CI/CD AWS + Serverless Framework

Análisis de arquitectura alternativa

He decidido realizar la arquitectura alternativa teórica, realizando el cambio tal y como aparece en la siguiente tabla:

Elemento	Actual	Alternativa
Despliegue infra	Manual	Terraform
Control de código	Github	AWS CodeCommit
Orquestación CI/CD	Jenkins	AWS CodePipeline
Build	SAM + Jenkins	AWS CodeBuild + Serverless Framework
Unit Test	unittest/PyTest	Hypothesis
Coverage Test	coverage	coverage.py
Quality Test	flake8	Pylint + Flake8
Security Test	bandit	Snyk
Complexity Test	radon (PyPI)	-
Deployment APPlication Infra	SAM + Jenkins	AWS CodeBuild + Serverless Framework
Integration Test	unittest/PyTest	Serverless Framework
Aplicación	AWS Lambda	AWS Lambda
Backend	DynamoDB	Aurora Serverless
API	API Gateway	API Gateway + Lambda auth + AWS WAF
Monitorización	CloudWatch	Dashbird

Despliegue de Infraestructura

Para dejar de hacerlo manualmente, me he decantado por el uso de **Terraform** para realizar el **despliegue de los servicios** y sus servicios dependientes (Buckets S3, roles y permisos en IAM, etc.) a través del uso de infraestructura como código.

Terraform es Open-Source y uno de los líderes de mercado en el aprovisionamiento de infraestructura, es fácil de configurar, tiene capacidad para funciones complejas, permitir la realización de test automatizados y es muy modulable gracias a los providers (plugins) para todo tipo de servicios.

Una de las principales **ventajas** de usar Terraform es la **capacidad de generar módulos**. Los módulos, en Terraform, son contenedores de múltiples recursos (resource) que son usados de manera conjunta, y consiste en una colección de archivos .tf y/o .tf.json contenidos en un mismo directorio. Dichos módulos son reusables y pueden (y deben) ser usados para crear unidades ligeras, de modo que se pueda describir la infraestructura según su propia arquitectura y no en términos de objetos físicos.

Una de las posibilidades que nos deja Terraform es que a través de su CDK (CDKTF), que usa el Cloud Development Kit de AWS, podemos usar cualquier lenguaje de programación que permita el CDK de AWS para definir la infraestructura.

Una de las **desventajas** que puede tener el uso de Terraform es que los archivos de configuración están escritos en HCL, que es un lenguaje declarativo específico con sintaxis de bajo nivel.

Añadido a Terraform, he elegido usar terratest para realizar **test de integración** de infraestructura automatizados al código generado por medio de Terraform.

CONTROL DE CÓDIGO

Como sustituto para el control de código con GitHub me he decidido por el servicio de AWS **CodeCommit** que podemos usar para realizar el control de versiones desde AWS, ya que vamos a realizar el proceso de CI/CD completamente desde servicios de AWS.

La ventaja con respecto al uso de GitHub es que tenemos el repositorio en el mismo entorno que el resto del proceso, por lo que las peticiones provenientes de otros servicios de AWS a dicho repositorio se podrán hacer directamente por una VPC, sin necesidad de pasar por el "exterior". También dejamos de tener un repositorio público, y mientras no pasemos de 5 usuarios activos no supondrá un coste adicional.

ORQUESTACIÓN CI/CD

Como orquestador, en sustitución de Jenkins y siguiendo con la suite de AWS, tenemos a nuestra disposición **CodePipeline**. Este servicio nos va a permitir controlar el despliegue de todo el proceso CI/CD.

Lo interesante de CodePipeline con respecto a Jenkins, es la sencillez de su configuración y la facilidad para configurar nuevas 'steps' con servicios independientes según necesidad.

Uno de los beneficios del uso de CodePipeline respecto a Jenkins puede ser el factor económico, puesto que no es necesario levantar una instancia EC2, sino que podemos ejecutar este servicio serverless de manera independiente y solo nos van a cobrar cuando dicho servicio realice una ejecución.

Para poder ejecutar cada pipeline cuando se realiza un merge request, se usa Cloudwatch para encontrar dicho evento que será el desencadenante.

BUILD

En sustitución del uso de SAM + jenkins para la etapa de Build he decidido usar **Serverless Framework + AWS CodeBuild**.

Serverless Framework es un servicio de aprovisionamiento de infraestructura que también puede trabajar como FaaS, es Open Source e independiente de la plataforma Cloud que usemos, usa sintaxis YAML y su integración en procesos CI/CD es sencilla.

Para mejorar la implementación y la fiabilidad al ser usado en entornos de AWS, Serverless Framework convierte su configuración a formato template de CloudFormation lo cual puede ser una ventaja para facilitar la integración con otras etapas o servicios.

Una vez entendido Serverless Framework vamos a realizar su ejecución desde **CodeBuild** que previa configuración de buildspec.yml realizará la ejecución de los comandos necesarios para las dependencias y el proceso de compilación (serverless package).

TEST UNITARIOS Y ESTÁTICOS

Con la ayuda de **CodeBuild** también podemos realizar los test unitarios, de cobertura, calidad y seguridad ya que permite la ejecución de servicios de terceros siempre que lo definamos en el archivo de configuración buildspec.yml para que sean instalados y ejecutados respectivamente:

- **Unit Test** - Los test unitarios son dependientes de cuál es el lenguaje de programación usado en el código que va a ser probado, por ello para sustituir a Unittest he elegido a Hypothesis.

Se trata de una librería de testing avanzado en Python que usa property-based testing. Esto funciona generando datos arbitrarios que concuerdan con tus especificaciones y comprueba que la garantía se cumple en ese caso. Si encuentra una parte del código donde no se cumple, coge esta parte y la reduce, simplificandola hasta que tiene un trozo de código más pequeño que esté dando el problema. Entonces guarda ese trozo como ejemplo para más tarde, de manera que cuando encuentra un problema en tu código no lo va a olvidar en el futuro.

Esto hace que sea más sencillo tener una biblioteca de ejemplos de fallos que nos permitan ahorrar tiempo cuando en el futuro se produzca el mismo error.

- **Coverage Test** - Los test de cobertura van ligados al test unitario que se esté ejecutando, y si bien Hypothesis permite llevar un control de la "salud" del código, no tiene una implementación de coverage óptima, por lo que lo más seguro es seguir usando coverage.py.
- **Static Tests**
    - **Quality Test**: Para complementar a flake8 he encontrado Pylint, que permite analizar el código sin necesidad de ejecutarlo, hace check de errores y realiza sugerencias sobre código que podría ser refactorizado. Es muy configurable y permite escribir plugins para añadir tus propios checks.
    - **Security Test** - Para buscar problemas de seguridad me he decidido por **Snyk** que busca y arregla automáticamente vulnerabilidades en tu código, dependencias, contenedores e IaC. Realizando procesos de testing, con la configuración que deseemos, por ejemplo: snyk test --severity-threshold=medium , y creando un nuevo proyecto de monitorización snyk monitor para poder verlo cómodamente desde la consola de Snyk.

DEPLOY

No he usado CodeDeploy ya que este servicio está orientado al despliegue de aplicaciones en instancias (EC2, etc.) mientras que lo que nosotros queremos es realizar un despliegue de infraestructura serverless junto a la aplicación contenida en un bucket S3. Por ello realizo el Deploy en una nueva etapa de CodePipeline usando **CodeBuild + Serverless Framework**, lo cual me permitirá con el uso del comando serverless deploy realizar el despliegue de la aplicación previamente compilada en la etapa "build".

Serverless Framework se configura en un archivo dedicado denominado serverless.yml, pero en su integración con AWS este archivo se traduce en un template de CloudFormation para facilitar y asegurar que el proceso sea exitoso. Es por esto que el despliegue en AWS se realiza igual que con SAM, es decir, se crea un Stack en CloudFormation y se almacena en un bucket S3.

Serverless Framework tiene la capacidad de elegir en que stage se realizan las acciones. Añadiendo --stage como opción al realizar las acciones de build o deploy y con --package en la fase de deploy podemos ignorar la creación de un nuevo artefacto, ya que serverless lo genera automáticamente cuando realiza el deploy por defecto.

TESTS DE INTEGRACIÓN

Serverless Framework permite realizar test básicos de integración para funciones con endpoints HTTP. Los test están definidos en un nuevo archivo denominado serverless.test.yml y son testeados usando el comando serverless test. Al finalizar el deploy podemos crear una nueva etapa en CodePipeline que lleve a cabo los test de integración por medio de CodeBuild.

BACKEND

Es francamente difícil sustituir un servicio tan sofisticado como DynamoDB, podría realizar una implementación parecida con otras bases de datos NoSQL como MongoDB, pero uno de los requisitos es que sea un servicio Serverless y no hay otro servicio que pueda hacer sombra a DynamoDB, ya que no solo es Serverless y NoSQL sino que la naturaleza de este servicio en AWS, para estar dividido entre varios servidores de manera dinámica, lo convierte en un tipo de base de datos difícil de mejorar.

Pero quizás si la necesidad ficticia de nuestra aplicación se beneficiase de un modelo de base de datos relacional se podría sustituir por una base de datos **Aurora Serverless**. A parte de este cambio de no relacional a relacional, la mayor ventaja de Aurora es que el modelo de pago es por uso, lo cual va más en la línea de una arquitectura Serverless.

Otra gran ventaja de Aurora Serverless es que tiene la capacidad de activar su propia API para permitir realizar peticiones HTTP sin necesidad de conexiones persistentes por TCP y es capaz de manejar las conexiones entrantes por lo que no es necesario indicar limites en las funciones Lambda. Esta API usa credenciales almacenadas en AWS Secrets Manager por lo que no es necesario pasar las credenciales en las llamadas a la API.

API

Para sustituir a AWS API Gateway como servicio API serverless podría haber escogido Application Load Balancer (ALB) ya que también es serverless y permite realizar las funciones de impas de entrada y receptor de solicitudes externas. Pero la integración de API Gateway con los demás servicios internos de AWS, e incluso externos, que hacen uso de HTTP requests es mejor y más sencilla de implementar.

Además, como punto débil de ALB comparado con API Gateway es que para ALB es necesario especificar más de una zona de disponibilidad por región para que alcance un nivel alto de disponibilidad y el coste de los servicios no va por requests sino basado en uso del tiempo y los recursos, lo que lo hace una opción menos factible para proyectos como el de la "To Do List".

Así que al haber decidido quedarme con API Gateway voy a suplirlo con un par de mejoras que podrían ayudar en cuanto a la seguridad y la integración:

Para mejorar la integración de API Gateway sería interesante la creación de una función Lambda que controle el acceso a la API (**Lambda Authorizer**) por medio de autentificación de tipo OAuth, SAML o que solicite parámetros para determinar la identidad del usuario.

También para mejorar la seguridad de la API se puede añadir un servicio firewall como **AWS WAF** como medida de protección contra ataques de peticiones masivas brindando herramientas para controlar el tráfico que recibe la API Gateway.

MONITORIZACIÓN

Para sustituir a CloudWatch he decidido usar **Dashbird**. Con dashbird podemos observar el log completo de nuestras aplicaciones serverless en tiempo real. A nivel de costes puede ser una ventaja que es gratuito siempre que no nos pasemos del millón de ejecuciones Lambda al mes, mientras que Cloudwatch se basa en el número de alarmas o métricas de las que queramos disponer.

Si hablamos de ventajas técnicas en Dashbird con respecto a CloudWatch no vamos a encontrar ninguna especialmente significativa pues, a la hora de realizar la monitorización, ambos cumplen bien. Pero en lo que si está Dashbird por encima es en la facilidad para presentar la información, ya sean métricas, logs, trazas de datos, etc.

Lo que hace dashbird es crear un template de CloudFormation que automáticamente crea un rol en tu cuenta de AWS que le delega solo permisos de lectura a los servicios que haya en tu cuenta. Es este rol el que va a usar para conseguir métricas, logs y trazas de datos.