ULPG

Tecnologías de Servicios para Ciencia de Datos

Git-Radar

Autores:
Jorge Hernández Hernández
Adrian Perera Moreno
Alvaro Juan Travieso Garcia

1 de febrero de 2024

1. Introducción

El desarrollo de software es un proceso fundamental en la era digital actual, donde la eficiencia y la precisión son cruciales para el éxito de cualquier proyecto. En este contexto, surge Git-Radar, una plataforma diseñada para proporcionar análisis detallado y predicciones inteligentes sobre el código fuente.

El propósito de este paper es presentar en detalle el proyecto Git-Radar, desde su arquitectura hasta su despliegue en la infraestructura de Amazon Web Services (AWS). Este sistema se centra en ofrecer métricas de código, así como en entrenar modelos predictivos para sugerir nombres de funciones, mejorando así la productividad y la calidad del desarrollo de software.

A lo largo de este documento, exploraremos los componentes clave de Git-Radar y su integración con AWS, destacando cómo esta plataforma puede transformar el proceso de desarrollo de software al proporcionar herramientas avanzadas de análisis y predicción de código.

2. Metodología

En este apartado, detallaremos la metodología utilizada para el desarrollo e implementación de Git-Radar, la plataforma innovadora diseñada para análisis y predicción de código. Exploraremos los enfoques y técnicas empleados para la construcción de la infraestructura en AWS, la integración de datos, el entrenamiento de modelos predictivos y la implementación de funcionalidades clave. Además, discutiremos los procesos de evaluación y validación utilizados para garantizar la efectividad y confiabilidad de la plataforma. La metodología adoptada se basa en principios de buenas prácticas de desarrollo de software, así como en técnicas avanzadas de análisis de datos y aprendizaje automático. A lo largo de este apartado, se proporcionará una visión detallada de los pasos y decisiones tomadas durante el desarrollo de Git-Radar, destacando los desafíos encontrados y las soluciones propuestas.

2.1. Herramientas

Cloud building

Para la construcción de la infraestructura se uso el software terraform, el cual permite tratar la infraestructura como codigo. Mediante un lenguaje declarativo basado en json. Se declaran los distintos recursos que se desean construir. Un recurso es la unidad basica de ejecución en terraform, pudiendo ser desde un bucket s3 hasta una regla de cloudwatch. Terraform permite modularizar la creacion de la infraestructura, agrupando conjuntos de recursos para que se creen a la vez, compartiendo algunas configuraciones.

Local

Dado que el proyecto se ha desarrollado simulando la plataforma AWS en local se han usado herramientas consecuentemente. Para poder simular el entorno se uso Localstack, un emulador de AWS que permite simular el despliegue de una arquitectura serverless en local. Esta usa docker para correr y ejecutarse a traves de varios contenedores.

Existen 2 wrapers para poder interactuar más facilmente con localstack. Estos permiten usar herramientas que normalmente interactuan con AWS para que lo hagan con localstack. Aunque el mismo efecto se puede obtener con las herramientas originales, se tendrian que configurar con cuidado lo que puede producir errores facilmente. Además de que a la hora de usarlas de nuevo en la nube tendrían que revertirse las configuraciones. Con el uso de los wrapers podemos simplificar el proceso.

- tflocal: Wraper para terraform, para que actue sobre localstack directamente sin más configuraciones

- awslocal: Wraper para el CLI de AWS, para interactuar con localstack como se haría con AWS.

2.2. Arquitectura

El diseño arquitectónico del proyecto Git-Radar se ha concebido para ofrecer una plataforma robusta y escalable que proporcione análisis detallado y predicciones inteligentes sobre el código fuente de los usuarios. Esta arquitectura se basa en el uso de servicios clave de AWS, como S3 para el almacenamiento de datos, DynamoDB para la gestión de bases de datos NoSQL, y AWS Lambda para el procesamiento de datos y la ejecución de tareas específicas.

Almacenamiento de Datos

Dos buckets S3 juegan un papel fundamental como datalakes en la arquitectura de Git-Radar. El primero se utiliza para almacenar los códigos fuente proporcionados por los usuarios, mientras que el segundo almacena los modelos ya entrenados. Esta división permite una gestión eficiente de los datos y asegura la disponibilidad de los modelos actualizados para su uso en la plataforma.

Procesamiento de Datos

El procesamiento de datos se realiza mediante el uso de funciones Lambda, que permiten ejecutar código en respuesta a eventos específicos. Git-Radar utiliza cuatro funciones Lambda diferentes, cada una dedicada a una tarea específica:

1. **Tokenización de Datos:** Cuando un nuevo código llega al bucket `code-files` en S3, la función Lambda correspondiente se activa para tokenizar los datos y almacenarlos en una tabla DynamoDB. Este proceso es fundamental para preparar los datos para su posterior análisis.
2. **Generación de Métricas:** La función Lambda asociada a esta tarea se activa cuando un usuario realiza una solicitud de métricas a través de la API. Utilizando el identificador del usuario y el nombre del archivo, esta función recupera las métricas previamente calculadas y almacenadas en DynamoDB, ofreciendo así una respuesta rápida y eficiente al usuario.
3. **Entrenamiento del Modelo:** Para mantener los modelos siempre actualizados con los datos más recientes de los usuarios, Git-Radar programa la ejecución de esta función Lambda todas las noches a las 2 am. Esto garantiza que los modelos estén constantemente adaptados a los patrones de código más recientes, mejorando así su capacidad predictiva.

4. **Predicción:** Cuando un usuario solicita autocompletar un código a través de la API, la función Lambda correspondiente toma el código proporcionado por el usuario y genera una predicción basada en el modelo entrenado. Esta capacidad de predicción en tiempo real permite a los usuarios obtener sugerencias precisas y útiles mientras escriben su código.

Modulos de terraform

Para crear la infraestructura se crearon 6 modulos de terraform. Cada uno crea una parte de la infraestructura:

1. **EventBus y Datalake:** El primer modulo se ocupa de crear y configurar el datalake. Así como el EventBus del sístema. Como parte de la configuración del datalake se configura la regla que copia todos los eventos al datalake
2. **Databases:** Este modulo crea las distintas bases de datos de dynamo y configura sus indices.
3. **tokenizer, metrics, sugester:** Estos modulos crean y configuran la parte de procesamiento de datos. Crean las lambdas y sus roles, crean la politica de loggin y logger de dichas lambdas. Tambien configuran sus reglas de activación. Tokenizer tambien crea el bucket s3 de code-files.
4. **ApiGateWay** Modulo que crea y configura una API REST así como 2 endpoints. Además tiene la configuracion para routear las peticiones a sus respectivas lambdas.

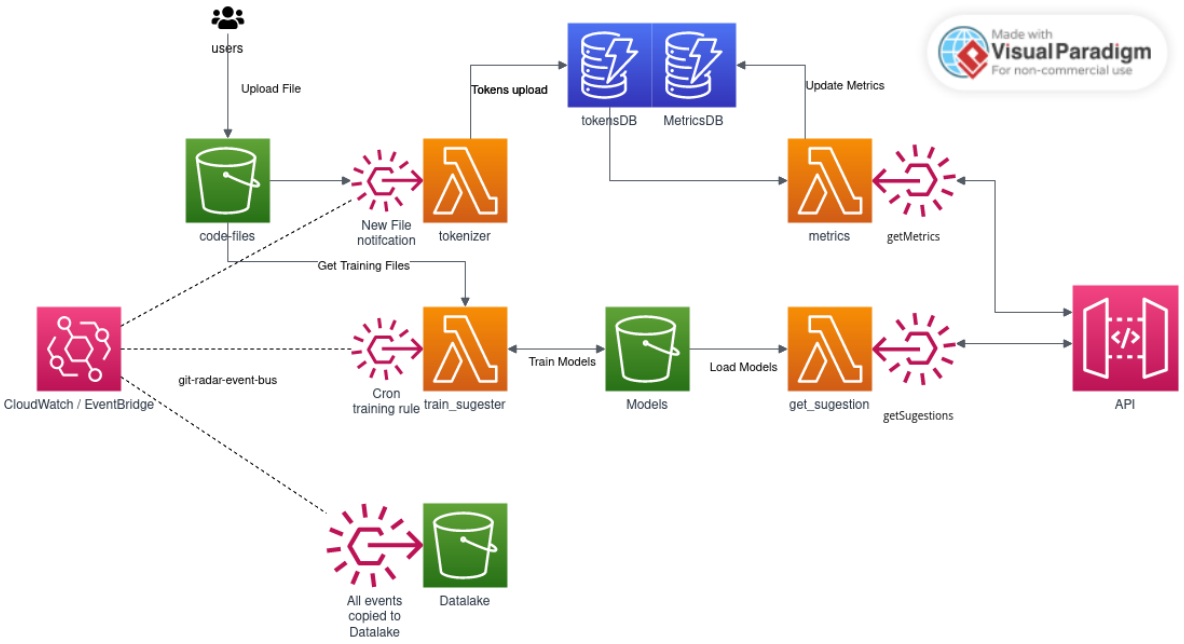

Figura 1: Diagrama de la Arquitectura de la aplicación.

3. Modulos

El programa se basa en la utilización de 4 módulos:

Módulo de Tokenización

Este módulo se encarga de la tokenización del código fuente proporcionado por los usuarios. Utiliza varias clases para llevar a cabo su cometido, siendo la más importante la clase `PyTokenizer`. Esta clase es responsable de realizar la tokenización del código, convirtiéndolo en una secuencia de tokens procesables. Además, el módulo cuenta con otras clases básicas como:

- `S3FileRetriever`: Descarga la información almacenada en el S3 una vez llega la notificación a la Lambda.
- `JSONSerializer y JSONDeserializer`: Se encargan de serializar y deserializar los JSON utilizados durante el proceso de tokenización.
- `JSONData`: Sus variables se inicializan al deserializar el JSON de entrada, facilitando su uso en la Lambda.
- `DynamoDBManager`: Clase encargada de subir el resultado a DynamoDB mediante la función `add_item`.

Módulo de Métricas

Este módulo también cuenta con su respectivo serializador, deserializador y la clase `TokenizedData`, cuya utilidad es similar a la encontrada en el módulo de tokenización. Además, incluye la clase `EventParser`, la cual, con un evento personalizado de nuestra API, extrae la información necesaria para obtener el JSON tokenizado guardado en DynamoDB. A partir de este JSON, se realizan las métricas, las cuales se almacenan en otro DynamoDB a través de la clase `DynamoDBManager`. El cálculo de las métricas se lleva a cabo mediante la clase `PyMetrics`.

Modulo de Entrenamiento

Este código utiliza el módulo trainingDataGenerator para generar datos de entrenamiento a partir de un archivo de código fuente. Luego, utiliza estos datos para entrenar un modelo de máquina de soporte vectorial (SVM) en un clasificador de funciones. El modelo entrenado se evalúa en un conjunto de prueba y se imprime la precisión (accuracy) del modelo. Además, se guarda el modelo entrenado en un archivo con el nombre especificado.

El código consta de tres clases principales: Main, model, y trainingDataGenerator. La clase Main maneja la ejecución principal del programa, creando una instancia de trainingDataGenerator para generar datos de entrenamiento, luego utiliza esos datos para entrenar un modelo de la clase model. La clase model se encarga de la preparación de datos, el entrenamiento del modelo SVM y la evaluación del modelo. Por último, la clase trainingDataGenerator utiliza el módulo ast para analizar un archivo de código fuente y extraer información sobre las funciones presentes en él, devolviendo una lista de tuplas que contienen el nombre y el contenido de cada función.

Modulo de Predicción

Este código en Python define tres clases: loadModel, Main, y predict. El propósito principal del código es cargar un modelo previamente entrenado y realizar predicciones sobre nuevos datos utilizando ese modelo.

La clase loadModel se encarga de cargar un modelo previamente guardado utilizando la biblioteca joblib. Al cargar el modelo, crea una instancia de la clase predict para realizar predicciones sobre nuevos datos utilizando el modelo cargado. La clase predict tiene un método predict que toma los nuevos datos como entrada y devuelve las predicciones realizadas por el modelo.

La clase Main actúa como el punto de entrada principal del programa. En su método main, crea una instancia de la clase loadModel, carga el modelo y realiza predicciones sobre los nuevos datos proporcionados. Luego, devuelve las predicciones.

4. Utilización

4.1. Instalación e Inicialización

Para comenzar a utilizar AWS CLI Local y Terraform Local con LocalStack, sigue los siguientes pasos:

1. **Crear un Entorno Virtual en Coda:** Para mantener las dependencias de tu proyecto aisladas, es recomendable crear un entorno virtual. Puedes hacerlo utilizando herramientas como `virtualenv` o `conda`.
2. **Instalar Paquetes:** Una vez que tengas tu entorno virtual activado, instala los paquetes `awscli-local` y `terraform-local` usando `pip`:

```
pip install awscli-local terraform-local
```

4.2. Despliegue con LocalStack

Para desplegar tus recursos utilizando LocalStack, sigue estos pasos:

1. Navega a la carpeta `infrastructure`.
2. Ejecuta el comando `tflocal init` para inicializar Terraform.
3. Luego, ejecuta el comando `tflocal plan -out "plan"` para planificar tus recursos. Esto generará un archivo de plan llamado `plan`.
4. Finalmente, ejecuta el comando `tflocal apply "plan"` para aplicar los cambios y desplegar tus recursos en LocalStack.

Con estos pasos, deberías poder ejecutar git-radar.

4.3. API URL

La URl de la API tendrá el siguiente formato (Sin endpoint):

`http://<apiId>.execute-api.localhost.localstack.cloud:4566/test/`

Dado que cada vez que se despliegue tendrá un nuevo apiID. Se podrá obtener con el comando:

`awslocal apigateway get-rest-apis`

Este devolverá la información sobre la API, entre ella el id.

5. Future Work

Como trabajo futuro para seguir avanzando y mejorando este proyecto, se identifican diversas áreas de enfoque:

- **Mejora en la Diferenciación de Usuarios:** Implementar una base de datos específica para los usuarios permitiría asignar a cada uno de ellos un identificador único interno. Esto facilitaría la diferenciación entre los usuarios y permitiría personalizar aún más la experiencia de cada uno.
- **Ampliación de Funcionalidades de Análisis:** Explorar la posibilidad de ampliar las funcionalidades de análisis del código, como la detección de patrones de uso, la identificación de posibles mejoras de rendimiento o la evaluación de la calidad del código. Esto proporcionaría a los usuarios una visión más completa y detallada de sus proyectos de desarrollo de software.
- **Implementación de Funciones de Colaboración:** Integrar funcionalidades que fomenten la colaboración entre usuarios, como la revisión de código entre pares, la creación de comentarios y la gestión de tareas. Esto promovería la interacción y el trabajo en equipo dentro de la plataforma, mejorando así la productividad y la calidad del desarrollo de software.
- **Investigación en Modelos de Predicción Avanzados:** Explorar modelos de predicción más avanzados y técnicas de aprendizaje automático para mejorar la precisión y la relevancia de las sugerencias de autocompletado de código. Esto requeriría investigar y experimentar con diferentes algoritmos y enfoques de modelado para encontrar la mejor solución para las necesidades de los usuarios.
- **Optimización de los recursos** Optimizar los recursos de terraform y configurciones. Así como el despliegue de las distintas funcionalidades para minimizar el posible costo de mantenimiento del sístema.

6. Conclusión

En este trabajo, hemos presentado Git-Radar, una plataforma innovadora diseñada para el análisis y la predicción de código fuente. A lo largo de este documento, hemos explorado la arquitectura de Git-Radar, detallando su infraestructura basada en AWS y la implementación de diversos módulos para la tokenización, generación de métricas, entrenamiento de modelos y predicción de código.

Este proyecto no solo nos ha proporcionado una oportunidad invaluable para aplicar y ampliar nuestros conocimientos en desarrollo de software, sino que también nos ha permitido adentrarnos en el emocionante mundo de la infraestructura como código. A

través de la implementación de Git-Radar en AWS, hemos tenido la oportunidad de aprender sobre herramientas como Terraform, que nos ha facilitado la creación y gestión de la infraestructura necesaria para el despliegue y funcionamiento del proyecto.

Además, hemos delineado posibles áreas de mejora y expansión para el proyecto, como la implementación de funciones de colaboración, la investigación en modelos de predicción avanzados y la integración con herramientas externas de desarrollo de software. Estas futuras mejoras podrían llevar a Git-Radar a convertirse en una plataforma aún más completa y útil para los desarrolladores.