données
En plus des données de journal, qui sont stockées dans un outil genre Logstash ou Splunk, il reste une grande quantité de données à explorer lorsqu'un problème se produit. Il existe cependant un autre problème : la pile d'observabilité elle-même.

Avec 3 systèmes indépendants qui ne partagent aucune donnée ou corrélation de données, l'interprétation des différentes sources de données est compliquée et demande du temps. Faire correspondre les horodatages, rechercher les corrélations, établir la connexion et finalement créer le contexte pour résoudre l'énigme en question, tout cela revient à l'utilisateur.

Gardez à l'esprit que l'interprétation de tout cela est le plus grand problème posé par les outils d'observabilité open source. Nous avons maintenant plus de systèmes individuels et plus de données, mais sans le contexte agrégé dont nous avons besoin.

```
public class FooHandler {
  Counter counter = Counter
        .build()
        .namespace("my-app")
        .name("foo-handler")
        .help("number of requests")
        .register();

private JaegerTracer tracer;

public FooHandler() {
  Configuration.SamplerConfiguration samplerConfig =
   Configuration.SamplerConfiguration
    .fromEnv()
    .withType("const")
    .withParam(1);

  Configuration.ReporterConfiguration reporterConfig =
   Configuration.ReporterConfiguration
    .fromEnv()
    .withLogSpans(true);

  Configuration config = new Configuration()
   .withSampler(samplerConfig)
   .withReporter(reporterConfig);

   this.tracer = config.getTracer();
  }

public ResponseEntity handler(Request request) {
   Span span = tracer.buildSpan("foo handler").start()
   try {
    counter.inc(1);
     // do some business thing here
   } catch (Exception e) {
     span.setTag("http.status_code", 500);
   } finally {
     span.finish();
   }
  }
}
```

Figure 9 Fragment de traçage Java avec Jaeger

06

Être ou ne pas être... open source

Le coût réel de l'open source
Vous vous souvenez que j'ai dit plus haut que les solutions étaient "gratuites" et que nous en reparlerions ? Nous y voici.

Contrairement à la croyance commune, l'open source n'est pas gratuit. Il est vrai que son utilisation n'est pas assujettie à une licence payante. Le coût principal de l'observabilité open source se mesure en temps : le temps que les entreprises doivent payer pour que leurs équipes comprennent comment faire fonctionner la pile d'observabilité, comment l'utiliser et comment interpréter les données. Ce temps, qui est souvent notre bien le plus rare, est soustrait aux tâches de livraison de la fonctionnalité métier.

Il peut également être nécessaire de payer des ingénieurs supplémentaires qui devront ajuster les solutions d'observabilité open source. N'oublions pas non plus le temps nécessaire pour intégrer la surveillance et le traçage réparti. Ces tâches représentent toutes des coûts supplémentaires en plus du temps nécessaire pour implémenter l'observabilité open source à la base source.

L'alignement de ces intégrations avec nos objectifs déjà définis peut facilement ajouter de 5 % à 10 % au temps de développement - un temps que nous tentions en fait d'éviter à la fin de l'époque de Zabbix et de Nagios.

Si l'on regarde le cas des équipes DevOps et des opérations, ce temps ne fait qu'augmenter, car plusieurs systèmes sont concernés, mais c'est une autre histoire. Nous nous limiterons à notre histoire du développement.

Dernier point, mais non le moindre, l'exploitation de la pile, l'espace de stockage des données, le temps de traitement nécessaire à la nouvelle implémentation des corrélations automatiques, même très simples, tout cela a un coût.

Subitement, ces outils open source "gratuits" sont devenus très coûteux, et accusent de plus encore des lacunes sur le plan de l'observabilité et du contexte.

La solution des fournisseurs de logiciels
Les solutions du commerce, quant à elles, offrent un grand nombre, sinon la totalité, des fonctions mentionnées, en les regroupant dans une solution unique. Si certaines parties ne sont pas fournies directement, elles intègrent des services externes de façon à donner l'impression d'un stockage interne. La corrélation des données des systèmes internes et externes est incluse.

La corrélation des données entre toutes les parties de nos systèmes est cependant l'élément le plus important. Infrastructure, applications, services, réseau, journalisation... toutes les données sont préalablement traitées de manière à ce que les informations nécessaires soient faciles à collecter et à interpréter.

Certains fournisseurs vont plus loin en corrélant les problèmes détectés dans différentes parties du système pour créer des alertes plus exploitables et plus ciblées. La mise en corrélation de tous les problèmes ou événements permet de comprendre encore plus rapidement l'étendue et la cause d'un incident, les services connexes et de savoir s'il y a un impact éventuel sur l'utilisateur.

En outre, certains fournisseurs offrent des informations immédiates sur les changements avant et après les versions, comme le montre la Figure 5. L'obtention de ces informations requiert une intégration aux pipelines CI/CD courants, aux services DevOps et aux outils de développement.

06 Être ou ne pas être… open source

Au-delà de l'open source
Il n'en reste pas moins qu'en tant que développeurs, ce qui nous intéresse est le travail de développement. Au lieu de tout paramétrer manuellement, certaines solutions du commerce proposent une instrumentation de code entièrement automatique.

Vous vous souvenez de notre exemple de code de tout à l'heure, et vous vous demandez à quoi il ressemblerait lorsque l'on utilise une instrumentation de code entièrement automatisée ?

Vous le voyez par vous-même : aucune modification du code n'est nécessaire pour ajouter la surveillance, le traçage réparti ou la collecte des performances. L'automatisation récupère les services, peu importe qu'ils soient démarrés directement ou non, dans un conteneur Docker ou dans Kubernetes, sur la plateforme Red Hat® OpenShift® ou dans des environnements similaires. Après l'avoir découvert automatiquement, le service est instrumenté instantanément, des tableaux de bord sont générés en fonction des meilleures pratiques connues pour le service ou la structure, et le tour est joué. Aucune intervention manuelle n'est nécessaire. Que demander de plus ?

L'avantage de la corrélation automatique
Comme on l'a vu tout à l'heure, le principal avantage d'une solution d'observabilité d'entreprise est sa capacité à corréler les métriques et les traces des machines, de l'infrastructure, des applications et des services. Les traces réparties permettent de comprendre le flux de la requête, tandis que les métriques fournissent les points de performance nécessaires.

La corrélation manuelle est toutefois assez lourde. Le raison principale pour laquelle les utilisateurs n'aiment pas avoir plusieurs tableaux de bord pour différents services est qu'il est presque impossible de faire correspondre toutes les séquences et de collecter le contexte général du problème.

Le plus grand avantage de la corrélation automatique, comme on l'a expliqué, est l'obtention immédiate des informations. Lors nous examinons un problème ou un incident, la solution du fournisseur a déjà fait tout le travail d'enquête pour nous fournir les éléments d'information importants, tels que les preuves contextuelles, et elle nous oriente tout droit sur les aspects intéressants.

```
public class FooHandler {
   public ResponseEntity handler(Request
request) {
      // do some business thing here
   }
}
```

Figure 10. Code utilisant une instrumentation entièrement automatisée avec Instana AutoTrace

07

Se simplifier la vie

Le coût des solutions des fournisseurs
Contrairement aux outils open source, les solutions des fournisseurs ne revendiquent pas la gratuité.

Avec l'open source, nous devons faire fonctionner le système et payer le stockage de données et le traitement. En revanche, les solutions des fournisseurs offrent tous les outils nécessaires pour fournir une observabilité d'entreprise complète sous la forme d'un environnement SaaS hébergé, ou un environnement sur site lorsque ce critère est requis.

En éliminant le coût de l'intégration de la surveillance et du traçage aux services et à l'infrastructure, les solutions des fournisseurs sont en définitive souvent plus abordables, avec des coûts d'exploitation et de développement inférieurs. Par ailleurs, n'oublions pas le coût du stockage des informations et des données.

Une prise de décision éclairée
Le choix entre l'open source ou une solution du commerce est une question de temps - le temps que nous voulons consacrer au développement d'une fonctionnalité non liée à l'activité métier, et non une question de coût, car aucune des deux solutions n'est gratuite.

Jusqu'à présent, nous nous sommes surtout intéressés aux développeurs qui n'aiment pas ajouter des éléments de surveillance et de traçage dans leur code. Ils forment certainement le groupe le plus important. Il n'empêche que certains développeurs aiment bricoler. Pour ma part, je me considère comme quelqu'un qui a hérité du gène du bricolage. Je préfère cependant jouer avec les nouvelles technologies et pas avec une quelconque métrique de performance.

En dehors de cela, mon expérience me dit que nous oublions les collectes des données les plus importantes lorsque nous les faisons manuellement, et cela plus d'une fois. Ou bien nous retombons dans la vieille habitude qui consiste à ajouter tout ce qui est possible au lieu de nous concentrer sur nos domaines de défaillance déjà définis.

Les solutions des fournisseurs offrent un ensemble toujours plus vaste de meilleures pratiques pour la collecte automatisée des métriques et la compréhension des langages de programmation, des structures et des intégrations de services. Ces meilleures pratiques éliminent la majorité des tâches lors de la définition des domaines de défaillance potentiels dans les systèmes nouveaux ou inconnus. Avec les solutions des fournisseurs, il est rare de passer de nouveau à côté d'informations importantes.

07

Se simplifier la vie

La création de logiciels d'aujourd'hui diffère considérablement des méthodes traditionnelles des décennies précédentes. Les applications sont décomposées en petits microservices, tandis que les déploiements sont basés sur Docker, Kubernetes et les pipelines automatiques CI/CD.

Le suivi de tous ces changements pendant les opérations et l'analyse des problèmes est marqué par une complexité croissante. Il est important de trouver de nouvelles méthodes de collecte de l'information sur les événements, surtout lorsque nous devons intervenir et que nous essayons de comprendre un problème posé par un composant inconnu.

L'observabilité d'entreprise est la clé. Choisissez le bon combat : ne luttez pas contre votre solution d'observabilité.

© Copyright 2021 Instana, une société d'IBM

Compagnie IBM France
17 avenue de l'Europe
92275 Bois-Colombes Cedex

Produit aux États-Unis d'Amérique
Avril 2021

IBM, le logo IBM et ibm.com sont des marques d'IBM Corporation, enregistrées auprès de nombreuses juridictions dans le monde. Les autres noms de services et de produits peuvent être des marques d'IBM ou d'autres sociétés. Une liste actualisée des marques d'IBM est disponible sur ibm.com/trademark.

Instana est une marque déposée d'Instana, Inc, une société d'IBM.

Red Hat et OpenShift sont des marques de Red Hat, Inc. ou de ses filiales aux États-Unis et dans certains autres pays.

L'information contenue dans ce document était à jour à la date de sa publication initiale, et peut être modifiée sans préavis par IBM. Les offres mentionnées dans le présent document ne sont pas toutes disponibles dans tous les pays où IBM est présent.

LES INFORMATIONS CONTENUES DANS LE PRÉSENT DOCUMENT SONT FOURNIES "EN L'ÉTAT", SANS AUCUNE GARANTIE EXPLICITE OU IMPLICITE. IBM DÉCLINE NOTAMMENT TOUTE RESPONSABILITÉ RELATIVE À CES INFORMATIONS EN CAS DE CONTREFAÇON AINSI QU'EN CAS DE DÉFAUT D'APTITUDE À L'EXÉCUTION D'UN TRAVAIL DONNÉ. Les produits IBM sont garantis conformément aux dispositions des contrats.