Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

gh-ai: Generación de Código Asistida por LLMs

gh-ai: LLM Assisted Code Generation

Raimon José Mejías Hernández

La Laguna, 23 de mayo de 2024

D. **Casiano Rodríguez León**, con N.I.F. 42.020.072-S profesor Catedrático de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

C E R T I F I C A

Que la presente memoria titulada:

gh-ai: Generación de Código Asistida por LLMs

ha sido realizada bajo su dirección por D. **Raimon Jose Mejías Hernández**, con N.I.F. 51.779.774-N.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 23 de mayo de 2024

Agradecimientos

A todos mis amigos, compañeros, familia y personas que me han ayudado a lo largo de la carrera.

Licencia

© Esta obra está bajo una licencia de Creative Commons Reconocimiento 4.0 Internacional.

Resumen

La utilización de inteligencia artificial capaz de interactuar con los seres humanos a través de lenguaje natural es una tecnología que ha llegado para quedarse, con los avances recientes en el procesamiento de lenguaje natural liderado por OpenAI con sus modelos `gpt-3` y `gpt-4`.

En este trabajo se busca aprovechar al máximo los modelos de lenguajes a través de la ingeniería de comandos con el objetivo de generar código que sirva de plantilla para desarrollar nuevas extensiones para GitHub *Command Line Interface*. Esta idea es posible gracias a la creación de una programa que, utilizando la información proporcionada por el usuario a través de un fichero de entrada, construya instrucciones previamente diseñadas que serán utilizadas para realizar la comunicación con la inteligencia artificial.

Además, haciendo uso de tecnologías de procesadores de lenguaje se ha diseñado un compilador de Markdown con el objetivo en mente de permitir al usuario crear su extensión a través de un simple fichero readme.md|.

Palabras clave: github, git, openai, ai, javascript, nearley, modelo, api, llm

Abstract

The use of artificial intelligence capable of interacting with humans through natural language is a technology that is here to stay, with recent advances in natural language processing led by OpenAI with its models `gpt-3` *and* `gpt-4`*.*

This work aims to maximize the potential of language models through command engineering with the goal of generating code that serves as a template for developing new extensions for the GitHub Command Line Interface. This idea is made possible by the creation of a program that, using the information provided by the user through an input file, constructs pre-designed instructions that will be used to communicate with the artificial intelligence.

Furthermore, by using natural language processing technologies, a Markdown compiler has been designed with the goal of allowing the user to create their extension through a simple Readme file.

Keywords: *github, git, openai, ai, javascript, nearley, model, api, llm*

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción, Antecedentes y Estado del Arte

1.1. Introducción

En los últimos años el mundo de la inteligencia artificial se ha visto en auge gracias a los recientes avances realizados en la rama del procesamiento de lenguaje natural. Dicha rama busca, a través de métodos y técnicas eficientes, comprender los diferentes lenguajes que utilizan los seres humanos para comunicarse y a su vez ser capaz de generar textos convincentes y coherentes que utilicen dichos lenguajes. Con la llegada de los modelos de aprendizaje profundo el procesamiento de lenguaje natural ha alcanzado un nivel de refinamiento capaz de simular la generación e interpretación de texto de manera correcta.

Los modelos de lenguajes son todos aquellos modelos de aprendizaje profundo que han sido entrenados con una gran cantidad de datos con el objetivo principal de generar, traducir e interpretar texto. En los últimos años, y con la llegada de modelos de lenguaje como `GPT-3`, las capacidades de dichos modelos se han aplicado en múltiples ámbitos, entre ellos la generación de código.

La gran expansión del procesamiento de lenguaje natural en años recientes ha dado como resultado la aparición una nueva rama de la ingeniería, conocida como *Prompt Engineering* (ingeniería de comandos) que busca analizar, comprender y manipular las capacidades de los modelos de lenguaje para conseguir resultados eficaces y reducir la aleatoriedad de las respuestas generadas.

En este trabajo se busca realizar una investigación acerca de cómo diseñar un programa que ayuda a los usuarios a generar código en un contexto específico utilizando el modelo de lenguaje `GPT-3.5`. El contexto del trabajo busca facilitar el desarrollo de nuevas extensiones para la herramienta GitHub *Command Line Interface* (CLI). Utilizando distintas técnicas de ingeniería de comandos se busca generar plantillas, que con la ayuda de la inteligencia artificial, permiten reducir el trabajo a realizar por parte del desarrollador.

1.2. Antecedentes y estado actual del tema

Por un lado la llegada de `GPT-3` por parte de OpenAI a mediados del año 2020 supuso un extraordinario avance en los estudios sobre el procesamiento de lenguaje natural, con una base de datos de `499` mil millones de `tokens` extraídos de grandes corpus[1] como *CommonCrawl*, Wikipedia, *Books*1 y 2, etc.

Se desarrollaron además una serie de métodos que permitían mejorar los resultados

obtenidos por parte del modelo de lenguaje, por ejemplo:

- *One-shot*: Consiste en indicarle un único ejemplo al modelo de lenguaje acerca de la tarea que debe realizar.
- *Few-shots*: Consiste en indicar varios ejemplos al modelo sobre la tarea que debe realizar.

Tras el lanzamiento al mercado en julio de 2021 de la primera Interfaz de Programación de Aplicaciones (API) de OpenAI ocurrió un nuevo auge para la investigación y utilización de modelos de lenguajes para multitud de escenarios como pueden ser la creación de asistentes virtuales, etiquetado de elementos, etc. A su vez dio lugar al avance de las investigaciones [2, 3] acerca de la ingeniería de comandos, dando como resultado nuevas técnicas como pueden ser:

- Salidas estructuradas: Indicarle al modelo de lenguaje que debe estructurar la respuesta bajo un esquema de *YAML Ain't Markup Language* (YAML) o *JavaScript Object Notation* (JSON).
- Razonar la tarea a realizar: Indicar al modelo que realice un análisis de la tarea a realizar.
- Generación de pruebas: Solicitar a la vez que se pide un nuevo código al modelo de lenguaje, un conjunto de pruebas para comprobar el correcto funcionamiento del código generado.

Por otra parte, la creciente e imparable popularidad de GitHub ha dado lugar a la expansión por parte de la empresa a nuevos horizontes y dispositivos. En los últimos años GitHub ha puesto un gran esfuerzo en desarrollar interfaces externas a la ya conocida interfaz web de GitHub, entre estas nuevas interfaces se encuentran: GitHub *Mobile*, GitHub *Desktop* y GitHub *Command Line Interface* (CLI).

Mientras *Mobile* y *Desktop* Modifican la interfaz web de GitHub, `GitHub CLI` crea una interfaz completamente nueva y muy potente para trabajar con GitHub a través de la línea de comandos de una terminal, eliminando por completo toda interfaz visual. Esta característica facilita la creación de *scripts* utilizados para automatizar la gran mayoría procesos comunes y repetitivos de GitHub. La naturaleza de código abierto de GitHub CLI ha conseguido mantener una gran comunidad activa de colaboradores a través de la creación de extensiones que permiten añadir funcionalidades adicionales al programa original.

Capítulo 2

Tecnologías

En la actualidad, la creación de programas completamente novedosos es prácticamente imposible sin la utilización de herramientas y tecnologías previamente creadas por terceros. La creación de la extensión `gh-ai` no sería posible sin la existencia de tecnologías externas como pueden ser el caso de la plataforma de desarrollo colaborativo `GitHub` o la Interfaz de Programación de Aplicaciones (API) de `OpenAI` así como los grandes avances realizados en la creación de modelos de lenguaje. A su vez la necesidad de desarrollar la extensión dentro de un plazo de tiempo razonable requiere la utilización de multitud de paquetes y librerías externas para agilizar y simplificar el desarrollo de la extensión.

En este capítulo se listan todas las herramientas externas de vital importancia para el desarrollo de la extensión `gh-ai`

2.1. Control de versiones y GitHub

Se ha utilizado la herramienta de control de versiones Git junto con la plataforma de desarrollo GitHub para llevar a cabo el desarrollo de la extensión `gh-ai`, el repositorio creado para alojar el código del programa puede ser consultado en el apéndice A del trabajo. La decisión de utilizar GitHub no proviene solamente de la necesidad de gestionar las distintas versiones del programa, sino que facilita la distribución de la extensión.

Pese a que la extensión `gh-ai` no utiliza en gran medida las funcionalidades que provee la API de GitHub, las nuevas extensiones que sean descritas por los usuarios pueden llegar a necesitar la utilización de la API.

2.1.1. GitHub Command Line Interface

La extensión `gh-ai` utiliza principalmente la herramienta de línea de comandos GitHub CLI para gestionar su distribución, haciendo uso del gestor de paquetes integrado en el programa `gh` La extensión puede ser instalada y actualizada con mucha facilidad, siendo necesario insertar una única vez el enlace al repositorio de GitHub.

Como se ha comentado en el capitulo 1, el objetivo de la extensión `gh-ai` es facilitar y guiar al usuario en la creación de nuevas extensiones para el programa `gh`. Por este motivo la utilización de GitHub CLI así como los conocimientos requeridos para trabajar con ella son cruciales a la hora de realizar pruebas sobre el funcionamiento de la extensión.

```
<Símbolo> -> <Expresión>
```

Figura 2.1: Gramática en Forma Normal de Backus utilizando la sintaxis del paquete Nearley.

2.2. El Lenguaje de Programación JavaScript

Una de las primeras decisiones a la hora de empezar con el proyecto fue decidir el lenguaje de programación a utilizar para generar la extensión de GitHub CLI. la herramienta permite emplear cualquier lenguaje de programación siempre y cuando exista un `script` en lenguaje Bash y cuyo nombre sea el nombre de la extensión, dicho `script` deberá ejecutar el programa correspondiente.

Los lenguajes de programación planteados para el desarrollo del proyecto fueron JavaScript y TypeScript. Tras realizar un análisis de los paquetes a utilizar para facilitar la creación de la extensión se llegó a la conclusión de utilizar JavaScript como lenguaje de programación, debido a que paquetes como `Moo` y `Nearley`, fundamentales para desarrollar la extensión `gh-ai`, no poseen una traducción oficial al lenguaje TypeScript.

2.2.1. Análisis léxico y sintáctico utilizando los Paquetes Moo y Nearley

La extensión `gh-ai` hace uso de una sintaxis propia para la elaboración de sus ficheros de entrada. La creación de una sintaxis propia requiere de una serie de elementos previos conocidos como analizadores léxicos y sintácticos. Una sintaxis requiere que ambos analizadores trabajen en conjunto para poder leer, analizar y comprobar que la información suministrada cumple con las reglas establecidas en dicha sintaxis.

Existen dos paquetes de JavaScript dedicados a la creación, diseño y generación de analizadores léxicos, usando el paquete `Moo Tokenizer` y semánticos, usando el paquete `Nearly`. El generador de analizadores léxicos, `Moo`, altamente optimizado[4], permite clasificar el contenido de una cadena de texto en diferentes `tokens`, los cuales son objetos que almacenan información útil para el manejo y manipulación de los mismos. Los `tokens` generados por un analizador léxico posteriormente serán utilizados por un analizador sintáctico para diseñar y generar las posibles reglas gramaticales de una sintaxis. Una vez obtenido el analizador léxico se puede utilizar el paquete `Nearley` para hacer uso de los `tokens` generados, `Nearley` es un generador de analizadores sintácticos que, utilizando el algoritmo Earley, es capaz de soportar todo tipo de gramáticas en Forma Normal de Backus[5], En la figura 2.1 se puede apreciar la sintaxis básica de una regla gramatical escrita en `Nearley`.

El generador de analizadores sintácticos viene acompañado de una serie de herramientas de depuración y de un lenguaje de dominio específico diseñado para escribir las reglas gramaticales de manera cómoda y eficiente.

2.2.2. Generación de esquemas con el Paquete Zod

Para prevenir posibles errores de la extensión `gh-ai` ante la lectura de información incorrecta, debido al mal uso de la sintaxis propia o de los posibles errores que puedan

ocurrir al recibir una respuesta en formato JSON por los modelos de lenguaje, se ha utilizado el paquete `Zod`, el cual es una librería para la generación y validación de esquemas. Un esquema puede ser cualquier tipo de dato, desde un valor numérico hasta una lista de objetos complejos[6].

Diseñando un esquema se puede realizar una comprobación previa antes de permitir que la extensión `gh-ai` Utilice información incorrecta antes de iniciar la comunicación con la inteligencia artificial, comprobando si los datos extraídos del fichero de entrada son correctos. Esta misma metodología puede ser utilizada para verificar si la respuesta recibida por el modelo de lenguaje cumple con el formato especificado durante el transcurso de la conversación.

2.2.3. Utilización de plantillas con el Paquete Mustache

Debido a la creación de un sistema automatizado para la extracción de la información de los ficheros de entrada, la generación de las instrucciones utilizadas para realizar la comunicación con el modelo de lenguaje debe tener en cuenta que secciones concretas del texto contienen información dinámica, la cual varía dependiendo de los datos suministrados por el usuario, mientras que el resto del texto contiene información inmutable. Para solventar este problema se utiliza el paquete `Mustache`, el cual permite la expansión de etiquetas dentro de un fichero de texto plano denominado plantilla.

El paquete implementa un sistema de expansión de etiquetas. Las etiquetas de `Mustache` se representan a partir de encapsular dos veces un texto dentro de llaves: `"{{Variable}}"`. El sistema de `Mustache` se denomina "sin lógica"[7] debido a que no implementa de manera explícita condicionales y bucles, en su lugar se deben escribir etiquetas especiales para indicar los casos anteriormente mencionados.

Gracias a la utilización de las plantillas, la generación de las instrucciones se puede realizar de manera dinámica e independiente de la información suministrada por el usuario, aunque añadiendo un grado de complejidad a la escritura del fichero de texto debido a la inclusión de los diferentes tipos de etiquetas que implementa el paquete.

2.3. La Interfaz de Programación de Aplicaciones de OpenAI

Pese a que la extensión `gh-ai` ha sido diseñada con el objetivo de permitir la inclusión de diferentes interfaces para la realización de la comunicación entre el usuario y un modelo de lenguaje. La única interfaz implementada de manera oficial es la interfaz de programación de aplicaciones(API) de OpenAI.

La API de OpenAI permite la comunicación entre usuarios y las diferentes herramientas proporcionadas por la empresa, dichas herramientas van desde la generación de texto con modelos de lenguaje como `ChatGPT` hasta la generación de imágenes con modelos como `DALL.E`. Dicha API se encuentra disponible, de manera oficial, para los lenguajes de programación Python, JavaScript y TypeScript. Un aspecto importante a tener en cuenta antes de realizar una comunicación con la API es disponer de una clave de autenticación, la cual se puede obtener a través de la página web oficial de OpenAI[8].

La utilización de las diferentes herramientas de OpenAI generan un coste monetario. Los modelos de lenguaje como `ChatGPT` procesan el texto a través de `tokens`, los cuales permiten entender la relación estadística entre las palabras. Dependiendo

del modelo de lenguaje utilizado el coste de lectura y escritura de `Tokens` por parte de la API puede variar. El modelo de lenguaje por defecto utilizado por la extensión `gh-ai` es `gpt-3.5-turbo-0125` el cual posee, a día 22 de mayo de 2024, un coste de `0.50 US$(0.46€)` por cada millón de `tokens` leídos y `1.50 US$(1.38€)` por cada millón de `tokens` escritos[9].

2.3.1. Interfaz beta de asistentes

Recientemente OpenAI ha lanzado al mercado su nueva interfaz de asistentes. Un asistente es una entidad que, utilizando los modelos de lenguaje como ChatGPT, es capaz de realizar tareas para un usuario. Los asistentes poseen un contexto y una ventana de memoria y que permiten la comunicación con el usuario a través de múltiples sesiones repartidas en el tiempo. Generalmente los asistentes tienen acceso a herramientas que les permiten realizar tareas más complejas como la ejecución de código o la lectura y de un fichero[10].

La interfaz de asistentes se encuentra actualmente en fase de desarrollo, constantemente la API se actualiza con nuevas funcionalidades y mejoras en la interfaz. La API de asistente cuenta con la capacidad de generar asistentes inteligentes, como puede ser un `Chatbot` dentro de una aplicación, capaces de aprovechar las herramientas previamente explicadas para generar una respuesta o acción ante las peticiones del usuario.

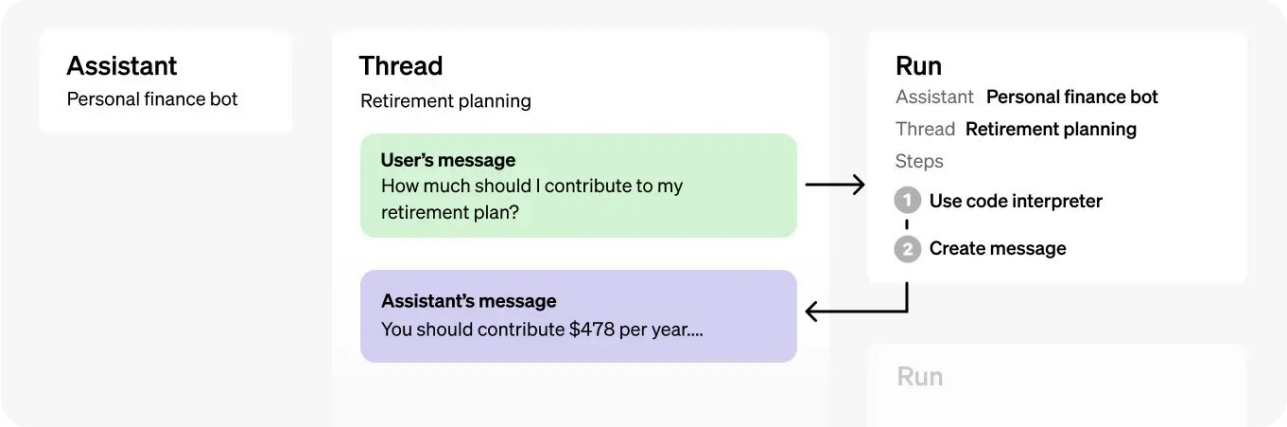

Figura 2.2: Ejemplo de los diferentes conceptos que intervienen en la API de Asistentes

Como se observa en la figura 2.2[11], cada asistente interactúa con uno múltiples usuarios a través de hilos, cada hilo permite al asistente entablar una conversación con un único usuario a la vez que almacena tanto las peticiones como las respuestas del modelo de lenguaje. El proceso de comunicación entre el asistente y el usuario se realiza a través de múltiples ejecuciones, cada ejecución empieza con el envío de una petición por parte del usuario y acaba tras la generación de la respuesta por parte del modelo de lenguaje, durante la ejecución el asistente es capaz de generar texto y utilizar las herramientas que tenga a su disposición. Actualmente los asistentes disponen de tres herramientas a utilizar durante una conversación con el usuario, las cuales son:

- Intérprete de código: Permiten al asistente escribir y ejecutar código en el lenguaje de programación Python. Conlleva, a día 22 de mayo de 2024, un gasto de `0.03US$(0,028€)` por sesión.

- Lectura de ficheros: Permiten al asistente realizar una lectura optimizada de un fichero, realizando una extracción de la información de manera precisa. La información leída dentro del fichero será utilizada para responder las peticiones del usuario, es una herramienta ideal para proporcionar al asistente de información confidencial o privada. El coste de utilizar la lectura de ficheros dependerá de el consumo de `embeddings`. OpenAI permite almacenar `1GB` de ficheros en forma de `embeddings`, a partir del primer `GB`de uso se empezará, a día 22 de mayo de 2024, a cobrar `0.1US$(0,092€)/GB/Tiempo almacenado en memoria`.
- Ejecución de herramientas externas: La interfaz de OpenAI permite al usuario escribir sus propias herramientas para luego poder ser utilizadas por un asistente. A nivel interno un asistente no es capaz de ejecutar herramientas externas, por este motivo se envía al usuario un mensaje que contiene la información en formato JSON con los argumentos a introducir en la herramienta. La ejecución de herramientas externas debe ser gestionada por el usuario que esté desarrollando el programa.

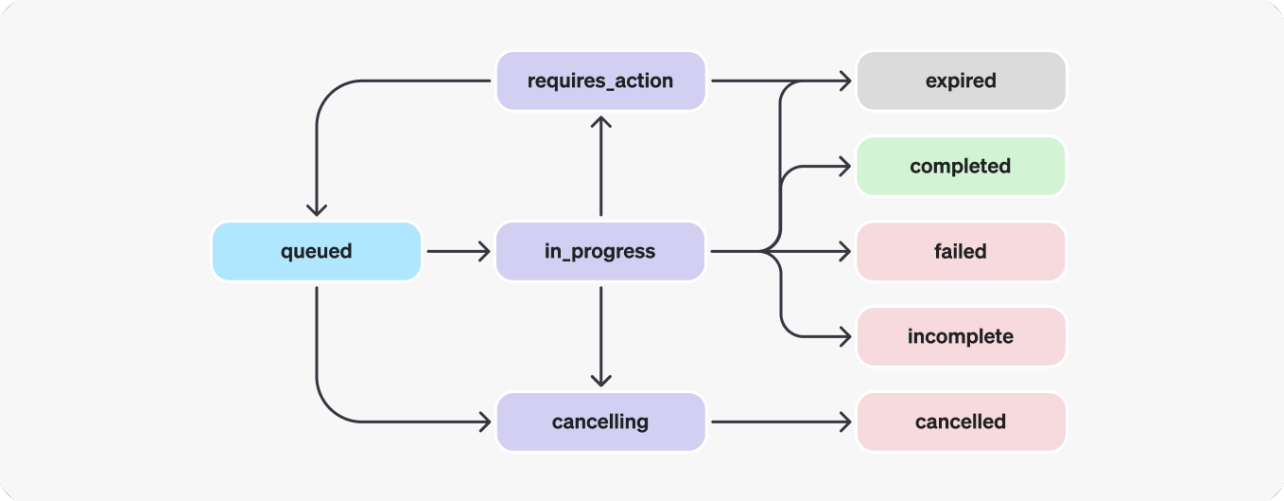

Figura 2.3: Diagrama de relación entre los múltiples estados de una ejecución.

Como se puede observar en la figura 2.3, la ejecución de un hilo es un proceso complejo capaz de pasar a través de múltiples estados dependiendo de las acciones que el asistente realice. Empezando siempre desde el estado inicial `"En cola"` y acabando en uno de los diferentes estados terminales. El estado de la ejecución permite abstraer al usuario final de las diferentes acciones que la API de OpenAI realiza como consecuencia de la introducción de una nueva instrucción, a su vez permite al programa, en este caso a la extensión `gh-ai` responder ante los resultados, empaquetando y organizando los resultados para entregar la información crucial al usuario final. El significado de los diferentes estados que pueden ocurrir durante la ejecución de un hilo se encuentran disponibles en la documentación de la API de OpenAI[12].

Capítulo 3

Modo de uso

En este capítulo se detallan todos los pasos a seguir para poder realizar una ejecución satisfactoria de la extensión `gh-ai`. Empezando con la instalación y configuración de la extensión, pasando por la escritura del fichero de entrada hasta obtener los resultados de la ejecución del programa.

3.1. Instalación del intérprete Node.js

El primer paso necesario para poder ejecutar la extensión `gh-ai` consiste en instalar el intérprete del lenguaje de programación JavaScript conocido como Node.js, este primer paso puede ser realizado de múltiples formas siendo las más recomendables:

- Windows/MacOS: Utilizar el instalador de paquetes oficial disponible en la página web de Node.js[13]
- General: Instalar un gestor de paquetes para gestionar las diferentes versiones de Node.js. Un gestor muy popular es *Node Version Manager* (NVM), se puede encontrar una guía paso a paso de la instalación en la página web oficial de Node.js[14].

Independientemente del método de instalación el resultado debe ser el mismo: El entorno Node.js correctamente configurado y listo para poder ejecutar la extensión `gh-ai`.

3.2. Instalación de Git y GitHub CLI

Para hacer uso de la extensión `gh-ai` es necesario tener instalado tanto el sistema de control de versiones Git como la herramienta para la línea de comandos GitHub *Command Line Interface* (CLI).

3.2.1. Instalación de Git

La instalación del sistema de control de versiones Git[15] varía dependiendo del sistema operativo en el que sea necesario instalarlo. En caso de utilizar MacOS, una opción viable es utilizar el instalador de paquetes `Homebrew`[16] y ejecutar el siguiente comando en una terminal:

```
$ brew install git
```

Para sistemas operativos Windows existen dos posibles opciones para instalar Git los cuales son:

- Descargar e instalar el gestor de paquetes oficial de la página web de Git.
- Para los sistemas Windows 10 y Windows 11 se puede utilizar el instalador de paquetes oficial de Windows conocido como `Winget`[17] y ejecute el siguiente comando dentro de una terminal o Powershell.

```
winget install --id Git.Git -e --source winget
```

Instalar Git en sistemas operativos Linux depende de la distribución de Linux y del gestor de paquetes que este contenga. Para distribuciones Debian/Ubuntu se puede utilizar el siguiente comando dentro de una terminal:

```
apt-get install git
```

3.2.2. Instalación de GitHub Command Line Interface

De la misma manera que el sistema de control de versiones Git, la herramienta GitHub CLI se encuentra disponible en la gran mayoría de sistemas operativos y por tanto su método de instalación varía. En el repositorio oficial de GitHub CLI se listan todas las posibles opciones de instalación de la herramienta[18].

A continuación muestran las opciones más comunes para instalar GitHub CLI:

- MacOS:

```
brew install gh
```

- Windows:

```
winget install --id GitHub.cli
```

- Debian/Ubuntu:

```
sudo apt install gh
```

Para ser capaz de ejecutar todas las funcionalidades de gh, una vez terminada la instalación, es necesario autenticarse utilizando una cuenta de GitHub. Este proceso se puede realizar a través de la página web de GitHub utilizando el comando: `gh auth login`. Una vez terminado el proceso se creará y guardará una clave de acceso. Dicha clave puede ser reutilizada para autenticarse en la misma cuenta.

3.3. Instalación de la extensión gh-ai

La herramienta GitHub CLI permite la gestión de diferentes extensiones del programa a través de su propio gestor de paquetes `gh extension`. El gestor de paquetes permite: Instalar, desinstalar y actualizar las distintas extensiones de la herramienta.

Gracias a este gestor de paquetes, instalar la extensión `gh-ai` se puede realizar de manera sencilla ejecutando los siguientes comandos:

- Instalar la extensión `gh-ai`:

```
gh ext install https://GitHub.com/gh-cli-for-education/gh-ai
```

- Actualizar la extensión `gh-ai`:

```
gh extension upgrade gh-ai
```

- Desinstalar la extensión `gh-ai`:

```
gh extension remove gh-ai
```

Una vez realizado este paso ya es posible ejecutar la extensión siempre que se requiera utilizando el comando:

```
gh ai <Fichero de entrada> <Directorio de salida> [Opciones]
```

3.4. Configuración de las variables de entorno

Para un correcto funcionamiento de la extensión es necesario configurar las variables de entorno utilizadas por el programa. En el caso de no existir ningún fichero con extensión `.env` el programa no será capaz de realizar la conversación con el modelo de lenguaje seleccionado. En la figura 3.1 se puede observar un ejemplo del fichero `.env` en el cual se detallan las posibles variables que puede contener.

```
1 OPENAI_API_KEY= # <Contiene la clave de acceso a la API de OpenAI>
2 OPENAI_ORG= # <(opcional)Contiene la clave de acceso a una organización dentro de OpenAI>
3 ASSISTANT_ID= # <(opcional)Contiene la identificación de un asistente previamente creado
    .>
4 THREAD_ID= # <(opcional)Contiene la identificación de una conversación entre un asistente
    y el usuario.>
```

Figura 3.1: Ejemplo del archivo `.env` necesario para la ejecución de la extensión

3.5. Escritura del fichero de entrada

Para poder construir las diferentes instrucciones a utilizar durante la conversación con el modelo de lenguaje es necesario que la extensión `gh-ai` tenga a su disposición toda la información crucial para estructurar de manera correcta el contenido de la instrucción, detallando los aspectos más importantes y eliminando toda información innecesaria.

Por este motivo es necesario que el usuario introduzca la información a través de un fichero de entrada escrito en un lenguaje al estilo Markdown[19]. El lenguaje se describe a través de dos tipos de bloques o secciones. Los bloques principales del lenguaje son aquellos representados por un único símbolo numeral #. Se caracterizan por influir, en gran medida, el comportamiento del programa. Actualmente existen solo dos bloques principales, siendo uno de ellos la configuración de la conversación con el modelo de lenguaje.

Por otra parte los bloques secundarios son dependientes del bloque de nivel anterior de anidación. Cada nuevo bloque secundario añade un nuevo nivel de anidación dentro del fichero y tienen como objetivo describir y especificar la información del bloque en el cual están anidados.

3.5.1. Bloque *ChatSettings*

A la hora de realizar la conversación el usuario puede indicar una serie de parámetros para modificar el comportamiento y respuesta del modelo de lenguaje ante las instrucciones generadas, estas configuraciones permiten al usuario personalizar su experiencia con la inteligencia artificial, por este motivo la inclusión de este bloque es completamente opcional.

Por ahora las configuraciones del bloque *Chat Settings* son puramente estéticas, pero se espera que en próximas versiones permita al usuario modificar parámetros más importantes como pueden ser la temperatura, la variable `top_p` y el formato de respuesta de la inteligencia artificial. El bloque *Chat Settings* soporta actualmente las opciones:

- *Language*: Indica el lenguaje natural a utilizar para la respuesta del modelo.
- *Nickname*: Indica el nombre que utilizará el modelo para referirse al usuario.

```
1 [comment]: # (La separación entre las palabras Chat y Settings es arbitraria)
2 [comment]: # (Ambas palabras son indiferentes a las mayúsculas y minúsculas)
3 # Chat Settings
4
5 - language: English
6 - Nickname: User
7 - Another: Option
8
9 [comment]: # (Pese a que el analizador sintáctico permite la inclusión de opciones
      inexistentes, el esquema del objeto de entrada detectará el error)
```

Figura 3.2: Ejemplo del bloque *Chat Settings*

3.5.2. Bloque *Extension*

Una de las primeras acciones que el usuario debe realizar a la hora de escribir el fichero de entrada es indicar el nombre de la extensión a crear, esto se realiza añadiendo el bloque *Extension* al fichero de entrada seguido de el nombre de la extensión:

```
# Extension <gh-extension-name>‘.
```

Haciendo uso de los diferentes elementos del lenguaje Markdown se deberá escribir una descripción detallada del funcionamiento del programa a crear junto con una serie de bloques secundarios que permiten incluir detalles de utilidad a la descripción de la extensión, sirviendo como puntos de apoyo para el usuario para obtener una visión clara y estructurada de la extensión deseada.

````
# Extension gh-rate-limit

[comment]: # (La descripción soporta párrafos de Markdown)
gh-rate-limit is an extension of the GitHub Command Line Interface ‘(GitHub CLI)‘
whose purpose is to show the user their existing **rate limits** and when its resets.
The program does exactly the same as executing the *following command*...

[comment]: # (Acepta bloques de código)

‘‘‘bash
curl -fSsL -H "Authorization: token $(GitHub_pat)" -X GET \
  https://api.GitHub.com/rate_limit \
  | jq --raw-output ’.resources | to_entries[] | {name: .key} + .value | "\(.name)  \(.
    remaining)/\(.limit)  \(.reset | strflocaltime("%H:%M:%S") )"’ \
  | column -t
‘‘‘

[comment]: # (También soporta listas tanto ordenadas como desordenadas)
The program has some **Prerequisites** that are:

+ It is necessary to have ‘GitHub CLI (gh) installed‘, so the program must be able to
    verify that said program is installed.
1. It is necessary to have ‘js installed‘, so the program is able to execute it.
````

Figura 3.3: Ejemplo de bloque *Extension*. Extraído del fichero: 2

Bloque *Language Settings*

El bloque *Language Settings* es una de las secciones secundarias dentro del bloque *Extension*. Dicho bloque secundario debe ser escrito de manera obligatoria debido a que contiene información acerca del lenguaje de programación a utilizar para la generación de la extensión.

A nivel de contenido el bloque *Language Settings* funciona exactamente de la misma manera que el bloque principal *Chat Settings*, con la diferencia de que acepta parámetros distintos, los cuales son:

- *Language*: Indica el lenguaje de programación a utilizar por el modelo de lenguaje para generar la extensión.

- *Specification*, (Opcional): Indica la especificación a utilizar del lenguaje de programación seleccionado previamente.
- *Style*, (Opcional): Indica la guía de estilo a seguir durante la generación del código.

```
[comment]: # (La separación entre las palabras Language y Settings es arbitraria)
[comment]: # (Ambas palabras son indiferentes a las mayúsculas y minúsculas)
## Language Settings

- language: JavaScript
- Nickname: ECMAScript2021
- Style: Google
- Another: Option

[comment]: # (Pese a que el analizador sintáctico permite la inclusión de opciones
    inexistentes, el esquema del objeto de entrada detectará el error)
```

Figura 3.4: Ejemplo del bloque *Language Settings*

Bloque *Function*

Para especificar las distintas funciones que la extensión ha de contener se debe utilizar el bloque secundario *Function* dentro del bloque *Extension* para indicarle a la extensión `gh-ai` que debe generar una instrucción específica para indicarle al modelo de lenguaje que debe codificar el contenido de dicha función.

El bloque *Function* abstrae la definición y el formato de una función para permitir que se pueda describir sin la necesidad de escribirlo en el lenguaje de programación especificado previamente. Por este motivo el bloque permite todos los tipos de elementos descriptivos mencionados anteriormente en el bloque *Extension*, y a la vez permite especificar dos secciones adicionales.

Como se observa en la figura 3.5, pueden existir dos apartados que permiten añadir información adicional, útil para el modelo de lenguaje. El bloque *Query* permite indicarle al programa que la función debe llamar a la API de GitHub, construyendo la instrucción correspondiente. El bloque *Template* permite indicarle a la inteligencia artificial una plantilla en la que basarse para generar la función.

```
[comment]: # (La palabra clave Function debe ir acompanado el nombre de la función)
## Function print-notifs

[comment]: # (Aquí va la descripción de la función)

The print_notifs function is responsible for printing on the command line the information
obtained from the notifications after making the API call...
...In each iteration, a maximum of 50 notifications can be obtained from the API. For **
example**: If a user requests 56 notifications, the API must be called twice with a
maximum of 50 notifications each time, and stop displaying the information once the limit
set by 'num_notifications' is reached. It is important to keep this in mind, otherwise
the program will malfunction.

### Query

[comment]: # (Aquí va la descripción de la query)

### Template

[comment]: # (El template debe ir dentro de un CodeBlock)

'''JavaScript
  print_notifs() {
    # código
  }
'''
```

Figura 3.5: Ejemplo de bloque *Function*. Extraído del fichero: 3

Bloque *Help*

Un aspecto importante de todo programa creado para la línea de comandos es la implementación de una función de ayuda que muestre la información relevante sobre el uso del programa. Por este motivo se puede especificar dentro del bloque secundario *Help*, dentro del bloque principal *Extension*, todo lo que debe incluir dicha función. Esta sección cumple varios roles a la vez ya que permite a su vez indicarle al modelo de lenguaje los parámetros y argumentos que puede recibir la extensión a crear.

Como se observa en la figura 3.6, el bloque está diseñado para replicar el formato común utilizado en este tipo de funciones. Empezando con el apartado obligatorio *Usage* y su esquema de uso, seguido de una serie de párrafos adicionales a modo de cabecera para luego indicar la lista de argumentos en el apartado *Arguments* y parámetros del programa en el apartado *Parameters*, por último se puede añadir información adicional a modo de pie de página.

```
## Help

[comment]: # (La subsección usage debe ir todo dentro de una misma línea)

### Usage gh-cpissues <git-repo> --label <label> [--verbose]

[comment]: # (Aquí pueden ir los párrafos de la cabecera)

### Parameters

[comment]: # (Dentro de la subsección Parameters se muestra una lista de parámetros)
[comment]: # (Siguen el siguiente formato)

- --label Specify the <label> of the issue to copy (Modifies the value of the variable `label`).

### Arguments

[comment]: # (Dentro de la subsección Arguments se muestra una lista de argumentos)
[comment]: # (Siguen el siguiente formato)

- input-object The input file used to feed the llm

[comment]: # (Aquí pueden ir los párrafos del pie de página)

```

Figura 3.6: Ejemplo de bloque *Help*. Extraído del fichero 4

Bloque *Examples*

Una manera de mejorar la respuesta del modelo de lenguaje es indicarle una serie de ejemplos que pueda utilizar como referencia para generar el código de la extensión, por este motivo el fichero de entrada permite la introducción de ejemplos a través del bloque secundario *Examples*, dentro del bloque principal *Extension*. Una vez abierta la sección se pueden indicar cuantos ejemplos el usuario vea necesarios siempre y cuando sigan el siguiente formato:

Cada ejemplo debe indicar la entrada por la línea de comandos así como el resultado correspondiente de ejecutar la extensión con dicha configuración.

```
## Examples

[comment]: # (El ejemplo debe ir en una sola línea)

`gh branch; gh poi; gh branch`

[comment]: # (El resultado de la ejecución debe ir dentro de un Codeblock)

```console
foo@bar:~$ gh poi
# Fetching pull requests...
# Deleting Branches...

foo@bar:~$ gh branch
- inproving-api-call
- main
```
```

Figura 3.7: Ejemplo de bloque *Examples*. Extraído del fichero 5

Bloque *Readme*

El último bloque secundario que puede contener el bloque principal *Extension* es la sección *Readme*, esta sección le indica a la extensión `gh-ai` que debe construir una instrucción que le ordene al modelo de lenguaje generar un fichero `Readme.md` con la información proporcionada por el usuario dentro de la descripción del bloque.

La descripción del bloque *Readme* puede no ser rellenado, en este caso el programa indicará una serie de secciones por defecto que el fichero debe contener.

```
## Readme

[comment]: # (La descripción detallada del contenido del fichero readme)
[comment]: # (Se construye en base a diferentes cabeceras escritas por el usuario)

# Instalation
1. Write how to **install** th gh-poi extension using the GitHub CLI program.

# Program usage
2. Write the **help** and usage of the gh-poi extension.

# Example
3. Write some **examples** of use `Don't use any provided example from the prompt`.
```

Figura 3.8: Ejemplo de Bloque *Readme*.

3.6. Ejecución de la extensión gh-ai

Una vez escrito el fichero de entrada y configurado las variables de entorno dentro del fichero `.env` se puede proceder a la ejecución del programa. Como se ha explicado en el apartado 3.3 para ejecutar la extensión se ejecuta el siguiente comando:

```
gh ai <Fichero de entrada> <Directorio de salida> [Opciones]
```

3.6.1. Argumentos

La extensión `gh-ai` recibe dos argumentos necesarios para el correcto funcionamiento del programa, los cuales son:

- Fichero de entrada: Fichero cuyo formato es una simplificación de Markdown. Debe contener una sintaxis correcta, de lo contrario el programa devolverá el error concreto por la terminal.
- Directorio de salida: Directorio en el cual se almacenarán los resultados obtenidos de entablar la conversación con el modelo de lenguaje. Se espera un camino correcto hasta el directorio, de lo contrario el programa no será capaz de guardar la información.

3.6.2. Parámetros

A su vez el programa espera recibir una serie de opciones que determinan su comportamiento a la hora de ejecutarse, tanto los argumentos como los parámetros pueden ser consultados en todo momento utilizando el parámetro `-h` o `--help`. A continuación se listan todos los parámetros aceptados por la extensión `gh-ai`:

- `-v --version`: Muestra la versión actual de la extensión.
- `-h --help`: Muestra la información de ayuda al usuario.
- `--tokens-verbose`: El programa almacenará el consumo de `Tokens` generado por el programa.
- `-m --llm-model <modelo>`: Indica al programa cual modelo de lenguaje utilizar.
- `-l --llm-api <api>`: Indica al programa cuál API de modelos de lenguaje utilizar, por defecto utilizará la API de OpenAI .
- `-t --command-type <type>`: Indica al programa qué tipo de ayuda necesita, por defecto pedirá ayuda para generar extensiones.
- `--save-thread`: Impide al programa eliminar la ID de la conversación con el modelo de lenguaje, guardando la información de la conversación dentro del reporte de usuario. Solo está disponible para la API de OpenAI.
- `--save-assistant`: Impide al programa eliminar la ID y el asistente generado durante la ejecución del programa, guardando la información del asistente dentro del reporte de usuario. Solo está disponible para la API de OpenAI.

3.7. Registro de la conversación

Tras finalizar la ejecución, la extensión `gh-ai` genera un fichero de registro escrito en formato Markdown cuyo contenido incluye información útil acerca de la ejecución del programa. Por cada instrucción generada se almacena la respuesta del modelo de lenguaje ante la instrucción así como el consumo total de la conversación. En caso de activar los parámetros `--save-thread` o `--save-assistant` la información del hilo o asistente respectivamente serán almacenados dentro del fichero.

El contenido del fichero de registro se divide en varios apartados, cada sección tiene como objetivo mostrar información útil al usuario acerca del resultado y ejecución de la conversación con el modelo de lenguaje. El primer apartado del fichero contiene una serie de advertencias sobre el uso de código generado por inteligencia artificial. De manera opcional existe una sección con información acerca del objeto de entrada generada por el compilador de Markdown simplificado, para activar esta sección es necesario utilizar la opción de depuración.

Durante el transcurso de ejecución de la extensión, el modelo de lenguaje es capaz de generar ficheros siempre que le resulte conveniente. Debido al comportamiento errático de los modelos de lenguaje es imposible asegurar que el asistente genere el fichero con el código escrito. Por este motivo es imprescindible la utilización del fichero `conversation-log` para observar el resultado de la conversación. Cabe mencionar que el programa está preparado para devolver el fichero de registro independientemente del resultado de la ejecución del programa, permitiendo así a los usuarios observar el progreso conseguido hasta la finalización de la extensión incluso ante un caso de error. En la figura 3.9 se puede observar un ejemplo de la organización de directorios esperado por la extensión.

```
Directorio
    +--- <Directorio de Salida>
    |       +--- conversation-log.md
    |       +--- <Nombre de la extensión>.<Extensión>
    +--- <Fichero de Entrada>.md
```

Figura 3.9: Organización de un directorio de ejemplo

Capítulo 4

Diseño

Para desarrollar la extensión `gh-ai` se ha necesitado llevar a cabo un proceso de toma de decisiones diseño para conseguir un programa de calidad y fácilmente escalable. Como se observa en la figura 4.1, la extensión se ha dividido en tres módulos bien diferenciados: Extracción de los datos de entrada, análisis del esquema y generación de las instrucciones y el proceso de conversación con el modelo de lenguaje. Este diseño permite la modificación de los distintos módulos sin poner en riesgo la integridad del resto, siempre y cuando se cumplan unos requisitos mínimos para que la comunicación entre los módulos sea posible.

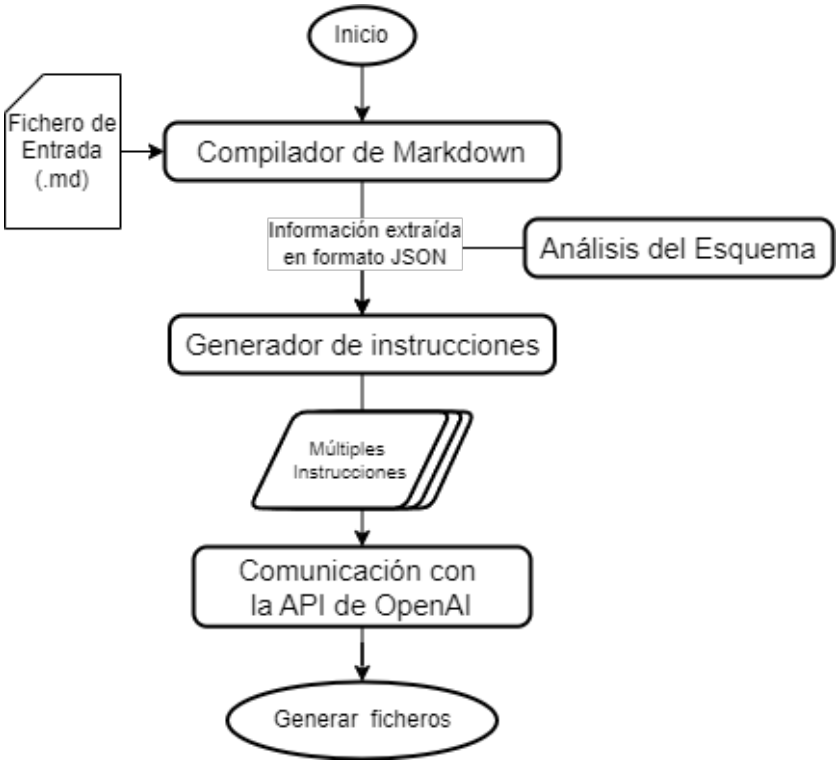

Figura 4.1: Diagrama de flujo del programa

A lo largo de este capítulo se detallarán las decisiones tomadas para cada uno de los diferentes módulos que conforman la extensión `gh-ai`.

4.1. El Compilador de Markdown Simplificado

La extensión `gh-ai` Utiliza un analizador sintáctico propio para extraer la información del fichero de entrada. Este analizador sintáctico está programado para ser capaz de leer ficheros con un formato muy similar a Markdown, el objetivo de este lenguaje propio es permitirle al usuario una gran expresividad para describir la extensión a generar a la vez que otorgarle al programa la capacidad de reconocer secciones específicas del fichero para poder extraer la información necesaria.

El lenguaje Markdown se convirtió rápidamente en la mejor opción para dar completa libertad al usuario a la hora de escribir la descripción del fichero de entrada. El mayor inconveniente del lenguaje Markdown reside en su sintaxis debido a que la gran expresividad del lenguaje viene de la mano de una carencia para controlar el contenido del mismo.

La solución a este problema se ha conseguido a través de la creación de una simplificación del lenguaje Markdown, dicha simplificación añade nuevos tipos de `tokens` así como la eliminación de todos aquellos `tokens` que no eran necesarios para el fichero de entrada así como otras características del lenguaje Markdown como puede ser la capacidad de anidar listas, bloques de código, citas, etc.

4.1.1. Consideraciones del lenguaje

Recrear la sintaxis de Markdown no es una tarea sencilla. Durante la realización del trabajo se planteó la utilización del analizador `remark`[20] para extraer la información del fichero de entrada, pero tras varios intentos se optó por la creación de un analizador propio.

Para lograr simular las características del lenguaje Markdown se le ha dado una mayor importancia a los `tokens` que el analizador léxico es capaz de procesar para que así, en conjunto con el analizador sintáctico sea posible escribir y extraer la información del lenguaje simplificado de Markdown.

4.1.2. Analizador Léxico

La primera decisión a tomar para la creación del lenguaje simplificado de Markdown es seleccionar los `tokens` a incluir del lenguaje original, tras varias pruebas se concluyó que los `tokens` necesarios para escribir el fichero serían todos aquellos que fueran capaces de dar una estructura al fichero, para ello necesario permitir la inclusión de títulos o cabeceras que permitan separar los contenidos en distintas secciones. La sintaxis de Markdown contiene los `tokens` de tipo *header*.

```
/(?<HEADER>^(?:[#]{1, 6})[ ]+(?:.*))/
```

Como añadido al lenguaje se han especificado una serie de cabeceras clave que permiten identificar las distintas secciones que puede incluir el fichero de entrada, estas cabeceras llevan consigo una expresión propia que las diferencia del resto cabeceras, siendo algunas de estas palabras claves: *Extension*, *Chat Settings*, *Language Settings*, etc. Dependiendo de la cabecera clave su nivel de anidación así como posible contenido puede variar.

Para mantener la filosofía de libertad expresiva todas las cabeceras claves del lenguaje Markdown simplificado no distinguen entre mayúsculas y minúsculas. Un aspecto importante a tener en cuenta al utilizar el analizador léxico `Moo` es la implementación manual de dicha no distinción.

```
/(?<DESCRIPTION>^#{2,3}[ ]*[dD][eE][sS][cC][rR][iI][pP][tT][iI][oO][nN])/
```

Otro aspecto importante de la capacidad expresiva del lenguaje es permitir que el usuario sea capaz de resaltar elementos dentro de un párrafo, por este motivo se incluyen varios tipos `tokens` que permiten ordenar y resaltar distintas secciones de una misma descripción. Es posible indicarle al modelo de lenguaje la importancia de ciertos elementos a partir de la utilización del `token` de tipo *highlight*.

```
/(?<HIGHLIGHT>[‘](?:.*)[‘])/
```

Para indicar elementos de código que contengan más de una línea se deberá utilizar el `token` *codeblock* el cual permite insertar código en diferentes lenguajes de programación dentro de una descripción.

```
/(?<CODEBLOCK>^[‘]{3}(?:[a-zA-Z]+)\n(?:([^\n]|\n)*?)[‘]{3}$)/
```

Para describir una serie de elementos contiguos se debe hacer uso del `token` *item list*, este token es capaz de almacenar listas tanto ordenadas como no ordenadas.

```
/(?<ITEM_LIST>^[ ]*(?:(?:[-+*])|(?:[0-9]+[.]))[ ]+(?:.*))/
```

De manera adicional a los `tokens` extraídos del lenguaje Markdown se han añadido una serie de nuevos `tokens` para elementos y secciones específicas del lenguaje. Estos `tokens` Tienen como objetivo extraer información importante de la descripción realizada por el usuario, como pueden ser los parámetros o argumentos del programa, la utilización de listas en un formato clave-valor para la especificación de opciones y permitir la escritura de comentarios, entre muchos otros.

```
/(?<KEY_VALUE>^[ ]*[-][ ]+(?:[a-zA-Z]+)[ ]*(?:[:]|[=])[ ]*(?:[a-zA-Z]+)/
/(?<SHORT_PARAMETER>^[ ]*[-][ ]+(?:[-][a-z])(?:[ ]+(?:.*))?/
/(?<COMMENT>^[[]comment[]][:][ ]+[#][ ]+[(].+[)]/
```

4.1.3. Analizador Sintáctico

Para tener un mayor control sobre la información proporcionada por el usuario se ha diseñado una serie de reglas gramaticales cuyos objetivos son estructurar la información lo máximo posible para poder almacenar los datos extraídos de forma ordenada para luego ser utilizadas por la extensión gh-ai para generar las correspondientes instrucciones.

De la mano de una mayor estructuración se busca, además, una separación de los contenidos para ayudar al usuario a centrar sus esfuerzos en escribir pequeñas y detalladas descripciones que favorezcan la visión general de la extensión a crear. Por ello el diseño de las diferentes secciones del fichero de entrada se basan en las secciones más comunes de un fichero Readme.md, por este motivo muchos bloques buscan detallar los diferentes casos y modos de uso de la extensión, fomentando la introspección del usuario para diseñar el programa en base a los bloques ya pre-establecidos del fichero.

Una de las reglas gramaticales más importantes del lenguaje es el bloque `description`, como se puede ver en la figura 4.2 el bloque está conformado de una cabecera opcional para indicar el inicio de la descripción seguido de una serie de bloques literales que en conjunto forman la descripción de una sección. Esta regla está contenida en la gran mayoría de los bloques que conforman el lenguaje, ya que permite añadir una descripción a la sección especificada.

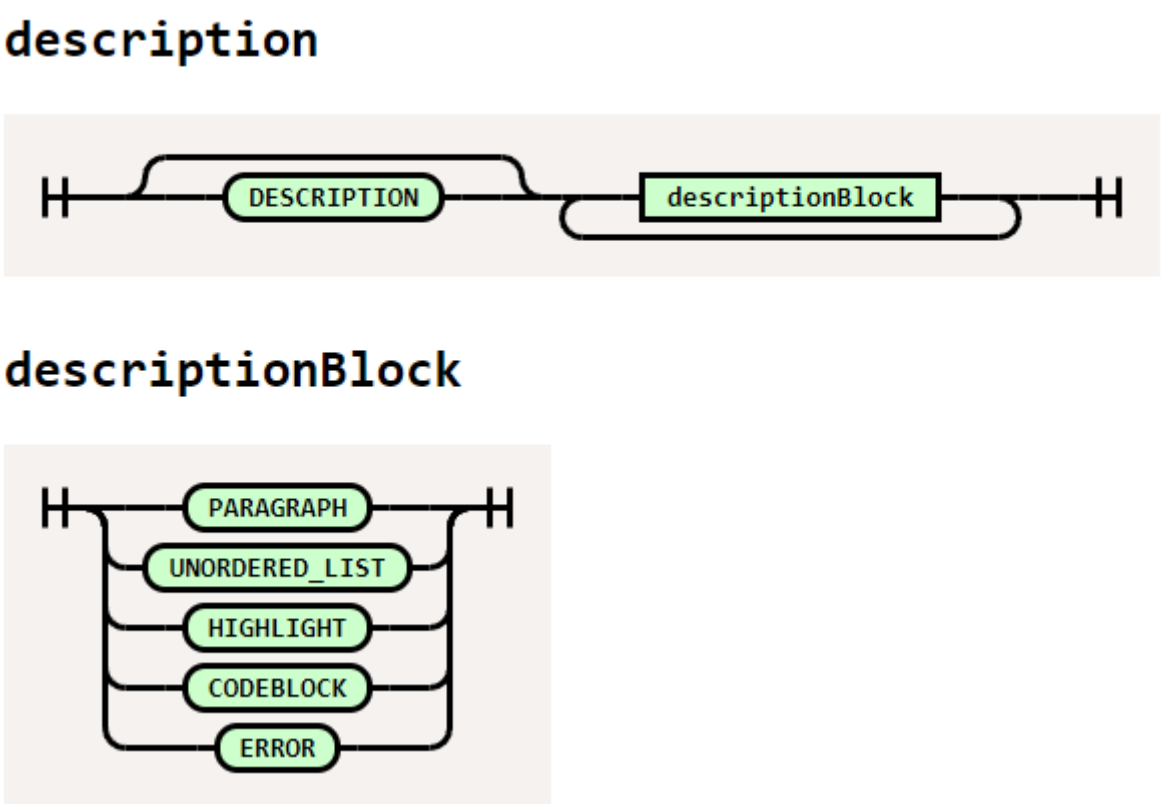

Figura 4.2: Reglas gramaticales para la creación de la sección descripción

Como se ha comentado anteriormente en los apartados 4.1 y 4.1.2, en la búsqueda de una filosofía de libertad expresividad se ha decidido permitir que el usuario sea capaz de escribir las diferentes secciones del lenguaje en cualquier orden, salvo excepciones. Para conseguir conseguir esta libertad las reglas gramaticales se han diseñado en base a la posibilidad de cambiar el orden las consecuentes reglas que lo conforman. Un claro ejemplo de esta filosofía de diseño puede ser vista en la figura 4.3, se puede observar que las diferentes propiedades o apartados que conforman al bloque *Extension* pueden ser escritas en cualquier orden.

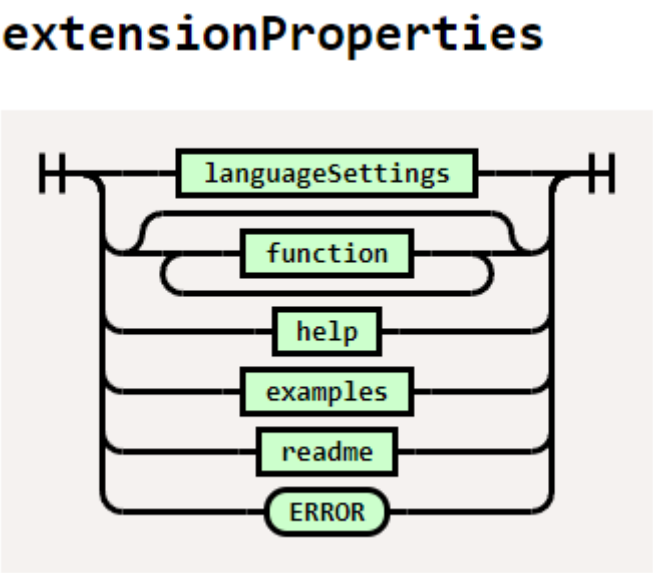

Figura 4.3: Ejemplo de orden indeterminado de las distintas secciones del fichero

Esta decisión conlleva a diseñar las correspondientes acciones semánticas, código que se ejecuta si se cumple una determinada regla gramatical, de manera que tengan no solo tengan en cuenta que las posibles propiedades que conforman un bloque no aparezcan en orden, sino que además cabe la posibilidad de que el usuario repita un mismo tipo de bloque. Este último caso tiene una solución sencilla al solo tener en cuenta el primer bloque de un mismo tipo que se haya declarado dentro de una sección.

4.1.4. Análisis del Esquema del Objeto de entrada

Otro problema derivado de la filosofía de libertad expresiva del lenguaje diseñado radica en la posibilidad de que el usuario no incluya las secciones mínimas requeridas para que el programa funcione correctamente. Una opción para evitar este problema sería obligar al usuario a añadir dichas secciones al fichero a través de las reglas gramaticales, esta idea contradice a lo explicado en el apartado 4.1.3 acerca de permitir al usuario escribir en el orden que desee. A su vez los bloques cuyo propósito es configurar parámetros de la conversación, como los que se pueden apreciar en la figura 4.4 no poseen la capacidad de detectar las configuraciones válidas y descartar cualquier otra opción que no sea correcta.

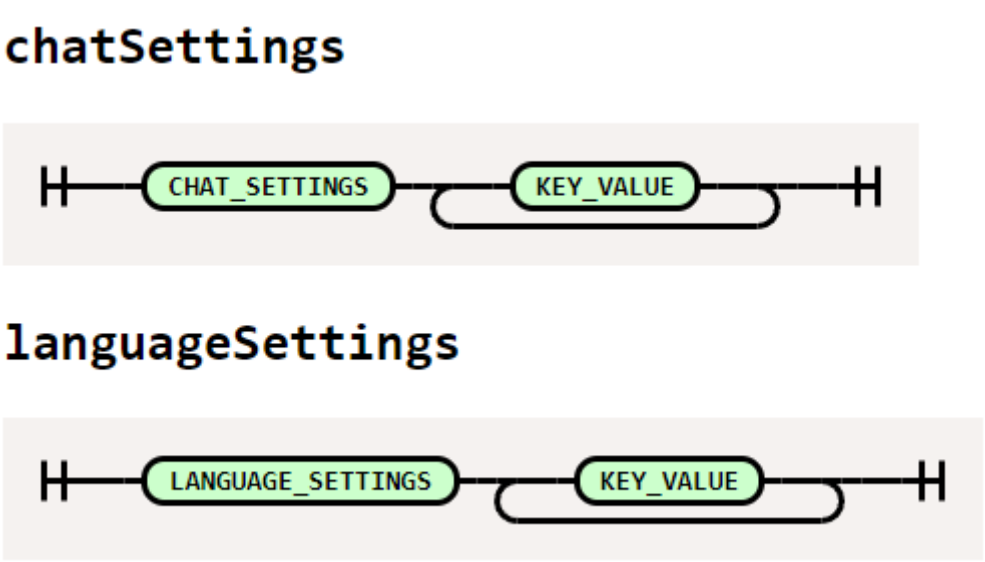

Figura 4.4: Bloques dedicados a la configuración de los parámetros de la conversación

El usuario es capaz de rellenar la información utilizada para construir las instrucciones con datos incorrectos, inexistentes o incoherentes. Para evitar que la información incorrecta sea suministrada al generador de instrucciones se ha optado por la creación de un esquema para el objeto JavaScript creado por el analizador sintáctico tras terminar su ejecución.

El esquema mostrado en la figura 4.5 tiene como objetivo comprobar si la información extraída del fichero de entrada contiene todo los elementos necesarios para la correcta generación de las instrucciones que van a ser utilizadas para realizar la conversación con el modelo de lenguaje. Para lograr este objetivo se ha decidido utilizar el paquete Zod para generar y validar el esquema.

La generación del esquema sirve como paso intermedio entre el primer y segundo módulo del programa, un cambio en el esquema requiere modificar ambas partes para adecuarse al nuevo formato, por otra parte, la validación del esquema complementa el procesamiento de los errores del fichero de entrada, en casos donde el analizador sintáctico no sea capaz de identificar un error, como los ya mencionados anteriormente, el esquema es el que se encargará de gestionar el error. Combinando ambos es posible indicar al usuario la posición y sección específica del fichero de entrada en el cual se encuentra el error.

4.2. Generador de Instrucciones

Una vez extraída la información crucial del fichero de entrada y comprobada por el esquema, el siguiente paso a realizar consiste en la construcción de las instrucciones por medio de diferentes plantillas. por cada una de estas instrucciones se construye un objeto como el se puede observar en la figura 5.9.

Pese a no existir una afirmación explícita acerca del entendimiento por parte de los modelos de lenguaje de diferentes sintaxis como pueden ser XML o Markdown. Se tiene un consenso general debido a la naturaleza de la base de datos con la cual dichos modelos han sido entrenados. La primera decisión consiste en utilizar la sintaxis Markdown para delimitar las distintas secciones de la instrucción además de permitir resaltar elementos y diferenciar código de texto.

```
{
  "chatSettings": {
    "language": "string"
    "nickname": "string"
  },
  "extension": {
    "languageSettings": {
      "language": "string",
      "style": "string",
      "specification": "string",
    },
    "files": [
      {
        "name": "string",
        "description": "string",
        "functions": [
          {
            "name": "string",
            "description": "string",
            "query": "string",
            "template": "string",
          }
        ],
        "help": {
          "usage": "string",
          "header": "string",
          "parameters": [
            {
              "parameter": "string",
              "argument": null | "string",
              "description": "string",
            },
          ],
          "arguments": [
            {
              "argument": "string",
              "description": "string",
            },
          ],
          "footer": "string",
        }
      }
    ],
    "examples": [
      {
        "input": "string",
        "output": "string",
      }
    ]
  }
}
```

Figura 4.5: Esquema del objeto de entrada

Antes de empezar la conversación con el modelo de lenguaje, es necesario otorgarle un contexto, una personalidad y una serie de reglas para modificar el comportamiento de la inteligencia artificial. A la hora de trabajar con la interfaz de asistentes de OpenAI esta primera instrucción se realiza a través de la instrucción del sistema del agente. Utilizando la información recopilada en el objeto de entrada es posible configurar al agente para seguir un contexto específico en referencia a la generación de extensiones para GitHub CLI.

Por otra parte se ha decidido separar las instrucciones a utilizar durante la conversación para permitir al modelo de lenguaje trabajar en una única tarea por cada petición, la separación sigue uno de los patrones comentados en el capítulo 1, el cual consiste en realizar una primera fase de análisis e introspección[2] con el objetivo de forzar al modelo a extraer los requisitos fundamentales de la extensión. Para complementar el análisis se busca mantener en todo momento dicha información recopilada en forma de tres listas que contienen: Los requisitos, variables utilizadas y función que componen la extensión. Cuya función busca simular una pequeña memoria.

Los conceptos de *one-shot* y *few-shot* dependen completamente del usuario que escriba el fichero de entrada debido a que el programa no es capaz de generar casos de ejemplo para suministrarle al modelo de lenguaje.

Una vez realizada la fase de introspección se busca realizar una petición por cada función indicada en el fichero de entrada. Empezando con la función *main* la tercera instrucción generada busca gestionar los parámetros de entrada de la extensión haciendo uso de la información objeto de entrada. A su vez se construye una cuarta instrucción para la creación de la función *help* del programa. Tras ello, por cada otra función se construye su correspondiente instrucción en base a una misma plantilla genérica que contiene la información importante acerca de la función a generar.

Por último se realiza una fase de post-procesado en el cual se busca cohesionar todo el código generado dentro de un mismo fichero, seguido se realiza la escritura de comentarios para permitir al usuario entender el código escrito por el modelo y por último una corrección general para adecuar el código a una guía de estilo proporcionada por el usuario. Tras ejecutar esta instrucción el modelo de lenguaje deberá crear el fichero correspondiente dentro del directorio de salida del programa, haciendo uso de las herramientas externas que es capaz de ejecutar.

En caso de especificarse la generación de un fichero `readme.md`, se construye una instrucción especial con el objetivo de generar dicho fichero. A su vez de manera excepcional la creación del registro de la conversación sigue la misma metodología comentada en este apartado. En la figura 4.6 se puede observar un extracto de la plantilla utilizada.

```
# User prompts

For each file requested by the user, its corresponding code has been generated.

{{#userPrompts}}
## {{#extension}}File{{/extension}} {{title}}

{{#prompts}}
### Petition {{title}}

```md
{{text}}
```

Assistant Response

{{response}}

Petition Usage

Total tokens used: **{{usage.total_tokens}}**
* Tokens used by the gh-ai generated prompt: **{{usage.prompt_tokens}}**.
* Tokens used by the LLM to generate the answer: **{{usage.completion_tokens}}**.

{{/prompts}}
Total Usage

Total tokens used to generate the {{#extension}}file{{/extension}}: **{{usage.totalTokens}}**.
* Total tokens used by the gh-ai generated prompts: **{{usage.totalPromptTokens}}**.
* Total tokens used by the LLM to generate the answer: **{{usage.totalCompletionTokens}}**.

{{/userPrompts}}
```

Figura 4.6: Plantilla utilizada para el registro de la conversación.

4.3. Modularidad del programa

Un aspecto importante del diseño de los `tokens`, descrito previamente en el aparato 4.1.2, es su capacidad para funcionar como bloques de construcción para las diferentes reglas gramaticales mostradas en el apartado 4.1.3, por este motivo el compilador de Markdown simplificado es escalable al permitir la construcción de nuevas reglas sin la necesidad de modificar las previamente diseñadas. De la misma manera la utilización de zod como generador de esquemas permite que los cambios realizados en la construcción del objeto de entrada sean fácilmente aplicables al esquema mostrado en la figura 4.5.

Mientras no se realicen modificaciones que cambien por completo la estructura del esquema del objeto de entrada, el generador de instrucciones podrá seguir realizando su trabajo con tan solo unas pocas modificaciones. La adición de nuevas propiedades al objeto de entrada nunca interferirá con el proceso de generación, debido a que el generador no tendrá en cuenta las nuevas propiedades.

Desde el comienzo de la realización del proyecto se tuvo la idea de permitir que la extensión `gh-ai` fuera capaz de aceptar no solo diferentes modelos de lenguaje sino además permitir la utilización de distintas APIs, y dependiendo de esta indicar el modelo específico. Pese a solo implementar la API de OpenAI, el programa acepta la incorporación de nuevas interfaces.

Ampliando la idea de modularidad conseguida en el uso de las APIs, el programa permite además la generación de instrucciones en base al tipo de ayuda requerido por el usuario permitiendo así seleccionar los elementos concretos y necesarios del fichero de entrada, como se había explicado previamente en el apartado 3.5, para la generación de las instrucciones y seleccionar las plantillas correspondientes.

Capítulo 5

Implementación

Tras introducir todas las tecnologías a utilizadas para desarrollar la extensión `gh-ai` en el capitulo 2 y explicar las diferentes decisiones de diseño en el capítulo 4 es momento de enseñar la implementación realizada. En este capítulo se desglosan las diferentes partes que conforman la extensión `gh-ai` siguiendo un orden similar, indicando los detalles y el porqué de ciertas diferencias con respecto al diseño planteado originalmente.

5.1. Compilador de Markdown Simplificado

Como se ha comentado en el capítulo anterior, el compilador del lenguaje Markdown simplificado consta de tres componentes claramente diferenciados: El analizador léxico, el analizador semántico junto con las correspondientes acciones semánticas y, de manera adicional, el esquema del objeto de entrada.

5.1.1. Analizador Léxico

El paso previo a la implementación de la sintaxis del lenguaje es la creación del analizador léxico que permita construir los `tokens` que el analizador sintáctico utilizará para la generación de las reglas gramaticales. A lo largo de la explicación realizada en el apartado 4.1.2 se mostraron una serie de expresiones regulares que han sido utilizadas para diferenciar ciertos `tokens` de otros. La lista completa de `tokens` utilizados por el lenguaje puede ser consultada en el repositorio de la extensión disponible en el apéndice A.

```
const HEADER_TOKEN = {
    match: /^(?:#{1,6})[ ]+(?:.*)/,
    value: (value) => {
        const HEADER_CAPTURING = /^(#{1,6})[ ]+(.*)/;
        const RESULT = HEADER_CAPTURING.exec(value);
        return {
            depth: RESULT[1].length;
            title: RESULT[2];
        };
    }
};s
```

Figura 5.1: Implementación del token *header*.

Para generar un analizador léxico es necesario indicar todos los `tokens` a tener en cuenta por parte del lenguaje. Por ello se ha creado una lista con todos los `tokens` del lenguaje, siguiendo el formato especificado por el generador de analizadores léxicos Moo. La figura 5.1 muestra la implementación de una cabecera de sección genérica, utilizada especialmente para realizar gestión de errores. Para gestionar las diferentes secciones específicas del lenguaje se han definido `tokens` A modo de cabeceras clave, en la figura 5.2 se pueden observar dos ejemplos de los distintos tokens reservados. los `tokens` *Extension*, *Function* y *Usage* poseen parámetros adicionales que les permiten almacenar la variable especificada en su `token`.

```
// Token sin variable asociada
DESCRIPTION: { match: new RegExp(`^[#]{2,3}[ ]+${toCaseInsensive('description')}`) },

// Token con variable asociada.
EXTENSION: {
  match: new RegExp(`^[#][ ]+${toCaseInsensive('extension')}[ ]+(?:gh-[a-z][a-z0
    -9]*(?:[-][a-z0-9]+)*)`),
  value: (value) => {
    const EXTENSION_CAPTURING = new RegExp(`^[#][ ]+${toCaseInsensive('extension')}[ ]+(
    gh-[a-z][a-z0-9]*(?:[-][a-z0-9]+)*)`);
    const RESULT = EXTENSION_CAPTURING.exec(value);
    return RESULT[1]
  },
}
```

Figura 5.2: Extracto de la implementación de cabeceras.

Se puede observar una función recurrente en todos los `tokens` correspondiente a las cabeceras del lenguaje, la función *toCaseInsensive*. Como se observa en la figura 5.3 la función tiene como objetivo construir una expresión regular que tenga en cuenta tanto mayúsculas como minúsculas dada una cadena de entrada.

```
/** ... */
const isLetter = (char) => {
  const LETTER_REGEXP = /[a-zA-Z]/;
  return LETTER_REGEXP.test(char);
};

/** ... */
function toCaseInsensive(word) {
  const REGEX_SOURCE = word.split('').map((char) => {
    if (isLetter(char)) {
      return `[${char.toLowerCase()}${char.toUpperCase()}]`;
    }
    return char;
  });
  return REGEX_SOURCE.join('');
}
```

Figura 5.3: Funciones auxiliares para la generación de los tokens.

5.1.2. Analizador Sintáctico

Una vez construido el analizador léxico generado por Moo ya es posible realizar la lectura de un fichero de entrada para recuperar los distintos `tokens` que lo conforman. Utilizando la lista de `tokens` declarados es posible generar todas las reglas gramaticales necesarias para la sintaxis del programa. Las primeras reglas declaradas dentro del fichero con extensión `.ne` Son aquellas correspondientes a los bloques principales del lenguaje Markdown simplificado. Como se ha comentado en el apartado 3.5 solo existen dos bloques principales, en la figura 5.4 se puede observar la implementación de la sintaxis de dichos bloques, así como de la inclusión de reglas gramaticales auxiliares para facilitar el trabajo de las acciones semánticas y dar legibilidad al código. A su vez se puede observar la implementación de la filosofía de libertad expresiva explicada en el apartado 4.1.3 a través de la regla gramatical *extensionProperties* que permite escribir los distintos bloques secundarios en un orden indeterminado.

```
#################################### Main Rule ####################################
inputObject -> (mainProperties {% id %}):* %EOF {% buildInputObject %}

mainProperties -> chatSettings {% id %} | extension {% id %}

############################## Chat Settings block rules ##############################
chatSettings -> %CHAT_SETTINGS (%KEY_VALUE {% id %}):+ {% buildSettings['chatSettings'] %}

############################## Extension block rules ##############################
extension -> %EXTENSION description (extensionProperties {% id %}):* {% buildExtension %}

extensionProperties ->
    languageSettings {% id %}
  | function         {% id %}
  | help             {% id %}
  | examples         {% id %}
  | readme           {% id %}

```

Figura 5.4: Reglas gramaticales correspondientes a los bloques principales del lenguaje. Extraído del fichero 6.

La implementación de los distintos bloques secundarios puede ser observada en la figura 5.5. Todas las reglas gramaticales presentes en la figura pueden ser anidadas solamente dentro de un bloque *Extension*, salvo aquellos apartados que forman parte de un bloque secundario como se pueden observar en las reglas *function*, *help*, etc. Un elemento recurrente tanto en la figura 5.4 como en la figura 5.5 es la inserción de código JavaScript cuyo propósito es realizar una llamada a la función correspondiente en cada una de las reglas gramaticales, haciendo especial énfasis en la función `id` la cual se encuentra en la mayoría de reglas y cuyo propósito es devolver el único token detectado dentro de la regla gramatical.

```
########################## Language Settings block rules ##############################
languageSettings ->
    %LANGUAGE_SETTINGS (%KEY_VALUE {% id %}):+ {% buildSettings['languageSettings'] %}

############################### Description block rules ###############################
description ->
    (%DESCRIPTION {% id %}):? (descriptionBlock {% id %}):+ {% buildDescription %}

descriptionBlock ->
    %PARAGRAPH {% id %}
  | %ITEM_LIST {% id %}
  | %HIGHLIGHT {% id %}
  | %CODEBLOCK {% id %}

################################ Function block rules #################################
function ->
    %FUNCTION description (query {% id %}):? (template {% id %}):? {% buildFunction %}
# Más reglas gramaticales del bloque function

################################## Help block rules ###################################
help ->
    %HELP usage (%PARAGRAPH {% id %}):* (arguments {% id %}):? (parameters {% id %}):? (%
    PARAGRAPH {% id %}):* {% buildHelp %}
# Más reglas gramaticales del bloque help
################################ Examples block rules #################################
examples ->
    %EXAMPLES (example {% id %}):+ {% buildExamples %}
# Más reglas gramaticales del bloque Examples

################################ Readme block rules ###################################
readme ->
    %README sections {% buildReadme %}
# Más reglas gramaticales del bloque readme
```

Figura 5.5: Bloques secundarios que pueden ser anidados dentro de un bloque *Extension*. Extraído del fichero 6.

5.1.3. Acciones semánticas

Como se ha comentado en el apartado 5.1.2, cada regla gramatical del lenguaje Markdown simplificado tiene asociada una o varias acciones semánticas. Una acción semántica no es sino código JavaScript, cuando el analizador sintáctico detecta que una regla gramatical ha sido encontrada dentro del fichero ejecuta automáticamente su acción semántica correspondiente. El código contenido dentro de la acción semántica es de longitud indefinida, pero por motivos de legibilidad es recomendado realizar una única llamada a una función, las distintas funciones utilizadas en las acciones semánticas son importadas de un fichero JavaScript a través de la cabecera del fichero de la gramática.

En la figura 5.6 se puede observar un extracto de las funciones que contiene el fichero de acciones semánticas del proyecto. Las funciones cuyo nombre empiezan por *build* son todas aquellas que construyen el objeto de entrada, estas funciones tienen como objetivo extraer toda la información crucial contenida en los `tokens` captados por la regla

gramatical. Para evitar problemas relacionados con la introducción de múltiples bloques del mismo tipo, como por ejemplo dos bloques *Language Settings*, se ha creado la función auxiliar *findFirstPropertyType* la cual permite buscar el primer bloque de cada tipo dentro del fichero de entrada.

```
/** ...*/
function findFirstPropertyType(properties, targetType) {
  return properties.find((property) => {
    return property.type === targetType;
  })?.content;
}

/** ... */
function buildInputObject([properties, eof]) {
  return {
    chatSettings: findFirstPropertyType(properties, 'chatSettings'),
    extension: findFirstPropertyType(properties, 'extension'),
  };
}

/** ... */
function buildExtension([extensionToken, description, extensionProperties]) {
  const FUNCTIONS = extensionProperties.filter((property) => {
    return property.type === 'function';
  });
  const HELP = findFirstPropertyType(extensionProperties, 'help');
  return {
    type: 'extension',
    content: {
      languageSettings: findFirstPropertyType(extensionProperties, 'languageSettings'),
      files: [buildFile([extensionToken, description, FUNCTIONS, HELP])],
      examples: findFirstPropertyType(extensionProperties, 'examples'),
      readme: findFirstPropertyType(extensionProperties, 'readme'),
    }
  };
}
```

Figura 5.6: Extracto del fichero 7. Contiene todas las acciones semánticas del lenguaje.

5.2. Esquema del Objeto de Entrada

Con el objeto de entrada construido a partir de las acciones semánticas, el siguiente paso a implementar ha consistido en la creación del esquema del objeto de entrada para realizar la validación del objeto y llevar a cabo la gestión de los posibles errores no resueltos por el analizador sintáctico.

5.2.1. Generación del Esquema utilizando Zod

La construcción del esquema se realiza en base a objetos separados que representan los diferentes tipos posibles que puede contener una propiedad dentro de un objeto JSON, para la creación de del esquema del objeto de entrada se han utilizado los tipos

correspondientes a cadenas de texto, listas y objetos literales, así como una multitud de opciones adicionales que permiten añadir más complejidad al esquema como pueden ser la posibilidad de que ciertas propiedades del objeto sean opcionales. En la figura 5.7 se puede observar un extracto de la implementación del esquema, en el que se muestra la construcción por bloques mencionada anteriormente. Por otra parte en la figura 4.5 se encuentra el esquema completo esperado por parte de la extensión `gh-ai`.

```
const LANGUAGE_SETTINGS_SCHEMA = z.object({
  language:      z.string(),
  specification: z.string().optional(),
  style:         z.string().optional().default('Google'),
})
.required({ language: true }).strict();

const EXTENSION_SCHEMA = z.object({
  files:             z.array(FILES_SCHEMA),
  languageSettings: LANGUAGE_SETTINGS_SCHEMA,
  examples:          z.array(EXAMPLES_SCHEMA).optional(),
  readme:            FILES_SCHEMA.optional(),
})
.required({ languageSettings: true, files: true,}).strict();

const INPUT_SCHEMA = z.object({
  chatSettings: CHAT_SETTINGS_SCHEMA,
  extension:    EXTENSION_SCHEMA,
})
.required({ chatSettings: true, extension: true }).strict();
```

Figura 5.7: Extracto del fichero 8 con la implementación del esquema.

La utilización del esquema del objeto de entrada permite controlar ciertos aspectos del fichero de entrada que al analizador sintáctico le supondría un aumento en la complejidad y en la estructura de las reglas gramaticales. Como se observa en la figura 5.7 tanto el esquema de *Chat Settings* como el de *Language Settings* permite controlar las propiedades que recibe por parte del fichero de entrada, permitiendo solo aquellas configuraciones que hayan sido previamente indicadas.

Para facilitar la escritura del fichero de entrada se ha implementado, además del esquema, la gestión y manejo de errores del esquema. Por defecto el paquete Zod devuelve mensajes de error poco útiles para los usuarios finales, por este motivo se ha sobrescrito el manejo de errores por defecto del paquete. La figura 5.8 muestra solo el primer error posible que se puede encontrar al validar el objeto de entrada, se puede observar como la función crea un nuevo mensaje de error, utilizando la información proporcionada por el error, el cual será devuelto por la función.

```
const customErrorMap = (issue, ctx) => {
  let amountOfHashs = 0;
  let errorMsg = '';
  switch (issue.code) {
    case z.ZodIssueCode.invalid_type:
      if (issue.path.length === 0) {
        return { message: 'Expected an object. Received nothing' };
      }
      issue.path.map((path) => amountOfHashs += isNaN(path));
      errorMsg = `Expected a ${'#'.repeat(amountOfHashs)}${issue.path[issue.path.length -
    1].toString().toUpperCase()} property `;
      amountOfHashs = 1;
      errorMsg += 'in ';
      issue.path.forEach((path, index) => {
        const ARROW_STRING = (index < issue.path.length - 1)? ' -> ' : '';
        if (isNaN(path)) {
          errorMsg += `${'#'.repeat(amountOfHashs)}${path.toUpperCase()}${ARROW_STRING}`;
          amountOfHashs++;
        } else {
          errorMsg += `at index ${path} inside the array${ARROW_STRING}`;
        }
      });
      errorMsg += `with a ${issue.expected.toUpperCase()} value. Received `;
      errorMsg += `${(issue.received === 'undefined')? 'nothing' : `a(n) ${issue.received
    .toUpperCase()} value instead`}.`;
      return { message: errorMsg };
      // Más código...
    }
};
```

Figura 5.8: Extracto del manejo de errores del esquema

5.3. Generación de Instrucciones

Siguiendo las decisiones de diseño tomadas en el apartado 4.2 se han escrito las diferentes plantillas necesarias para la realizar la comunicación con el modelo de lenguaje. Estas plantillas se encuentran almacenadas en el directorio `gh-ai/src/templates`, dentro de este directorio se han escrito todas las plantillas que no dependen del tipo de ayuda y la API seleccionadas. Dentro de esta carpeta se ha escrito un fichero JavaScript cuyo propósito es gestionar la lectura y escritura de las diferentes plantillas del programa.

Como se observa en la figura 4.6, las plantillas escritas para el programa consisten únicamente en texto plano intercalado con la sintaxis de etiquetas utilizada por el paquete Mustache para interpolar los datos extraídos del fichero de entrada dentro de la plantilla. Se ha utilizado Markdown para escribir el contenido de las plantillas debido a que permite una mejor visualización del texto gracias a las ayudas visuales proporcionadas por los editores de texto como Visual Studio Code, además de por los motivos expuestos en el apartado 4.2 acerca de cómo los modelos de lenguajes leen perfectamente la sintaxis de Markdown.

Por otro lado, la extensión `gh-ai` hace uso de un segundo fichero JavaScript cuyo propósito es recorrer el objeto de entrada para generar las distintas instrucciones a

utilizar durante la conversación con el modelo de lenguaje, el tipo de generador de instrucciones dependerá del tipo de ayuda. Como se muestra en la figura 5.9 un objeto instrucción esta conformado por su contenido y la respuesta correspondiente, además se pueden observar distintas propiedades utilizadas durante la ejecución de la conversación como puede ser la propiedad `executeTool` la cual le indica al modelo de lenguaje que herramienta externa utilizar.

```
{
    title: `general idea of ${MAIN_FILE.name}.`,
    text: TEMPLATES.EXTENSION.FILE_GENERAL_IDEA(MAIN_FILE),
    reponse: undefined,
    usage: {},
    executeTool: undefined,
    askForChanges: false,
}
```

Figura 5.9: Ejemplo de un objeto instrucción.

Todas las instrucciones generadas son almacenadas en una lista, la cual será utilizado para calcular el gasto de `tokens` de la llamada a la API así como para generar el registro de la conversación. El fichero de registro sigue la misma metodología que el resto de plantillas, con la única diferencia de utilizar los datos de salida del modelo de lenguaje en lugar del fichero de entrada.

5.4. Comunicación con la API

Como se ha comentado previamente en el apartado 4.3, la extensión `gh-ai` está preparada para aceptar diferentes implementaciones de llamadas a una API de modelos de lenguaje. Actualmente `gh-ai` solo posee una única implementación oficial, la Interfaz de asistentes de OpenAI.

5.4.1. Interfaz de OpenAI

La interfaz de asistentes de OpenAI, previamente explicada en el apartado 2.3, ha sido implementaba a través de tres ficheros JavaScript diferentes separando la comunicación con la API de OpenAI de la implementación propia por parte de la extensión `gh-ai`. Como se puede observar en la figura se muestra un pseudocódigo de la llamada a OpenAI realizada por la extensión `gh-ai`, se puede señalar la construcción del objeto OPENAI donde se puede observar la capacidad del programa de extraer la información de las variables de entornos explicadas en el apartado 3.4.

```
async function(responseObject, outputDirectory, options) {
  // Creación de la API de OpenAI
  const OPENAI = new OpenAI({
    apiKey: process.env.OPENAI_API_KEY,
    organization: process.env.OPENAI_ORG
  });

  // Creación del asistente y del hilo a utilizar.
  let assistant = await createOrRetreiveAssistant(...);
  let thread = await createOrRetreiveThread(...);

  for (const PROMPT of responseObject.userPrompts) {
    // Se realiza la llamada a la API de OpenAI
    let result = await call(...);

    // Si el modelo de lenguaje requiere ejecutar una función
    while (result.status === 'requires_action') {
        // Se ejecuta la función y se vuelve a llamar a la API con el resultado.
        const TOOL_OUTPUT = await manageToolAction(...);
        result = await call(...);
    }

    // Se inserta la respuesta del modelo en el objeto del prompt.
    await addResponseToPrompt(...);

    if (result.status === 'error') { return; }
  }

  // Se calcula el uso de tokens del programa.
  await calculatePromgramTotalUsage(...);
}
```

Figura 5.10: Pseudocódigo de la llamada a la API de OpenAI por parte de gh-ai.

Entrando más en detalle acerca de la utilización de la API de OpenAI cabe mencionar la facilidad con la que se pueden construir los diferentes elementos que conforman la conversación con el modelo de lenguaje. La creación tanto de asistentes como de Hilos se realiza a través de dos funciones distintas, la extensión `gh-ai` permite conservar el asistente creado durante la ejecución del programa así como de su hilo correspondiente para que el usuario pueda volver a insertar dicho asistente y conservar la conversación y el contexto generado previamente, por este motivo la implementación tiene en cuenta si se va a construir un nuevo asistente o utilizar uno ya creado, este mismo proceso se realiza para los hilos. En caso de utilizar un asistente ya creado el programa comprobará si su contexto concuerda con la configuración actual del fichero de entrada, en caso de discrepancias se preguntará al usuario si se quiere actualizar el asistente o ejecutar el programa con el contexto antiguo.

Antes de ejecutar la creación del asistente se realiza una extracción de las herramientas externas que dicho asistente será capaz de ejecutar, la extensión `gh-ai` Implementa dos herramientas externas que permiten al asistente generar un fichero de texto con el código previamente generado por el modelo.

Como se observa en la figura 5.10 por cada instrucción generada se realiza una

```
async function call(...) {
    // Se anade el mensaje al hilo
    await openai.beta.threads.messages.create(
      threadID,
      { role: 'user', content: prompt }
    );
    // Se crea una ejecución para obtener la respuesta del modelo de lenguaje
    while (currentTry < MAX_TRIES) {
        // Se crea la ejecución.
        let run = await openai.beta.threads.runs.create(
          threadID,
          {
            assistant_id: assistantID,
            tool_choice: executeTool ?? 'none',
          }
        );
        // Se realiza un segundo bucle para coprobar el estado de la ejecución.
        const RESULT = await checkRunStatus(run);

        // Ante cualquier estado terminal se termina la ejecución.
        if (RESULT.status !== 'rate_limit') { return RESULT; }

        // En caso de Rate_Limit, se reintenta la ejecución
        currentTry++;
        await sleep(DELAY);
    }
}
```

Figura 5.11: Pseudocódigo de la creación de una ejecución.

comunicación con la inteligencia artificial. El proceso de comunicarse con el modelo de lenguaje se realiza a través de las ejecuciones. Como se comentaba en el apartado 2.3, cada ejecución comienza al momento de insertar un nuevo mensaje en la conversación, para ello primero se debe envolver la instrucción dentro del formato de mensaje de OpenAI antes de crear la nueva ejecución.

En la figura 5.11 se muestra pseudocódigo representativo de la llamada a OpenAI a través de la utilización de ejecuciones. Se puede notar que existe un número máximo de intentos, esto es debido a la política de restricción de llamadas impuesta por OpenAI dentro de un determinado rango de tiempo, dependiendo del plan y modelo pagado por el usuario los límites serán más o menos restrictivos. Por este motivo durante la ejecución de la extensión es bastante probable encontrarse con errores pertenecientes al límite de llamadas por rango de tiempo. La extensión `gh-ai` tiene en cuenta esta limitación y por tanto se ha implementación el número de intentos, permitiendo volver a ejecutar la conversación pasados tras terminar la restricción de tiempo, pese a los esfuerzos para evitar la terminación del programa, si la ejecución falla más de diez veces, el programa cancelará la conversación con el modelo de lenguaje y terminará la ejecución del programa.

Entrando más en detalle acerca de la función `checkRunStatus` mostrada en la figura 5.11 y en la figura 5.12 se encarga de comprobar constantemente el estado actual de la ejecución. Dependiendo del estado, el programa se encarga de mostrar información útil

```
while (!process.env.GRACEFUL_SHUTDOWN) {
  // Se obtiene la ejecución actual
  const RUN = await openai.beta.threads.runs.retrieve(threadID, runID);

  // Dependiendo del estado de la ejecución se realizan distintas acciones
  switch (RUN.status) {
    case 'failed':
      if (RUN.last_error.code === 'rate_limit_exceeded') {
        return 'rate_limit';
      }
      return RUN.status;

    case 'expired':          return 'failed';
    case 'cancelled':        return 'failed';
    case 'requires_action':  return RUN.status;
    case 'completed':        return RUN.status;
    case 'queue':            break;
    case 'in_progress':      break;
    case 'cancelling':       break;
    default: // Error
  }
  // Para evitar un Rate_Limit se realiza un pequeno retraso
  await sleep(DELAY);
}
```

Figura 5.12: Pseudocódigo de la gestión de los diferentes estados de una ejecución.

al usuario. En caso de que la ejecución se encuentre en un estado terminal se devuelve dicha información al bucle principal del apartado 5.11.

Tras completar la llamada, como se ve en la figura 5.10, `gh-ai` Comprueba que la ejecución haya sido exitosa, en caso contrario cancela la conversación con el modelo de lenguaje, guardando la respuesta de la inteligencia artificial en la correspondiente instrucción que empezó la ejecución, para ello se debe extraer los mensajes generados del modelo a través del objeto hilo, comentado en la figura 2.2, buscando los últimos mensajes generados por el asistente y extrayendo su contenido. En la figura 5.13 se puede observar un ejemplo de la lista de mensajes que contiene el objeto hilo, cuanto más larga sea la conversación mayor será el tamaño de la lista.

```
{
  "object": "list",
  "data": [
    {
      "id": "msg_abc123",
      "object": "thread.message",
      "role": "user",
      "content": [
        {
          "type": "text",
          "text": {
            "value": "Given the following *description* your job is to generate code...",
            "annotations": []
          }
        }
      ],
    },
    // Más mensajes...
  ]
}
```

Figura 5.13: Ejemplo de una lista de objetos mensaje.

Por último se realiza el cálculo de la cantidad de tokens utilizados tanto por el programa al introducir la instrucción, como por el modelo de lenguaje para generar la respuesta. Toda la información es recopilada y enviada al generador de instrucciones para generar el fichero de registro de la conversación como se ha comentado previamente en el apartado 5.3.

Cabe mencionar que las figuras mostrada durante este apartado no son más que pseudocódigo para dar una idea general de la implementación realizada, para una visión más detallada y completa sobre cómo se realiza la llamada a la API de OpenAI el repositorio del proyecto se encuentra enlazado en el apéndice A.

5.5. Pruebas, Rendimiento y Resultados

Durante la realización del proyecto se han llevado a cabo múltiples pruebas para comprobar la efectividad del código y el diseño implementado. Para realizar las pruebas se ha hecho uso del foro público `gh-extension` alojado en la página web de GitHub, se han buscado ejemplos con diferentes grados de complejidad. Dichos ejemplos pueden ser encontrados en el directorio `gh-ai/examples` en el que se encuentran un total de cinco ejemplos de uso en conjunto con sus respectivos resultados. La investigación acerca del rendimiento de las plantillas utilizadas no ha sido realizada de manera exhaustiva por lo que los resultados obtenidos no son del todo fiables, aun así se han encontrado los siguientes resultados:

- A mayor descripción de la extensión a crear más probable es que el modelo de lenguaje realice el código pedido.
- En caso de que el modelo de lenguaje no sea capaz de entender la descripción, su

modo de respuesta será escribir una plantilla con comentarios, generalmente con puntos claves encontrados por el modelo de lenguaje.

- Cuanto mayor sea el código generado, más probable es que el modelo de lenguaje corte o resuma código previamente escrito resultando muchas veces en ficheros con información incompleta que debe ser rellenada con los datos del registro de la conversación.
- El código generado por el modelo de lenguaje requiere de modificación por parte del usuario para poder ser ejecutado correctamente.

Por último se puede observar una comprobación entre el código de ejemplo, en la figura 5.14[21], y el código generado por el modelo de lenguaje a través de utilizar la extensión `gh-ai`, en la figura 5.15.

```
gh issue list --search "sort:created-asc" --label $LABEL -R $TEMPLATEREPO --json 'title,
    body' > $TEMPLATEISSUES

# Check if issues were found
if jq -e '. | length == 0' "$TEMPLATEISSUES" > /dev/null; then
    echo "No issues found in $TEMPLATEREPO with label \"$LABEL\""
fi

for row in $(cat $TEMPLATEISSUES | jq -r '.[] | @base64'); do
    _jq() {
        echo ${row} | base64 --decode | jq -r ${1}
    }
    TITLE=$(_jq '.title' | sed -e 's/^[[:space:]]*//' -e 's/[[:space:]]*$//')

    if gh issue list | grep -e ".*$TITLE.*" >/dev/null ; then
        echo "Issue already exists: ($TITLE)"
    else
        echo Copying issue: \'$TITLE\'
        echo "$(_jq '.body')" > $TMPFILE
        cmd="gh issue create --title \"$(_jq '.title')\" --body-file $TMPFILE"
        eval $cmd
        # Clean up
        if [ "$VERBOSE" = false ]; then
            rm $TMPFILE
        fi
    fi
done
```

Figura 5.14: Extracto del código escrito para la extensión gh-cpissues

```
# Get issues with the specified label from source repository
issues=$(gh issue list -R $source_repo -l "$label" --json title,body)

if [ -z "$issues" ]; then
  echo "No issues with label '$label' found in $source_repo."
else
  for row in $(echo "${issues}" | jq -r '.[] | @base64'); do
    _jq() {
      echo ${row} | base64 --decode | jq -r ${1}
    }

  issue_title=$(_jq '.title')
  issue_body=$(_jq '.body')

  # Check if issue already exists in the current repository
  if ! gh issue list --json title | grep -q "\"$issue_title\""; then
    gh issue create -t "$issue_title" -b "$issue_body" -l "$label"
    echo "Issue '$issue_title' copied successfully to the current repository."
  else
    echo "Issue '$issue_title' already exists in the current repository. Skipped."
  fi
  done
fi

echo "Copying issues with label '$label' from $source_repo to the current repository
    completed."
```

Figura 5.15: Código escrito por el modelo de lenguaje gpt-3.5-turbo-0125

Capítulo 6

Conclusiones y líneas futuras

Es un hecho que la utilización de inteligencia artificial capaz de interactuar con los seres humanos a través de lenguaje natural es una tecnología que ha llegado para quedarse, pero mientras que tareas como realizar resúmenes, escribir correos y traducir texto son fácilmente resueltas por los modelos de lenguajes, otras como puede ser en este caso la generación de código todavía hace falta un mayor refinamiento.

Es necesario recordar que el objetivo principal de la extensión es simplificar las etapas tempranas del desarrollo de una extensión para Github CLI (gh). No se recomienda utilizar el código generado por el modelo de lenguaje sin realizar una supervisión previa acerca de la calidad y validez del código generado. Es bien sabido que los modelos de lenguaje tienden a sufrir alucinaciones y es posible que genere código innecesario o incorrecto si no es capaz de entender completamente los requerimientos.

El código generado por la inteligencia artificial debe ser utilizado como guía e inspiración para realizar el verdadero código de la extensión. La extensión `gh-ai` puede ser útil para buscar y probar distintos enfoques, nunca para generar una extensión completa y lista para ser lanzada al público.

En cuanto a la extensión `gh-ai`, pese a conseguir resultados mediocres todavía existe un gran margen de mejora en cada uno de los módulos que la conforman. El compilador de Markdown puede ser mejorado utilizando como referencia el compilador remark mencionado en capítulos anteriores. La interfaz beta de asistente de OpenAI esta en constante evolución por lo que la implementación realizada para el proyecto debe ser actualizada para no utilizar tecnología en desuso. El diseño e implementación de las plantillas así como la ingeniería de comandos ha quedado en un punto bastante primitivo, siendo el principal foco de mejora del proyecto a futuro.

Gracias a este trabajo he podido aprender gran cantidad de tecnologías nuevas. Las decisiones de diseño resultaron ser un gran reto a la hora de desarrollar el proyecto. He tenido que reconstruir el proyecto múltiples veces a lo largo del desarrollo. Puedo concluir que pese a todas las dificultades ha sido una experiencia gratificante.

Capítulo 7

Summary and Conclusions

It is a fact that the use of artificial intelligence capable of interacting with humans through natural language is a technology that is here to stay. While tasks such as summarizing, writing emails, and translating text are easily handled by language models, others, like code generation in this case, still require further refinement.

It is important to remember that the primary goal of the extension is to simplify the early stages of developing an extension for Github CLI (gh). It is not recommended to use the code generated by the language model without prior supervision regarding the quality and validity of the generated code. It is well known that language models tend to hallucinate, and it is possible that they may generate unnecessary or incorrect code if they cannot fully understand the requirements.

The code generated by artificial intelligence should be used as a guide and inspiration for writing the actual extension code. The `gh-ai` extension can be useful for exploring and testing different approaches, but never for generating a complete, ready-to-launch extension for the public.

Regarding the `gh-ai` extension, despite achieving mediocre results, there is still a significant margin for improvement in each of its modules. The Markdown compiler can be improved using the remark compiler mentioned in previous chapters. Since the beta interface of OpenAI's assistant is constantly evolving, the current implementation of the project must be updated to avoid being deprecated. The design and implementation of the templates, as well as the prompt engineering modules remain quite primitive, being the main focus for future project improvements.

Thanks to this work, I have been able to learn about many new technologies. The design decisions proved to be a great challenge during development, leading to multiple reconstructions of the project. Despite the difficulties, I must say it has been a rewarding experience.

Capítulo 8

Presupuesto

Todos los paquetes de JavaScript utilizados para la realización de este proyecto son de carácter *open source* y gratuitas, todo el hardware utilizado para la realización del proyecto es de uso personal por lo que no se incluye en el presupuesto. Se presupone un saldo hipotético de 10 euros la hora. Cabe mencionar el gasto de `tokens` utilizados por la extensión `gh-ai` durante el transcurso de la realización de este proyecto. El coste de utilizar la API de OpenAI a día 22 de mayo de 2024 puede ser consultada en el apartado 2.3 del capitulo 2.

Coste	Tipos	Descripción
1.240€	Diseño	El diseño e investigación del proyecto ha durado aproximadamente unas 124 horas.
4.960€	Desarrollo	El desarrollo de aproximadamente 4 meses con un horario medio de 4 horas, dando un total de 496 horas trabajadas.
560€	Elaboración	La elaboración de la memoria del trabajo así como de los distintos seminarios y talleres ha durado unas 56 horas aproximadamente.
2.5€	Utilización de la API de OpenAI	El coste monetario total asociado a la utilización de tokens por cada ejecución de la extensión.

Cuadro 8.1: Resumen de tipos

Esto supone un presupuesto total de, `6.762,5` euros en un periodo de 8 meses.

Apéndice A

Repositorio y Ficheros de la Extensión

A continuación se adjuntan enlaces al repositorio de GitHub donde se encuentra alojado todo el código utilizado para la creación de la extensión `gh-ai`.

1. Repositorio de gh-ai:
   `https://github.com/gh-cli-for-education/gh-ai`
2. Fichero de ejemplo gh-rate-limit: `gh-rate-limit.md`
3. Fichero de ejemplo gh-notify: `gh-notify.md`
4. Fichero de ejemplo gh-cpissues: `gh-cpissues.md`
5. Fichero de ejemplo gh-poi: `gh-poi.md`
6. Fichero con la sintaxis del lenguaje: `parser.ne`
7. Fichero con las acciones semánticas: `semantic-actions.js`
8. Fichero con el esquema del objeto de entrada: `input-schema.js`

Glosario

Interfaz de Línea de Comandos *Software* que utiliza una entrada de texto plano como medio de comunicación entre el usuario y el programa, careciendo de cualquier elemento gráfico..

Interfaz de programación de aplicaciones Conjunto de funciones, procedimientos y subrutinas que se ofrece una librería, o paquete, para ser utilizadas por otro *Software*.

Markdown Formato de texto plano cuya diseño busca ser fácil de leer y escribir, además funciona como herramienta de conversión de texto a HTML..

Modelo de Lenguaje Algoritmos matemáticos para la predicción de palabras que constan de una red neuronal entrenada con grandes cantidades de texto..

Paquete Un paquete, o librería, en un conjunto de código con una funcionalidad definida que ofrece una interfaz para ser utilizada por otros programas..

Siglas

API Application Program Interface(Interfaz de Programación de Apliaciones).

CLI Command Line Interface(Interfaz de Línea de Comandos).

GH GitHub.

NVM Node Version Manager.

Bibliografía

[1] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[2] T. Ridnik, D. Kredo, and I. Friedman, "Code generation with alphacodium: From prompt engineering to flow engineering," 2024.

[3] J. Shin, C. Tang, T. Mohati, M. Nayebi, S. Wang, and H. Hemmati, "Prompt engineering or fine tuning: An empirical assessment of large language models in automated software engineering tasks," 2023.

[4] K. Chandra and T. Radvan, "Moo: a highly-optimised tokenizer/lexer generator," octubre 2022. [Online]. Available: https://github.com/no-context/moo

[5] K. Chandra and T. Radvan, "Nearley: a parsing toolkit for JavaScript," junio 2020. [Online]. Available: https://github.com/kach/nearley/

[6] C. McDonnell, "Zod: Typescript-first schema validation," marzo 2020. [Online]. Available: https://zod.dev/

[7] C. Wanstrath, "Manual de uso mustache," 2009. [Online]. Available: https://mustache.github.io/mustache.5.html

[8] OpenAI, "Overview - OpenAI API," https://platform.openai.com/overview, (Visitado: 21 mayo,2024).

[9] OpenAI, "Pricing," 2015. [Online]. Available: https://openai.com/pricing

[10] OpenAI, "Function calling," mayo 2024. [Online]. Available: https://platform.openai.com/docs/guides/function-calling

[11] OpenAI, *diagram-assistants*, noviembre 2023. [Online]. Available: https://platform.openai.com/docs/assistants/how-it-works/objects

[12] OpenAI, *diagram-run-statuses-v2*, noviembre 2023. [Online]. Available: https://platform.openai.com/docs/assistants/how-it-works/runs-and-run-steps

[13] OpenJS Foundation, "Download | node.js," mayo 2024. [Online]. Available: https://nodejs.org/en/download

[14] OpenJS Foundation. (2024, mayo) Installing node.js via package manager | node.js. [Online]. Available: https://nodejs.org/en/download/package-manager

[15] S. Chacon and J. Long, *Manual de uso de Git*, Git Project, mayo 2024, disponible en https://git-scm.com/downloads.

[16] Howell, Max and Prévost, Rémi and McQuaid, Mike and Lalonde, Danielle, mayo 2009. [Online]. Available: https://brew.sh/

[17] Wojciakowski, Matt, "Use the winget tool to install and manage applications," abril 2023. [Online]. Available: https://learn.microsoft.com/en-us/windows/package-manager/winget/

[18] GitHub, "Manual," enero 2020. [Online]. Available: https://cli.github.com/manual/

[19] John, Gruber, "Markdown," dec 2004. [Online]. Available: https://daringfireball.net/projects/markdown/

[20] Wormer, Titus, "Remark," julio 2019. [Online]. Available: https://remark.js.org/

[21] Schmidt, Luchas and H, Kevin and Kruse, Lars, enero 2024. [Online]. Available: https://github.com/thetechcollective/gh-cpissues/tree/main