



daveo

#Cloud

Du monolithe au Serverless

Focus sur l'un des services majeurs
d'Amazon Web Services : AWS Lambda

Sommaire

Préface

Introduction

- 1 - Le Serverless quèsaco ?
- 2 - Comment penser l'architecture ?
- 3 - Je développe & je construis mon usine de développement.
- 4 - Atteindre la maturité technologique
- 5 - Envie d'aller encore plus loin ?

Conclusion

Culture Daveo

Remerciements

Préface

Dès 2006, de nombreuses start-up ont pu se développer grâce au cloud. **L'accès à la demande en quelques minutes, par n'importe quelle équipe au meilleur de la technologie était une révolution.** Le modèle mental du cloud était « qu'un étudiant dans sa chambre pourrait avoir accès aux mêmes technologies d'infrastructure que les grandes entreprises ».

En 2014, Amazon Web Services lance le service AWS Lambda, qui devient la brique centrale de ce qu'on appellera peu après le Serverless. **Cette nouvelle approche permet de démocratiser davantage l'accès à la technologie et rend le développement et le déploiement d'applications encore plus rapide**, en affranchissant les créateurs d'applications de tâches complexes d'élasticité, de sécurisation et de haute disponibilité. Andy Jassy, CEO d'Amazon Web Services, a dévoilé en décembre 2020 que **plus de la moitié des nouvelles applications créées sur AWS étaient faites en Serverless.**

Là où une architecture basée sur des instances dans le cloud permet déjà de **réduire la responsabilité opérationnelle** par rapport à une application tournant en datacenters, une architecture Serverless constitue l'étape suivante logique, tirant parti autant que possible du savoir-faire et de l'amélioration continue des technologies AWS pour construire des **applications cloud natives.**

Le but d'une stratégie *Serverless-First* est de suivre cette tendance en priorisant l'adoption de technologies qui abstraient et prennent en charge le management de l'infrastructure sous-jacente, permettant d'être plus agile et de **se concentrer sur l'innovation et le développement de nouvelles applications** et fonctionnalités.

Ces technologies s'occupent de la montée en charge de manière automatique pour répondre à la demande des mises à jour, ainsi que du **remplacement des ressources défectueuses sans nécessiter d'intervention manuelle.**

La possibilité d'**utiliser les services Serverless permet également de faire gagner un temps précieux aux développeurs** : sur un panel de onze entreprises utilisant ces technologies, **le temps moyen de réalisation d'une fonctionnalité passe à seulement 9 jours en Serverless alors qu'il est de 5 semaines sur une solution traditionnelle.**

Au-delà de la vitesse de développement, il n'est pas rare de voir des clients réduire de plus de 80% leurs factures AWS à l'occasion d'une migration en Serverless. C'est autant de marge de manœuvre libérée pour investir dans des nouveaux projets.

Le Serverless est également une bonne manière de réduire son impact environnemental : réduire de 80% sa facture signifie réduire son impact environnemental d'un **facteur similaire.**

Grâce à cela, les équipes techniques peuvent **mettre en place des prototypes plus rapidement**, et donc multiplier les expérimentations pour valider la faisabilité et l'utilité de nouvelles fonctionnalités, plutôt que de rester dans la théorie par manque de ressources. Pour les entreprises mentionnées précédemment, cela se concrétise par une augmentation de 115% du nombre de fonctionnalités mises en production par an, tout en bénéficiant d'une réduction de 71% d'interruptions non planifiées en s'appuyant sur **l'expertise d'AWS.**

Dans un monde en constante évolution, **ce gain en flexibilité aide à répondre plus rapidement aux changements des besoins du marché et des utilisateurs, donnant un avantage compétitif significatif par rapport à l'utilisation de technologies plus traditionnelles.**

À travers le monde, ce sont des centaines de milliers de clients AWS qui utilisent d'ores et déjà les **technologies Serverless**. En 2019, 19% des entreprises les utilisent, et 42% supplémentaires s'attendent à faire de même **d'ici 2021** pour bénéficier de ses avantages et se concentrer sur comment répondre aux besoins de leurs clients de la meilleure façon possible.

Nous partageons cette vision avec Daveo, membre du réseau de partenaires AWS APN et élu « APN Rising Star Partner of the Year » en 2020. **Leurs équipes font preuve d'une accélération de la montée en compétences et de la qualité des projets délivrés sur le cloud** AWS. C'est donc avec enthousiasme que nous avons préfacé **ce livre blanc qui livre de nombreux conseils pour réussir son passage au Serverless.**

Nicolas Moutschen

Specialist Solutions Architect

Serverless at Amazon Web Services (AWS)

Introduction

Depuis la fameuse bulle internet de la fin des années 90, nous avons connu une croissance exponentielle de notre consommation de l'outil informatique. Informatique qui a profondément changé nos comportements et notre quotidien, et qui est également en perpétuelle évolution : intelligence artificielle, informatique quantique, blockchain...

Difficile de prédire l'après Covid-19 et les comportements que nous aurons, mais une chose est sûre, l'accélération forcée dans l'usage des nouveaux outils IT et les nouvelles méthodes qui permettent un travail distribué de qualité marqueront durablement les pratiques. L'usage massif des outils comme Google Meet ou Microsoft Teams n'est que la partie émergée d'un changement de fond majeur, d'une nouvelle ère de l'informatique.

Si cette informatique a déjà connu bien des virages importants, celui qui est en cours sera bien moins visible pour le commun des mortels que celui de l'ère ubiquitaire, l'ère de la mobilité. Nous parlons ici du virage du cloud. Les data centers « on-premises » (privés, opérés par les entreprises en direct) devraient se raréfier dans la prochaine décennie. Pourquoi ? Quelques raisons à cela : les grands fournisseurs de cloud tels qu'Amazon, Microsoft, Google offrent des services, des performances, une résilience bien supérieure à ce qu'une entreprise lambda peut mettre en œuvre dans un data center privé. La mutualisation des ressources permet aussi de réelles économies sur les coûts des infrastructures ainsi que de leur gestion. D'autre part, cette mutualisation des ressources, couplée à des data centers plus écologiques, nous permettra de tendre vers une informatique moins gourmande, plus respectueuse de l'environnement, tout en étant capable de continuer à supporter les besoins toujours grandissants de l'humanité.

Introduction

Ce virage du cloud, c'est aussi l'opportunité pour les entreprises de diminuer drastiquement et sur le long terme le traitement subi de l'obsolescence des systèmes, imposé par des fins de supports, des machines en fin de vie, des failles de sécurité... Ce traitement que nous devons inclure dans la roadmap des projets, en bataillant souvent pour justifier son coût, est comme un poids mort n'apportant aucune valeur. Déployer nos applicatifs sur le cloud, c'est donc choisir le niveau d'obsolescence que nous aurons à traiter demain. Plus il sera faible, plus le temps accordé au développement des applicatifs, créateur de valeur pour l'entreprise, sera important.

Ces changements fondamentaux dans la gestion des infrastructures informatiques ne sont pas sans impact sur l'Homme. Des métiers se créent, se transforment, tandis que d'autres disparaissent. Adresser ce virage cloud dans les règles de l'art ne se fait pas de façon aisée. Il n'est pas rare dans une entreprise d'avoir un amoureux de son datacenter on-premises, tel un conducteur de train à vapeur voyant arriver avec méfiance l'électrification des lignes en se disant qu'on rigolera moins quand les câbles seront cassés. **Réussir une migration vers le cloud est un challenge aussi bien technique qu'humain !**

Ce livre blanc traite du bon usage d'un des types de composant du cloud, celui qui apporte le plus haut niveau d'abstraction de l'infrastructure, celui qui génère le moins d'obsolescence et celui qui est le plus éco-responsable : le Serverless, et plus précisément le composant AWS Lambda !



1

Le Serverless, quèsaco ?

Overview

Retrouvez un résumé avec
les informations vulgarisées,
en début de chaque chapitre !

Le Serverless quèsaco ?

Le mot Serverless a fait couler beaucoup d'encre depuis sa première apparition dans un article de **Ken Fromm** en 2012, puisque son étymologie peut laisser penser, tout au moins pour les non-initiés, que nous serions en présence d'une informatique sans serveur. Il n'en est rien !

Il reste difficile de synthétiser le terme Serverless, nous vous livrons notre définition :

"Se dit d'une technologie informatique affranchissant l'utilisateur de la gestion des serveurs, et qui ne consomme que les ressources strictement nécessaires à son fonctionnement."

1. Le Serverless quèsaco ?

Une certaine controverse

En pratique, les applications Serverless sont des services exposés par des plateformes cloud qui répondent aux critères suivants :

- > ne nécessitent aucune gestion des couches qui hébergent le service (serveurs, conteneurs, réseaux, processus du système d'exploitation...),
- > s'adaptent automatiquement à la charge/au niveau de ressources nécessaires,
- > offrent une facturation proportionnelle à l'usage du service,
- > permettent implicitement une haute disponibilité.

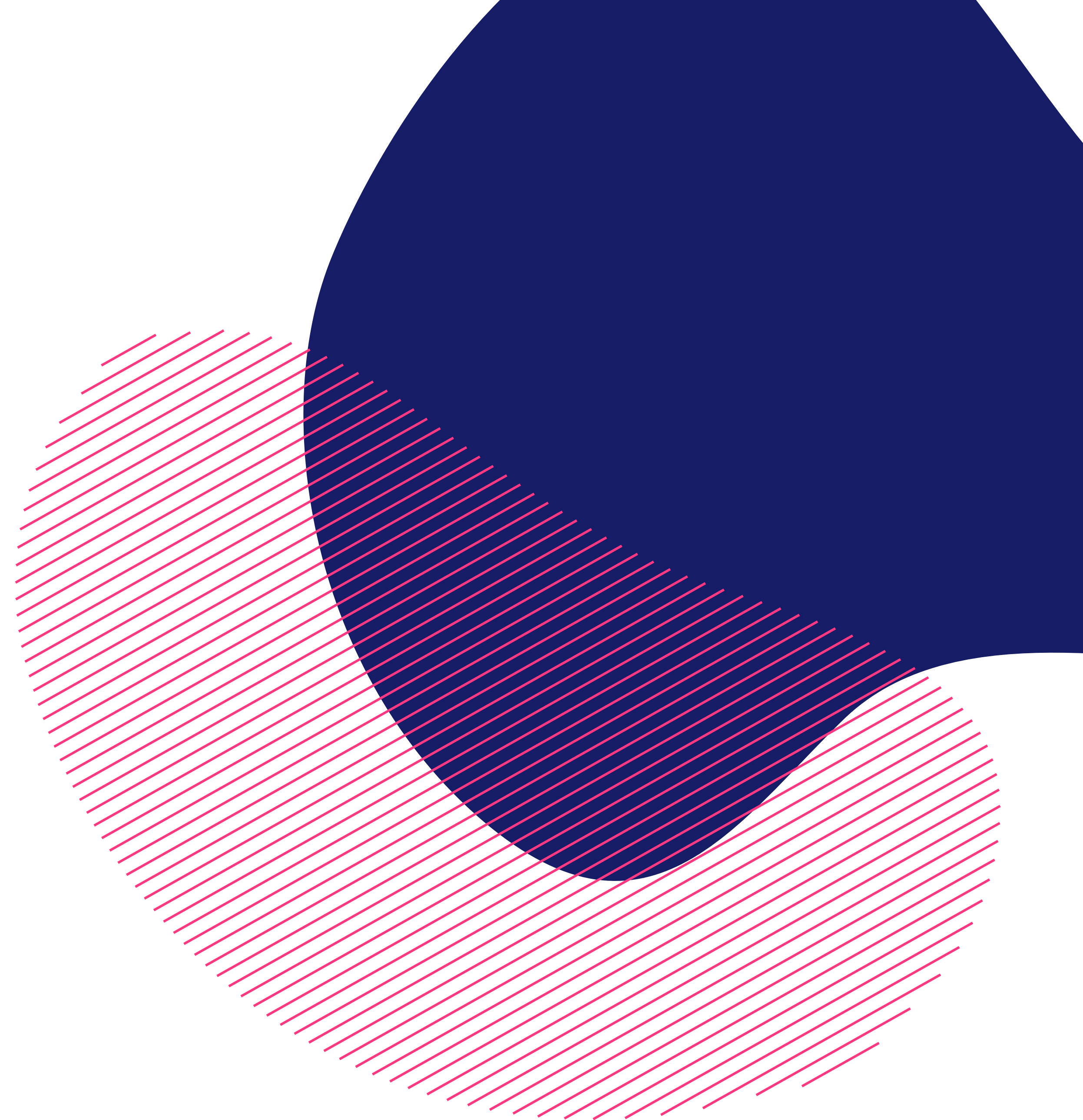


1. Le Serverless quèsaco ?

L'objectif d'un service Serverless est donc de libérer le client de la gestion des couches IT plus basses afin de lui permettre de se concentrer sur ses problématiques business, le tout en lui facturant uniquement l'usage précis des ressources utilisées.

Les services que nous pourrions qualifier de Serverless sont par exemple :

- un service en mode SaaS (Software as a Service) que l'on consomme et pour lequel nous sommes facturés au nombre d'appels,
- les services de bases de données non-relationnelles type DynamoDB (AWS), CosmosDB (Azure), Google Cloud Datastore (GCP). Au delta près du coût du stockage des données, qui entraîne une facturation même si il n'y a aucune activité sur la base de données,
- les services d'exposition de fonction type AWS Lambda, Google Cloud Functions, Azure Functions.



1. Le Serverless quèsaco ?

Une abstraction maximale

Le service "star" estampillé Serverless est actuellement le service d'exposition de fonction, appelé aussi FaaS comme **Function as a Service**. Ce livre blanc mettra d'ailleurs le focus sur ce service, notamment dans sa version amazonienne : **AWS Lambda**.

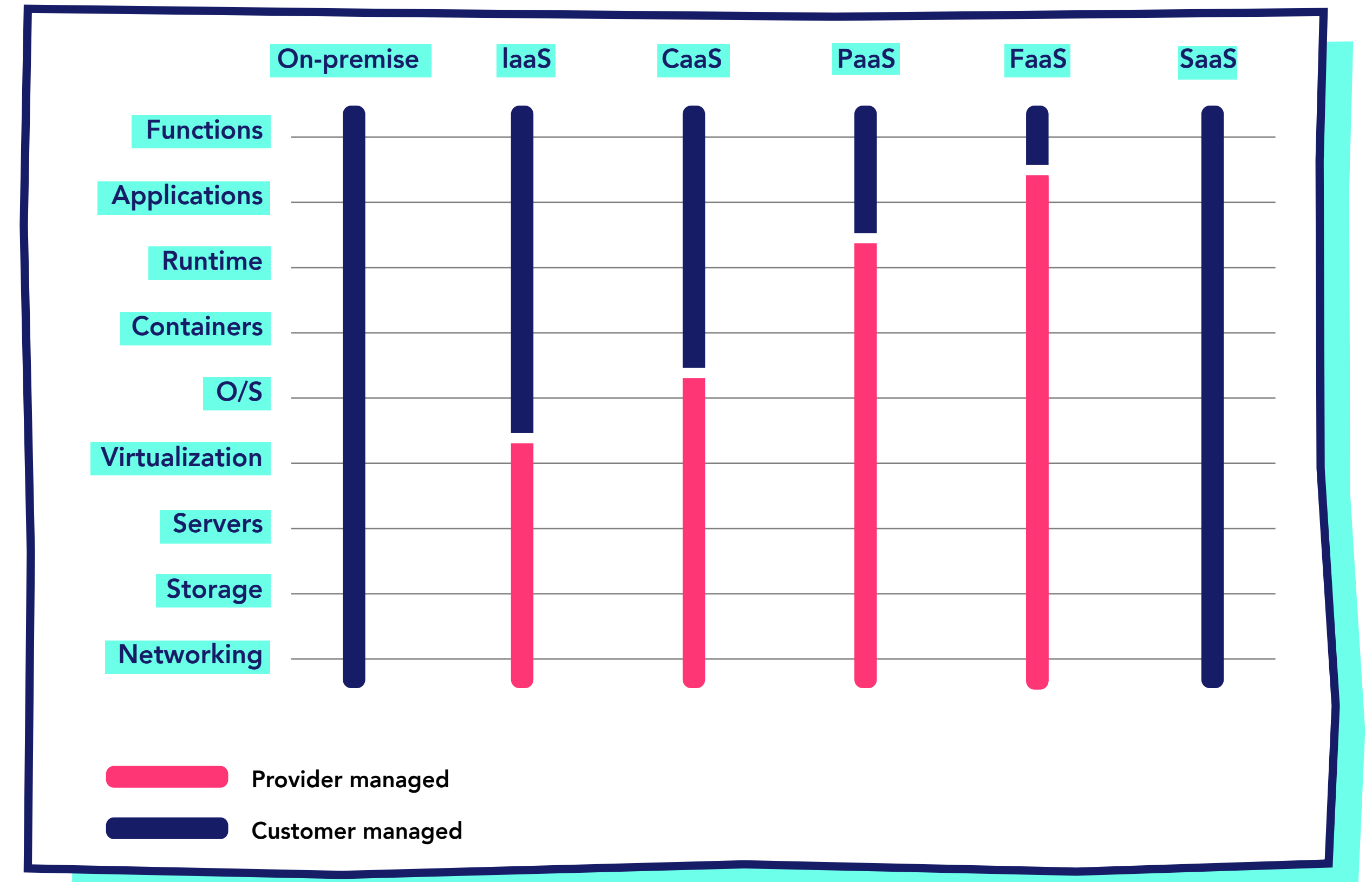
Pour bien comprendre ce que le terme FaaS représente, un parallèle simple peut être celui de l'alimentation électrique de son habitation.

À aucun moment nous nous soucions de la manière dont l'énergie est produite, ni véhiculée. Nous ne nous soucions pas non plus de savoir si l'énergie reçue le sera en quantité suffisante. Et nous sommes bien facturés par rapport à notre consommation.

Nous appelons simplement la fonction "fournis moi de l'électricité" exposée par son fournisseur d'énergie.

À l'opposé, un panneau solaire sur sa toiture ou une éolienne dans son jardin équivaldrait à la gestion d'un serveur dans un data center on-premises.

Le schéma ci-contre présente le positionnement du FaaS dans la grande famille des **aaS** (as a service).



Le FaaS se situe entre le PaaS (Platform as a Service) et le SaaS (Software as a Service), il ne reste à sa charge que le développement du code applicatif de sa fonction. La concentration se porte donc uniquement sur du code métier en laissant au cloud provider le soin de gérer les couches les plus basses.

Si la notion de SaaS est simple à comprendre, la différence entre les technologies PaaS et FaaS est plus subtile.

1. Le Serverless quèsaco ?

PaaS vs FaaS

Le PaaS présente beaucoup de similitudes avec le FaaS.

- ① **Les services PaaS participent à l'abstraction des couches basses** (serveur, OS...) au même titre que le FaaS.
- ② Ils bénéficient pour une majorité d'entre-eux d'une **mise à l'échelle automatique en fonction des besoins**, au même titre que le FaaS.

Des différences apparaissent d'une part sur les aspects de haute disponibilité et d'autre part sur les aspects financiers.

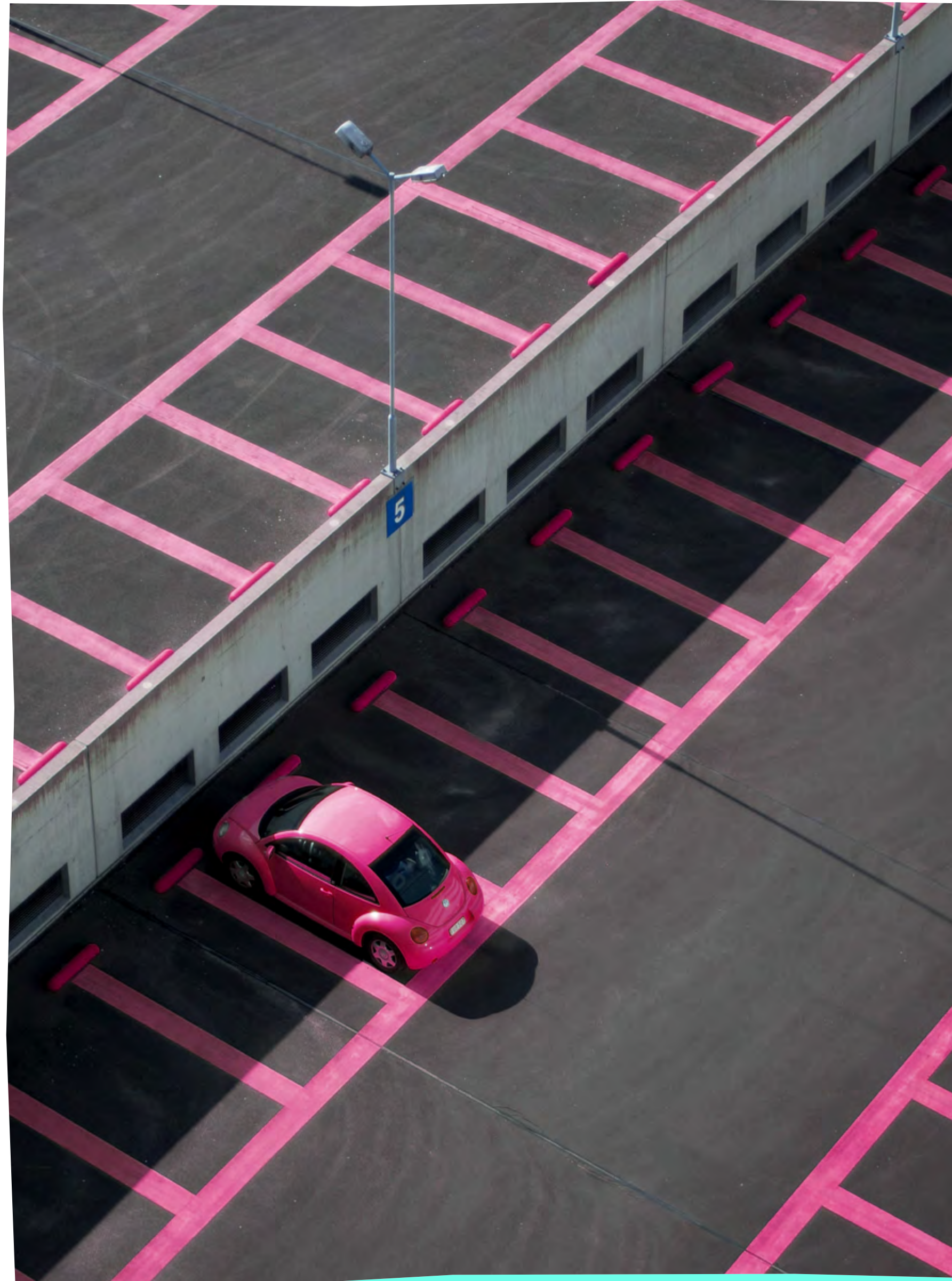
- ① **Si pour le FaaS, la haute disponibilité** est implicite et gérée par le cloud provider, pour le PaaS, si elle est disponible, elle doit être gérée.

Un exemple concret serait l'utilisation d'une base de données Amazon Relational Database Service (Amazon RDS) qui est bien un objet PaaS et pour lequel en fonction des SLAs attendus (service-level agreement), il est nécessaire ou non de mettre en place de la géo-réplication sur plusieurs régions différentes.

- ② **Du côté financier, sur les objets du PaaS**, même si un mode d'abonnement sans engagement ("Pay As You Go") semble dominer, il n'en reste pas moins que si l'objet instancié sur la plateforme cloud n'est pas utilisé, il sera tout de même facturé. Ce coût correspond aux ressources prises et réservées pour cet objet (mémoire, stockage...) sur l'infrastructure cloud.

Vu de cette façon, il est clair qu'**un raccourci pourrait être de ne plus vouloir produire que des architectures Serverless tant les avantages semblent évidents.** Nous verrons toutefois dans la suite de ce livre blanc que **le FaaS présente aussi certaines contraintes** (limitations techniques, une orchestration pouvant être plus complexe...) **qu'il conviendra de bien comprendre pour faire les bons choix.** Dit autrement, les architectures Serverless ne peuvent répondre à l'ensemble des problématiques.

1. Le Serverless quèsaco ?



Le Serverless c'est aussi du Green IT, ou encore être éco-responsable dans les usages des ressources cloud.

Lorsque nous utilisons des services cloud de type IaaS (Infrastructure as a Service), CaaS (Container as a Service), PaaS (Platform as a Service), le modèle de facturation est très souvent un mode de paiement à l'usage "Pay As You Go".

En simplifiant, le coût calculé est proportionnel à la durée durant laquelle l'objet est instancié sur la plateforme cloud. Ce coût correspond donc non seulement à l'usage des ressources nécessaires au fonctionnement du composant (RAM, CPU, disque, réseau...), mais aussi aux ressources immobilisées afin de garantir qu'il puisse fonctionner dans les limites hautes des performances promises.

Concrètement, lorsque l'on instancie un objet de type Amazon EC2 (Elastic Compute Cloud) de type t4g.large sur la région Paris d'AWS, on réserve 2 vCPU et 8 GB de RAM dans un data center qui seront facturés environ 62,05\$/mois, et ce même si nous n'utilisons que 10% des ressources CPU.

À l'échelle mondiale, en 2019, une estimation du coût des ressources de calcul gaspillées dans les data centers cloud serait de l'ordre de 17 milliards de dollars (fourchette basse, source [ParkMyCloud](#)).

1. Le Serverless qu'est-ce que ?

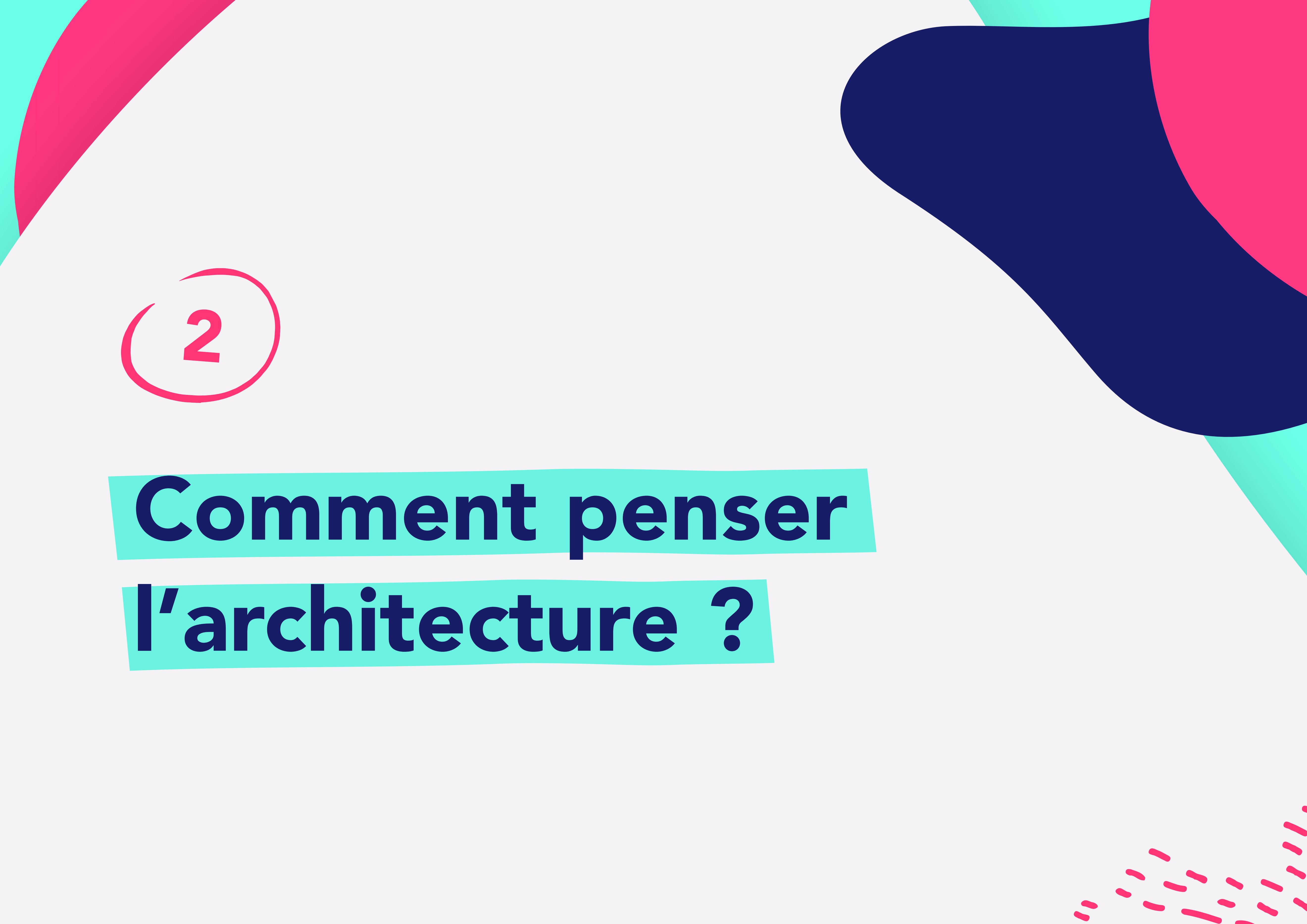
Le FaaS, et plus généralement les services Serverless, ne demandent quant à eux aucune réservation de ressources lorsqu'ils sont non utilisés. Au-delà de la réduction de coût de ces architectures, c'est bien aussi une réduction de l'utilisation des composants physiques et une baisse de la consommation énergétique.

Utiliser les technologies Serverless c'est aussi être éco-responsable !

Amazon, et de facto AWS, souhaite atteindre 100% d'énergie renouvelable avant 2025. L'objectif à terme étant de parvenir à éliminer toute empreinte carbone d'ici 2040. Pour y parvenir, des projets comme des parcs éoliens et solaires ou encore des systèmes réduisant la consommation d'eau dans les data centers voient le jour (référence : **ici**). Ce qui ouvre la voie à des data centers plus écologiques. Si certains grands cloud providers s'engagent de leur côté à une neutralité carbone (**Microsoft, AWS**) dans les années à venir, il nous appartient aussi en tant qu'individu d'être éco-responsable dans l'usage des ressources IT.

Dans un monde où il y a une prise de conscience sur les sujets environnementaux, où des postes touchant au domaine de la RSE sont de plus en plus nombreux dans les entreprises, et où des index de notation sont créés, exemple avec l'**Euronext Eurozone ESG 80**, la technologie Serverless pourrait être vue aussi comme la technologie verte du futur.

De manière plus globale, se tourner aujourd'hui vers le cloud, c'est avoir la garantie de profiter des évolutions "hardware" de la prochaine décennie. Les data centers de demain seront certes plus écologiques, mais offriront aussi des performances sans commune mesure avec ce que nous connaissons. **Dans un environnement concurrentiel, il est à craindre que les entreprises qui n'auront pas franchi ce cap dans les cinq à dix années à venir seront en grande difficulté.**



2

**Comment penser
l'architecture ?**

Overview

Comment penser l'architecture ?

Quel que soit le projet de cloudification (nouvelle application, réécriture d'une application existante, simple migration...), l'agilité et la flexibilité qu'apporte le cloud autorisent le droit à l'erreur.

Là où la construction d'une infrastructure on-premises faisait souvent l'objet d'investissements importants, de processus complexes et chronophages, il est aujourd'hui possible de créer, modifier, supprimer des éléments d'infrastructure très simplement, en quelques minutes, avec une incidence financière mineure en cas d'erreur.

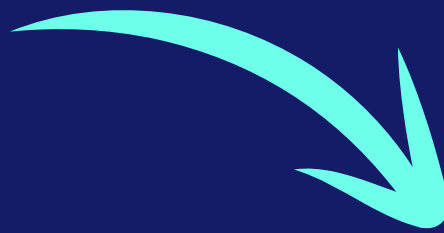
Les processus itératifs d'amélioration des produits issus d'une gestion agile des projets vont pouvoir s'appliquer aussi sur les éléments d'infrastructure.

2. Comment penser l'architecture ?

Dans ce livre blanc, nous nous penchons sur un cas de figure que bon nombre de sociétés désireuses de se tourner vers le cloud public rencontrent :

"Comment migrer une application monolithique hébergée on-premises vers le cloud ?" Mais aussi, "Peut-on se tourner dans notre cas vers une solution Serverless en connaissance des avantages que cela procure ?" (cf.chapitre 1).

Nous ne sommes pas ici dans un exercice de Lift & Shift (déplacement simple d'une application vers le cloud sans la retoucher) brutal d'un univers IaaS on-premises vers un autre univers IaaS d'un cloud provider, mais bien dans une **démarche de réécriture d'une application existante à destination d'une architecture Serverless.**



Tout au long de ce chapitre, nous prendrons en exemple un projet de réécriture/migration vers le cloud d'une application B2B d'une société d'assurance-crédit. Cette application permet aux partenaires et aux courtiers de suivre leurs contrats d'assurance et leurs commissions. Sur ce qui va suivre, nous allons nous focaliser sur la migration/réécriture des APIs de cette application.

2. Comment penser l'architecture ?

Étape 1, Identifier les services ?

Cette étape n'est pas propre au monde du cloud ni au Serverless, mais plutôt en lien direct avec la tendance des **architectures microservices**.

En synthèse, il s'agit de découper notre monolithe en x services faiblement couplés entre eux.

Ce type d'architecture offre la possibilité pour une équipe de travailler indépendamment sur un service sans impacter les autres, et sans être impactée par les autres services.

Cela permet d'améliorer la fréquence des mises à jour et le *time to market* des fonctionnalités à mettre en place. À chacun des services identifiés correspond généralement une ou n APIs exposée(s).

Sur notre projet, les premiers services identifiés ont été :

- > création d'un contrat d'assurance,
- > affichage des limites de remboursement,
- > affichage de la situation financière des clients et prospects,
- > calcul du taux de risque (client, région, pays),
- > affichage des commissions par partenaire,
- > suivi des échéances de paiement.

Étape 2, Comment passer d'un système 100% on-premises à un système 100% cloud ?

Deux options sont généralement possibles. Une migration progressive service par service versus une solution big-bang, où l'on switche brutalement entre l'ancien monde et le nouveau. Le choix de l'une ou l'autre solution dépend de multiples facteurs parmi lesquels : le niveau de maturité des équipes, la taille de l'application, les risques liés à l'obsolescence...

L'option de la migration progressive a toutefois plusieurs avantages... :

- > permettre de délivrer rapidement les premières briques du futur système, et donc rassurer les parties prenantes sur les choix qui sont faits,
- > prendre moins de risques qu'une solution big-bang.

Mais aussi une contrainte principale :

- > faire vivre les deux systèmes en parallèle durant toute la phase du projet, ce qui nécessite très souvent une mise à jour des données entre l'environnement on-premises et celui sur le cloud.

Cette option de migration est connue sous le nom de **Strangler pattern**.

Notre projet de migration et de réécriture concerne une centaine d'APIs. Ce nombre important nous a poussé à choisir l'option d'une migration progressive. Pour chaque API, le processus de migration commence par la réécriture de chacune d'elle tout en respectant les 5 piliers du **Well-Architected Framework** (excellence opérationnelle, sécurité, fiabilité, efficacité des performances et optimisation des coûts), passant par les tests fonctionnels sous Postman et les tests de charge sous Gatling avant la mise en production.

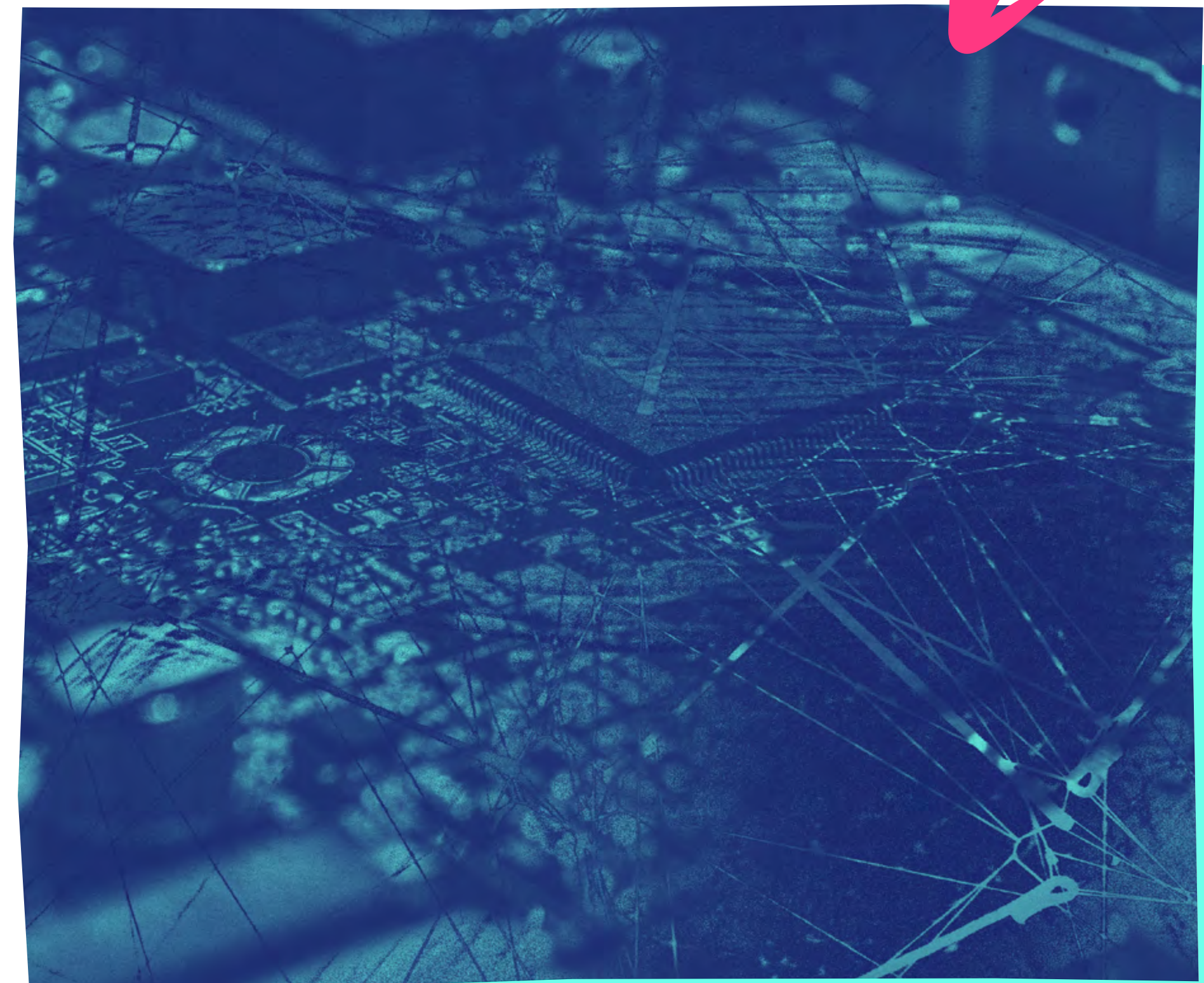
2. Comment penser l'architecture ?

Étape 3, Identifier les composants cloud en fonction des besoins ?

Le cloud apporte un changement majeur dans la façon dont nous construisons nos architectures techniques.

Si nous venons d'un monde "on-premises" où nos choix sont dictés par l'optimisation des ressources dont nous disposons, la construction des architectures cloud est quant à elle basée sur **l'optimisation des ressources utilisées** et donc plus directement basée sur les coûts des composants que nous souhaitons implémenter.

En imageant, si nous représentons une infrastructure on-premises comme l'achat d'un parking, et des voitures comme des applications, le challenge des architectes est bien d'arriver à optimiser le stationnement des véhicules en fonction de leurs gabarits.



2. Comment penser l'architecture ?

A contrario, les solutions cloud de type PaaS ou FaaS peuvent être vues comme de la location d'une place de parking. Le jeu étant de louer la juste dimension en fonction de ses besoins pour le côté PaaS tandis que pour le FaaS, la place s'adapte automatiquement aux besoins.

Dans la pratique, certains critères vont nous aider à choisir les composants à mettre en œuvre pour construire notre infrastructure. Pour exemple, concernant la couche d'exposition des APIs, les réponses aux questions suivantes peuvent orienter nos choix sur le type de composant à mettre en place (PaaS, FaaS) :

- > nombre d'appels des APIs par seconde ?,
- > quel SLA est attendu par le client ?,
- > quel est le niveau de scalabilité attendu ?,
- > quel est le RPO (Recovery Point Objective) attendu ?

Dans le cadre de notre projet :

- > La partie frontale est exposée en mode static website sur Amazon **S3**, un service Serverless qui permet d'héberger des sites web statiques à moindre coût.
- > La couche d'APIs est exposée via une **API Gateway** et des fonctions AWS Lambda.
- > La base de données a été migrée de IBM DB2 vers RDS PostgreSQL (Base de données en mode PaaS) pour réduire les coûts et profiter des avantages de services managés Amazon **RDS** (Relational Database Service).

2. Comment penser l'architecture ?

Les points d'attention du FaaS

> Limitation de la durée d'exécution

Cette contrainte est simple à comprendre. Le temps maximal d'exécution lors de l'appel d'une fonction est limité par défaut. Selon les fournisseurs, il diffère légèrement, sur AWS **15 minutes** et sur Azure, **entre 5 à 10 minutes**.

Dans la grande majorité des cas, l'entreprise ne devrait pas être impactée par ces limites tant leurs valeurs sont importantes. À savoir qu'en cas de dépassement, il existe des solutions de contournement via l'utilisation d'autres services cloud type **AWS Step functions** ou encore **Azure Logic App**.

> Persistance des informations

Le monde du FaaS est un monde sans persistance des données :

Stateless.

À l'opposé des états de sessions des serveurs web, les fonctions ne permettent pas de stocker des informations en mémoire qui seraient utilisables entre plusieurs appels.

Cette propriété n'est pas une contrainte à proprement parler mais plutôt une bonne pratique relative à l'application du modèle des APIs Rest. Cela permet à une fonction une mise à l'échelle automatique. Cette fonction est déterministe et l'ensemble des informations nécessaires aux appels est stocké par exemple dans des objets type **DynamoDB** ou encore dans un **cache**.

> Une orchestration plus complexe

Cette contrainte n'est pas propre au FaaS, mais plus en relation avec la tendance des microservices. Il faut simplement être vigilant sur le fait que l'agilité que fait gagner le découpage du monolithe en microservices n'est pas sans incidence sur la complexité ajoutée lors des phases de livraison.

La gestion des versions dans l'exposition des APIs est un moyen simple de limiter les risques de régression lors des livraisons.

2. Comment penser l'architecture ?

> Le démarrage à froid, "Cold Start"

Il s'agit certainement de la contrainte la plus forte à prendre en compte.

Au premier appel d'une fonction, l'instanciation d'un objet ajoute un délai à la réponse. Un ordre de grandeur d'un démarrage à froid peut être d'environ 5 secondes, et peut donc être considéré comme un point bloquant dans un projet, surtout dans le cas d'appels synchrones.

Deux parties distinctes composent ce phénomène de "cold start" :

① Le temps pris par la plateforme cloud pour créer le conteneur

dans lequel une fonction sera exécutée. Sur ce point, il n'y a aucun moyen d'action pour optimiser ce délai.

② Le temps pris par la plateforme pour instancier la fonction elle-même (chargement des frameworks, interprétation du code...).

Sur ce deuxième point, il est possible d'agir sur certains éléments pour minimiser cette partie du "cold start", parmi lesquels :

- > réduire le nombre de packages nécessaires à l'instanciation de la fonction,
- > simplifier/optimiser le code,
- > **choisir un langage** qui donne de meilleures performances sur cet aspect de démarrage à froid.

Si ce premier appel de fonction, qui peut durer plusieurs secondes, est un point bloquant, il reste tout de même la possibilité de :

- > mettre en place un processus d'appels réguliers afin de "garder la fonction en vie" et **garantir des appels à chaud**,
- > utiliser l'option Provisioned Concurrency qui permet d'exécuter les fonctions AWS Lambda avec une latence de démarrage constante, dédiée aux microservices sensibles à la latence et aux API appelées de façon synchrone.

Les deux solutions pourraient générer des coûts supplémentaires. La première à travers des appels Ping-Pong pour garder les fonctions en vie, et la deuxième à travers l'unité de calcul provisionnée pour faire vivre la fonction dans son conteneur.

En synthèse, prévoir le temps d'exécution du phénomène de "cold start" en situation réelle est compliqué. C'est bien l'expérience qui permettra de définir ses propres abaques afin de savoir estimer ce processus dans son propre contexte. Tester et mesurer sont les maîtres mots !

2. Comment penser l'architecture ?

Le modèle de facturation

Comme pour tout hébergement, on évaluera le prix à payer pour l'exposition de ces fonctions.

Les paramètres qui influent sur le prix étant le nombre d'appels, la durée d'exécution, la mémoire consommée, et éventuellement le coût de l'intégration d'une fonction dans un VPC, il est difficile d'avoir un chiffre juste.

Les calculatrices des cloud providers sont là pour se projeter et avoir un ordre de grandeur :

Azure : <https://azure.microsoft.com/fr-fr/pricing/calculator/>

AWS : <https://calculator.aws/>

GCP : <https://cloud.google.com/products/calculator>

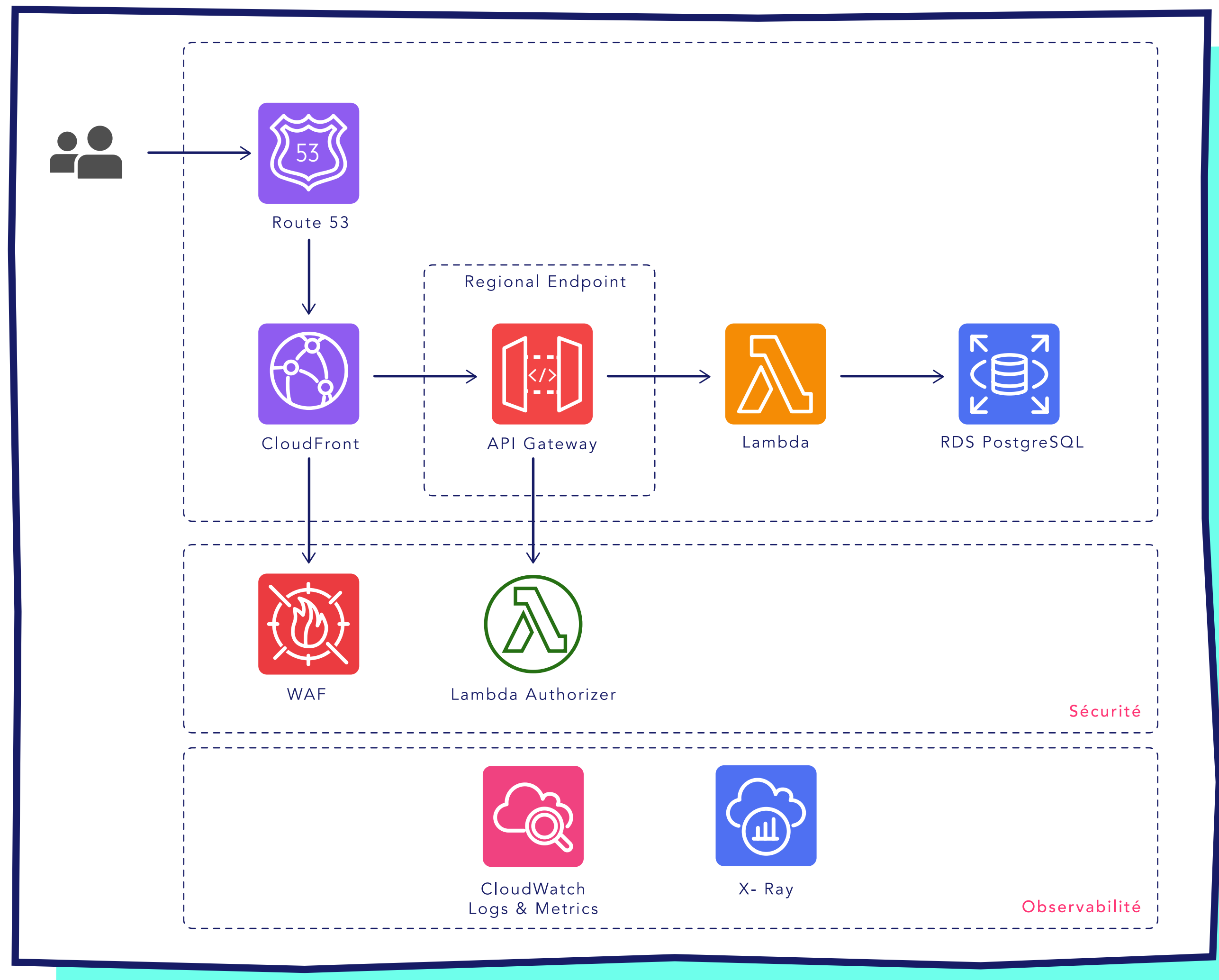
Là encore, les retours d'expérience aideront à affiner les estimations. Le grand jeu étant de savoir quel est le gain attendu dans la mise en place de ces fonctions par rapport à un site web plus standard basé sur du PaaS voire du IaaS.

Conclusion

L'agilité que nous apporte le cloud en termes de construction d'architecture nous permet de nous tromper en minimisant les impacts.

Pensez microservices, comprenez les limitations du monde Serverless, estimez approximativement les coûts et lancez-vous. Vous n'êtes pas à l'abri d'avoir de très bonnes surprises.

2. Comment penser l'architecture ?



La capture ci-contre représente l'architecture de la couche API du projet et se compose de :

- > Amazon **Route53** pour la gestion des DNS,
- > Amazon **CloudFront** comme CDN (Content Delivery Network) accompagné du **WAF** (Web Application Firewall),
- > **API Gateway** pour l'exposition des APIs accompagné de **Lambda Authorizer** pour la gestion d'authentification des APIs.
- > AWS **Lambda** pour l'exécution du code et la base de données **RDS PostgreSQL**.

Toutes les ressources de cette architecture sont monitorées via CloudWatch pour suivre les logs et les métriques, et X-Ray pour déboguer les erreurs et la performance.



3

**Je développe &
je construis
mon usine de
développement**

Overview

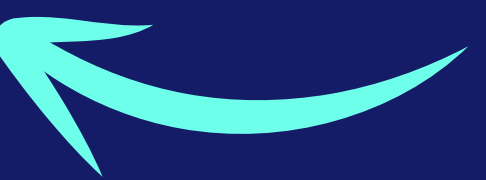
Je développe & je construis mon usine de développement

Les applications Serverless bien conçues sont découpées en services autonomes et utilisent un minimum de code (cf. cold start chapitre 2).

Au fur et à mesure qu'un projet se développe, l'objectif principal va être de maintenir un faible niveau de complexité.

La bonne gouvernance de l'usine de développement participe à garder une bonne maîtrise de l'application et indirectement un niveau de complexité le plus faible possible.

Ce chapitre fournit des recommandations pour la conception et la gestion des référentiels de code (stockage des codes sources applicatifs) dans des projets Serverless, et des bonnes pratiques en termes de déploiement de versions, CI/CD (Continuous Integration, Continuous Deployment).



Organisation des référentiels de code (repositories)

Dans l'univers du FaaS (et plus généralement des **microservices**), la tentation est grande de prendre des raccourcis en venant enrichir le périmètre d'une fonction existante. Ainsi, une fonction AWS Lambda simple au départ va voir son niveau de complexité augmenter au fil du temps pour devenir en quelque sorte un monolithe à elle seule.

Ce "monolithe" est alors représenté par une seule fonction AWS Lambda effectuant plusieurs tâches, et dont la logique applicative est contenue dans un seul et unique référentiel de code.

Si cela peut fonctionner pour des applications Serverless assez simples (tâches cron, traitement de données, processus asynchrones), à mesure qu'elles évoluent en intégrant de nouvelles fonctionnalités, il devient important de passer par une étape de refactoring.

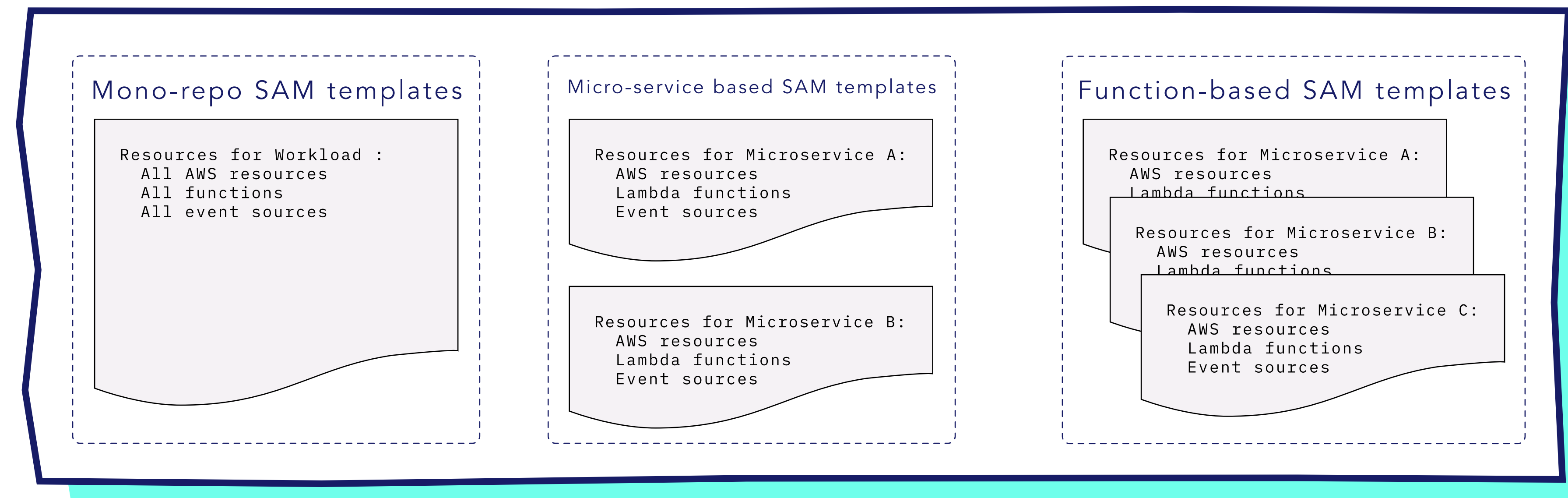
L'utilisation de frameworks tels que **AWS Serverless Application Model** (SAM) ou **Serverless Framework** facilite le regroupement de fonctionnalités dans des services plus petits.

Chacun d'eux peut avoir un référentiel de code distinct.

Pour SAM, le fichier template (.[yaml|yml]) contient toutes les ressources et définitions de fonctions nécessaires à une application.

Par conséquent, la division d'une application en microservices avec des modèles séparés est un moyen simple de diviser les référentiels et les groupes de ressources.

3. Je développe & je construis mon usine de développement



Dans la plus petite unité d'une application Serverless, il est également possible de créer un référentiel par fonction. Si ces fonctions sont indépendantes et ne partagent pas d'autres ressources AWS, cela peut être approprié. Les fonctions de traitement d'événements simples sont des exemples de candidats pour ce type de structure de référentiel.

Dans la plupart des cas, il est logique de créer des référentiels autour de groupes de fonctions et de ressources qui définissent un microservice. En prenant un exemple e-commerce, le « traitement des paiements » est un microservice avec plusieurs fonctions connexes plus petites qui partagent des ressources communes.

Comme pour toute application, la conception du référentiel dépend du cas d'utilisation, de la structure et de l'organisation des équipes de développement. Un référentiel centralisé rend plus difficile le travail sur des fonctionnalités différentes, la gestion du repository et les livraisons. À contrario, le fait d'avoir un trop grand nombre de référentiels peut amener à de la duplication de code et des difficultés de partage des ressources entre référentiels.

Trouver le bon équilibre pour son projet est une étape importante et aura un impact direct sur l'agilité et la facilité que l'on aura à maintenir son application.

3. Je développe & je construis mon usine de développement

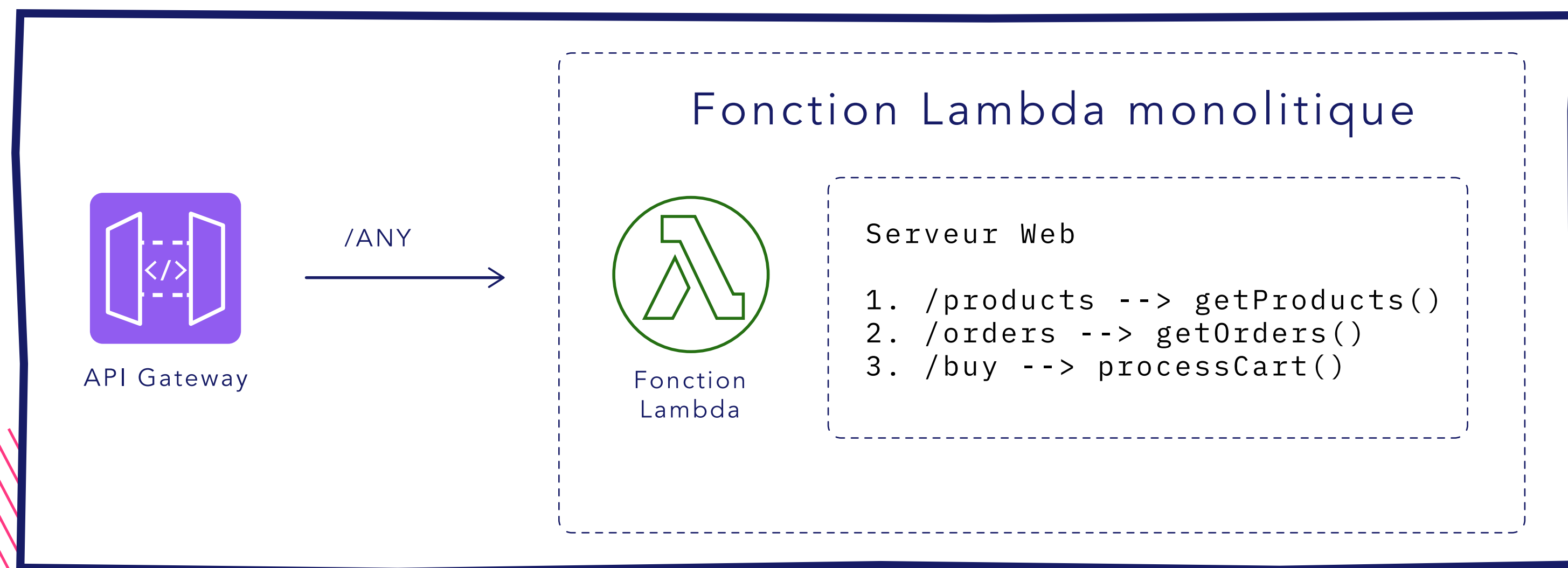
Utilisation de services en lieu et place de bibliothèques de code

Les services proposés par un cloud provider sont des éléments importants que les applications Serverless vont pouvoir consommer. Ces services offrent souvent une mise à l'échelle, des performances et une résilience supérieure à des frameworks fournissant des fonctionnalités similaires.

Exemple avec l'utilisation de l'API Gateway/Lambda d'AWS.

Amazon API Gateway est un service managé, qui permet aux développeurs d'exposer, de surveiller, de sécuriser facilement des APIs.

Des applications Web migrées vers des fonctions Lambda peuvent utiliser des frameworks tels que Flask (pour Python) ou Express (pour Node.js) qui prennent en charge le routage et la gestion des utilisateurs. Ces solutions conviennent bien lorsque l'application s'exécute sur un serveur Web.



L'utilisation de ces frameworks dans les fonctions Lambda donne des architectures comme celle-ci

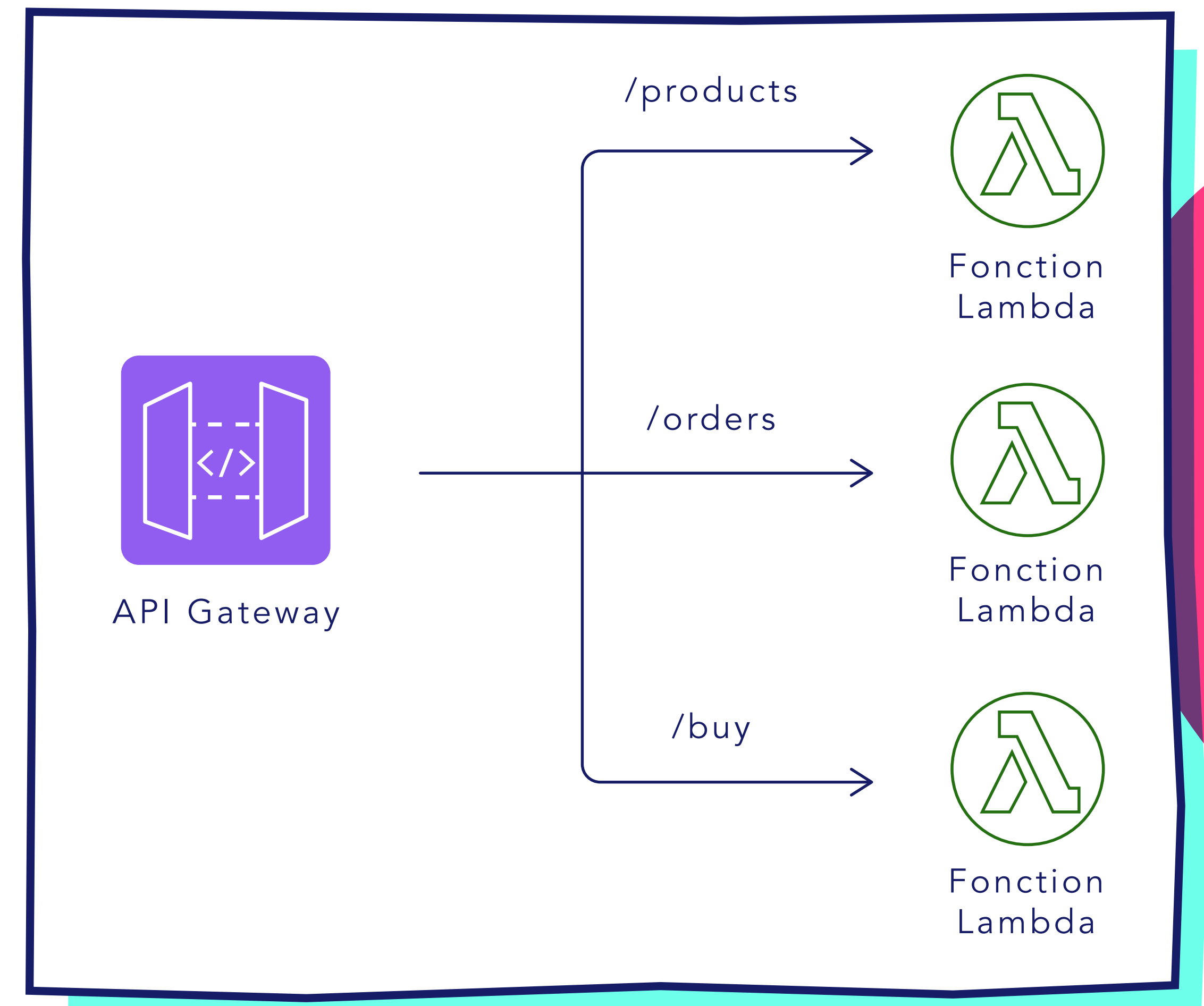
3. Je développe & je construis mon usine de développement

Dans ce cas, le service de gestion des APIs **Amazon API Gateway** envoie toutes les requêtes à la fonction AWS Lambda qui prend en charge elle-même le routage. À mesure que l'application grandit, de plus en plus de routes sont mises en place. La fonction Lambda devient donc plus complexe, et pour chaque montée de version/livraison, le code de la fonction est entièrement remplacé. Il devient donc plus difficile pour une équipe de développeurs d'intervenir sur ce projet pour modifier et/ou ajouter des fonctionnalités.

Dans cet exemple précis, il est préférable de tirer parti des fonctionnalités natives de routage qui sont offertes par l'API Gateway. Le framework Web n'est alors plus nécessaire dans la fonction AWS Lambda, ce qui diminue la taille du package de déploiement et va permettre de scinder une fonction Lambda complexe en plusieurs Lambda plus simples. Les mises à jour applicatives et la mise en place des fonctionnalités sont ainsi facilitées.

En outre, API Gateway est également capable de prendre en charge la validation des paramètres que vous n'avez plus besoin d'implémenter comme du code spécifique dans les fonctions AWS Lambda. Il peut également fournir une protection contre les accès non autorisés via AWS Cognito ou Lambda Authorizer.

L'utilisation de ces fonctionnalités avancées permet de modifier notre architecture.

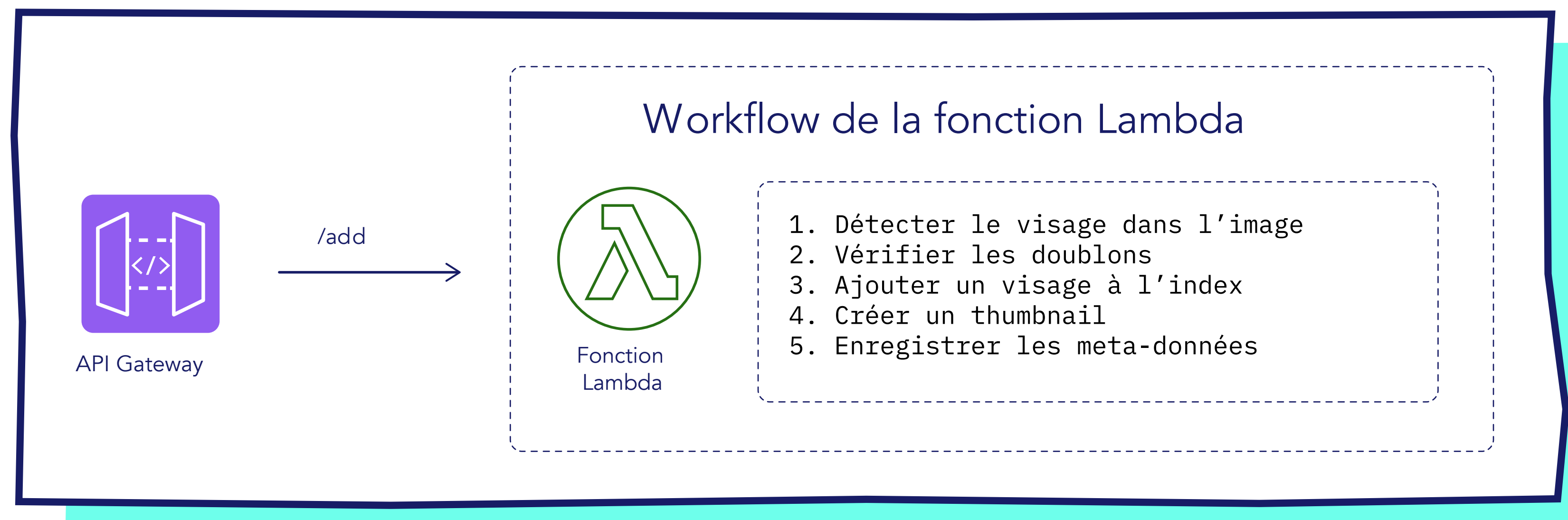


À noter, il est possible de configurer les itinéraires de l'API Gateway directement dans le modèle SAM de l'application.

Cinématique, orchestration des appels

Il est vivement déconseillé d'effectuer des orchestrations de flux de travail directement dans les fonctions AWS Lambda. De manière pratique, ce sont des sections de code qui appellent d'autres services et/ou fonctions, et effectuent d'autres actions en fonction d'une exécution réussie ou non, d'une donnée en retour.

Exemple ci-dessous :

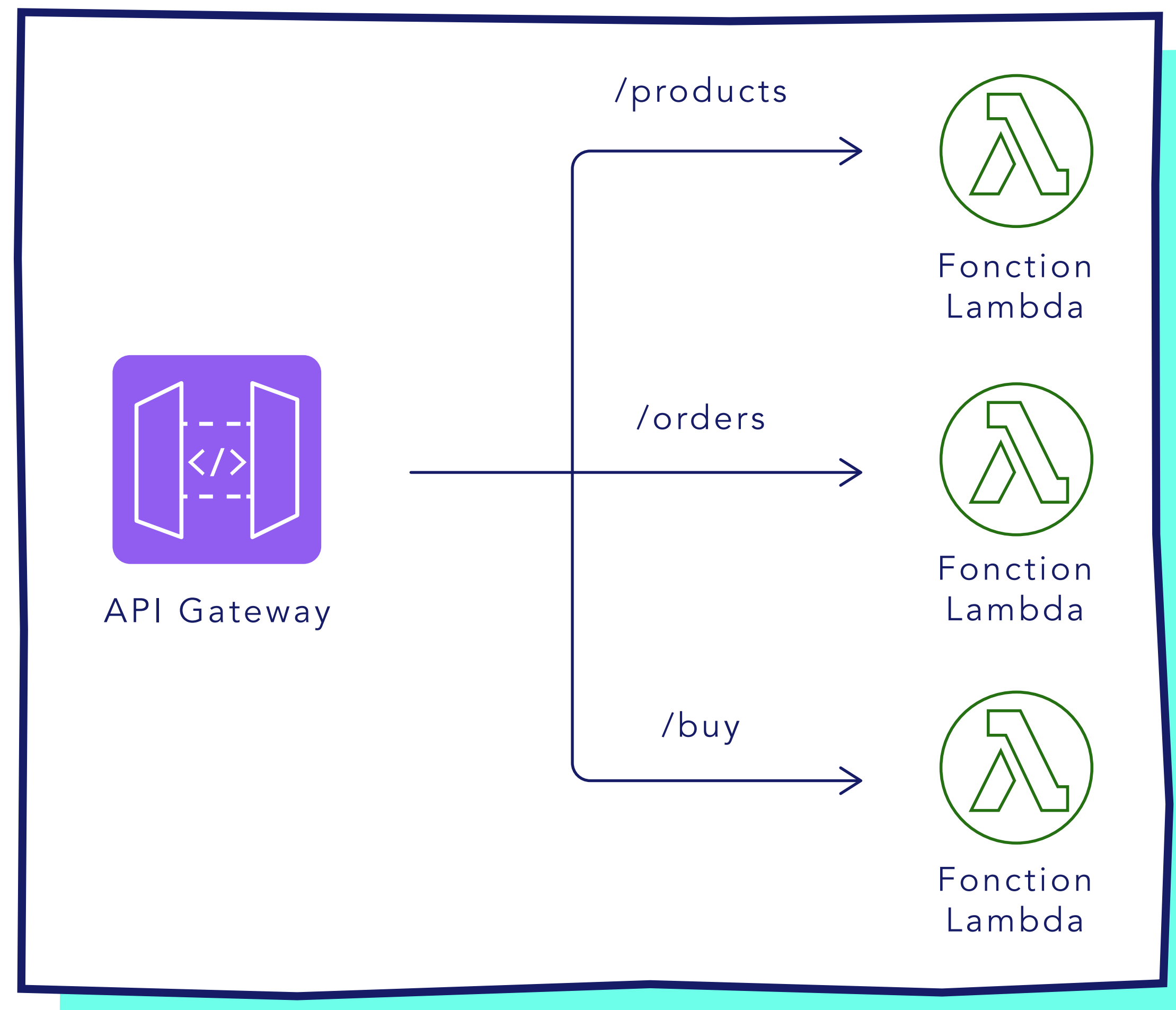


Ce type d'implémentation va certes complexifier la Lambda mais aussi et surtout rendre plus complexe l'observabilité lors de son exécution.

Une meilleure approche consiste à utiliser le service **AWS Step Functions**, qui peut orchestrer des flux de travail complexes. La définition de l'orchestration est définie au format JSON dans le modèle SAM de l'application. Ce service permet donc de réduire la quantité de code dans la Lambda et de gérer des workflows de plus longue durée.

3. Je développe & je construis mon usine de développement

L'exemple précédent, ré-architecturé avec un workflow Step Functions, ressemble à ceci :



Utilisation de plusieurs comptes AWS pour les équipes de développement

Il existe de nombreuses façons de déployer des applications Serverless en production. À mesure que les applications grandissent et deviennent plus importantes, les équipes de développement souhaitent généralement améliorer la robustesse du processus de déploiement.

Tout d'abord, il est fortement recommandé de créer un compte unique pour chaque membre/environnement de son organisation. À l'aide d'**AWS Organization**, nous pouvons gérer de manière centralisée la facturation, la conformité et la sécurité de nos comptes. Nous pouvons associer des stratégies à des groupes de comptes afin d'éviter des processus manuels, source d'erreurs et chronophages. Une approche simple consiste à fournir à chaque équipe un compte AWS de développement, puis à utiliser des comptes distincts pour les déploiements dans les environnements de Staging et de production.

Les comptes développeurs peuvent accéder à des ressources avec des autorisations de niveau administrateur sans risque pour l'environnement de production.

À noter qu'il est possible de tester localement l'exécution d'une fonction Lambda en utilisant les ressources AWS qui y sont liées via ces comptes individuels.

Une bonne pratique consiste aussi à utiliser **AWS Secrets Manager** afin de stocker les informations sensibles : authentification, chaîne de connexion... plutôt que de les laisser directement dans le code applicatif. Cela permet de pousser du code des environnements de développement vers le test et la production avec une sécurité optimale.

Gestion des versions et CI/CD pour les applications Serverless

Contrairement aux déploiements d'applications traditionnelles, les applications Serverless sont une combinaison de fonctions Lambda et de services AWS.

Nous pouvons citer comme exemple IAM, pour la gestion de rôles, et AWS Secrets Manager pour la gestion des mots de passe et clés d'authentification.

L'ajout ou la modification d'une fonctionnalité dans une AWS Lambda va entraîner un changement de version de celle-ci. Ce changement de version peut aussi être accompagné de l'ajout ou la modification d'un point de terminaison dans une API Gateway, ou encore par la création d'une nouvelle table DynamoDB.

Gestion des versions et CI/CD pour les applications Serverless

Il est donc important, lorsque nous implémentons des pipelines CI/CD pour des applications Serverless, de bien maîtriser les déploiements en termes de versionning et de mise à jour des différents services connexes.

AWS CodeDeploy (l'outil de déploiement AWS) livre les fonctions Lambda qui peuvent être décrites sous forme de template grâce à AWS SAM (Serverless Application Model). Il favorise aussi la configuration des déploiements de type "Blue/Green" pour chaque nouvelle version d'une fonction AWS Lambda. Cela permet ainsi de garder une parfaite maîtrise lors des livraisons et de minimiser les risques de régression.

La configuration suivante, SAM, illustre un exemple de déploiement Canary, avec une redirection de 10% de trafic vers la nouvelle version pendant 10 minutes. Le déploiement sera accompagné de deux alarmes basées sur des métriques d'erreur. Les deux métriques permettent de surveiller la montée de version et d'activer un rollback dans le cas où une des deux alarmes serait déclenchée. De plus, le déploiement est sécurisé via deux Hooks (PreTraffic et PostTraffic) qui permettent de déclencher des fonctions AWS Lambda avant et après le déploiement pour vérification et validation.

```
Resources:
  GetProducts:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: getProducts/
      Handler: app.handler
      Runtime: nodejs12.x

      AutoPublishAlias: live

    DeploymentPreference:
      Type: Canary10Percent10Minutes
      Alarms:
        # A list of alarms that you want to monitor
        - !Ref AliasErrorMetricGreaterThanZeroAlarm
        - !Ref LatestVersionErrorMetricGreaterThanZeroAlarm
      Hooks:
        # Validation Lambda functions run before/after traffic shifting
        PreTraffic: !Ref PreTrafficLambdaFunction
        PostTraffic: !Ref PostTrafficLambdaFunction
```



4

**Atteindre la
maturité
technologique**

Overview

Atteindre la maturité technologique

Les grands acteurs du cloud proposent tous des plateformes totalement "api"ées.

Ainsi, la totalité des actions que nous pouvons effectuer via les portails web sont aussi réalisables par des appels d'API. Cela apporte un changement majeur dans la manière de gérer les infrastructures. Il devient possible de les développer et ainsi de profiter de tous les avantages issus des pratiques du développement (DevOps, CI/CD...).

Si ce mode "As Code" apporte un gain non négligeable dans l'agilité avec laquelle nous pouvons gérer les infrastructures, cela les rend aussi plus volatiles. Il est important pour garder la maîtrise de ces environnements cloud de mettre en place une gouvernance technique et financière, gouvernance à l'origine de la création de nouveaux métiers tel que le convoité FinOps.

D'autres métiers techniques comme ingénieur système ou analyste d'exploitation sont aussi amenés à évoluer fortement au contact du cloud, en rapprochant leurs pratiques de celles des développeurs.

Réussir ce virage technique et humain, c'est la garantie de maîtriser son parc IT hébergé sur le cloud.

4. Atteindre la maturité technologique

Infrastructure as Code : moteur de l'agilité technique

Une des innovations que nous apporte le cloud et les pratiques issues de la mouvance DevOps est la possibilité d'automatiser la création et le déploiement de l'infrastructure, c'est ce que l'on appelle l'Infrastructure as Code ou IaC.

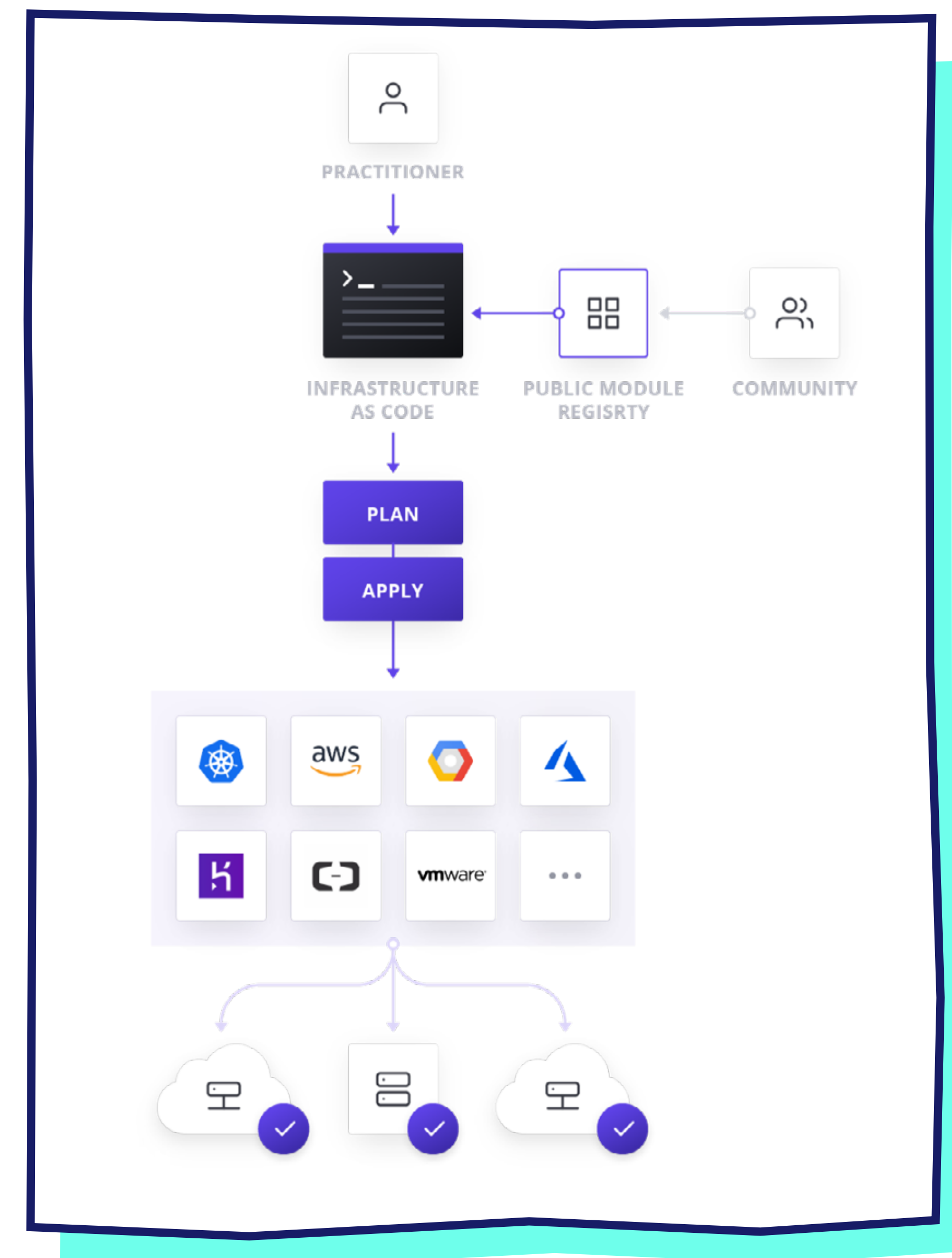
Ce mode offre de nombreux avantages, à savoir :

- > **création automatique d'une nouvelle infrastructure logicielle complète en quelques minutes** (base de données, fonctions, fichiers statiques...), idéale pour améliorer le time to market d'un projet,
- > **répétabilité du processus pour de multiples environnements** (développement, recette, pré-production, production), avec également la possibilité de créer autant d'environnements que désiré à n'importe quel moment, pour tester par exemple une nouvelle fonctionnalité,
- > **réduction des coûts, diminution de la fatigue** engendrée par un processus ayant peu de valeur ajoutée et également **réduction des risques** en cas d'erreur.

Pour atteindre cette agilité technique, il existe des outils propres à chaque cloud provider (AWS CloudFormation, Azure Resource Manager template), mais aussi des outils agnostiques qui présentent l'avantage de pouvoir adresser avec un seul outil/langage plusieurs plateformes cloud.

Terraform

Terraform est l'un des outils phare en matière d'IaC. De par sa nature, il permet de gérer des infrastructures dans les principaux cloud providers, ou même dans des cloud privés.



4. Atteindre la maturité technologique

Chaque brique logicielle est généralement représentée dans un module qui contient le code nécessaire pour créer, modifier et supprimer une ressource.

On peut facilement imaginer, créer un mono-repository contenant à la fois le code source permettant de faire fonctionner le logiciel, et le code source (Terraform) permettant de décrire l'infrastructure associée. Cette pratique où chaque commit représente l'unique source de vérité du logiciel à un instant T a pour nom GitOps. Il ne peut donc plus exister de différences entre ce qui est demandé (code source) et ce qui sera délivré (software & hardware).

En admettant que l'on dispose du code Terraform pour déployer notre infrastructure, il se passera toujours au mieux les 2 étapes suivantes.

- ① **La planification** permet de faire la différence entre l'architecture qui existe actuellement (si déjà déployée) et celle que l'on tente de créer. Cette étape a aussi l'avantage de permettre la visualisation des différentes modifications à appliquer. Si ces changements ne sont pas cohérents ou ne sont pas valides, il est possible de revenir en arrière et d'annuler la « terraformation ».
- ② **L'application du plan** consiste à exécuter le plan précédemment obtenu. Une fois l'application du plan terminée, notre infrastructure est désormais prête à l'emploi.

Un exemple de code ci-dessous.

```
resource "aws_lambda_function" "test_lambda" {
  filename      = "lambda_function_payload.zip"
  function_name = "lambda_function_name"
  role          = aws_iam_role.iam_for_lambda_arn
  handler       = "exports.test"

  source_code_hash = filebase64sha256("lambda_function_payload.zip")

  runtime = "nodejs12.x"

  environment {
    variables = {
      foo = "bar"
    }
  }
}
```

D'autres outils d'IaC existent parmi lesquels : Pulumi, Serverless Framework, Chalice, Claudia.js. À noter que certains sont entièrement orientés sur la technologie Serverless.

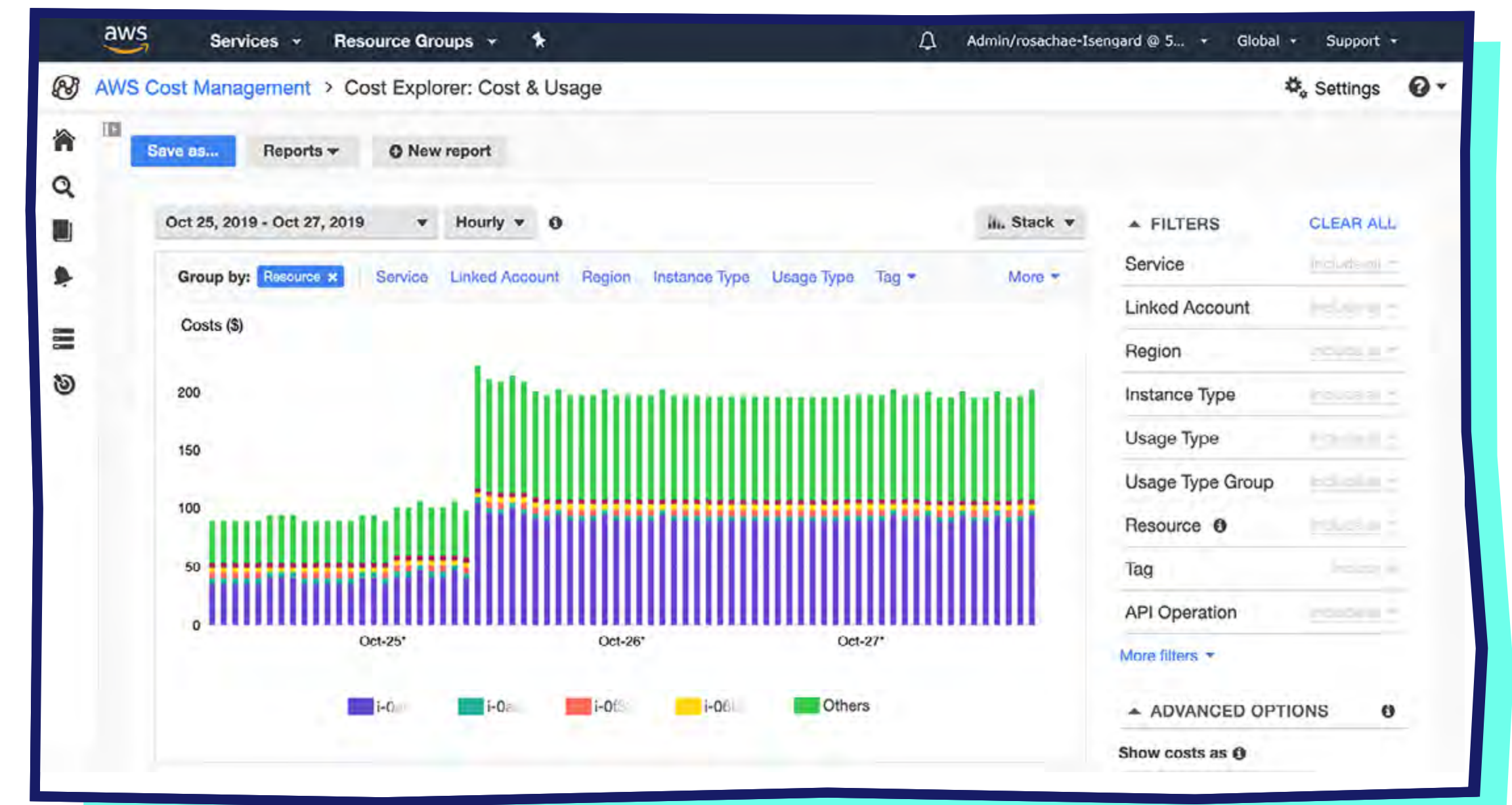
En résumé, le choix d'un outil d'Infrastructure as Code est plus que fortement recommandé afin de permettre une parfaite maîtrise de nos infrastructures, au même titre que ce que nous faisons déjà avec nos applications.

4. Atteindre la maturité technologique

FinOps - Le bon produit au juste prix

Un autre enjeu dans l'appropriation des plateformes cloud est la maîtrise des coûts. S'il est souhaitable de laisser aux équipes projets le maximum d'autonomie et d'agilité, il est important de contrôler que la présence de chaque objet sur un environnement cloud est utile et correctement dimensionné. Cet aspect de la démarche fait donc le rapprochement entre l'aspect financier et l'aspect opérationnel, autrement appelé FinOps.

Afin d'optimiser les coûts, il faut dans un premier temps avoir la capacité de les mesurer individuellement et globalement. Il existe pour cela des outils dans la majorité des cloud providers. C'est le cas par exemple d'**AWS Cost Explorer**.



Cet outil est capable de collecter et d'agréger les données de consommation et de facturation sur l'ensemble des services AWS. Afin de mesurer précisément où et comment réduire ses coûts intelligemment, il est possible d'étudier les graphes sous plusieurs angles :

- > pour une ressource en particulier (ex : AWS Lambda, DynamoDB...),
- > pour un projet en particulier en utilisant au préalable une nomenclature de tags,
- > pour une ressource en particulier en fonction de l'activité business.

4. Atteindre la maturité technologique

Les tags font partie intégrante des solutions cloud. Ils sont majoritairement destinés à mettre une étiquette sur les projets et les ressources. Bien que facultatifs, les tags doivent être utilisés pour classifier les ressources et affiner l'analyse des coûts. On peut prendre comme exemple de tags : le nom du projet associé, le type d'environnement (développement ou production) ou la Business Unit en charge du projet. Il est même conseillé de créer automatiquement ces tags à l'aide d'un outil d'Infra as Code.

De la même manière que pour les tags et dans un souci de clarté, il est possible de définir une convention de nommage commune pour toutes les ressources et gérer celle-ci automatiquement par IaC.

En complément des bonnes pratiques précisées par les principaux cloud providers (**AWS** et **Azure**), **nous conseillons de choisir une règle de nommage de ces ressources qui comprend au minimum :**

- > **les initiales du type de la ressource,**
- > **le nom du projet** (ou ses initiales si le nom est trop long),
- > **l'environnement cible** (dev, recette, preprod ou prod).

Il est aussi possible de mettre en place des alertes lors du dépassement d'un budget, afin de réagir immédiatement en cas de surconsommation anormale.

Qui dit FinOps dit aussi Green IT. Il devient possible de réduire l'impact environnemental en surveillant la consommation budgétaire, étroitement liée à la consommation énergétique.

Observabilité

La meilleure description que l'on peut donner de l'observabilité est d'être une mesure du bon fonctionnement du système dans son ensemble. Cela implique le monitoring de chaque brique logicielle. Pour simplifier, là où le monitoring offre des informations sur ce qui ne fonctionne pas et à quel moment un service est en erreur, l'observabilité permet de comprendre le pourquoi de l'erreur et de réduire le temps d'analyse menant à la solution du problème.

Une plateforme d'observabilité doit :

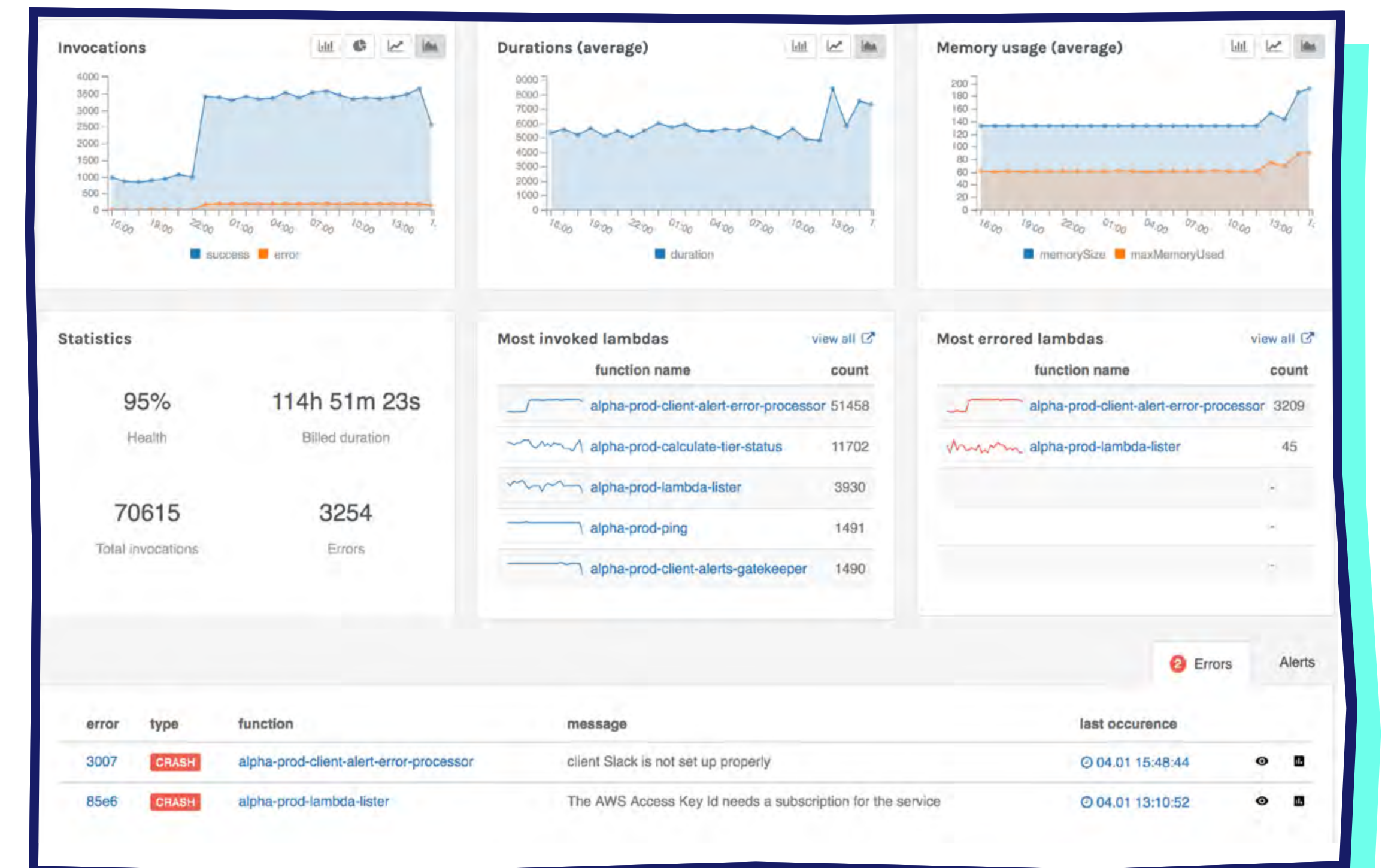
- > **collecter et agréger toutes les données d'analyse,** de n'importe quelle source, à un seul endroit. Cette manière d'opérer permet de réduire drastiquement le nombre d'outils à maîtriser et évite donc de devoir changer d'outil constamment, ce qui est très important en cas de problème,

4. Atteindre la maturité technologique

- > **former des connexions entre les différentes sources de données et les entités** afin de mieux comprendre le problème et son origine. Mettre en lumière le contexte du problème est la clé permettant de le résoudre au plus vite,
- > **offrir une interface de développement (API) au-dessus de la plateforme** pour permettre aux développeurs de créer des applications afin d'automatiser certaines tâches. La programmation permet de découpler les possibilités offertes par la plateforme d'observabilité.

Avant de créer sa plateforme, il faut dans un premier temps être en mesure d'obtenir des informations d'analyse et donc d'instaurer un processus de monitoring et une stratégie de logs efficiente.

L'intérêt d'une plateforme d'observabilité va évidemment bien au-delà du Serverless mais il n'en reste pas moins que certains outils ciblent spécifiquement ce type d'architecture. On y retrouve différentes métriques comme la fréquence d'utilisation des fonctions, le nombre d'erreurs récentes, et ce qui nous intéresse plus particulièrement : le détail de ces erreurs (nom de la fonction, message d'erreur, stacktrace...) !





5

**Envie d'aller
encore plus loin ?**

Overview

Envie d'aller encore plus loin ?

Après avoir vu ce qu'il est possible de mettre en place dans le cadre d'une architecture Serverless AWS et les outils qui gravitent autour, il devient alors primordial de nous assurer que cette architecture soit fiable et robuste dans la durée.

Que ce soit pour du monolithe, des microservices ou du Serverless, la qualité est un enjeu majeur et il faut à la fois fiabiliser le code et l'architecture. La qualité peut être considérée comme un investissement, puisque les efforts de temps et de budget supplémentaires nécessaires au lancement du projet se traduiront par un ROI positif lors de la maintenance et l'évolution de l'application.

Nous allons donc parler de deux approches permettant d'initier et d'améliorer la qualité, en faisant un premier focus sur la partie code et un second sur la partie infrastructure/architecture.

Sur le code contenu dans les AWS Lambda, il est possible de mettre en place des tests vérifiant simplement le bon déploiement des fonctions, mais aussi des tests fonctionnels plus avancés afin de s'assurer qu'elles répondent correctement au besoin.

Sur l'architecture, Netflix est à l'origine d'un concept appelé le "chaos engineering" ou "l'ingénierie du chaos". Cette approche consiste à améliorer la résilience de l'architecture en introduisant volontairement des dysfonctionnements afin de tester régulièrement que les palliatifs mis en place sont efficaces.

Commençons avec les tests !



5. Envie d'aller encore plus loin ?

Tester son application Serverless

Pour les applications monolithes, la qualité peut être vérifiée et mesurée par des équipes expertes sur le sujet, comme des chargés de qualification ou des testeurs produits. Leur but est essentiellement de s'assurer que les principaux scénarios n'engendrent pas de bugs. Une partie de ce travail est manuel mais peut aussi être automatisée à l'aide d'outils prévus à cet effet. Chaque langage possède au moins un outil de tests unitaires (JUnit, PHPUnit, Jest...). Le fait d'automatiser certains scénarios permet aux testeurs de ne pas rejouer systématiquement les mêmes tests avant chaque livraison. Enfin, il est plus intéressant et plus enrichissant pour ces équipes de se concentrer uniquement sur la valeur ajoutée (bugs corrigés et nouvelles fonctionnalités) au cours du sprint ou de la release.

La constitution de ce harnais de tests automatiques peut se faire au fur et à mesure des itérations en collaboration avec les testeurs et les développeurs. On parle même d'un métier faisant le liant entre les 2 précédents, celui d'automaticien.

Mais pour nos applications Serverless ? Comment est-ce que cela se passe-t-il ?

Pour les applications monolithes, c'est plus facile car l'application représente un seul et même bloc. Toutes les fonctions applicatives ont fait l'objet d'une unique livraison applicative. Les testeurs

peuvent donc dérouler l'ensemble des scénarios sur une version bien précise de l'application.

Lorsque nous sommes sur des architectures microservices ou Serverless, la pratique du test manuel reste pertinente mais il devient plus difficile d'identifier la source du problème. Comme nous sommes dans une logique de création d'un grand nombre de fonctions réutilisables et les plus génériques possibles, il peut être plus difficile d'identifier la fonction qui déclenche le bug.

Pire encore, est-ce qu'en modifiant une fonction, je ne risque pas de compromettre d'autres systèmes qui y font aussi appel ?

Il n'est pas non plus concevable de demander à une tierce personne de tester unitairement chaque fonction de façon manuelle. Il convient donc de s'outiller correctement et d'adopter une stratégie de tests.

Les approches que nous allons vous présenter vont de la plus simple (et la moins coûteuse, mais la moins efficace) à la plus complète (plus coûteuse mais avec plus de possibilités).

5. Envie d'aller encore plus loin ?

Le test d'intégration simple

La première approche consiste à créer des tests d'intégration simples. Pour cela, AWS nous propose un outil (intégré dans le portail AWS) permettant de tester nos endpoints HTTP (s'assurer que nos AWS Lambda soient bien déployées) ou de faire des tests de charge sur nos Lambda.

Cet outil est simple d'utilisation et permet d'initier notre harnais de tests sans développer une seule ligne de code puisque tout se fait à l'aide de fichiers JSON. Les résultats des tests peuvent être stockés dans un espace de stockage cloud de type DynamoDB, ou Amazon S3.

Un exemple de test de charge avec l'outil AWS :

```
{
  "operation": "load",
  "iterations": 100,
  "function": "TestHarness",
  "event": {
    "operation": "unit",
    "function": "HTTPSInvoker",
    "resultsTable": "unit-test-results",
    "event": {
      "options": {
        "host": "customers-list.us-east-1.amazonaws.com",
        "path": "/prod/Customers",
        "method": "GET"
      }
    }
  }
}
```

Ce fichier JSON va permettre de lancer un test de charge en exécutant 100 fois l'appel à une AWS Lambda.

Retrouvez plus d'informations sur cet article de blog AWS :
<https://aws.amazon.com/fr/blogs/compute/serverless-testing-with-aws-lambda>

Ce type de tests est le plus simple et ne nécessite qu'un fichier de paramétrage.

Cependant, nous vérifions simplement que nos Lambda sont correctement déployées. Nous ne testons pas le contenu des réponses aux appels.

5. Envie d'aller encore plus loin ?

Le test d'intégration plus poussé

La seconde approche va nécessiter de nous équiper un peu plus en termes d'outils. Cependant, nous n'avons toujours pas besoin de code particulier, simplement de connaître le langage YAML. YAML n'est pas un langage de programmation mais un format de représentation de données au même titre que JSON ou XML.

En y associant le framework **Serverless**, il est possible de créer un harnais de tests où nous pouvons vérifier le contenu renvoyé par une Lambda.

Serverless Framework permet de tester ses fonctions aussi bien en local qu'à distance.

```
- name: says hello to the world by default
  endpoint: {function: hello}
  response:
    body: "Hello, world"
- name: says hello to Daveo
  endpoint: {function: hello}
  request:
    body:
      name: Daveo
  response:
    body: "Hello, Daveo"
```

Un exemple plus fourni est disponible sur le site officiel du framework Serverless : <https://www.serverless.com/blog/serverless-test-framework>

Il existe un autre framework open-source similaire basé sur du JSON permettant de faire des tests d'intégration encore plus poussés, moyennant une certaine montée en compétences. Ce dernier est disponible ici: <https://github.com/blackflux/lambda-tdd>

5. Envie d'aller encore plus loin ?

Le test unitaire

L'écriture de tests d'intégration est plus simple que celle des tests unitaires.

En effet, nous appelons simplement notre fonction et contrôlons la réponse.

Cependant, lorsque la réponse d'un appel dépend d'une source externe (base de données, APIs...) il peut être compliqué de valider le fait que la réponse reste correcte.

Dans des cas comme celui-ci, il est plus aisé d'écrire des tests unitaires, notamment afin de nous assurer que nos tests soient pérennes dans le temps.

Qu'est-ce qu'un test unitaire ?

Un test unitaire est un traitement automatisé permettant de valider une portion de code comme une fonction ou une classe. Les caractéristiques d'un test unitaire sont les suivantes :

- > **isolée**, aucune dépendance externe n'est réellement appelée (base de données, API),
- > **small ou suffisamment petite** une vérification (assert) par test,
- > **indépendante** : peut être joué seul et n'est pas dépendant d'un autre test,
- > **rejouable à l'infini** : le résultat du test doit être toujours le même dans la durée.

Que peut-on tester ?

Nos appels aux fonctions doivent être isolés des dépendances externes. Il nous faut donc les simuler.

Dans la pratique du test unitaire, nous pouvons identifier deux types de tests :

- > **Veut-on simuler une dépendance et vérifier comment la fonction traite le résultat ?** Pour ce faire, nous avons besoin d'utiliser un **stub**, c'est-à-dire un objet permettant de simuler une classe, une API, une base de données.
- > **Veut-on nous assurer que la dépendance est bien appelée avec les bons paramètres ?** Dans ce cas, nous allons utiliser ce que l'on appelle un **spy**.

5. Envie d'aller encore plus loin ?

Peut-on faire des tests unitaires quand on fait du Serverless ?

Bien sûr ! Il se trouve d'ailleurs que nous pouvons appliquer les mêmes principes et méthodologies aussi bien sur une application monolithe que sur une application Serverless.

Si les outils diffèrent selon la technologie et le langage utilisé, l'approche reste identique.

Le plus simple est de s'appuyer une nouvelle fois sur le framework Serverless, puisque les outils pour écrire des tests unitaires (à l'aide de **Jest** notamment) sont intégrés. Tous les concepts de *stubs*, *spys* et *mocks* sont disponibles.

Les tests unitaires peuvent être joués sur un poste local, mais il est également possible de les faire tourner dans une chaîne d'intégration continue (CI/CD) afin d'assurer la non-régression dès lors que l'on décide de modifier une Lambda.

Le blog officiel de Serverless explique comment écrire ses premiers tests unitaires pour des AWS Lambda : <https://www.serverless.com/blog/unit-testing-nodejs-serverless-jest>

Intégrer l'écriture des tests dans le développement

L'approche TDD (ou Test Driven Development) consiste à ne plus considérer l'écriture des tests unitaires comme une étape après le développement mais bien comme faisant partie intégrante du développement de nos fonctions.

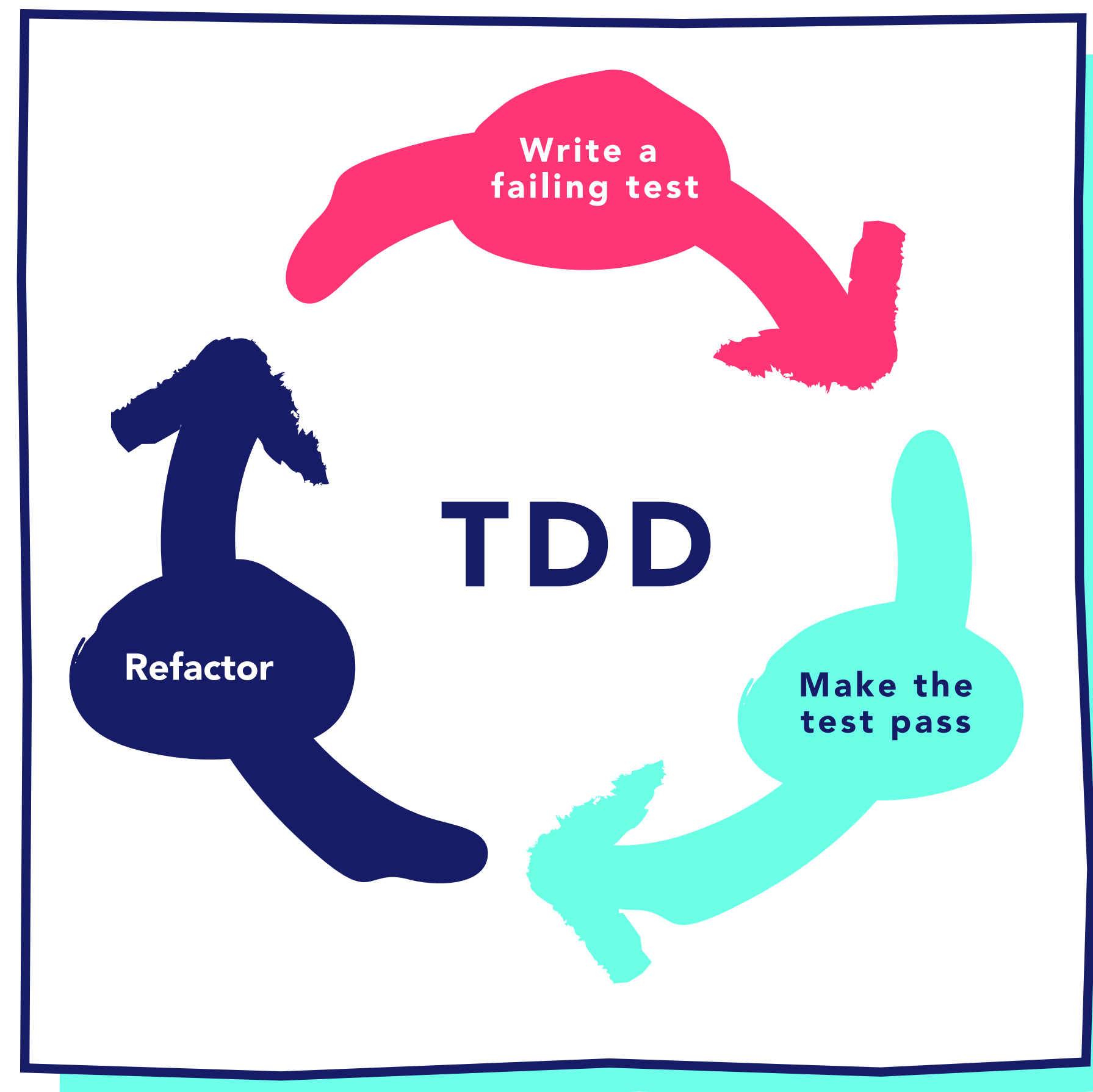
TDD est une méthodologie de développement où l'on va se servir des tests pour spécifier du code et non le valider.

La démarche en 3 étapes :

- ① Écrire un premier test unitaire qui échoue
- ② Écrire le code minimal permettant de faire passer le test au vert
- ③ Refactoriser/réorganiser le code si besoin

... et de répéter cette boucle de 3 étapes dès lors qu'une nouvelle fonctionnalité est demandée ou qu'un bug est à corriger.

5. Envie d'aller encore plus loin ?



Notre harnais de tests est constitué dès le départ, et notre CI/CD est là pour valider que le code produit ne provoque pas de régression sur l'existant.

Si vous souhaitez en savoir plus sur TDD, cet article du référentiel Institut Agile est disponible : <http://referentiel.institut-agile.fr/tdd.html>

Pour commencer à appliquer TDD sur une application Serverless, nous recommandons cet article : <https://hackernoon.com/tdd-means-zen-along-with-serverless-means-nirvana-a39a76ee8e63>

Cette discipline demande de la pratique et de l'expérience, pouvant nécessiter un investissement en temps durant la phase initiale du projet. Cet effort aura pour effet d'entraîner du retard sur les premières itérations, le temps que les développeurs acquièrent cette nouvelle compétence. Cependant, cet investissement sera rentabilisé par la suite puisque le risque de régression se retrouve diminué, la qualité de l'application est améliorée ainsi que la facilité de maintenance. Les tests permettront d'indiquer immédiatement la présence d'un bug.

5. Envie d'aller encore plus loin ?

Chaos engineering : un monde en perpétuel changement

L'un des principes fondamentaux d'une architecture Serverless est que chaque service doit être résilient. Même si l'écriture de tests unitaires est honorable, cela ne signifie pas que le système final sera résilient. Étant donné la nature toujours plus complexe des systèmes, il devient important de s'assurer qu'un système reste opérationnel en mode dégradé.

Le Chaos Engineering est la discipline visant à réaliser des expérimentations sur le système pour améliorer la confiance dans la capacité du système à résister à des conditions extrêmes, en particulier dans un environnement de production.

Avant de mettre en place du Chaos Engineering, il faudra s'assurer que les différents environnements soient les plus similaires possibles à l'environnement de production (Infrastructure as Code) et il faudra aussi s'armer d'une plateforme d'Observabilité suffisamment mature pour être en mesure d'évaluer l'impact des expérimentations sur le système. L'objectif est dans un premier temps de savoir où nous nous situons entre un état stable versus un état instable du système. Lorsque tout ce travail est accompli, il devient alors assez facile de décrire les points faibles de notre système et de les corriger.

Parmi les différentes expérimentations pouvant être créées, nous retrouvons par exemple :

- > **la mise en place d'un timeout** sur une fonction AWS Lambda,
- > **l'échec continu ou discontinu** d'une fonction Lambda (erreur HTTP 500),
- > **l'indisponibilité d'une ou plusieurs dépendances** d'un service (ex : base de données, API externe),
- > **la surcharge momentanée des services** (nombreux appels concurrents sur une fonction Lambda),
- > **l'indisponibilité partielle ou totale d'une région.**

Des offres en mode SaaS existent pour nous aider à vacciner le système (ex : **gremlin.com**).

N'hésitez pas à mettre en place ces expérimentations dès les phases de développement pour faire prendre conscience aux équipes projets des aléas possibles qui pourraient être vécus par les clients/utilisateurs de l'application en production.

Conclusion

Nous l'avons vu dans ce livre blanc : l'architecture microservices hébergée sur une infrastructure Serverless est une nouvelle façon de concevoir les logiciels. L'avantage immédiat pour l'entreprise est l'abstraction de la gestion de l'infrastructure sous-jacente. C'est le propre d'un service managé. **Il devient alors possible de se focaliser davantage sur la création de valeur au sein de la structure - c'est-à-dire - travailler sur la partie métier des applications. Cela amène également une amélioration implicite du *time to market*.** En d'autres termes : qui dit absence d'infrastructure à gérer, dit davantage de temps pour le développement. Et cela bouscule les habitudes.

Par conséquent, le passage vers ce modèle Serverless ne sera pas sans impact sur une organisation. Si l'émergence du cloud public n'a pas encore transformé les façons de travailler chez certains, ce n'est qu'une question de temps. Si l'on n'a plus besoin de gérer les infrastructures, alors la question centrale est de savoir ce que feront demain les *ops* ? C'est le sujet abordé par le mouvement *noOps* qui prophétise la fin du mouvement DevOps. Rappelons que le *noOps* est un concept selon lequel la gestion de l'infrastructure a atteint un niveau d'automatisation tel qu'il n'est plus nécessaire d'avoir une équipe en charge de son administration. Il est préférable d'anticiper ces futurs impacts en accompagnant ce changement.

Conclusion

Le métiers d'ops va en réalité continuer d'exister... mais plus sous sa forme actuelle. De la même manière, nous pouvons imaginer que l'émergence des intelligences artificielles ne fera pas disparaître le métier de développeur. Et oui, James Cameron s'est trompé, en 2021 toujours pas de Skynet à l'horizon. Cela n'est pas une surprise : l'évolution n'est-elle pas l'histoire des métiers de la technologie ? Rappelons que le métier de développeur a évolué au début de ce siècle avec l'émergence d'Internet pour quitter notamment des solutions en client lourd-serveur et aller vers des applications web. Ce type d'application ne requiert plus la même façon de penser, de développer et de déployer, les possibilités n'étant plus les mêmes.

Les nouvelles missions potentielles des ops seront maintenant - entre autres - de co-construire et participer à la maintenance des pipelines de CI/CD (Continuous Integration / Continuous Delivery), perfectionner le monitoring et l'observabilité, mettre en oeuvre du chaos engineering ou encore gérer les fichiers de configuration du **serverless framework**. Bref, il s'agira de s'orienter vers un positionnement définitivement plus DevOps.

Les nouvelles technologies liées au cloud nécessitent que les acteurs soient formés. Chez Daveo, nous avons la conviction qu'il est important de maîtriser ces nouveaux outils. Il est conseillé de se lancer dans une migration vers le cloud avec un accompagnement autour de la formation, sans toutefois négliger le travail préparatoire. Nous aimons reprendre les propos d'Abraham Lincoln qui nous prévenait déjà au 19e siècle : «Si j'avais 6 heures pour abattre un arbre, je passerai les 4 premières à affûter ma hache». Sage conseil !

Culture Daveo

Cabinet de conseil en management et technologies

La naissance et un peu plus...

Daveo est avant tout une histoire humaine. L'histoire de deux associés, poussés par l'envie de faire plus mais surtout de faire mieux. En bref, apporter la preuve que l'on peut réussir en se fondant sur la transparence, le travail collaboratif et l'épanouissement.

L'aventure commence en 2008 à Paris. Douze ans plus tard, le Groupe Daveo rayonne dans 6 villes en France, rassemblant ainsi plus de 300 collaborateurs.

La clé du succès ?

Cette même culture de la différence avec l'envie de bien faire et d'avoir un impact positif, faisant ainsi de Daveo une référence en matière de management, de coaching et d'innovations technologiques. les mêmes.

Notre vocation ?

Réunir ceux qui imaginent et ceux qui réalisent en se basant sur trois piliers pour réussir :

- > Représenter une expérience professionnelle inoubliable pour les collaborateurs
- > Innover et créer de la valeur en continue sur les offres clients
- > Développer une économie à impact positif avec un programme de RSE

Pour quel impact ?

L'accompagnement réussi de clients grands comptes (CAC 40 et SBF 120) dans leurs projets ! Une transformation technologique s'opère avant tout avec l'implication de femmes et d'hommes. Ce serait une erreur d'oublier l'humain. Au-delà du côté technique, le projet de quelques-uns doit devenir le projet de tous pour en garantir le succès. Les équipes de Daveo accompagnent et co-construisent avec les directions opérationnelles, RH, Innovation ainsi que les DSI autour de leurs enjeux. Pour ce faire, nos consultants activent différents leviers spécifiques à chaque organisation : de la formation, en passant par la réorganisation ou encore l'analyse d'outils de marketing numérique. Daveo défend cette vision consumer centric pour un impact significatif !

2008
DATE DE CRÉATION

+300
COLLABORATEURS

80
CLIENTS

6
AGENCES

90/100
INDEX ÉGALITÉ
FEMMES-HOMMES

26
MILLIONS D'€
DE CA EN 2019

Remerciements

Après plusieurs mois de travail en équipe, de workshops, de points de synchro et de communication à distance, le nouveau livre blanc Daveo "Du monolithe au Serverless, focus sur l'un des services majeurs d'Amazon Web Services : AWS Lambda" est arrivé !

Merci à tous les contributeurs pour leur expertise, leur savoir-faire et leur soutien tout au long de l'écriture de cet ouvrage. Leur niveau d'exigence et de professionnalisme ainsi que la contribution exceptionnelle des équipes AWS nous permettent d'aller toujours plus loin.

Ce livre blanc a été pensé afin d'apporter un nouveau regard, vulgariser (au travers des pages overview) et marquer l'expansion de l'utilisation des technologies liées au Serverless. Dans un monde en perpétuelle évolution et en recherche d'optimisation constante des technologies, le cloud et le Serverless sont devenus incontournables. Chez Daveo, nous sommes convaincus à la fois de leur influence positive et de leur productivité. De l'approvisionnement du service au développement d'une architecture sur mesure, nous avons exposé, au travers de cet ouvrage collaboratif, les coulisses du cloud, en vous livrant notre vision et notre retour d'expérience pour réussir votre transition.

Un grand merci et félicitations aux auteurs de ce livre :

- > David Bottiau, ingénieur étude et développement chez Daveo
- > Richard Buttin, Head of Engineering & Operations, Supply & Fullfilment chez Adeo
- > Nicolas Moutschen, Specialist solutions architect serverless chez AWS
- > Jean-Baptiste Vigneron, leader technique agile chez Daveo
- > Marc Vincent, Head of Technology & Cloud chez Daveo
- > Kamal Zerta, AWS solutions architect chez Daveo

Et des remerciements tout particuliers pour le soutien sans faille d'Amazon Web Services : Stephan Hadinger et Jeremy Drillon pour ne citer qu'eux, ainsi que l'équipe Daveo : Cyril Harpoutlian, Thibault Gonce, Eléa Cygan, Géraldine Seuleusian et Clara Vallino pour son accompagnement créatif.

Serverless : prêt à sauter le pas ?

Contactez-nous pour savoir comment Daveo vous accompagnera de façon personnalisée.

Cyril Harpoutlian

Directeur associé

cyril.harpoutlian@daveo.fr

01 73 78 14 40

Bordeaux

Espace Mama Works
51 Quai Lawton 33300

-

Lille

6 Rue Jean Roisin
59000

Lyon

4 Place Amédée Bonnet
69289 Lyon Cedex 02

-

Nantes

27 Rue du Calvaire
44000

Paris

19 rue du Quatre Septembre
75002

daveo
daveo.fr

Rejoignez notre communauté !

