COMPLEMENT

Ayant fini la réalisation du pipeline plutôt que prévu, j’ai pu faire d’autre tâches dans l’entreprise. La partie qui suit est un aperçu global et chorologique des tâches supplémentaires qui m’ont été conféré.

1. BASCAC – Blu Age (J’ai oublié la suite).

BASCAC est une petite application web qui permet de simplifier la gestion de la facturation de Velocity au client. Celle-ci s’appuient sur la base de données client (identique à la base de données qu’utilise le pipeline pour récupérer les ACL).

L’application permet entre autres, d’ajouter/retirer la permission à un client d’utilisé Velocity, ou encore de géré le pourcentage de réduction accorder au client. En conséquence, son accès doit être limité car elle gère des données de facturation.

L’objectif de cette tâche est de mettre en ligne BASCAC de manière sécurisée, en limitant son accès par un mot de passe. J’ai utilisé l’IDE Visual Studio Code pour y ajouter mes modifications.

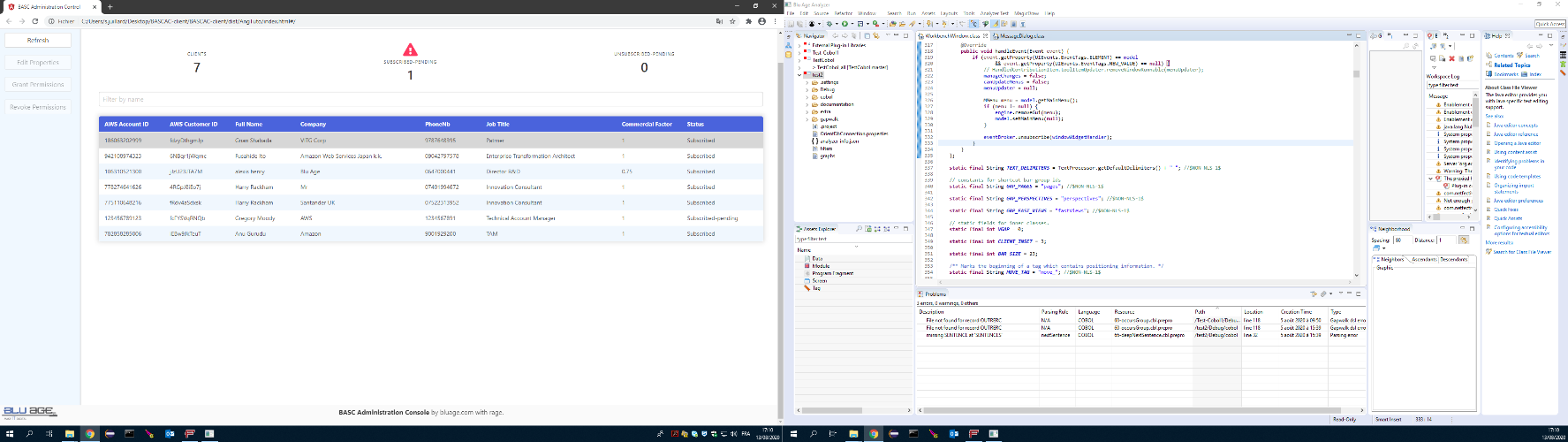
L’outils qui m’as été préconisé est AWS Cogito, un service d’amazone, pour les applications web, qui propose de gérer les mots de passe.

L’application a été réalisé avec Angular, soit codé en TypeScript, html et css. L’application été nativement une “single-page application” ou SPA. J’ai dû rajouter à cette application un mécanisme de routage afin d’y ajouter une page d’authentification.

Une fois cela terminer, j’ai essayé d’intégrer cogito à ma page d’authentification. C’est à cette étape que j’ai pris connaissance de Amplify. Un autre service d’amazone qui permet de mettre en ligne des applications. Pour la mise en place de Cognito, son utilisation été nécessaire.

En me renseignant plus sur cet outil, je me suis aperçu que je pouvais déployer BASCAC, après l’avoir compilé, de manière non-programmatique sur la console graphique d’Amazone. Puis une fois cela fait, ajouter la gestion des mots de passe, qui apparait comme un pop-up en haut de la page. (Cf. image)

Au final, je suis revenu sur mon travail d’ajout de page d’authentification et d’utilisation de Cognito et j’ai opté pour un déploiement et une gestion des mots de passe non-programmatique, à l’aide d’amazone.

N.B : Sur demande de son créateur, j’ai également modifié légèrement le style de la page en ajoutant du contenu aux feuilles de styles de l’application.

2 – Exportation de la Vue Problème de Blu Age Analyzer.

Analyzer est un produit de Blu Age de la V7 <ALORS LA J’AI BESOIN D’EXPLIQUATION POUR SON USAGE>.

Lorsqu’un client rencontre des problèmes avec son code, il peut contacter Blu Age afin de recevoir de l’aide. On lui demander alors le contenu de l’onglet « problème », l’onglet dans lesquels est affiché les erreurs et les warnings issue de l’a compilations du/des projets en cours.

L’objectif de cette tâche est d’ajouter, dans l’onglet problème une option pour exporter le contenu de la vue au format .csv (tableurs).

Blu age Analyser est une IDE développé sur la base d’Eclipse. Elle possède à la fois des onglet, vues et fonctionnalités développées par Blu Age et d’autre issue de Eclipse.

En xml, on peut ajouter des éléments graphiques à Analyzer, et les liés avec des handlers, commandé en java.

La difficulté de cette tâche réside dans l’adaptation de code interne à Eclipse. L’onglet « problèmes » d’Analyzer, est une vue dont le code est interne et il est donc difficile d’en récupérer les éléments. Pour cette raison, il n’est pas possible de récupérer le contenu de la vue problème à proprement parlé. Pour récupérer ce contenu, il a fallu aller à la source de ce contenu : On va récupérer les informations « à la source », soit dans les markers d’un projet. Un marker est un ensemble de donnée généré dans notre cas à la racine du projet dans lesquels les informations relatives à celui-ci son stocker.

C’est en récupérant ces informations que j’ai pu reconstituer la vue problème, puis l’exporter sous le format csv. Après son implémentation, j’ai pris connaissance d’une option similaire, qui avait déjà été implémenter, mais qui avais un usage différent. Cette option proposait d’exporter, à la racine du projet, dans plusieurs formats possibles, les problèmes relatifs à un projet en faisant une clique droite sur celui-ci dans la vue navigation.

J’ai donc dans un deuxième temps, adapter ma solution pour quelle s’intègre au travail déjà effectuer. Ainsi, sans retirer l’option déjà implémenté, j’ai ajouté à la vue problème une option qui propose d’exporter la vue problème (problèmes de tous les projets ouvert), dans plusieurs formats différents, dont le csv, et de l’enregistre par un menu contextuel en choisissant le nom et l’emplacement du rapport d’erreurs.

Une fois la tâche terminer, j’ai pris connaissance des protocoles de Blu Age afin de manager le suivit des commit. Avec l’aide de Oliver <Nom de famille> j’ai revu l’écriture de mon code, j’ai rempli mon message de commit avec un identifiant qui correspondais à ma tâche, puis on a rempli ensemble cette fiche pour présente l’option implémenté.

Conclusion :

La forme finale du pipeline a pu être développé jusqu’au bout. Tous les aspects de la chaîne ont pu être traité : De la gestion des erreurs en passant par les notifications, la récupération des objets nécessaires au déploiement (fichier ACL, fonction teste), ainsi que la documentation de la chaîne.

Le dernier aspect qui pourrait éventuellement manquer est la réalisation de teste unitaire pour vérifier le bon fonctionnement du pipeline. La fiabilité du pipeline n’a à se jour qu’était tester par des teste manuellement fait au fur es à mesure de son développement. A cause de problématiques de droit d’accès, mon pipeline est encore en stade de prototype (elle n’a pas été paramétré pour déployer les vrais fichiers de Velocity) mais tout à été mise en place pour que ce travail soit simple. Une documentation détaillée de 14 pages, en anglais é été crée dans cette optique. De plus, j’ai porté une attention toute particulière à la clarté de mon code.

Les 4 tâches supplémentaires que j’ai pu effectuer j’ai pu toute les terminer également. J’ai pu push mon travail sur le git de Blu Age. Parfois, mes push on crée des problèmes en particulier au niveau des importation de packages, lorsque j’ajouter des nouvelles dépendances. Mon code à également pu être revu par les personnes qui m’encadrées.

D’un point de vu personnelle, ce stage m’a permis de prendre en compétence, et ce dans diverse domaine : devOps, devWeb et dev logiciel. J’ai codé dans plusieurs langages différents, utilisé et tester divers outils. Cette expérience m’as permis de développer ma capacité d’adaptation et ma flexibilité. Grâce au revus de code, j’ai eu également appris des bonnes pratiques de développement que je ne connaissais pas.

J’ai dû moi-même concevoir des solutions, chercher par moi-même les outils qui pourrais me servir et faire le choix des plus adéquats. Pour la tâche d’implémentation d’une fonctionnalité d’export, j’ai pu me familiariser avec les problématiques de la programmation d’un grand projet, ou plusieurs acteurs différent commit du code. Ce que l’on ne rencontre pas forcément dans un projet en milieu scolaire. J’ai aussi du crée de la documentation pour le pipeline que j’ai créée, pour qu’elle puisse être reprise plus tard. Enfin, j’ai eu l’immense satisfaction de terminer mon travail et d’en être fière.

1. Plugin Blu Age Cobol

Afin de permettre à un développeur COBOL de compiler leur application dans un autre langage, Blu Age met disposition de leurs client un Plugin VSC. Ce plugin ajoute à l’IDE plusieurs commandes pour compiler leurs codes, par l’intermédiaire d’une requête http.

Plusieurs types d’opérations sont proposées par l’extension : D’une part, la compilation du code Cobol vers java, et d’autre part, la compilation de fichiers BMS uniquement, afin de générer une application web. Le fichier jar produit, issue de la compilation par l’extension, pourra ensuite être exécuter sur une fonction Lambda d’Amazone ayant une Layer contenant le Framework Velocity (c’est la layer dont nous avons construit la chaine de déploiement). (cf. contexte technique).

Lorsque le client appelle une de ces commandes sur Visual, une phase de vérification, va au préalable empêcher la compilation si le fichier sélectionné n’est pas du cobol, et ce, peut-importe la nature de la commande invoquée. Or, pour la compilation d’un fichier BMS, la présence d’un fichier Cobol n’est pas nécessaire.

L’objectif de cette tâche est de séparé les phases de vérification pour qu’elle soit adapter au type de compilation invoqué. La modification un fois effectué, j’ai généré à partir du code un fichier ‘vsix’ ce fichier pourra permettre à un utilisateur d’ajouté l’extension.

1. Plugin Blu Age Cobol

Les application CICS  (Customer Information Control System) sont des systèmes qui permet d'effectuer des [opérations transactionnelles](https://fr.wikipedia.org/wiki/Transaction_informatique) (en général consultation ou mise à jour de [bases de données](https://fr.wikipedia.org/wiki/Base_de_donn%C3%A9es) ou de [fichiers](https://fr.wikipedia.org/wiki/Fichier_informatique)). Généralement codé en COBOL, c’est couramment ce type d’application qui doit être transcrite par l’extension de Blu Age. En plus des fichiers COBOL, des fichiers bms, pour l’apparence de l’application et un fichier textuel de description de ressource sont généralement présent. Lors de la compilation, ces fichiers doivent également être traité. Entre autres, le fichier textuel de description de ressource est parsé pour en une suite de requêtes SQL qui permet de former une base de données lié à l’application.

Au moment de la compilation, cette opération n’est pas visible pour l’utilisateur de l’extension. Et il n’a pas accès à la suite de requête SQL qui permet de crée la base de données.

L’objectif de cette tâche est d’ajouter une commande à l’extension Blu Age Cobol afin de récupérer un fichier SQL généré à partir d’un fichier textuel de description de ressources.

Tous d’abord, coté frontend, une commande nommée « Blu Age parsing CSD to SQL » a été créé en type script avec l’IDE Visual Studio Code. Lorsque celle-ci est invoquée, le fichier CSD sélectionné va être transmis au server COBOL par le biais d’une requête http.

Puis côté server, la requête est traitée : A l’aide des outils déjà présents, le fichier CSD réceptionné est traduit en SQL, zippé, puis renvoyé au client. (Développement en java, avec Eclipse).

Ensuite, à nouveau côté client, la réception des réponses du server a été adapter pour que le fichier soit récupéré correctement.

Le développement des mécanismes présenter a évidemment été inspiré des commandes de compilations Blu Age déjà présente. Mon travail à été d’ajouter à une architecture déjà présente, une fonctionnalité supplémentaire.

Le choix de cette solution est motivé d'un part, par sa flexibilité, mais également pour son cout très faible comparé aux autres solutions proposées. Enfin, l'atout de cette méthode comparativement l’utilisation de Jenkins par exemple est que l'on utilise quand même des outils Amazone, et les opérations sont effectuer de manière interne au compte amazone de l'entreprise. En d’autres termes, on élimine la nécessité de devoir se logger et donc de transmettre des identifiants ni en claire. Cette solution propose le meilleurs compromis entre sécurité, flexibilité et cout.

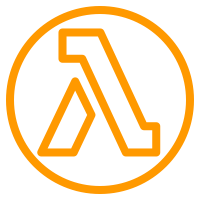
# Aperçu global du prototype

Le déploiement démarre lorsque la chaine d’intégration se termine. A l'issue de ce traitement, la nouvelle version du Framework, sous forme de jar va être chargé dans le Bucket Source (cf fig). Ce fichier, c'est celui indiquer sur le schéma sous le nom de "Layer File". Cette action va alors déclencher la chaîne de déploiement. Une fonction que l'on nomme "Main Pipeline", exécuté sur une Lambda va alors récupérer le fichier, publié la Layer, la tester, et ajouter les permissions aux compte AWS qui doivent avoir accès aux Framework. Toute ces opérations sont effectuées dans la région par default (canoniquement, en Virginie du Nord us-east-1). Une fois le déploiement de la Layer effectuer dans la région par défaut, et si l’opération c’est déroulé correctement, alors le déploiement sur toute la région pourra commencer.

La fonction "Main Pipeline” avant de se terminer, va déclencher de manière asynchrone (C'est à dire simultanément, comme des threads) N fonction lambda nommé "Déployer". Chacune avec comme argument une région différente. Leurs rôles, effectué le déploiement de la layer dans la région qui leurs a été attitré. Dans la région qui leurs ai assigné, les fonctions vont créer une copie du contenu du Bucket Source dans un bucket temporaire, pour pouvoir ensuite reproduire les étapes de déploiement (publié, tester, ajouter les permissions).

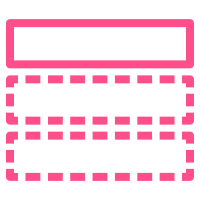
Les étapes de déploiement, étant commune à la fois de la fonction Main Pipeline et Déployer seront effectué par une autre Lambda lancé de manière synchrone par leurs mères ; ainsi on évite de dupliquer du code.

**Schéma fonctionnelle de l’utilisation « serverless » du Framework Velocity**



***Fonction Lambda***

(environnement d’exécution de l’application)



***.jar***

(Issue de la compilation par l’extension Visual Studio code)

***Layer Velocity***

(dépendances)

***COBOL***

**Schéma fonctionnelle de la compilation par l’extension Blu Age Cobol**

Client utilisant Visual avec l’extension

Server de Blu Age

***Copie du Projet COBOL***

***Projcet.jar***

Compilation

Requête https POST

Attente d’une réponse

***Projcet.jar***

***Projet COBOL***

Attachement : project.zip

Réponse https

Attachement : project.jar

# Contexte Technique

La solution serverless de Velocity a pour but de procurer au client une solution pour déployer, sur une Lambda Amazone, une application java initialement codées en COBOL.

Elle est constituée de deux produits : d’une part un compilateur COBOL vers java, déployer sous forme d’une extension Visual Studio code. Cette extension permet au client de généré un fichier jar à partir de son projet Cobol.

Ce fichier contient le code minimum de l’application : Il ne contient pas les librairies nécessaires à son fonctionnement. Ces dépendances sont fournies par le second produit, le Framework Velocity de Blu Age. Ce Framework est déployé sous Amazon Web Services en tant que « Layer ».

Ainsi, pour déployer son application, le client devra crée une Lambda, a laquelle il ajoutera le Framework Velocity en tant que Layer, puis il pourra y exécuter le fichier jar fournit par l’extension VSC.

# L’Environnement Amazon

Amazon Web Services est une plateforme qui propose des services informatiques à destination des entreprises comme des particuliers. Elle propose un grand nombre de services, pour stocker, manager, et déployé des données et des applications. En particulier, cette plateforme est spécialisée dans le Could Computing.

Dans les parties précédentes, nous avons mentionné certain des services qu’il propose :

Tous d’abord, les Lambda AWS. C’est l’environnement d’exécution sur laquelle le client pourra déployer son application. Une Lambda est un service qui fournit un environnement d’exécution de code qui ne nécessiter ni mise en services, ni gestion de server. C’est le fournisseur d’accès (c-a-d Amazon) qui se charge de l’administration des ressources, entre autres, la maintenance des servers, le dimensionnement et la mise l’échelles de capacité, la surveillance et la journalisation des exécutions. De plus quand la Lambda et appelée plusieurs fois, des fichiers temporaires sont conservés permettant ainsi d’économiser sur le temps de préparation de l’environnement d’exécution.

Pour l’utilisateur, ce service permet de déployer simplement n’importe quel type d’application en étant facturer uniquement pour le temps de calcul utilisé. Lorsque la fonction Lambda ne s’exécute pas, l’utilisateur débourser rien.

Ce service d’Amazon et également très adapter à la gestion des autres ressource et services que Amazon propose.

Ensuite les Layers. C’est une couche supplémentaire que l’on peut ajouter à une Lambda. : Une couche est une archive ZIP qui contient des bibliothèques, un [environnement d’exécution personnalisé](https://docs.aws.amazon.com/fr_fr/lambda/latest/dg/runtimes-custom.html) ou d'autres dépendances. Elle permette de crée un runtime personnalisé. Ainsi, il n’est plus nécessaire d‘inclure les bibliothèques utilisées par la fonction Lambda dans le package de déploiement.

Pour déployer un Framework sur une Layer, une pratique possible (et c’est celle que nous allons utiliser pour la chaîne de déploiement), est de stocker les fichiers de la Layer sur une unité de stockage d’Amazon (un bucket) qui se trouve dans la même région que la Layer, puis importer le code de la Layer depuis ce bucket.

Enfin, ces unités de stockage appelé bucket que nous allons utiliser : Un bucket est un compartiment de stockage fournit par le service Amazon Simple Storage Services (Amazon S3). Un compartiment permet de stocker des Object ; un objet est la somme d’un fichier et de tout le méta data qui le décrive.

En résumé, le flow d’exécution du pipeline est codé par 3 fonctions Lambdas codé en python. Grâce à la bibliothèque d’Amazon Boto 3, ces fonctions peuvent interagir avec les autre ressources et services que propose Amazon.

La première fonction principale déclenché par le déploiement d’une nouvelle version de la Layer sur un bucket auxquelles celle-ci est relié, déclenchera à son tour les autres fonctions.

Additionnellement, une fonction Lambda codé en Java devra être charger de tester la Layer (fonction que je n’ai pas codée, elle est matérialisée dans le prototype par une fonction qui ne fait rien). Et enfin, une dernière fonction qui sera en charge de géré les journaux d’exécution et l’envoie des notifications du statue du déploiement. Cette dernière permet de centraliser les informations des différent maillions de la chaîne qui s’exécute indépendamment et en parallèle.

## Paramétrage des foncions.

Tous les paramètres susceptibles de devoir être modifié sont entré sous forme de variable d’environnement de la fonction Lambda principale. Voici la liste des paramètres modifiables :

* SOURCE\_BUCKET : Nom unique du bucket source.
* LAYER\_NAME : Nom de la Layer que l’on souhaite donner à la future Layer
* FCT\_NAME : Nom de la fonction en charge de tester la Layer
* ACCOUNT\_ID : ID du compte sur laquelle d’exécute la Pipeline
* FILE\_NAME\_REQUIRMENT : Règle sur le nom du fichier qui permettra de filtré les fichiers qui déclencherons effectivement la pipeline.
* FILE\_EXTENTION : Règle sur l’extension du fichier (soit zip soit jar pour qu’elle soit reconnu en tant que Layer).
* DEPLOYER\_NAME : Nom de la fonction de Déploiement
* PUBLISHER\_NAME : Nom de la fonction Publisher
* region\_list : Liste des régions sur lesquelles on souhaite déployer la Layer.
* TOPIC\_SNS : Nom du topic ‘SNS’ sur lequel envoyer les notifications.

Prérequis pour le fonctionnement du pipeline : La pipeline requière l’existence de toute les ressources mise en paramètre (bucket source, topic SNS, fonction Lambda de teste) ainsi que la présence de la fonction lambda de teste dans toutes les régions où l’on souhaite déployer la Lambda.

Le bucket source doit également contenir une fichier ACL.json (pour la gestion des permissions).

Ce fichier d’ACL (Access Control) devra suivre le Template fig[ ??]. Il servira à l’administration des permissions de la layer.

## Journaux d’exécution

Par défaut, Amazon stock les journaux d’exécution à travers un service nommée Cloud Watch. Le principal inconvénient de ces journaux pour notre pipeline multifonctions est que les logs de chaque fonction se retrouve séparé les uns des autres. Pour faciliter la lecture, un système de journaux qui centralise les logs de toutes les fonctions de la chaine à été mise en place. Dans le bucket source va se trouver un dossier Log (ou crée s’il n’existe pas déjà). A chaque exécution, un nouveau fichier texte contenant les logs est créé. Pour se différentier des autres exécutions, le nom de se fichier est l’heure et la date, à la milliseconde près, du début de l’exécution de la chaîne.

Cette information d’horodatage est générée par la 1er fonction Lambda (au démarrage) et est transmise en argument à toute les autres fonctions invoquées.

Une fonction Lambda que l’on nommera Notification-Manager sera chargé de gérer l’écriture des Logs. Lorsqu’une des fonctions voudra écrire dans les logs, il invoquera cette fonction avec en paramètre l’information d’horodatage mentionné plus tôt ainsi que le message et sa nature (Erreur /warning/info).

## Système de Notification

Le service SNS (Simple Notification Service) d’Amazon fournit un système de messagerie pour la communication A2P (Application à personne), ou system à système. Le service est basé sur un système de publication/Abonnement : un « topic » est créé, la/les applications/ micro-services lié au topic vont push les informations sur le topic. Les utilisateurs peuvent s’abonner à un topic et récupéré ces informations.

La fonction Notification-Manager (celle mentionnée dans la partie précédente pour la gestion de journaux d’exécution) est aussi chargé de la gestion des messages de notifications SNS.

Un message SNS est publié pour chaque layer déployer avec succès ou lorsqu’une erreur se produit

Le schéma ci-dessous montre comment est managé la gestion de l’envoie des messages pour que même dans le cas où une erreur non-intercepté se produise, un message SNS soit tous de même envoyé, ou pour éviter la répétition des messages.

<Schémas>

Dans le cas où l’erreur n’as pas été intercepté par la Lambda elle-même, et qu’elle n’est pas en capacité d’invoqué. C’est grâce au mécanismes mise en place par Amazon que l’on envoie un message SNS généré automatiquement. Ce mécanisme est identique à celui qui lie le bucket source avec la 1er fonction du pipeline et est configurable dans la console d’aws (cf fig).

## Gestion des ACL.

Pour éviter de devoir éditer à la main le fichier ACL.json, une fonction lambda annexe a été créé, afin de gérer automatiquement la mise à jour ce fichier. Blu Age stock les informations de ces clients dans une ressource Amazon sous forme de base de données : DynamoDB.

La fonction ACL Conversion a pour but de convertir la base de données en un fichier ACL.json.

Cette fonction est déclenchée par la modification de la base de données. Elle effectue une requête pour récupérer la liste des clients ainsi que leurs identifiant, puis édite le fichier ACL.json à partir de ces informations. Enfin, elle charge se fichier sur le bucket source donnée en paramètre (en tant que variable d’environnement de la fonction).

## Génération de bucket temporaire

Un bucket doit avoir un nom unique. Si un bucket porte le même nom il sera alors impossible de le crée. Le nom d’un bucket temporaire. Le nom d’un bucket temporaire est généré de la manière suivante : [SOURCE\_BUCKET] -[REGION]-temporary. (Ex : in us-east-2 🡪 Test-Pipeline--Layer-us-east-2-temporary). Si ce nom est déjà pris, une fonction implémentée de manière récursive sera chargée de trouver un nom disponible en générant une chaine de caractère aléatoire et de taille variable qui sera ajouter au nom par défaut.