ULPGC
Escuela de Ingeniería Informática

Git-Radar: Analítica y Sugerencias Inteligentes para Mejorar el Desarrollo de Código en GitHub

Victoria Torres Rodríguez[1] [1]*victoria.torres101@alu.ulpgc.es*
José Juan Hernández Gálvez[2] [2]*jose.hernandez219@alu.ulpgc.es*
Joel del Rosario Pérez[3] [3]*joel.del101@alu.ulpgc.es*

Git-Radar es un proyecto destinado a potenciar a los usuarios de GitHub al proporcionar métricas en tiempo real para sus repositorios de código cada vez que se carga un nuevo archivo. La herramienta no solo ofrece información sobre la calidad y actividad del código, sino que también incorpora una función de sugerencias. Aprovechando una base de datos almacenada de fragmentos de código del usuario en GitHub, este sugeridor propone la siguiente palabra más probable durante el desarrollo de código a través de una API. Construido sobre la infraestructura de AWS e implementado en Java, Git-Radar mejora la experiencia de desarrollo al proporcionar análisis valiosos y sugerencias inteligentes de código a los usuarios, fomentando prácticas de codificación mejoradas y eficiencia.

Palabras clave: Git-Radar, GitHub, Métricas de código, Desarrollo de software, Sugerencias inteligentes, AWS, Java, Repositorio de código, Análisis de código, Eficiencia de desarrollo

1 Introducción

Git-Radar representa una solución integral para el desarrollo de software, aprovechando GitHub Actions y la arquitectura de AWS. Este proyecto automatiza el análisis de código, proporcionando métricas esenciales como indentación máxima, número de líneas, funciones e imports. Además, incorpora un avanzado suggester de código basado en n-gramas, utilizando fragmentos de código almacenados en la base de datos del usuario.

Con una arquitectura respaldada por servicios en la nube y Docker, Git-Radar asegura escalabilidad y eficiencia. La integración de AWS facilita la gestión de datos, mientras que Docker garantiza la portabilidad. En resumen, Git-Radar redefine el desarrollo de software al ofrecer métricas detalladas y sugerencias contextuales, impulsando la excelencia en el proceso de codificación.

2 Metodología

2.1 Herramientas

En el desarrollo de Git-Radar, se emplearon diversas herramientas que desempeñaron roles fundamentales en la construcción, despliegue y operación del sistema. A continuación, se detallan las principales herramientas utilizadas:

2.1.1 Docker y LocalStack:

Docker fue esencial para la creación de entornos de desarrollo y despliegue consistentes. La extensión de LocalStack para Docker facilitó la simulación de servicios de AWS localmente, permitiendo el despliegue de la infraestructura y la ejecución de los JAR resultantes del trabajo. Esto posibilitó un desarrollo y pruebas eficientes en un entorno controlado y reproducible.

2.1.2 IntelliJ IDEA:

IntelliJ IDEA se destacó como el IDE principal durante el desarrollo en Java. Su potente conjunto de herramientas, depuración integrada y compatibilidad con tecnologías modernas facilitaron la creación y refactorización del código, contribuyendo a la eficiencia y calidad del desarrollo.

2.1.3 AWS (Amazon Web Services):

AWS sirvió como la infraestructura general de Git-Radar. Utilizando servicios como S3 para el almacenamiento de datos, Lambda para ejecutar código sin servidor, DynamoDB para la base de datos NoSQL y CloudFormation para la implementación de recursos en la nube, AWS proporcionó una base sólida y escalable para la aplicación.

2.1.4 GitHub Actions:

La integración continua se gestionó mediante GitHub Actions, una herramienta que automatiza el flujo de trabajo en GitHub. Se configuró un flujo específico llamado `Tokenizer CI` que se activaba en eventos de push y pull request en el repositorio.

Este flujo se ejecutó en un entorno basado en Ubuntu, utilizando las últimas versiones disponibles de las herramientas necesarias. La secuencia de pasos incluyó la configuración del entorno Java con JDK 21, seguido de la construcción y prueba del módulo `Tokenizer` utilizando Maven.

GitHub Actions proporcionó una integración continua efectiva, ejecutando automáticamente pruebas y asegurando la calidad del código en cada confirmación o solicitud de extracción, contribuyendo así a la fiabilidad y consistencia del desarrollo.

3 Arquitectura del Proyecto

El proyecto Git-Radar se sustenta en una arquitectura sólida que utiliza diversos servicios de AWS para gestionar eficientemente los artefactos, códigos y métricas esenciales. A continuación, se detallan los componentes clave de la arquitectura:

3.1 Buckets S3

- **S3 de Artifacts (artifacts):** Almacena los módulos esenciales y archivos de CloudFormation necesarios para el funcionamiento de las lambdas y otros componentes del proyecto.
- **S3 de Códigos (code-files):** Contiene códigos fuente en formato JSON, proporcionando información vital como nombre y contenido del archivo para operaciones de tokenización y métricas.
- **S3 de Métricas (metrics):** Reserva espacio para almacenar métricas obtenidas de los archivos en el S3 de códigos. La Lambda code-metric calcula métricas como el número de líneas e indentación máxima en respuesta a nuevos archivos en este bucket.
- **S3 de Datalake (datalake):** Almacena códigos tokenizados generados por la Lambda tokenizer, activada por notificaciones del S3 de códigos.

3.2 Lambdas

- **Lambda Tokenizer:** Activa la tokenización de archivos de código en respuesta a notificaciones del S3 de códigos, almacenando las versiones tokenizadas en el S3 de Datalake.
- **Lambda Code Metric:** Calcula métricas, como el número de líneas e indentación máxima, en respuesta a notificaciones del S3 de códigos, almacenando los resultados en el S3 de Métricas.
- **Lambda Token-Datamart-Builder:** Activa la construcción de n-gramas a partir del texto tokenizado almacenado en el S3 de Datalake. Los n-gramas se almacenan en una base de datos DynamoDB para proporcionar datos ricos en información para el suggester.

3.3 Base de Datos DynamoDB

Almacena en formato JSON los diversos n-gramas generados por la Lambda token-datamart-builder. La estructura incluye campos como "context," que representa las palabras dentro de la ventana, y "next," que indica la siguiente palabra a completar.

3.4 APIs

Todas las cuestiones que se quieran realizar, se deben llevar a cabo a partir de la api a la que da soporte el módulo token-suggester, tanto como para obtener el próximo token dado una lista de palabras, como para obtener una métrica determinada sobre uno de los archivos procesados. Por tanto, podemos diferenciar dos rutas principales dentro de esta api.

Por un lado, tenemos la encargada de dar al ususario el siguiente token. Para acceder a ella, se debe serguir la ruta */gitradar/token-suggester/next/:word*, siendo word un parámetro de uso obligatorio.

Por otro lado, tenemos la ruta /gitradar/code-metric/:file/:metric, que devuelve el valor de la métrica deseada para el fichero indicado.

Toda esta infraestructura puede verse reflejada en el siguiente gráfico de la Figura 1.

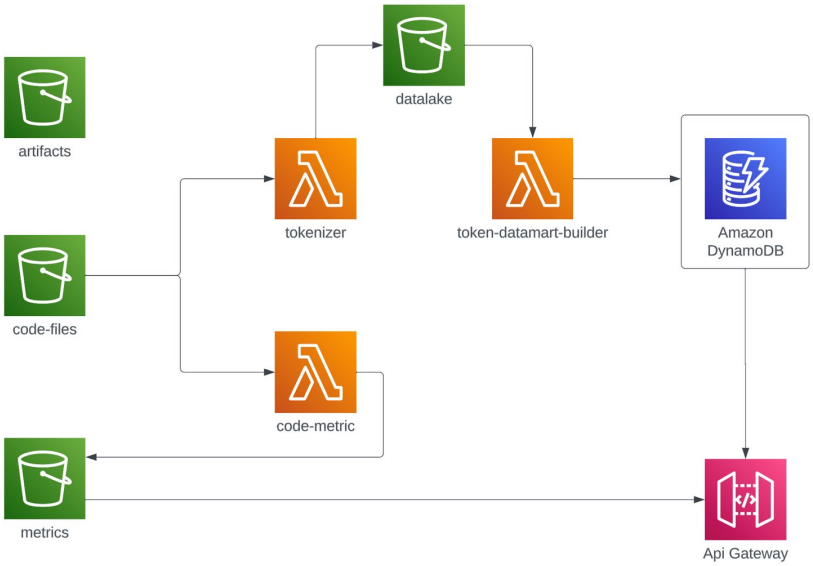

Figure 1: Arquitectura general de Git-Radar

4 Módulos

4.1 Módulo Launcher

El Módulo Launcher es esencial para poner en marcha el entorno de desarrollo local de Git-Radar. Define la configuración del servicio Localstack, una herramienta que simula servicios en la nube, como Lambda, EventBridge, IAM y DynamoDB, de manera local. Este módulo utiliza el archivo `docker-compose.yml` para configurar la imagen de Localstack y exponer los puertos necesarios. Además, incluye un script Bash (`init.sh` y `dynamobdlocalstack.sh`) que crea tablas DynamoDB específicas necesarias para el proyecto.

1. **docker-compose.yml**: Define la configuración de Docker Compose para el servicio de LocalStack, una herramienta que proporciona entornos de prueba locales de AWS. Este archivo establece la imagen de LocalStack, configura los puertos y los servicios de AWS que se simularán, y especifica volúmenes para almacenar datos persistentes. Inicia un entorno local simulado de AWS con servicios como Lambda, eventos, IAM y DynamoDB utilizando LocalStack.

2. **dynamobdlocalstack.sh**: Script de shell que crea tablas de DynamoDB en el entorno de LocalStack. Toma el nombre del contenedor de LocalStack y el número de tablas a crear como parámetros y utiliza la interfaz de línea de comandos de AWS para crear tablas de DynamoDB con nombres específicos y una definición de atributos predefinida.

3. **init.sh**: Script de inicialización para la plantilla de CloudFormation utilizada en el proyecto. No se proporciona el código específico, pero se explica que este script realiza configuraciones y acciones iniciales necesarias para el despliegue del proyecto en la nube.

4.2 Módulo Tokenizer

Este módulo, implementado en Java, tiene como principal objetivo procesar archivos de código después de recibir una notificación de AWS sobre la adición de un nuevo archivo al bucket S3 `code-files`. Su función es devolver un objeto que represente una lista de tokens del archivo recién subido.

El módulo utiliza el patrón de diseño Factory para gestionar el procesamiento de archivos, ya que estos pueden ser de dos tipos: Java (.java) o Python (.py). Se mantiene una clase abstracta, `Tokenizer`, que contiene los métodos comunes para procesar cualquier archivo de código. Además, existen clases que extienden `Tokenizer`, como `JavaTokenizer` y `PythonTokenizer`, permitiendo la incorporación sencilla de nuevos lenguajes mediante clases adicionales.

Por otro lado, la clase `TokenizerMapper` se encarga de gestionar las extensiones de archivo disponibles (.py y .java) mediante un diccionario, permitiendo al `TokenInitializer` utilizar esta información para determinar cómo procesar el archivo, facilitando así la expansión a nuevos lenguajes en el futuro gracias al patrón *Factory*.

La salida del módulo es un objeto `TokenizedCode`, que contiene un identificador de marca de tiempo (*TimeStamp*) y la lista de tokens procesados. Este resultado se presenta en formato JSON para ser almacenado en el S3 `datalake`.

4.3 Módulo Metrics

El módulo `Metrics`, también implementado en Java, opera de manera similar al módulo `Tokenizer` y sigue el patrón de diseño *Factory*. Su funcionalidad principal es analizar archivos de código después de recibir una notificación de AWS sobre la adición de un nuevo archivo al bucket S3 `code-files`. El módulo utiliza el Factory Pattern para elegir el analizador de código adecuado entre `JavaMetricsAnalyzer` y `PythonMetricsAnalyzer`, ambas clases que extienden de `MetricsAnalyzer`. Esta flexibilidad facilita la incorporación de nuevos lenguajes mediante la creación de clases adicionales.

Al recibir la notificación, el módulo `Metrics` activa el analizador de código correspondiente y devuelve un objeto `SourceCodeMetrics`. Este objeto contiene un identificador de marca de tiempo (*TimeStamp*) y las métricas obtenidas del archivo procesado, que incluyen el número de líneas de código, la indentación máxima, el número de imports y el número de funciones.

Al igual que en el caso anterior, el resultado se presenta en formato JSON para ser almacenado en el S3 `metrics`, proporcionando una visión detallada de las métricas del código recién agregado.

4.4 Módulo Token-Datamart

El módulo `token-datamart` está diseñado para facilitar la interacción entre aplicaciones y las bases de datos de DynamoDB, especialmente en contextos donde se requiere la generación dinámica de texto. Al ofrecer una interfaz de programación de aplicaciones (API) intuitiva, permite la fácil inyección de nuevos modelos de generación de palabras, lo cual es esencial para aplicaciones que buscan mejorar o actualizar continuamente su capacidad para generar contenido textual relevante y coherente. Esta facilidad de integración asegura que los desarrolladores puedan adaptar y expandir sus aplicaciones con un esfuerzo mínimo.

Para optimizar el almacenamiento y la recuperación de datos, `token-datamart` implementa una estrategia de diseño en DynamoDB que involucra la creación de múltiples tablas, con una tabla por cada tamaño de contexto utilizado. Esta metodología evita la necesidad de una única "mega tabla", facilitando así una gestión de datos más eficiente y un rendimiento mejorado en las operaciones de consulta. Esta estructura de múltiples tablas permite una escalabilidad más efectiva y una organización de datos que se alinea mejor con las necesidades específicas de la aplicación, asegurando que el rendimiento no se vea comprometido a medida que el volumen de datos crece.

La API que provee `token-datamart` soporta operaciones POST y GET, cruciales para la manipulación de datos en aplicaciones de generación de texto. La operación POST permite a los usuarios subir tanto el contexto como la palabra siguiente (nextWord), facilitando así la alimentación de la base de datos con información relevante para el entrenamiento de modelos. En contraste, la operación GET posibilita la recuperación de la palabra siguiente basándose en un contexto previamente suministrado, jugando un papel clave en la generación de contenido textual coherente y contextualmente relevante. Estas operaciones de la API subrayan la versatilidad y la potencia del módulo `token-datamart`, destacando su capacidad para soportar aplicaciones dinámicas y ricas en contenido.

4.5 Módulo Token-Datamart-Builder

El módulo `Token-Datamart-Builder` desempeña un papel crucial en la generación de n-gramas que se almacenarán en la base de datos DynamoDB. Activado por una notificación del S3 `datalake` cuando se añade un nuevo archivo, este módulo procesa los tokens previamente almacenados y utiliza la clase principal `WordContextDTOProcessor`.

La clase `WordContextDTOProcessor` es la pieza central de este módulo. Su función principal es transformar la lista de tokens de un archivo en objetos `WordContextDTO`, los cuales representan contextos y palabras siguientes dentro de una ventana definida. Esta ventana tiene un tamaño específico, inicialmente entre 2 y 25. La clase `WordContextDTOProcessor` verifica si se debe omitir el procesamiento en ciertos casos y crea un objeto `WordContextDTO` con el contexto y la palabra siguiente correspondientes.

Cada
texttttWordContextDTO generado representa un n-grama, donde el contexto son los tokens dentro de la ventana definida y *next* es la palabra almacenada como una posible predicción. Este proceso se repite hasta que el último elemento de la lista de tokens sea nulo, momento en el cual el último *next* se establece como EOF (*End Of Sequence*).

Una vez que se ha procesado y creado el objeto `WordContextDTO`, se envía en formato JSON como un nuevo registro a la base de datos DynamoDB de la arquitectura, enriqueciendo así la colección de datos que alimentará el suggester del proyecto.

4.6 Módulo Token-Sugester

Todas las cuestiones que se quieran realizar, se deben llevar a cabo a partir de la api a la que da soporte el módulo`token-suggester`, tanto como para obtener el próximo token dado una lista de palabras, como para obtener una métrica determinada sobre uno de los archivos procesados. Por tanto, podemos diferenciar dos rutas principales dentro de esta api.

Por un lado, tenemos la encargada de dar al ususario el siguiente token. Para acceder a ella, se debe serguir la ruta */gitradar/token-suggester/next/:word*, siendo word un parámetro de uso obligatorio. Este deberá ser usado por el ususario para redactar el número de palabras previas que se posean, todas ellas separadas por comas. El resultado será una palabra que es la más probable que acompañe a las pasadas según nuestro modelo. Para obtener este dato, nuestro módulo se comunica internamente con otra api; con token-datamart, que es la encargada de tratar todas aquellas cuestiones de get y post sobre la base de datos dynamodb que contiene el modelo de datos. A partir del número de tokens pasados en el request, se cuestiona a la tabla apropiada y se obtiene el siguiente token esperado, puediendo ser devuelto al usuario.

Por otro lado, tenemos la ruta */gitradar/code-metric/:file/:metric*, que devuelve el valor de la métrica deseada para el fichero indicado. En :file se debe poner el nombre completo, estando la extensión incluida (por ejemplo, .java) del archivo procesado del que se quiera la métrica. En cuanto a :metric, se usa para indicar la métrica deseada. Hay varias posibilidades.

1. **lines**: Indica el número de líneas que tiene el archivo procesado.
2. **maxIndentation**: El número indica la indentanción máxima encontrada en alguna parte del código. Esta medida también es conocida como complejidad ciclomática, que es una métrica del software en ingeniería del software que proporciona una medición cuantitativa de la complejidad lógica de un programa.
3. **imports**: Se usa para obtener el número de import que contiene el código. Muy útil para saber el número de dependencias que tiene.
4. **functions**: Muestra el número de funciones que contiene el código procesado.

Su funcionamiento es bastante sencillo. En nuestra arquitectura contamos con un bucket, llamado `gitradar-metrics`, donde se van almacenando las métricas de todos los ficheros procesados. Internamente, este módulo se encarga de, dado un fichero por parámetros, llevar a cabo una cuestión al s3 para obtener todas la métricas de este. Posteriormente, se deserializa el objeto JSON y se obtiene el valor de la métrica que haya sido requerida, dejándolo listo para ser devuelto en la respuesta.

5 Modo de Uso

Para que el programa se pueda desarrollar de forma adecuada, es necesario comenzar creando toda la infraestructura virtual que será usada y lanzar todos los programas que darán respuesta a nuestras solicitudes. Para que este proceso se lleve a cabo correctamente, debemos seguir una serie de pasos. Posteriormente, tendremos el programa listo y disponible para hacer cuestiones y subir archivos de códigos para que sean desarrollados.

5.1 Ejecución de script

1Comenzamos ejecutando el script desarrollado en bat para lanzar y crear todos los servicios de localstack que van a ser usados en nuestro programa, como los buckets de s3 y las funciones lambdas. Asimismo, añadimos las notificaciones necesarias para asegurar la correcta comunicación entre toda la infraestructura.

Para lanzar el script, el usuario debe ubicarse en la ruta donde tenga el script y ejecutar:

```
init.bat {ruta_artifacts} {ruta_jsons}
```

- `ruta_artifacts`: Ruta donde se encuentren todos los ficheros que se ejecutaran en las lambdas. En nuestro caso, son los archivos jar de cada uno de los módulos.
- `ruta_jsons`: Ruta donde tengas todos los jsons que se tengan que subir al docker para configurar correctamente la infraestructura

5.2 Creación de tablas

Ejecutamos la siguiente instrucción que creará todas las tablas de dynamoDB necesarias para el modelo.

```
docker exec localstack bash -c 'for i in {1..25}; do awslocal dynamodb create-table \
  --table-name ngram$(printf "%02d" $i) \
  --attribute-definitions AttributeName=context,AttributeType=S \
  --key-schema AttributeName=context,KeyType=HASH \
  --billing-mode PAY_PER_REQUEST \
  --region eu-central-2; done'
```

Como vemos, esta creará 25 tablas que se irán rellenando con cada fichero de código que se suba al bucket.

5.3 Lanzamiento de token-datamart

Posteriormente, debemos ubicarnos en la carpeta del `token-datamart`, crear y ejecutar el contenedor del `token-datamart`. Esta será la api a la cuál se conectarán todos aquellos módulos que quieran consultar o añadir algo al datamart de dynamoDB, logrando así que este sea el único módulo que tenga acceso directo. Para completarlo, se debe ejecutar:

```
docker build -t token-datamart .
docker run -d -p 8080:8080 --name token-datamart \
  --network gitradar_executable_default \
  --env AWS_ACCESS_KEY_ID=test \
  --env AWS_SECRET_ACCESS_KEY=test token-datamart
```

Destacar que el parámetro `--network` se añade para que el contenedor se ubique en la misma red que localstack, logrando así que ambos tengan una comunicación muy sencilla. Como se ve, también pasamos las credenciales necesarias como variables globales del programa. Esté modulo se encontrará escuchando por el puerto 8080.

5.4 Lanzamiento de token-suggester

Finalmente, repetimos exactamente el paso 3, pero esta vez con el módulo `token-suggester`, que será el encargado de dar respuesta a las cuestiones de los usuarios. Las instrucciones para ponerlo en marcha son:

```
docker build -t token-suggester .
docker run -d -p 9090:9090 --name token-suggester \
  --network gitradar_executable_default token-suggester
```

6 Trabajo a Futuro

Aunque Git-Radar ha logrado integrar eficazmente métricas de código y un suggester basado en n-gramas para mejorar la experiencia de desarrollo en GitHub, existen diversas áreas de mejora y expansión para futuros trabajos. Algunos aspectos a considerar incluyen:

6.1 Soporte para Nuevos Lenguajes de Programación:

La inclusión de soporte para una variedad más amplia de lenguajes de programación permitiría a los usuarios de Git-Radar aprovechar sus funcionalidades avanzadas en un espectro más amplio de proyectos.

6.2 Optimización de Suggester:

Explorar técnicas de aprendizaje automático para mejorar la precisión y relevancia de las sugerencias generadas por el suggester. La implementación de modelos más sofisticados podría resultar en recomendaciones de código más contextuales y personalizadas.

6.3 Mejora de Métricas de Código:

Refinar y expandir las métricas de código ofrecidas por Git-Radar para proporcionar una visión más completa y detallada del código fuente. Incluir métricas específicas para distintos tipos de proyectos o contextos de desarrollo.

6.4 Integración con Plataformas Adicionales:

Explorar la integración de Git-Radar con otras plataformas populares de desarrollo de software, ampliando así su alcance y utilidad en entornos más diversos.

6.5 Interfaz de Usuario Mejorada:

Desarrollar una interfaz de usuario más intuitiva y visual para Git-Radar que permita a los usuarios interactuar de manera más efectiva con las métricas y sugerencias proporcionadas, facilitando la comprensión y acción sobre los datos presentados. Estos puntos de trabajo futuro representan solo algunas de las posibles direcciones para evolucionar y mejorar Git-Radar, consolidándolo como una herramienta integral para el desarrollo de software colaborativo en plataformas como GitHub.

7 Conclusiones

En este trabajo, presentamos Git-Radar, una herramienta integral diseñada para mejorar la experiencia de desarrollo en GitHub al proporcionar métricas de código y un suggester basado en n-gramas. La arquitectura de Git-Radar, basada en AWS y Java, ha demostrado ser robusta y escalable, permitiendo a los usuarios obtener información valiosa sobre sus repositorios y recibir sugerencias inteligentes durante el proceso de codificación.

El análisis de métricas de código, como la indentación máxima, el número de líneas, funciones e imports, ofrece a los desarrolladores una visión detallada de la calidad y estructura de su código. Además, el suggester basado en n-gramas proporciona sugerencias contextuales para mejorar la eficiencia y consistencia en el desarrollo de código.

La implementación del patrón Factory en los módulos Tokenizer y Metrics facilita la extensibilidad del sistema, permitiendo la inclusión sencilla de soporte para nuevos lenguajes de programación y analizadores de código.