**Documentación de Funciones del Sistema de Inteligencia Artificial para SENAsoft 2025**

**Abel Moreno Cervantes**

**Brajhan Medina**

**Juan Sebastián Lizcano**

**Servicio Nacional de Aprendizaje – SENA**

**Análisis y Desarrollo de Software**

**Bogotá, Colombia**

**Agosto 2025**

**Índice**

1. **Metodología**  
    1.1 Enfoque de desarrollo  
     1.1.1 Modelo incremental – iterativo  
    1.2 Etapas del proceso  
     1.2.1 Análisis de requerimientos  
     1.2.2 Diseño  
     1.2.3 Implementación  
     1.2.4 Pruebas  
     1.2.5 Documentación  
    1.3 Herramientas utilizadas
2. **Documentación de Funciones – SENA-Mate API (Backend)**  
    2.1 Introducción  
    2.2 Resumen de funciones  
    2.3 Detalle de funciones  
     2.3.1 load\_prompt(path: str)  
     2.3.2 to\_keywords(text: str)  
     2.3.3 score\_template(pr\_kw, tpl\_title, tpl\_desc)  
     2.3.4 build\_checklist(title, description, prompt\_data)  
     2.3.5 call\_chat\_model(model, messages, timeout\_seconds)  
     2.3.6 root()  
     2.3.7 ask\_ai(request)  
     2.3.8 hf\_ask(request)  
     2.3.9 prepare\_pr(request)
3. **Documentación de Funciones – SENA-Mate (Frontend)**  
    3.1 Introducción  
    3.2 Variables clave  
    3.3 Funciones  
     3.3.1 addMessage(text, sender = "bot")  
     3.3.2 addChatItem(text)  
     3.3.3 sendMessage()  
    3.4 Eventos y almacenamiento  
     3.4.1 Envío de mensaje con Enter  
     3.4.2 Cargar historial al iniciar  
     3.4.3 Guardar historial al cerrar
4. **Conclusión**

**Metodología**

**Enfoque de desarrollo**

En este proyecto se utilizó un **enfoque incremental** basado en el modelo **Iterativo**:

* Se construyó una primera versión funcional con las características esenciales.
* Posteriormente se añadieron mejoras, como trazas de depuración y manejo de errores.
* Cada iteración fue probada antes de continuar.

**Etapas del proceso**

1. **Análisis de requerimientos**
   * Se identificaron las funcionalidades necesarias: ejecución de modelos de IA, manejo de datos de entrada y salida, y control de errores.
   * Se definieron los criterios de seguridad, como evitar la impresión de tokens completos.

**Diseño**

* + Se organizó el proyecto en módulos para backend y frontend.
  + En backend: separación de funciones y controladores.
  + En frontend: componentes modulares para la interfaz de usuario.

**Implementación**

* + Se desarrolló el código del backend (main.py) con funciones para procesar datos y comunicarse con la API.
  + Se incorporaron mensajes de depuración para seguir el flujo.
  + En frontend se diseñaron pantallas para la interacción con el usuario.

**Pruebas**

* + Pruebas unitarias para verificar funciones específicas.
  + Pruebas integradas para confirmar que backend y frontend se comunican correctamente.
  + Simulación de errores para validar el manejo de excepciones.

**Documentación**

* + Se documentaron las funciones en código y en este documento Word.
  + Se añadieron ejemplos de entrada y salida.

**Herramientas utilizadas**

* **Backend:** Python, FastAPI/Flask (según el caso), librerías para IA.
* **Frontend:** HTML, CSS, JavaScript, React.
* **Control de versiones:** Git y GitHub.
* **Pruebas:** Postman para API y Jest/React Testing Library para frontend.

Documentación de Funciones – SENA-Mate API (Backend)

1. Introducción

La SENA-Mate API es un backend desarrollado en Python utilizando FastAPI. Su objetivo principal es ofrecer una interfaz de comunicación entre un modelo de lenguaje (utilizando el router de Hugging Face con la interfaz de OpenAI) y un frontend, permitiendo procesar preguntas, generar respuestas y crear checklists para revisión de Pull Requests.

Este documento describe cada una de las funciones implementadas en el archivo main.py, detallando sus parámetros, valores de retorno y posibles errores.

| **Función** | **Descripción** | **Parámetros** | **Retorna** | **Errores/Excepciones** |
| --- | --- | --- | --- | --- |
| **load\_prompt(path: str)** | Carga el archivo prompt.json y valida su estructura mínima. Devuelve valores por defecto si no existe o hay error. | path *(str)* – Ruta al archivo del prompt. | dict con claves system y few\_shots. | Error de lectura de archivo (manejado devolviendo valores por defecto). |
| **to\_keywords(text: str)** | Convierte texto en conjunto de palabras clave, eliminando stopwords y palabras cortas. | text *(str)* – Texto de entrada. | set con las palabras clave. | Ninguno (manejado internamente). |
| **score\_template(pr\_kw, tpl\_title, tpl\_desc)** | Calcula coincidencias de palabras clave entre un PR y una plantilla. | pr\_kw *(set)* – Keywords del PR. tpl\_title *(str)* – Título de la plantilla. tpl\_desc *(str)* – Descripción de la plantilla. | int con cantidad de coincidencias. | Ninguno. |
| **build\_checklist(title, description, prompt\_data)** | Genera checklist combinando una base común con ítems de plantillas coincidentes. | title *(str)*, description *(str)*, prompt\_data *(dict)*. | dict con checklist, plantillas coincidentes y prompt usado. | Ninguno (devuelve checklist vacío si no hay coincidencias). |
| **call\_chat\_model(model, messages, timeout\_seconds)** | Llama al modelo de chat configurado (router HF vía cliente OpenAI). | model *(str)*, messages *(list[dict])*, timeout\_seconds *(int, opcional)*. | Objeto de respuesta del modelo. | Excepciones de conexión o respuesta inválida. |
| **root()** | Endpoint raíz para verificar que la API está funcionando. | Ninguno. | dict con mensaje de estado. | Ninguno. |
| **ask\_ai(request)** | Endpoint /ask que envía una pregunta al modelo y retorna la respuesta. | request *(AskRequest)* – Pregunta validada. | dict con pregunta, respuesta, prompt y modelo usado. | HTTPException (400, 502, 500). |
| **hf\_ask(request)** | Igual que /ask, pero pensado para llamadas específicas con más trazas y control de modelo. | request *(AskRequest)*. | dict con respuesta y modelo usado. | HTTPException (400, 502, 500). |
| **prepare\_pr(request)** | Endpoint /prepPR que genera un checklist de revisión de PR. | request *(PrepPRRequest)* – Título y descripción del PR. | dict con título, descripción, checklist, plantillas y prompt usado. | HTTPException (400, 500). |

**3. Detalle de Funciones**

**3.1 load\_prompt(path: str)**

* **Propósito:** Cargar el archivo prompt.json y garantizar que tenga la estructura mínima requerida.
* **Parámetros:**
  + path *(str)*: Ruta del archivo JSON a cargar.
* **Retorno:** Diccionario con claves:
  + "system" *(str)* – Texto de configuración del sistema.
  + "few\_shots" *(list)* – Lista de plantillas de ejemplo.
* **Errores manejados:** Si el archivo no existe o no se puede leer, se devuelve un diccionario con valores por defecto.
* **Ejemplo de uso:**
* prompt\_data = load\_prompt("prompts/prompt.json")

**3.2 to\_keywords(text: str)**

* **Propósito:** Extraer palabras clave de un texto eliminando stopwords y palabras de menos de 3 caracteres.
* **Parámetros:**
  + text *(str)*: Texto a procesar.
* **Retorno:** Conjunto (set) de palabras clave.
* **Ejemplo de uso:**
* keywords = to\_keywords("Este es un título de ejemplo para prueba")

**3.3 score\_template(pr\_kw, tpl\_title, tpl\_desc)**

* **Propósito:** Calcular el número de palabras clave que coinciden entre un PR y una plantilla.
* **Parámetros:**
  + pr\_kw *(set)*: Keywords del PR.
  + tpl\_title *(str)*: Título de la plantilla.
  + tpl\_desc *(str)*: Descripción de la plantilla.
* **Retorno:** Número entero con la cantidad de coincidencias.
* **Ejemplo de uso:**
* coincidencias = score\_template(keywords, "Fix bug", "Soluciona error en módulo X")

**3.4 build\_checklist(title, description, prompt\_data)**

* **Propósito:** Generar un checklist combinando ítems base con ítems adicionales de plantillas coincidentes.
* **Parámetros:**
  + title *(str)*: Título del PR.
  + description *(str)*: Descripción del PR.
  + prompt\_data *(dict)*: Datos del archivo prompt.json.
* **Retorno:** Diccionario con:
  + "checklist" *(list)* – Lista final de ítems.
  + "matched\_templates" *(list)* – Plantillas usadas.
  + "prompt\_used" *(str)* – Prompt del sistema.
  + Ejemplo de uso:
  + checklist = build\_checklist("Agregar feature X", "Se añade funcionalidad para Y", prompt\_data)

**3.5 call\_chat\_model(model, messages, timeout\_seconds)**

* **Propósito:** Llamar al modelo de chat configurado, centralizando trazas y manejo de errores.
* **Parámetros:**
  + model *(str)*: Modelo a utilizar.
  + messages *(list[dict])*: Lista de mensajes con roles (system, user).
  + timeout\_seconds *(int, opcional)*: Tiempo máximo de espera.
* **Retorno:** Objeto de respuesta del modelo.
* **Errores:** Excepciones de conexión o respuesta inválida.
* **Ejemplo de uso:**
* resp = call\_chat\_model("openai/gpt-oss-20b", messages)

**3.6 root()**

* **Propósito:** Verificar el estado de la API.
* **Parámetros:** Ninguno.
* **Retorno:** Diccionario con mensaje de confirmación.
* **Ejemplo de uso:** Acceder a / en el navegador o con GET.

**3.7 ask\_ai(request)**

* **Propósito:** Recibir una pregunta desde el frontend, enviarla al modelo y devolver la respuesta.
* **Parámetros:**
  + request *(AskRequest)*: Objeto con la pregunta.
* **Retorno:** Diccionario con la pregunta, respuesta, prompt usado y modelo.
* **Errores:** HTTP 400, 502, 500.
* **Ejemplo de uso:** POST a /ask con:
* { "question": "¿Qué es FastAPI?" }

**3.8 hf\_ask(request)**

* **Propósito:** Similar a /ask pero con más trazas y control del modelo.
* **Parámetros:** Igual que ask\_ai.
* **Retorno:** Diccionario con respuesta y modelo.
* **Errores:** HTTP 400, 502, 500.

**3.9 prepare\_pr(request)**

* **Propósito:** Generar checklist de revisión de PR usando título y descripción.
* **Parámetros:**
  + request *(PrepPRRequest)*: Datos del PR.
* **Retorno:** Diccionario con checklist y metadatos.
* **Errores:** HTTP 400, 500.
* **Ejemplo de uso:** POST a /prepPR con:
* {

"title": "Fix bug en login",

"description": "Se corrige un problema al autenticar usuarios"

}

**Documentación de Funciones – SENA-Mate (Frontend)**

**1. Introducción**

El frontend de SENA-Mate es una interfaz web en JavaScript puro, diseñada para interactuar con el backend (FastAPI).  
Permite enviar mensajes, mostrar respuestas, mantener un historial y renderizar una lista lateral (sidebar) de chats previos.

Este documento describe cada función y variable clave, incluyendo parámetros, retornos y manejo de errores.

| **Variable** | **Descripción** |
| --- | --- |
| messagesContainer | Elemento del DOM que contiene el chat principal. |
| userInput | Campo de texto donde el usuario escribe sus mensajes. |
| chatList | Contenedor de la barra lateral con historial de chats. |
| chatHistory | Array en memoria que guarda los mensajes enviados y recibidos. |

**3. Funciones**

**3.1 addMessage(text, sender = "bot")**

* **Propósito:** Agregar un mensaje (del usuario o del bot) al chat principal.
* **Parámetros:**
  + text *(string)*: Texto del mensaje.
  + sender *(string, opcional)*: Quién envía el mensaje, valores posibles "bot" o "user". Por defecto "bot".
* **Funcionamiento:**
  + Crea un <div> con clases "message" y el valor de sender.
  + Agrega una imagen de avatar.
  + Inserta el texto del mensaje.
  + Añade el mensaje al contenedor principal y hace scroll automático.
* **Retorno:** Ninguno.
* **Ejemplo:**

addMessage("Hola, ¿cómo estás?", "user");

**3.2 addChatItem(text)**

* **Propósito:** Añadir un resumen del mensaje a la lista lateral (sidebar).
* **Parámetros:**
  + text *(string)*: Texto a mostrar en la lista.
* **Funcionamiento:**
  + Verifica que chatList no sea null.
  + Crea un <div> con clase "chat-item".
  + Inserta un icono y el texto.
  + Añade el elemento a la barra lateral.
* **Retorno:** Ninguno.
* **Ejemplo:**

addChatItem("Consulta sobre IA");

**3.3 sendMessage()**

* **Propósito:** Enviar un mensaje al backend y procesar la respuesta.
* **Funcionamiento:**
  1. Lee el texto del userInput y valida que tenga al menos 5 caracteres.
  2. Muestra el mensaje del usuario en el chat.
  3. Guarda el mensaje en chatHistory y en la barra lateral.
  4. Limpia el campo de texto.
  5. Envía una solicitud POST al backend (/ask) con el contenido del mensaje.
  6. Si el backend responde correctamente, muestra la respuesta del bot y la guarda en el historial.
  7. Maneja errores de conexión o respuesta inválida.
* **Parámetros:** Ninguno (lee el valor directamente del input).
* **Retorno:** Ninguno.
* **Errores manejados:**
  1. Mensajes demasiado cortos (respuesta del bot indicando el error).
  2. Respuestas no OK del backend (response.ok falso).
  3. Errores de conexión (catch).
* **Ejemplo:**

await sendMessage();

**3.4 Evento: Enter para enviar mensaje**

* **Propósito:** Permitir enviar el mensaje presionando la tecla Enter.
* **Implementación:**

userInput.addEventListener("keypress", function(e) {

if (e.key === "Enter") sendMessage();

});

**3.5 Evento: Cargar historial al iniciar**

* **Propósito:** Recuperar mensajes previos guardados en localStorage y mostrarlos en el chat y la barra lateral.
* **Implementación:**

window.addEventListener("load", () => {

const savedHistory = JSON.parse(localStorage.getItem("chatHistory") || "[]");

savedHistory.forEach(msg => {

addMessage(msg.text, msg.sender);

addChatItem(msg.text);

chatHistory.push(msg);

});

});

**3.6 Evento: Guardar historial al cerrar**

* **Propósito:** Guardar el historial de chat en localStorage antes de cerrar o recargar la página.
* **Implementación:**

window.addEventListener("beforeunload", () => {

localStorage.setItem("chatHistory", JSON.stringify(chatHistory));

});

**Conclusión**

La documentación de las funciones tanto del backend como del frontend de la aplicación SENA-Mate permite comprender claramente el flujo de trabajo, la interacción entre los módulos y la lógica de procesamiento de datos.

Este registro técnico facilita el mantenimiento del sistema, la incorporación de nuevas funcionalidades y la detección temprana de posibles errores.  
Además, asegura que cualquier miembro del equipo de desarrollo pueda comprender la estructura y el propósito del código, optimizando la colaboración y reduciendo la curva de aprendizaje.