ESCUELA DE INGENIERÍA DE FUENLABRADA

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE LA TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

TEST PROCEDURE CREATOR: HERRAMIENTA DE GENERACIÓN DE PRUEBAS USANDO JINJA Y TEXTX

Autor: David Martínez Sanz

Tutores: Jesús María González Barahona y Raúl de la Torre Martín

Curso académico 2025/2026

Agradecimientos

A mis padres, María Isabel y Miguel Ángel, que han sido siempre mi mayor sostén. A mi madre por su paciencia inagotable y su forma de estar siempre. A mi padre, por enseñarme el valor del esfuerzo, de hacer las cosas bien y con constancia. Y a mi hermana, por su ejemplo de dedicación durante todos estos años.

A mis compañeros de carrera, con quienes he compartido conocimientos, risas, agobios, aprendizajes y momentos que van más allá de lo académico. Gracias por esas partidas de mus y esos viajes improvisados en últimos momentos que se convierten hoy en inolvidables.

Agradezco especialmente a mi tutor de TFG y a mi tutor de Airbus Helicopters, por su guía, su disponibilidad y su confianza. Gracias por orientarme sin imponer, por darme libertad y al mismo tiempo estructura, y por tomarse en serio mis ideas.

También quiero extender mi gratitud a todas personas que, de una manera u otra, han contribuido a que hoy me encuentre aquí, en este punto. Algunas ni siquiera lo sabrán, pero su influencia ha sido decisiva.

Y por último, gracias a mi pareja y a quien me acompaña de forma cercana en lo cotidiano, con una presencia que significa todo para mí.

A todos vosotros: gracias de corazón.

Resumen

Este trabajo presenta el desarrollo de una herramienta de software que automatiza la creación de pruebas para la certificación de software crítico en la industria aeronáutica. Los resultados obtenidos demuestran la viabilidad de unificar la validación de múltiples requisitos mediante la identificación de patrones de pruebas comunes bajo el principio de separar la lógica de validación de los datos concretos a evaluar. Este flujo permite optimizar el ciclo de verificación y superar las ineficiencias de métodos tradicionales basados en la creación manual de pruebas.

La solución propuesta se fundamenta en una aplicación que opera a través de diversas interfaces de edición, donde el usuario define de forma independiente los diccionarios de datos a evaluar y la lógica de validación, componentes que posteriormente son integrados en la interfaz principal por un motor de procesamiento central. Este núcleo orquesta el cruce de múltiples diccionarios de datos con una única plantilla, generando *scripts* de pruebas independientes listos para ser importados y ejecutados en entornos de validación externos. Además, el sistema incorpora un asistente basado en inteligencia artificial generativa, el cual actúa como soporte interactivo para guiar y ayudar al usuario en todo momento.

Esta integración tecnológica representa un avance significativo en la estandarización de los procedimientos de certificación de pruebas. Entre sus posibles aplicaciones, y gracias a su capacidad para adaptar el formato final de los *scripts*, la herramienta ofrece un marco de desarrollo flexible y escalable, fácilmente integrable en los procesos de validación y *testing* de cualquier organización o sector industrial.

Summary

This report presents the development of a software tool that automates the creation of tests for the certification of critical software in the aeronautical industry. The results obtained demonstrate the feasibility of unifying the validation of multiple requirements by identifying common test patterns, based on the principle of separating the validation logic from the specific data to be evaluated. This workflow enables the verification cycle to be optimised and overcomes the inefficiencies of traditional methods based on manual test creation.

The proposed solution is based on an application that operates through various editing interfaces, where the user independently defines the data dictionaries to be evaluated and the validation logic; these components are subsequently integrated into the main interface by a central processing engine. This core coordinates the cross-referencing of multiple data dictionaries with a single template, generating independent test scripts ready to be imported and executed in external validation environments. Furthermore, the system incorporates a wizard based on generative artificial intelligence, which acts as an interactive support to guide and assist the user at all times.

This technological integration represents a significant step forward in the standardisation of test certification procedures. Among its potential applications, and thanks to its ability to adapt the final format of scripts, the tool offers a flexible and scalable development framework that can be easily integrated into the validation and testing processes of any organisation or industrial sector.

Índice general

Índice de figuras

Capítulo 1

Introducción

En la industria del software, especialmente en el desarrollo de sistemas críticos, la certificación es un proceso crítico que garantiza el cumplimiento de estrictos estándares de seguridad. Este proceso se rige por exigentes criterios de calidad industrial, los cuales exigen procesos de verificación exhaustivos y una trazabilidad entre requisitos, código y pruebas para asegurar la máxima fiabilidad funcional. Para asegurar que los sistemas de alta complejidad funcionen correctamente, se realizan test exhaustivos, comprobando que los sistemas cumplan con los requisitos previamente especificados. Tradicionalmente, la generación de estos procedimientos de prueba se ha llevado a cabo de manera manual en las propias herramientas de validación.

Esta metodología clásica, en la mayor parte de los casos obliga a los ingenieros a construir cada prueba paso a paso, seleccionando instrucciones y parámetros uno a uno. Este enfoque presenta dos problemas fundamentales, por un lado, un alto consumo de tiempo y por otro, la dificultad para reutilizar las baterías de pruebas en futuros escenarios. Dado que una gran parte de estos procedimientos a menudo siguen patrones lógicos y secuencias repetitivas, la automatización moderna aboga por un principio de parametrización, es decir, separar la lógica que se desea probar de los datos concretos a evaluar.

Es en este escenario donde surge el presente Trabajo Fin de Grado[1], fruto de la experiencia técnica desarrollada dentro del departamento de software de Airbus Helicopters. Para intentar agilizar sus procesos de certificación, el departamento contaba con una pequeña aplicación de apoyo basada en este principio de separación. Por un lado, el sistema permitía crear plantillas parametrizadas, donde cada una encapsulaba la lógica de un test; por otro lado, se empleaban ficheros de

[1] URL del Trabajo Fin de Grado: https://daviidmartnez03.github.io/tfg-site/

parámetros, que actuaban como diccionarios de datos con la información específica de cada escenario de prueba (como nombres de equipos, señales o valores esperados).

El flujo de trabajo de la herramienta consistía en combinar ambos elementos, inyectando automáticamente los datos del fichero de parámetros en la plantilla correspondiente. Este enfoque resultaba extremadamente eficiente, ya que permitía que una única plantilla lógica pudiera reutilizarse y cruzarse con multitud de ficheros de parámetros diferentes. De este modo, se facilitaba la ampliación y reutilización de baterías de pruebas existentes. Como resultado de este procesamiento, la aplicación generaba unos archivos de texto (*scripts* finales), que los ingenieros debían guardar para su posterior importación en la plataforma de validación oficial, donde finalmente se ejecutaba el test.

Sin embargo, tras un periodo de integración en la compañía enfocado en el análisis de estas metodologías, y en colaboración con el equipo de ingenieros expertos en pruebas, se concluyó que, si bien el concepto era bueno, la aplicación era muy limitada, ya que carecía de un entorno fácil de usar y seguía obligando a los ingenieros a lidiar con formatos de código muy complejos y poco intuitivos.

Al ver el enorme potencial de mejora, se determinó construir una nueva generación de esta herramienta con la finalidad de transformar la forma en la que el ingeniero de validación trabajaba, siendo el objetivo proporcionar al usuario un flujo de trabajo basado en plantillas que abstrajeran la complejidad técnica del proceso de creación de pruebas. Además, para mitigar la curva de aprendizaje, reducir tiempos y facilitar la adopción de la nueva generación de la aplicación, se instruyó un asistente inteligente basado en inteligencia artificial generativa que proporciona al usuario soporte interactivo en tiempo real. En conjunto, se garantiza una reducción drástica en los tiempos de generación de las baterías de pruebas y se minimiza la tasa de error humano.

Finalmente, cabe destacar que, debido a las restricciones de seguridad industrial y confidencialidad, a lo largo de este trabajo se utilizarán datos y escenarios de prueba sintéticos. No obstante, estos han sido diseñados para simular y representar con absoluta fidelidad la complejidad y la casuística operativa del entorno real.

1.1. Objetivos

El objetivo principal de este Trabajo Fin de Grado es **dar respuesta a las limitaciones operativas detectadas en la herramienta base preexistente**, con el propósito de optimizar el proceso de creación y automatización de pruebas. Para ello, el sistema provee al usuario de capacidades que, en conjunto, abstraen la complejidad técnica subyacente y aceleran los ciclos de certificación de software.

Los **objetivos instrumentales** derivados son:

- **Evolucionar la aplicación inicial proporcionada:** Aprovechar la arquitectura y el conocimiento de la aplicación inicial, que ya operaba correctamente en Airbus Helicopters, para construir un sistema automatizado mucho más robusto.
- **Optimizar la experiencia de usuario:** Dotar de botones interactivos, atajos y menús que asistan al ingeniero de validación y agilicen todo el proceso de generación de plantillas.
- **Diseñar e implementar una sintaxis humanizada:** Crear e implementar un Lenguaje de Dominio Específico (DSL), que permita a los usuarios abstraerse de los formatos técnicos y rígidos requeridos por las herramientas de validación finales y permita la generación de plantilla de forma más intuitiva y legible.
- **Proporcionar soporte interactivo y reducir la curva de aprendizaje:** Proporcionar un asistente inteligente que actúe como complemento a la herramienta local, con el propósito de que resuelva dudas y guíe al usuario en la creación de pruebas, reduciendo drásticamente la curva de aprendizaje del nuevo sistema.

Como **condicionantes metodológicos** y **requisitos operativos** se plantean:

- **Garantizar la retrocompatibilidad de baterías de pruebas preexistentes:** Asegurar que la nueva herramienta permita interpretar y procesar sin conflictos baterías de plantillas y ficheros de parámetros heredados.

- **Asegurar la operatividad en ecosistemas Windows:** Diseñar la nueva arquitectura de la herramienta garantizando su ejecución en sistemas operativos Windows. Esta decisión de diseño responde a la estandarización de la infraestructura en Airbus Helicopters.
- **Salvaguardar la confidencialidad:** Proteger por completo la propiedad intelectual y los datos reales de Airbus Helicopters frente al exterior.
- **Validar la funcionalidad del sistema:** Demostrar que la nueva aplicación es plenamente funcional, segura y capaz de automatizar entornos de trabajo complejos.

1.2. Contribuciones

Las contribuciones principales de este trabajo se resumen en:

- **Desarrollo de un entorno de trabajo optimizado:** Se entrega una nueva generación de la herramienta, evolucionando la herramienta original mediante el uso de librerías como *CustomTkinter*, *Pygments* y *textX*, hacia una solución que mejora, facilita y optimiza todo el proceso de diseño y generación de pruebas, resolviendo las limitaciones encontradas en la herramienta original.
- **Implementación de un motor de traducción basado en un DSL:** Este módulo permite traducir un formato mucho más legible e intuitivo al lenguaje final exigido por las herramientas de validación, gracias a la creación de una gramática propia construida sobre el metalenguaje *textX*, orientada a la abstracción técnica de la sintaxis requerida.
- **Diseño de una arquitectura de procesamiento de doble flujo:** Se ha reestructurado el sistema para soportar dos vías de flujo distintas. Por un lado, se mantiene y optimiza el flujo de trabajo original basado en el motor Jinja2, para garantizar la compatibilidad con proyectos existentes y, por otro lado, se añade un nuevo flujo que combina el motor Jinja2 con la traducción del Lenguaje de Dominio Específico, dotando a la herramienta de una gran flexibilidad.

- **Creación de un asistente mediante Inteligencia Artificial:** Se ha incorporado un asistente inteligente basado en el modelo Gemini, alojado en la nube corporativa de forma que opera de forma independiente a la aplicación principal. Actúa como consultor para el usuario reduciendo la curva de aprendizaje, facilitando la redacción de pruebas y resolviendo dudas en tiempo real.
- **Optimización del ciclo de validación de software:** Se ha resuelto la problemática inicial asociada a ineficiencias encontradas en el proceso de creación de pruebas en las herramientas de certificación. La nueva aplicación no solo logra una drástica reducción en los tiempos de las campañas de certificación, sino que también facilita la generación de baterías de pruebas reutilizables y permite generar test mucho más exhaustivos.

1.3. Estructura del documento

Para comprender plenamente el diseño e implementación del sistema, es necesario conocer las tecnologías y conceptos en los que se apoya. Por ello, el documento se organiza de la siguiente manera:

- **Capítulo 2: Tecnologías utilizadas y trabajos relacionados.** Presenta una revisión de las tecnologías clave utilizadas en este trabajo (Python, Tkinter, Jinja2, textX, etc.), justificando su elección frente a otras alternativas. Además, se analiza el estado actual de las herramientas de *testing* automatizado y el uso de DSLs en la industria.
- **Capítulo 3: Descripción del resultado.** Presenta el resultado funcional del proyecto, presentando la arquitectura de la solución, la interfaz gráfica del usuario, los editores principales (parámetros, Jinja2, Jinja2 + DSL) y los flujos de ejecución disponibles, incluyendo la integración del asistente basado en Gemini.
- **Capítulo 4: Desarrollo del proyecto.** Documenta detalladamente el proceso de construcción y evolución del software organizado en fases. Abarca desde el análisis inicial y la interfaz base de la que se partía, hasta la integración del motor de renderizado sumado al desarrollo y procesamiento DSL y la configuración del asistente inteligente.

- **Capítulo 5: Pruebas y validación.** Analiza el rendimiento del motor de generación, evalúa la precisión del soporte proporcionado por el asistente inteligente y valida la reducción de tiempo obtenida mediante casos de uso con parámetros sintéticos, exponiendo también las limitaciones detectadas.
- **Capítulo 6: Conclusiones.** Resume los principales resultados y logros obtenidos, las contribuciones realizadas y plantea las líneas de trabajo futuro para la evolución de la herramienta.

Capítulo 2

Tecnologías utilizadas y trabajos relacionados

En este capítulo se presenta el conjunto de tecnologías que se han utilizado durante el desarrollo del sistema, así como trabajos previos que han servido de referencia o inspiración. El objetivo es justificar las decisiones técnicas tomadas durante el proyecto.

Durante el desarrollo, se han utilizado asistentes de inteligencia artificial como Gemini para resolver dudas técnicas puntuales, depurar fragmentos de código o explorar soluciones alternativas. No obstante, el diseño, la arquitectura y la implementación completa del sistema han sido realizados de forma independiente.

La aplicación, combina interfaces de usuario interactivas, un motor de renderizado de plantillas y un procesamiento léxico-sintáctico mediante un Lenguaje de Dominio Específico (DSL). Toda esta infraestructura se encuentra apoyada de forma independiente por un asistente inteligente basado en Inteligencia Artificial Generativa.

El uso de Python como lenguaje base ha sido el pilar del proyecto. Su elección frente a otros lenguajes compilados, se debe a su potencia para el procesamiento de cadenas de texto y a la manipulación de archivos, procesos que constituyen el núcleo de la generación de procedimientos de prueba.

Las tecnologías implementadas mediante Python, según sus áreas funcionales se pueden agrupar en:

- **Desarrollo de la Interfaz Gráfica (*Frontend*):** Debido a la confidencialidad de los datos, se optó por una arquitectura local (aplicación de escritorio), empleando para su construcción las siguientes librerías:
    - **Tkinter:** Utilizada como base para la creación de la interfaz y gestión de eventos de la aplicación.
    - **CustomTkinter:** Empleada para integrar componentes más modernos, como pueden ser botones, menús y ventanas de los editores, con una estética más actualizada.
    - **tksheet:** Proporciona una interfaz visual de cuadrícula, estilo Excel, que es esencial para la creación de un editor de diccionarios de datos, ya que facilita la definición de estos.
    - **Pygments:** Facilita la legibilidad de código, puesto que funciona como resaltador de sintaxis dentro de los editores de texto.
- **Lógica de Procesamiento (*Backend*):** Encargada del procesamiento y renderizado de los datos del sistema, apoyándose en:
    - **Jinja2:** Utilizado como motor de plantillas encargado de evaluar lógicas e inyectar dinámicamente los diccionarios de datos de los ficheros de parámetros en las plantillas parametrizadas.
    - **Lenguaje de Dominio Específico (DSL):** Sintaxis humanizada propia que abstrae la complejidad de los formatos de validación finales, permitiendo al ingeniero definir la lógica a probar con un lenguaje más natural, intuitivo y legible.
    - **textX:** Empleado para definir el Lenguaje de Dominio Específico (DSL) y permitir la traducción semántica al lenguaje final requerido por la herramienta de certificación.

- **xml.etree.ElementTree:** Implementada para el análisis de ficheros XML, permitiendo la búsqueda y resolución automática de rutas de datos definidos en las plantillas.

De soporte externo a la herramienta desarrollada en Python, el entorno de trabajo se completa con:

- **Gem de Gemini:** Modelo de I.A. Generativa alojado en la nube corporativa, que proporciona asistencia al ingeniero en tiempo real y durante todo el proceso de redacción de pruebas.

A continuación, en la siguiente sección se profundizará en cada uno de los aspectos mencionados anteriormente.

2.1. Tecnologías utilizadas

2.1.1. Tkinter

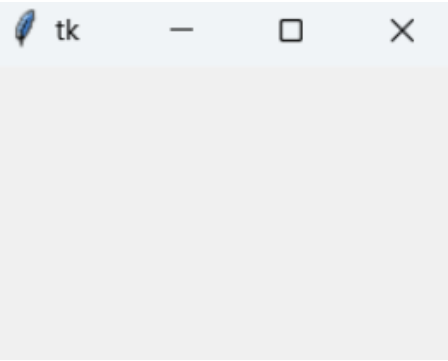


Figura 2.1: Ventana básica de Tkinter

Tkinter [1] es la biblioteca estándar utilizada para el desarrollo de Interfaces Gráficas de Usuario, incluida de forma nativa en Python. Su arquitectura se basa en la combinación de Python con el conjunto de utilidades Tcl/Tk, que posibilitan la creación de ventanas, cuadros de texto y una amplia variedad de controles interactivos denominados *widgets*.

Destaca por su ligera computación, su amplia documentación y su estabilidad, utilizándose en el desarrollo de herramientas internas y *scripts* ejecutables que necesitan operar en entornos locales seguros. Estas características, hicieron de ella la opción más correcta para esta herramienta.
Para ilustrar su funcionamiento, a continuación, se presenta el fragmento de código requerido para inicializar una ventana en Tkinter:

```
import tkinter as tk

root = tk.Tk()
# Widgets are added here
root.mainloop()
```

En el marco de este proyecto, Tkinter se ha empleado para construir la interfaz de usuario sobre la que se sustenta la aplicación. Su puesta en marcha ha posibilitado el establecimiento y gestión de eventos principal manteniendo la herramienta activa y a la escucha de las interacciones del ingeniero de validación.

También ha permitido controlar el enrutamiento de ventanas y la maquetación general de la interfaz gráfica, garantizando que el sistema se ejecute de forma robusta, local y cumpla las normativas de confidencialidad y ciberseguridad exigidas en Airbus Helicopters.

2.1.2. CustomTkinter

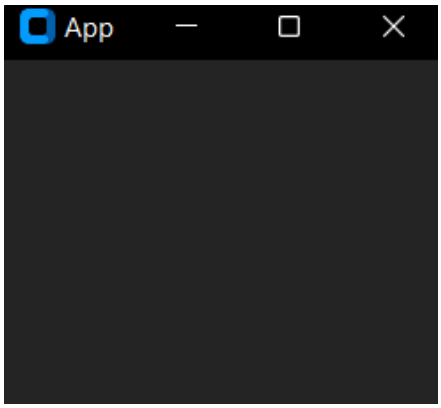


Figura 2.2: Ventana básica con CustomTkinter

El diseño visual que ofrece Tkinter presenta limitaciones en términos de modernidad estética, ello hace necesario el uso de CustomTkinter [2], que se sitúa en una capa superior sobre Tkinter y tiene la capacidad de modernizar y mejorar la estética de las aplicaciones de escritorio. Su principal ventaja es que ofrece un diseño más moderno, fluido y parecido al de las aplicaciones móviles o plataformas web actuales, pero manteniendo la lógica de programación de Tkinter. Esto se debe a que ofrece una amplia variedad de *widgets* adaptables con un diseño contemporáneo, incluyendo modos claro y oscuro, bordes redondeados y escalado automático según la resolución de pantalla.

Concretamente en este Trabajo Fin de Grado, CustomTkinter ha permitido transformar un diseño inicialmente elemental en un entorno optimizado y visualmente atractivo, elaborando entre otros, editores que disponen de menús, botones interactivos y ventanas de configuración. Esta actualización proporciona al ingeniero de pruebas una interfaz clara e intuitiva, disminuyendo la fatiga visual y atenuando la curva de aprendizaje de la nueva herramienta.

2.1.3. tksheet

La librería tksheet [3] está diseñada para integrarse dentro de interfaces gráficas creadas con Tkinter. Ofrece un *widget* de cuadrícula, que tiene la capacidad de manejar y visualizar datos tabulares de manera interactiva. Además, ofrece funcionalidades preintegradas, tales como la selección de múltiples celdas, el ajuste dinámico de filas y columnas, atajos de teclado (copiar, pegar, deshacer) y la capacidad de aplicar estilos a celdas.

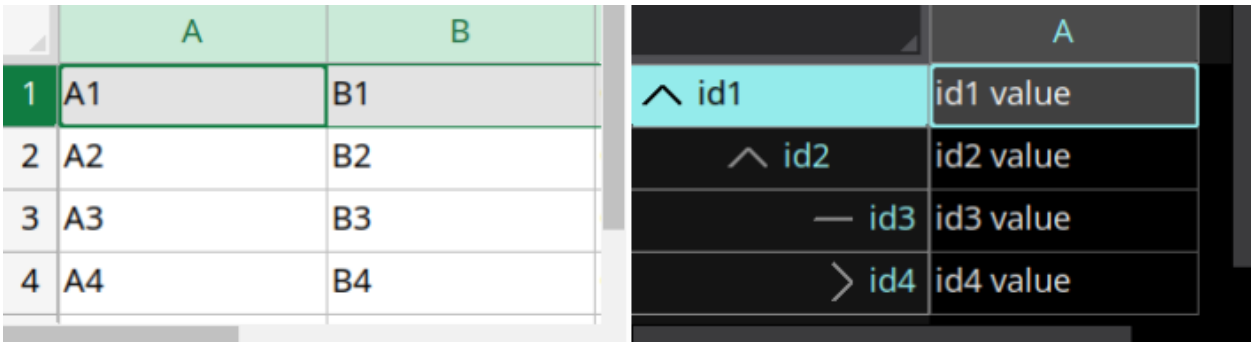


Figura 2.3: Aspecto de Editor de datos tabulares

Esta biblioteca ha hecho posible el diseño de un Editor de Parámetros en la arquitectura de la aplicación desarrollada, proporcionando al usuario un entorno tabular interactivo e intuitivo, en el que poder definir los diccionarios de datos (ficheros de parámetros) de cada escenario de prueba. Además, ha permitido implementar características como la codificación por colores, que facilitan enormemente al ingeniero la definición de los datos.

2.1.4. Pygments

Pygments [4] es un analizador léxico considerado un estándar en la industria del software para el resaltado de sintaxis en editores de código y generadores de documentación. Su motor funciona tomando bloques de texto y procesándolos mediante analizadores que identifican la estructura del lenguaje y le aplican el estilo visual correspondiente mediante formateadores. De fábrica soporta el análisis de más de 500 lenguajes de programación.

Su elección frente a otros analizadores es debida a su flexibilidad y facilidad de implementación. En el marco de este proyecto, su integración ha supuesto una amplia mejora en la experiencia de edición del usuario. Se ha implementado en los editores de texto, dedicados a la creación de plantillas, dotando a las cajas de texto (*textbox*) de capacidades propias de un Entorno de Desarrollo Integrado (IDE). Su finalidad en la aplicación es escanear en tiempo real el texto introducido por el ingeniero y aplicar el resaltado de colores que diferencia las etiquetas de código, las variables inyectadas, el texto plano y las instrucciones del DSL.

2.1.5. Jinja2

Es un motor de plantillas conocido en el desarrollo web y utilizado para la creación de documentos HTML, XML, CSV o cualquier otro formato basado en texto, incluyendo archivos de código.

Su funcionamiento gira en torno a un objeto principal denominado Entorno de plantillas (template Environment). En concreto, las instancias de esta clase se emplean para almacenar la configuración del motor. Además, se utilizan para cargar las plantillas desde el sistema de archivos.

Sus principales cualidades se describen a continuación:

- Recurre a una sintaxis similar a Python, lo que favorece su adopción.
- Presenta un trabajo de renderización rápido y eficiente.
- Es compatible con la herencia de plantillas y la creación de filtros y funciones personalizadas, lo que da lugar a estructuras de diseño reutilizables.
- Incluye elementos de control de flujo, como bucles y condiciones.

En el contexto de este Trabajo Fin de Grado, Jinja2 [5] actúa como el motor principal de la capa de procesamiento, permitiendo que una única plantilla (que contiene la lógica de pruebas a realizar) pueda cruzarse de forma iterativa con diferentes ficheros de parámetros (diccionarios de datos), resolviendo secuencialmente los bloques de código, inyectando las variables dinámicamente y finalmente generando de forma automatizada *scripts* de pruebas, garantizando la reutilización del código en el departamento. A continuación, se expone un ejemplo básico para visualizar el funcionamiento de la sintaxis y del motor Jinja2.

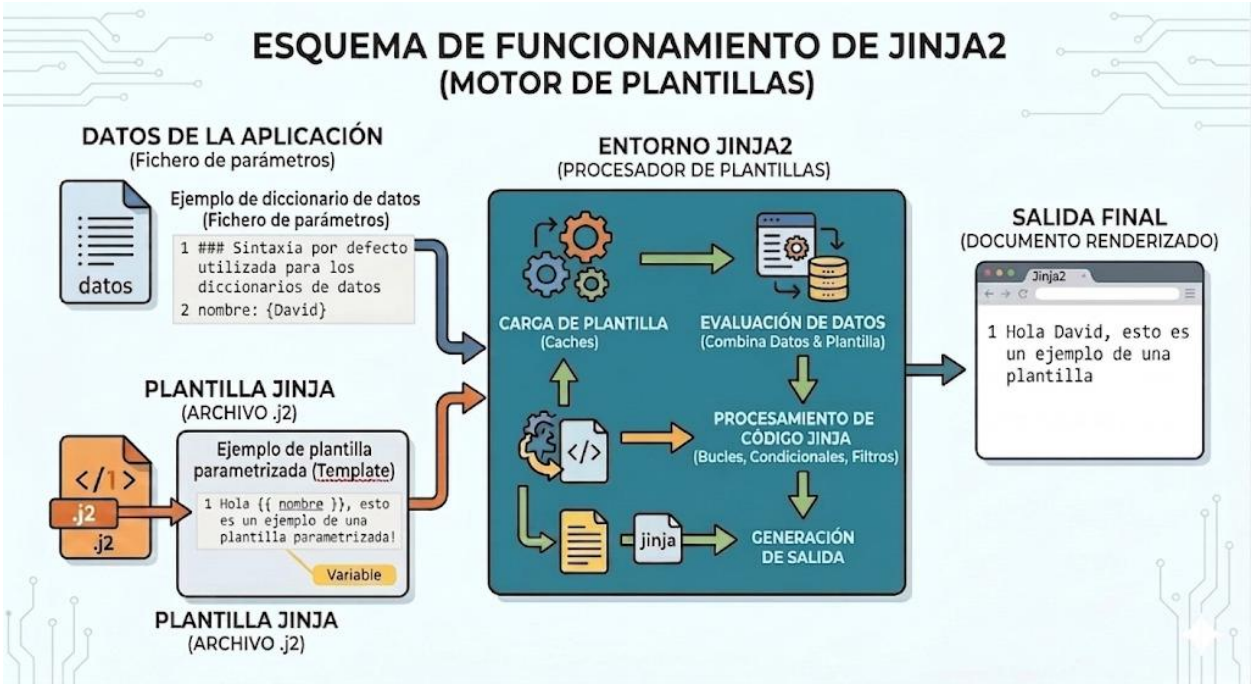


Figura 2.4: Esquema simplificado del motor de plantillas Jinja2

Ejemplo de plantilla parametrizada (Template)

```
1 Hola {{ nombre }}, esto es un ejemplo de una plantilla parametrizada!
```

Ejemplo de diccionario de datos (Fichero de parámetros)

```
1 ### Sintaxis por defecto utilizada para los diccionarios de datos
2 nombre:{David}
```

Resultado de renderización

```
1 Hola David, esto es un ejemplo de una plantilla parametrizada!
```

2.1.6. Lenguaje de Dominio Específico (DSL)

Un Lenguaje de Dominio Específico [6] es un tipo de lenguaje que se utiliza para resolver problemas específicos de programación o de especificación. A diferencia de los lenguajes de propósito general (como Python, C++ o Java), que están definidos para abordar cualquier tipo de problema computacional, un Lenguaje de Dominio Específico renuncia a la versatilidad a cambio de ofrecer una sintaxis más legible y humanizada que posibilita solucionar y optimizar problemas específicos.

Dependiendo de su definición se pueden clasificar en dos categorías:

- **DSL Internos**: Se construyen sobre un lenguaje ya existente, utilizando la sintaxis y el compilador del lenguaje base, pero restringiendo el vocabulario únicamente a las funciones necesarias. Su principal ventaja es que no requieren del desarrollo de un analizador sintáctico o de un compilador propio, pero están limitados a las reglas gramaticales del lenguaje existente escogido.
- **DSL Externos**: Son lenguajes creados desde cero, definiendo su propia gramática y sintaxis personalizada, ofreciendo una libertad de diseño absoluta. Su implementación, sin embargo, requiere del desarrollo de todo el flujo de procesamiento: un analizador léxico (*lexer*), un analizador sintáctico (*parser*) y, finalmente, un compilador de código.

Para las necesidades del proyecto, el diseño de un DSL externo ha sido una de las soluciones arquitectónicas centrales. Las herramientas de certificación de pruebas de Airbus Helicopters requieren un lenguaje rígido, compuesto por estructuras sintácticas complejas que dificultan el entendimiento humano, por ello la incorporación de una gramática propia adaptada al área de *testing* ha logrado que el ingeniero pueda redactar la lógica a probar utilizando una sintaxis humanizada y más intuitiva. De este modo, se abstrae la complejidad del lenguaje y permite al ingeniero de validación enfocarse en la definición de la prueba, delegando en el sistema la tarea de transformar la sintaxis humanizada en el lenguaje requerido por la herramienta de certificación final.

2.1.7. textX

El metalenguaje y herramienta textX [7] fue diseñado para la implementación de Lenguajes de Dominio Específico (DSL). Se fundamenta en los principios de Ingeniería Dirigida por Modelos (*Model-Driven Engineering*) y a diferencia de otros generadores (como LEX o ANTLR), que requieren una fase de compilación separada para traducir la gramática a código fuente intermedio, textX unifica la definición de reglas sintácticas y la construcción del modelo en tiempo de ejecución de Python. Se fundamenta en dos conceptos clave:

- El **Metamodelo (La Gramática)**: Define las reglas sintácticas del lenguaje a crear, estableciendo que palabras clave existen, cómo se estructuran las líneas y que jerarquía se debe seguir.
- El **Modelo (La Instancia)**: Constituye la representación en memoria de un archivo de texto concreto escrito según los estándares de dicho metamodelo, es decir, cuando el analizador lee un documento de pruebas redactado por el usuario, valida que su contenido cumple con las reglas del metamodelo y lo carga en el sistema como una estructura de datos.

Se decidió escoger esta librería frente a otras por su capacidad para realizar un mapeo directo y automatizado entre las reglas de la gramática y las clases de Python. Cuando la librería procesa un archivo de pruebas se llevan a cabo las siguientes etapas:

- **1º. Generación de un Grafo de Objetos (Object Graph)**: textX, a diferencia de otros analizadores, genera un Árbol de Sintaxis Abstracta (AST) e instancia objetos nativos de Python dinámicamente. A nivel práctico, esto significa que el texto se analiza línea por línea y se convierte automáticamente en clases y atributos en memoria, de forma que permite al desarrollador interactuar con los datos de forma directa (por ejemplo, accediendo a un atributo mediante instruction.message_name), simplificando el proceso de traducción al lenguaje destino.
- **2º. Validación semántica**: La biblioteca no solo comprueba que la sintaxis esté bien escrita, sino que permite la implementación de funciones de verificación propias, evaluando si el contexto es el correcto; por ejemplo, comprobando que un parámetro exista antes de proceder a traducirlo.

En el sistema desarrollado, textX actúa como el compilador de las pruebas diseñadas por el ingeniero de validación escritas utilizando el DSL humanizado invocando el metamodelo programado en textX. La herramienta procesa el texto en tiempo real para verificar que se cumplan las reglas del lenguaje, y en caso de fallos, se capturan las excepciones y se indican las líneas y columnas del fallo para dar asistencia al usuario. Por el contrario, si el código es correcto, textX construye el grafo de objetos correspondiente, procesándolo posteriormente para traducir las abstracciones del DSL en el lenguaje final rígido que demandan las herramientas de certificación de Airbus Helicopters, logrando una traducción determinista, segura y automática.

2.1.8. xml.etree.ElementTree

El módulo xml.etree.ElementTree [8] es una librería diseñada para el procesamiento de datos en formato XML, permite cargar documentos XML y representarlos en memoria con una estructura jerárquica de árbol, facilitando la búsqueda de nodos o atributos.

Su elección frente a otras librerías ha sido debida a su bajo consumo de memoria y su rapidez de ejecución, además de estar integrada de forma nativa en Python.

Dentro de este proyecto, se ha implementado como un procesador (*parser*) encargado de la resolución dinámica de rutas. En las lógicas de prueba, las señales y mensajes poseen rutas (por ejemplo, *equipment.module.message.signal*) que tradicionalmente se introducían a mano. Con el procesador se ha conseguido un escaneo automático de ficheros XML, de forma que se extraen y se completan automáticamente las rutas asociadas a cada mensaje o señal. Esto aporta a la herramienta una gran robustez y minimiza el tiempo de generación de lógicas de prueba a la par que minimiza la tasa de error humano por la introducción manual de datos.

2.1.9. Asistente Gem de Gemini

El Asistente Gem de Gemini [9] pertenece a la familia de la Inteligencia Artificial Generativa y los Modelos de Lenguaje de Gran Escala (LLM, *Large Language Models*). Estos sistemas se basan en las redes neuronales de tipo Transformer utilizando mecanismos de autoatención. Gracias a esta tecnología, el modelo no procesa las palabras de forma aislada, sino que es capaz de analizar fragmentos enteros de texto de forma simultánea, permitiendo captar el contexto global y el significado real de las frases con una enorme precisión.

Un "Gem" de Google consiste en un modelo cuyo comportamiento, tono y conocimiento se delimitan mediante instrucciones del sistema y la inyección de un contexto operativo cerrado. Gracias a capacidades como el aprendizaje en contexto, el modelo puede transicionar de ser un modelo de carácter generalista a operar como un sistema experto orientado a un nicho técnico concreto.

En cuanto a la gestión de su base de conocimientos, a diferencia de un modelo estándar, el Gem almacena un contexto persistente que permanece en su memoria. De forma que, si la herramienta de pruebas evoluciona actualizando las instrucciones del sistema, estas actualizaciones se propagan automáticamente a todas las sesiones de los ingenieros, de forma que siempre se trabaja con la última versión del asistente sin necesidad de reentrenar el modelo.

En este Trabajo Fin de Grado, el asistente ha sido instruido con la gramática del DSL desarrollado y con un ecosistema de datos y escenarios de prueba sintéticos.

El asistente tiene como propósito reducir la curva de aprendizaje de los ingenieros frente al nuevo sistema, dando asistencia en tiempo real a través de:

- **La resolución de dudas y ayuda interactiva**: Proporciona asistencia para generar código aclarando el uso de instrucciones, variables o reglas sintácticas del DSL.
- **La generación automatizada de plantillas**: Tiene la capacidad de generar plantillas con la lógica de las pruebas, basándose en la descripción proporcionada por el usuario.
- **La validación de código**: Detecta errores lógicos o de formato antes del procesamiento.

Debido a las exigentes políticas de ciberseguridad de Airbus Helicopters, el asistente se ha diseñado para operar de manera paralela a la aplicación, estando alojado en la nube corporativa, de forma que se garantiza que la herramienta ofrezca un soporte interactivo de alto nivel sin que ningún fragmento de información real sea expuesto a servidores externos, logrando un entorno de desarrollo eficiente y seguro.

2.2. Trabajos relacionados

En esta sección se revisan investigaciones previas relacionadas con los aspectos clave del proyecto realizado como Trabajo Fin de Grado, incluyendo referencias clave que ilustran el contexto y los enfoques más cercanos al sistema propuesto.

2.2.1. Parametrización de plantillas en arquitectura web

En el contexto del desarrollo web, la elección de motores de plantillas eficientes juega un papel fundamental. En [10] se analiza la adopción de plantillas Jinja2 en proyectos de Django, respondiendo directamente a las limitaciones de diseño del motor nativo del *framework*. El estudio destaca cómo la falta de evolución en las plantillas de Django ha impulsado a los usuarios a buscar alternativas que ofrezcan mayor flexibilidad y rendimiento.

El sistema propuesto en este trabajo traslada este paradigma de flexibilidad y reutilización al ámbito de certificación de *software*. A diferencia de los métodos de validación tradicionales, la herramienta desarrollada aplica el principio de separar los diccionarios de datos de la lógica de validación. Esta separación, impulsada por Jinja2, permite cruzar la lógica de validación con múltiples diccionarios de datos, posibilitando la generación masiva de pruebas sin necesidad de reescribir código.

2.2.2. Abstracción del lenguaje mediante DSLs

El uso de Lenguajes de Dominio Específico (DSL) para la abstracción de pruebas complejas, cuenta con un amplio recorrido en la industria tecnológica. En [11] se explora este paradigma mediante el uso de **textX**, un meta-lenguaje diseñado para facilitar la creación de DSLs intuitivos. En su investigación, los autores presentan *Pyflies*, un lenguaje creado específicamente para el diseño de experimentos cognitivos, demostrando cómo la sintaxis legible de los DSL permite generar automáticamente experimentos ejecutables, incrementando la productividad y la reproducibilidad.

El trabajo aquí desarrollado adopta este mismo enfoque de abstracción aplicándolo al contexto de generación de pruebas de software para sistemas aeronáuticos. Al igual que en el estudio referenciado, la herramienta se apoya en textX para implementar una gramática propia que ofrece al ingeniero una “sintaxis humanizada”. Esto permite a los usuarios diseñar lógicas de validación de una forma mucho más natural e intuitiva, delegando en el sistema la tarea de traducción, minimizando así la carga técnica y los errores humanos.

2.2.3. Inteligencia Artificial aplicada al testing de software

En los últimos años, se ha podido documentar y demostrar las ventajas de integrar modelos de lenguaje (LLMs) genéricos como asistentes en tareas de programación habituales. En [12] se aborda la intersección entre la inteligencia artificial, el aprendizaje automático y el aseguramiento de la calidad (*software testing*). A través de la recopilación de perspectivas de expertos en la industria, el artículo documenta el estado actual y las proyecciones futuras de la IA en este campo.

Los autores exponen lecciones aprendidas y estrategias de cómo utilizar la IA para probar software, cómo validar los propios sistemas de IA y los avances hacia los sistemas de auto prueba, presentando la posibilidad de un cambio de paradigma hacia la automatización inteligente en la industria.

El proyecto desarrollado adopta esta metodología mediante la integración de un asistente basado en IA generativa. Actuando como un soporte interactivo, el modelo ayuda a resolver dudas de sintaxis y a estructurar el código, reduciendo el periodo de adaptación y optimizando el proceso de certificación.

2.2.4. Automatización de validación en sistemas de software crítico

Debido al incremento continuo en la complejidad del software en dominios industriales como la aviación, el desarrollo de pruebas para alcanzar los estándares de certificación (como la normativa DO-178C) se ha convertido en un desafío técnico. En [13] se señala que las herramientas de pruebas existentes a menudo presentan limitaciones: o bien no logran generar un conjunto completo de pruebas que satisfaga las normativas, o bien requieren considerable intervención humana durante el proceso. Para abordar este problema, el artículo presenta un enfoque orientado a soluciones automatizadas basadas en requisitos, buscando eliminar cuellos de botella generados por las metodologías manuales.

La herramienta desarrollada en este Trabajo de Fin de Grado se alinea perfectamente con esta necesidad industrial. Al sustituir el método tradicional de creación manual de pruebas por una arquitectura de separación de componentes, el sistema elimina la intervención humana repetitiva. De esta forma, el flujo de trabajo acelera la obtención de pruebas a la vez que garantiza el nivel de rigor, estandarización y trazabilidad que exigen normativas aeroespaciales, mitigando los riesgos asociados a la intervención humana.

Capítulo 3

Descripción del resultado

Este capítulo presenta el resultado del sistema desarrollado, partiendo de la aplicación proporcionada inicialmente por Airbus Helicopters, hasta alcanzar el diseño técnico y funcional del sistema final desarrollado en este Trabajo Fin de Grado.

3.1. Descripción de la herramienta base

En el ámbito de la ingeniería de calidad, la automatización dirigida por capas de abstracción es la vía estándar para eliminar la rigidez de las herramientas de validación comerciales. Sin embargo, estas soluciones generales deben adaptarse a las realidades tecnológicas de cada organización. En el contexto específico de este proyecto en Airbus Helicopters, el estado de la herramienta inicial y los condicionantes de partida se analizan a continuación.

Al inicio del proyecto, la arquitectura de software proporcionada por Airbus Helicopters consistía en un sistema con un desarrollo funcional elemental, siendo su principal objetivo proporcionar al ingeniero una herramienta con las funciones básicas para el procesamiento de plantillas Jinja2. Si bien la aplicación funcionaba correctamente, la creación de plantillas parametrizadas carecía de una capa de abstracción orientada al usuario.

El ingeniero de validación se enfrentaba a un flujo de trabajo muy manual, ya que toda la interacción se realizaba a través de editores simples que no contaban con interactividad y asistencia. Esta metodología aumentaba la curva de aprendizaje, obligando al usuario a depender exclusivamente de su conocimiento técnico para la generación de pruebas, lo que ralentizaba el proceso de diseño y, por consiguiente, la certificación de pruebas.

Las principales limitaciones del sistema inicial se podían resumir en los siguientes puntos:

- **Falta de abstracción del lenguaje:** El ingeniero tenía que escribir las plantillas directamente en el lenguaje complejo y rígido que exigen las herramientas de certificación finales, lo que dificultaba enormemente la legibilidad de las pruebas.
- **Falta de asistencia en el editor de plantillas:** El editor funcionaba a modo de bloc de notas, careciendo de ayudas propias de los entornos de desarrollo modernos (como VSCode).
- **Detección tardía de errores:** Al no existir validación en tiempo real, los fallos humanos solo se descubrían al final de todo el proceso, durante el intento de generación de archivos finales. Esto provocaba retardos en la certificación de pruebas.
- **Introducción manual de rutas:** El usuario tenía que buscar, escribir y actualizar manualmente las rutas de los mensajes y señales en la plantilla que contenía la lógica de pruebas. Ello derivaba en que el sistema fuera muy vulnerable a errores de escritura.

Esta arquitectura inicial, aunque era funcional para tareas básicas, dificultaba en gran medida la eficiencia del ingeniero de validación, debido a la dependencia de procesos manuales, la ausencia de abstracción sintáctica y la falta de soporte interactivo. Lo que derivaban en una alta curva de aprendizaje para llevar a cabo el proceso de generación de pruebas.

Por ello, este trabajo expande los límites de la herramienta preexistente, integrando capacidades que resuelven de manera directa las limitaciones detectadas en el departamento.

3.2. Descripción del nuevo entorno de generación de pruebas

La finalidad de este Trabajo de Fin de Grado reside en la creación de un nuevo entorno de generación de pruebas partiendo de la aplicación base proporcionada por Airbus Helicopters, con el objetivo de revertir el flujo de trabajo manual, la vulnerabilidad a errores y la compleja curva de aprendizaje identificadas en el sistema previo.

Entre las implementaciones del nuevo entorno se encuentran:

- **Evolución y optimización del editor de plantillas base (*Template Editor*):** Incorpora mejoras para el análisis léxico y resaltado de sintaxis Jinja2 en tiempo real, un sistema de búsqueda y autocompletado de variables (estilo VSCode) y atajos para la inyección de código Jinja2 mediante botones. Todas estas mejoras facilitan la generación de plantillas parametrizadas.
- **Flujo de procesamiento (*Jinja2 + DSL*):** Emplea un motor de traducción semántica basado en un Lenguaje de Dominio Específico (DSL) propio, de forma que tras el renderizado mediante Jinja2, apoyándose en el DSL desarrollado, se traduce al lenguaje final que exige la herramienta de certificación de pruebas, generando el *script* de pruebas final.
- **Procesador de archivos XML (*Parser*):** Encargado de la extracción, validación y resolución dinámica de rutas de mensajes y señales, mediante un *parser* de un archivo XML que almacena la información de las rutas.
- **Editor de plantillas con DSL (*Humanized Syntax Template Editor*):** Complementa al editor de plantillas base e incorpora un procesador de sintaxis humanizada, capaz de resolver al mismo tiempo rutas dinámicas de mensajes y señales. Está diseñado para abstraer la complejidad de ciertas herramientas de pruebas con lenguajes poco legibles. Permite la generación de plantillas parametrizadas utilizando una sintaxis humanizada más intuitiva y legible, proporcionando al usuario una ayuda de abstracción del lenguaje que le facilita el enfocarse únicamente en definir la lógica a probar.
- **Módulo de Inteligencia Artificial Generativa (*Gem*):** Asistente inteligente instruido con la gramática del DSL y escenarios de prueba sintéticos, que proporciona soporte al ingeniero de validación durante el proceso de generación de plantillas.

A modo de síntesis, esta descripción da una visión preliminar sobre la evolución arquitectónica del proyecto, destacando el procesamiento y análisis léxico-sintáctico de lenguajes formales (tanto del DSL como de estructuras XML), el diseño de nuevos módulos, la optimización de la

experiencia de usuario y la integración de un asistente inteligente a modo de guía en la creación de pruebas. Todas estas características suponen un valor fundamental para la herramienta, logrando transicionar de un proceso manual, lento y propenso a errores; a un ecosistema automatizado que reduce la curva de aprendizaje y disminuye drásticamente los tiempos empleados para la certificación de pruebas.

3.3. Comparativa de la aplicación base con el nuevo entorno de trabajo

Para dimensionar el impacto de la evolución arquitectónica descrita en el apartado anterior, resulta fundamental contrastar las capacidades operativas de la herramienta base frente a las del nuevo sistema. La siguiente tabla sintetiza cómo la integración tecnológica ha resuelto las carencias iniciales:

Característica/Módulo	Aplicación Base	Nuevo Entorno de Trabajo
Interfaz Gráfica	Estática, sin optimización, ni asistencia interactiva	Modernizada con CustomTkinter, contando con ayuda visual, menús guiados y asistencia interactiva
Editor de Plantillas Estándar	Estilo bloc de notas, en el que el usuario dependía de su conocimiento técnico exclusivamente para la redacción de pruebas	Editor optimizado con resaltado de sintaxis en tiempo real, sistema de autocompletado de variables y menús de inyección de código Jinja2
Editor de Plantillas con Sintaxis Humanizada	Inexistente, obligado a utilizar el editor de plantillas estándar y programar con el lenguaje final	Permite la redacción de plantillas con una sintaxis humanizada (DSL). Incluye formularios interactivos que inyectan código humanizado y un sistema de procesamiento para previsualizar el script final y detectar fallos
Lenguaje y Sintaxis	Uso obligatorio del lenguaje final requerido (rígido, complejo y difícilmente legible)	Implementación de un DSL, permitiendo al ingeniero redactar pruebas en un lenguaje natural e intuitivo
Motor de Procesamiento	Flujo único: Renderizado directo con Jinja2	Multi-flujo: Permite elegir entre el renderizado estándar o un flujo avanzado

		(renderizado Jinja2 + procesamiento DSL)
Validación y Detección de Errores	Los fallos solo se detectaban al intentar generar los archivos finales, ralentizando la depuración	Permite compilar en el editor, capturando excepciones y validando la sintaxis antes de procesar la plantilla
Resolución Dinámica de Rutas	Obligación de escribir la ruta completa de mensajes y señales. Vulnerabilidad a errores	Integración de un analizador XML que autocompleta las rutas de mensajes y señales introducidas por el usuario
Asistencia al Usuario Integrada	Nula, sin manuales interactivos ni ayudas. Alta curva de aprendizaje	Menús interactivos, menús guiados y ayudas contextuales integradas en la propia interfaz
Asistencia Inteligente (Módulo IA)	Inexistente	Herramienta independiente complementaria a la aplicación que asiste en el proceso de creación de pruebas

Tabla 3.1: Comparación aplicación base vs nuevo entorno

Tras haber delimitado el punto de partida tecnológico proporcionado por Airbus Helicopters y los avances arquitectónicos implementados para alcanzar la solución final, las siguientes secciones detallan exhaustivamente el diseño funcional, las características de la interfaz y la operativa del usuario en cada uno de estos módulos.

3.4. Descripción funcional

En este punto, se explicarán los diferentes módulos partiendo desde la Interfaz Principal de la herramienta, desde la cual el ingeniero de pruebas posee acceso a toda la funcionalidad de la aplicación. Asimismo, se describirá el módulo del Asistente Inteligente, el cual opera de forma independiente a la aplicación local para garantizar la seguridad de los datos del entorno corporativo. A través de este ecosistema de trabajo, el usuario puede:

- Configurar los *paths* por defecto, asociados a los directorios de ficheros de parámetros, plantillas o archivos procesados.

- Definir una sintaxis por proyecto, empleada en los ficheros de parámetros.
- Crear ficheros de parámetros (diccionarios de datos) asociados a plantillas parametrizadas (lógica de test a probar).
- Crear plantillas parametrizadas utilizando una sintaxis humanizada e intuitiva.
- Seleccionar los ficheros de parámetros a procesar, la plantilla asociada y el directorio en el que se guardarán los archivos procesados (*scripts* de pruebas finales).
- Escoger la metodología de procesamiento (Jinja2 o Jinja2 + DSL), a la hora de procesar plantillas parametrizadas con sus correspondientes ficheros de parámetros.
- Consultar al Asistente Inteligente para generar esqueletos de lógicas de validación, resolver dudas técnicas y recibir asistencia guiada durante la creación y configuración de las plantillas.

3.4.1. Interfaz principal y configuración

La interfaz principal actúa como núcleo central de la aplicación. Facilita el ciclo de generación de pruebas proporcionando acceso a manuales, configuradores, herramientas de edición y motores de generación de pruebas. El flujo de trabajo se estructura en tres fases:

1. Preparación y configuración del entorno

Previamente a la redacción y procesamiento de pruebas, el usuario debe comprender y configurar la herramienta según las necesidades del proyecto en el que se encuentre trabajando. Para ello, dispone de un menú superior que contiene las siguientes opciones:

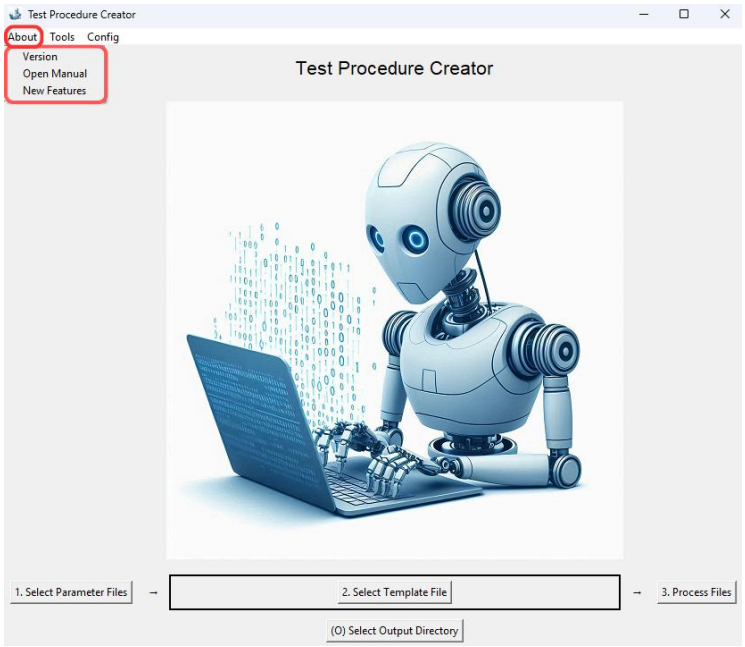


Figura 3.1: Opciones del menú About

- **About (Información del sistema y Documentación técnica):** Proporciona al usuario la versión actual de la aplicación, un manual técnico e información acerca de las nuevas características implementadas en la herramienta, de forma que se reduzca la curva de aprendizaje durante las primeras interacciones con el sistema.

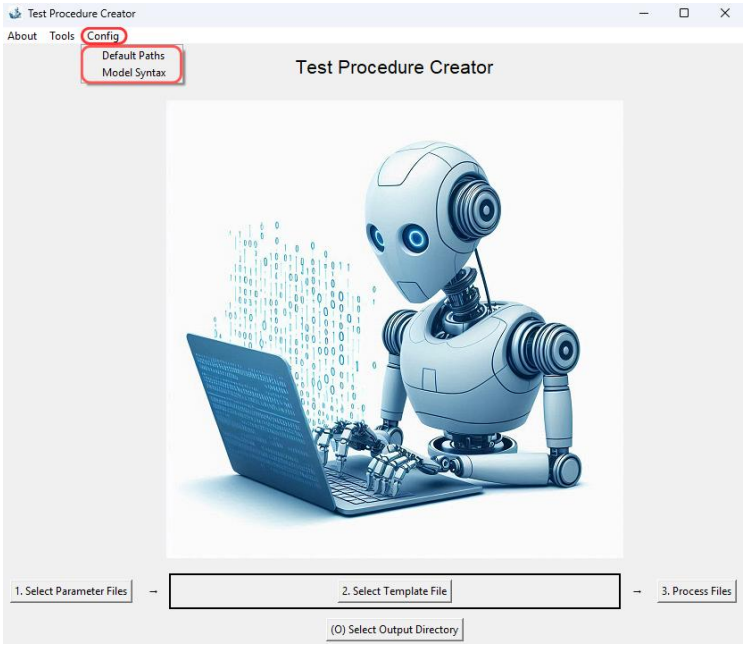


Figura 3.2: Opciones del menú Config

- **Config (Paneles de configuración del sistema):** Permite gestionar la configuración de los ajustes operativos de la aplicación, con el fin de garantizar la flexibilidad y adaptabilidad del sistema ante diferentes entornos de trabajo.

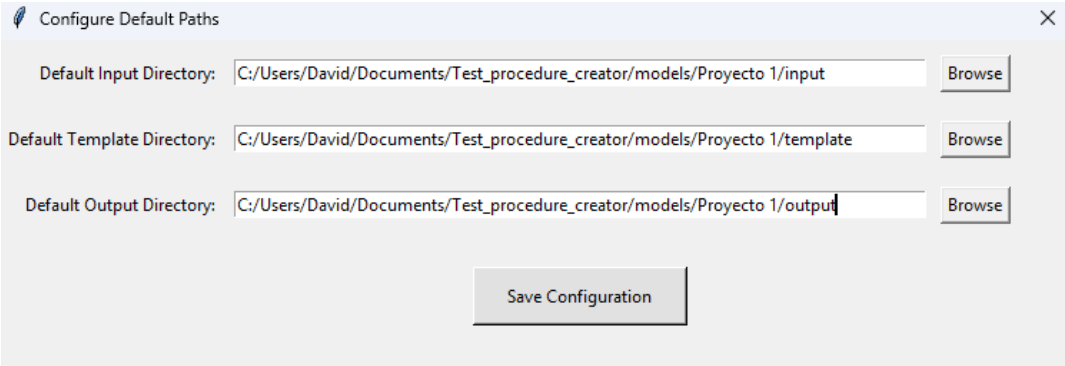


Figura 3.3: Configurador de directorios por defecto

- **Default Paths (Configuración de Rutas):** El ingeniero define los directorios de trabajo predeterminados (ficheros de parámetros, plantillas, *scripts* finales de pruebas), agilizando la selección de archivos durante el flujo de trabajo normal del ingeniero.

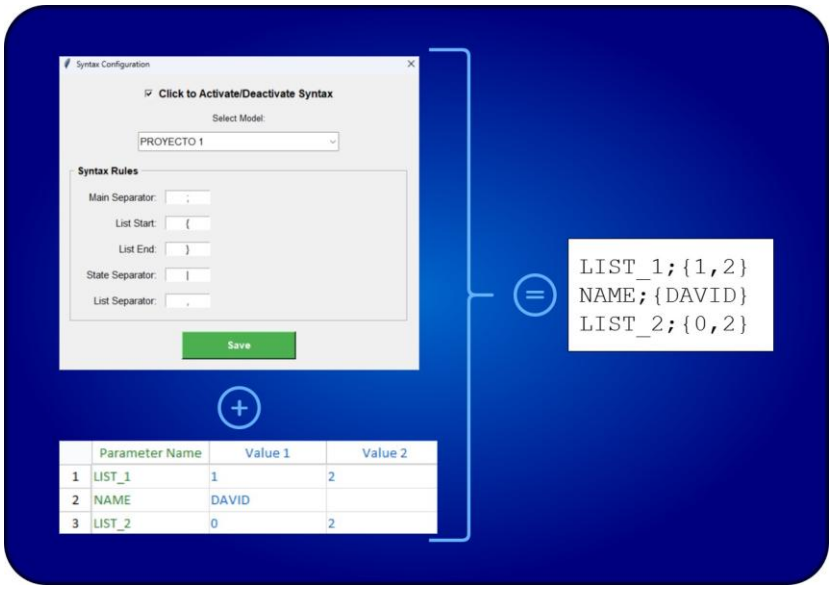


Figura 3.4: Configurador de sintaxis por proyecto

- **Model Syntax (Definición de Sintaxis):** El usuario define las reglas léxicas (delimitadores de listas y separadores de datos), que estructurarán y se utilizarán para el procesamiento de los ficheros de parámetros (diccionarios de datos). Además, para garantizar la retrocompatibilidad con baterías de pruebas anteriores, se permite desactivar la sintaxis definida, forzando a la aplicación trabajar con el flujo de procesamiento de la herramienta base.

2. Generación de pruebas

Una vez entendido y configurado el entorno, a través del menú desplegable ***Tools***, el ingeniero puede acceder de forma paralela a cada uno de los módulos de edición especializados, entre los que se encuentran:

- 1. El editor de ficheros de parámetros (*Parameter File Editor*).
- 2. Los editores de plantillas parametrizadas (*Template Editor* y *Humanized Syntax Template Editor*).
- 3. El visor de *scripts* de pruebas finales (*Processed File Viewer*).

Esta estructura, descrita en detalle en apartados siguientes, dota al usuario de capacidad para trabajar de forma paralela en la definición de los diccionarios de datos y en la construcción de la lógica de pruebas.

3. Procesamiento de plantillas

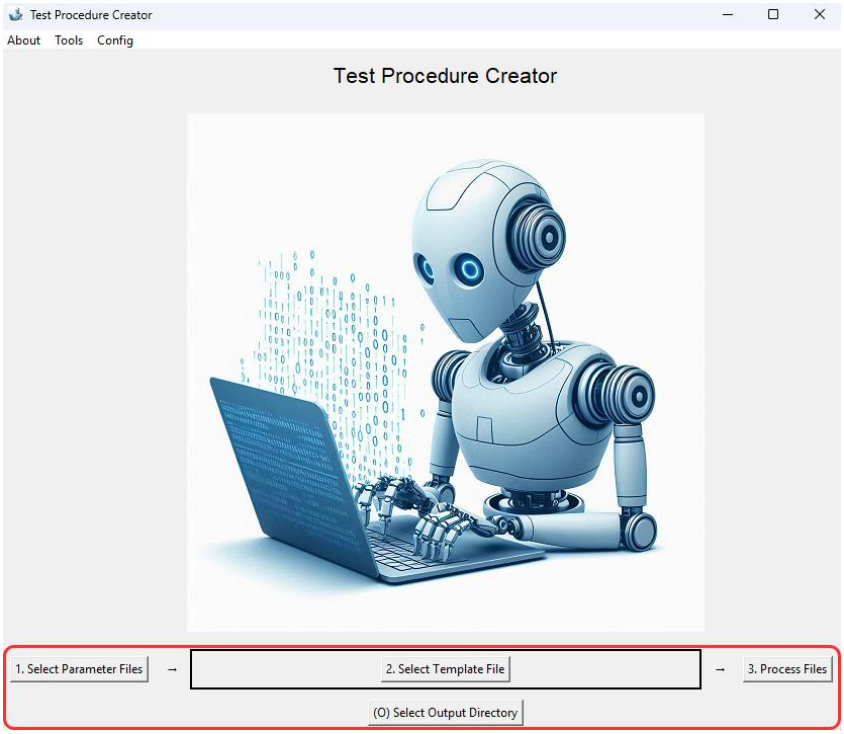


Figura 3.5: Menú de procesamiento de plantillas

El área inferior de la interfaz está diseñada para trabajar en un orden lógico secuencial, de manera que los pasos a seguir son:

- **Paso 1 (Select Parameter Files):** El usuario selecciona uno o múltiples ficheros de parámetros. Esto habilita el procesamiento por lotes y optimiza el flujo de ciclo de pruebas.
- **Paso 2 (Select Template File):** El ingeniero escoge la plantilla parametrizada que contiene la lógica de pruebas a ejecutar, en la que se inyectarán los datos procedentes de los ficheros de parámetros seleccionados en el paso anterior. Una vez escogida la plantilla, el usuario deberá seleccionar el método de procesamiento, mediante una ventana emergente con las siguientes opciones:

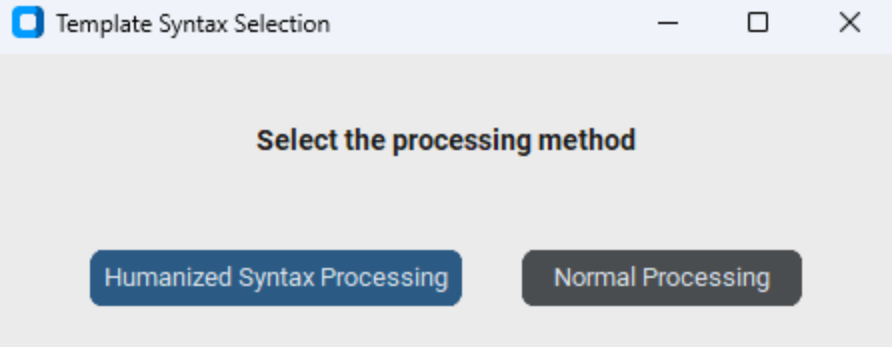


Figura 3.6: Ventana de selección de método de procesamiento

- **Humanized Syntax Processing**: Se ejecutará el flujo de renderizado Jinja2 seguido de un procesamiento del DSL, obteniendo como resultado el *script* de pruebas final en el lenguaje requerido.
- **Normal Processing**: Se ejecutará únicamente un renderizado Jinja2, obtenido el *script* de pruebas final.

- **Paso 3 (Select Output Directory):** Se determina el directorio donde se volcarán los *scripts* de pruebas procesados.
- **Paso 4: (Process Files):** Al pulsar este botón, se acciona el procesamiento en el que se siguen los siguientes pasos, de manera transparente para el usuario:
  - **Paso 4.1:** Se obtiene el diccionario de datos, extrayendo los parámetros respetando el *Model Syntax* activo (delimitadores de datos, separadores de variables).
  - **Paso 4.2:** Se ejecuta el método de procesamiento seleccionado por el usuario en el Paso 2, obteniendo los *scripts* finales de pruebas.
  - **Paso 4.3:** Se procede al volcado de los *scripts* finales de pruebas en el directorio seleccionado por el usuario (o en su defecto el definido en el *Default Paths*), listos para su importación directa en la correspondiente herramienta de certificación de pruebas oficial.

3.4.2. Módulo de edición especializada

Tal y como aparece en el punto anterior, una vez entendido y configurado el entorno a través del menú desplegable ***Tools***, el ingeniero puede acceder de forma paralela a cada uno de los módulos de edición especializados

3.4.2.1. Editor de Parámetros (Parameter File Editor)

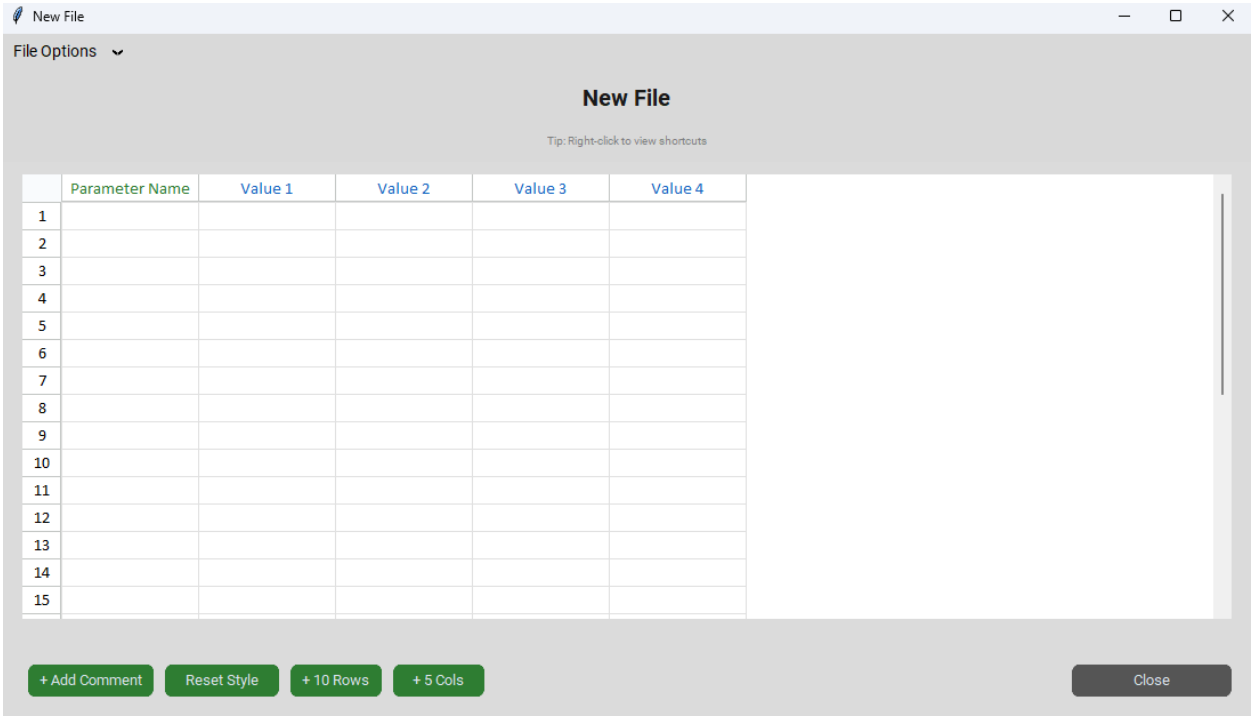


Figura 3.7: Editor de Parámetros (Parameter File Editor)

Para resolver la parametrización de las plantillas que contienen la lógica de pruebas, se necesita alimentar la plantilla con los datos específicos de cada escenario. Para facilitar la creación de estos datos, llamados en este proyecto como ficheros de parámetros, la herramienta cuenta con el Parameter File Editor, un entorno visual basado en una cuadrícula interactiva que simula una hoja de cálculo. A través de este editor, el flujo de trabajo para crear los diccionarios de datos se estructura de la siguiente manera:

- **Definición de variables y valores:** El usuario introduce los datos directamente en la tabla. La primera columna (*Parameter Name*) actúa como clave, mientras que las columnas adyacentes (*value 1, value 2...*) representan los diferentes valores de la variable. De manera que el usuario crea la estructura de datos de forma visual, abstrayendo el proceso de sintaxis, del cual se encarga el sistema.

- **Gestión de comentarios:** En caso de que el usuario necesite escribir comentarios, puede insertar notas mediante un botón dedicado (*+Add Comment)*. Esto despliega un modal textual que, al confirmarse, encapsula el texto entre delimitadores (###) y lo formatea en color gris sobre la celda seleccionada.

- **Gestión de la cuadrícula:** El ingeniero dispone de botones para expandir el área de trabajo dinámicamente (*+10 rows, +5 cols*), en caso de ser necesario.

- **Lógica de guardado:** Una vez definidos los datos, al invocar el comando *Save*, situado en el menú superior (*File Options*), el módulo recopila los datos de la cuadrícula y aplica las reglas definidas en el *Model Syntax* (insertando los separadores y delimitadores lógicos configurados) para generar un archivo de texto plano estructurado que el usuario ve en un modal a modo de *preview*, permitiendo su almacenamiento si el usuario aprueba el resultado.

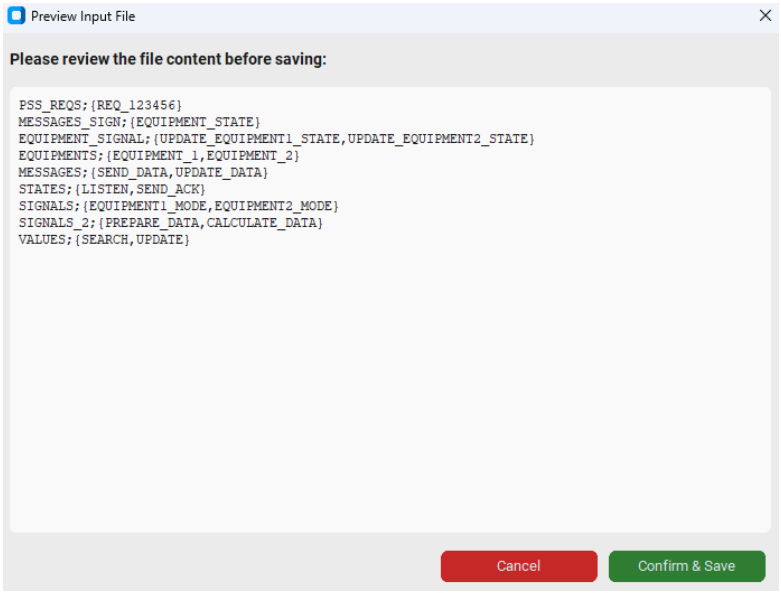


Figura 3.8: Preview de un fichero de parámetros

3.4.2.2. Editores de Plantillas

Una vez que los datos están definidos, el usuario debe redactar la plantilla que contiene la lógica de pruebas. La herramienta proporciona dos variantes para la redacción de plantillas, siendo su objetivo común dotar al ingeniero de un entorno con capacidades similares a un Entorno de Desarrollo Integrado (IDE), mitigando la curva de aprendizaje asociada a la sintaxis de programación y los posibles errores de formato asociados a la generación de *scripts* de pruebas.

Dependiendo de la herramienta de certificación de pruebas donde se vaya a ejecutar el test y del nivel de abstracción requerido, el usuario optará por uno de los dos entornos de trabajo diferenciados:

a) Template Editor (Estándar)

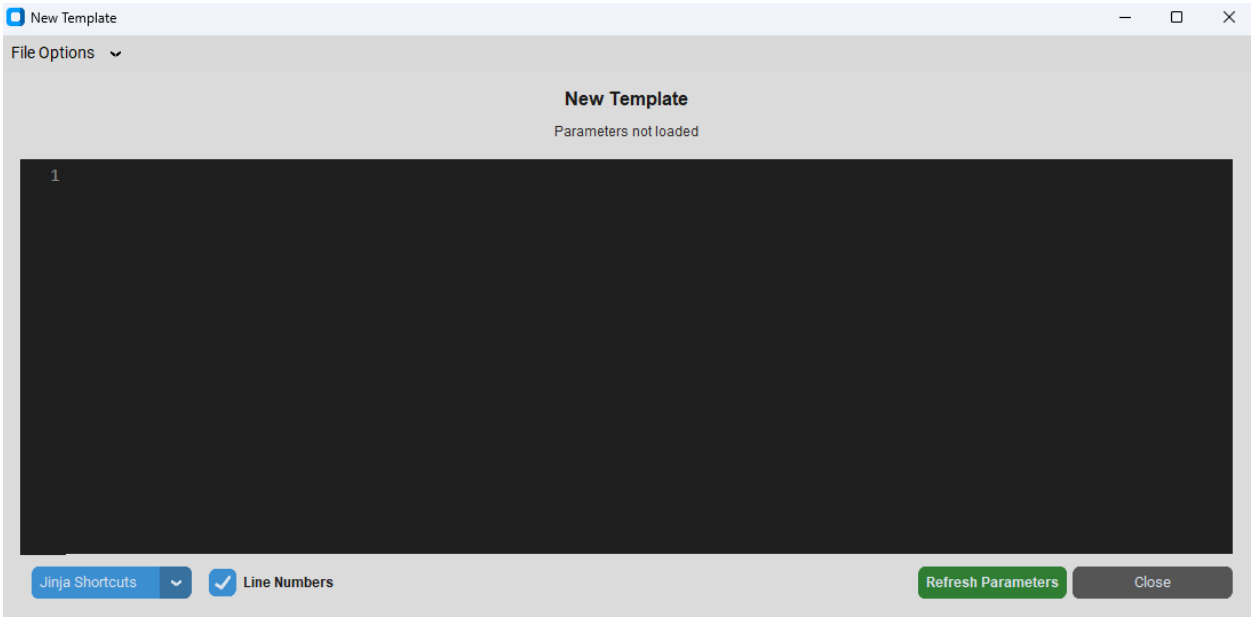


Figura 3.9: Editor Estándar (Template Editor)

Orientado a la creación de plantillas donde el ingeniero trabaja directamente con la sintaxis técnica de la herramienta de certificación de pruebas destino. Su diseño está enfocado en optimizar el proceso de generación de lógica de pruebas:

- **Precarga de contexto y Autocompletado de variables:** Previo a abrirse, se solicita opcionalmente la carga de un fichero de parámetros, esto alimenta un sistema de autocompletado de variables, asistiendo al usuario durante la instanciación de variables existentes y minimizando errores tipográficos. En caso de que se modifique el diccionario de datos en paralelo, el botón *Refresh Parameters* actualiza el sistema.

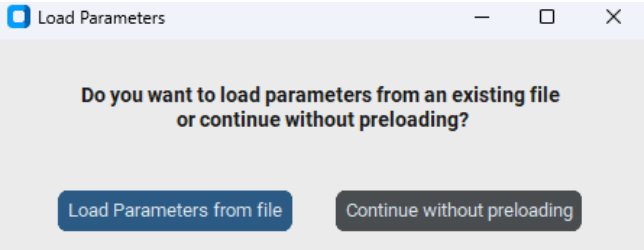


Figura 3.10: Ventana de selección de carga de parámetros

- **Gestión de archivos (File Options):** Menú superior que incluye la siguiente funcionalidad:
  - *Load parameters*: Permite seleccionar y cargar otro fichero de parámetros, para actualizar el sistema de autocompletado de variables.
  - *Open Template*: Carga el template seleccionado por el usuario dentro del editor de texto.
  - *Save y Save as*: Actualiza o guarda el contenido del template.
- **Editor de texto:** En el que el ingeniero de pruebas definirá la lógica a probar. Cuenta con funcionalidades propias de un entorno de desarrollo para facilitar la redacción de código como: resaltado sintáctico en tiempo real, asistiendo al usuario de manera visual para minimizar errores tipográficos, el sistema de autocompletado previamente explicado y funcionalidades como la de *Search and Replace* (Ctr + F), para facilitar la corrección de código y su mantenibilidad.
- **Panel de controles inferior:** Situado debajo del editor de texto, proporciona herramientas para optimizar el flujo de trabajo del usuario:

- ***Jinja Shortcuts**:* Menú desplegable que proporciona atajos para la inserción automática de bloques de código recurrentes (*Add Variable*, *Add For*, *Add Comment*, *Add Code*). Esta característica abstrae la complejidad sintáctica de Jinja2, permitiendo a usuarios no experimentados desarrollar plantillas de forma ágil y minimizando los errores de formato.
- ***Line Numbers**:* Un *toogle* que permite alternar la visualización de la numeración de líneas en el margen izquierdo del editor, facilitando la navegación y la depuración del documento.

b) Humanized Syntax Template Editor

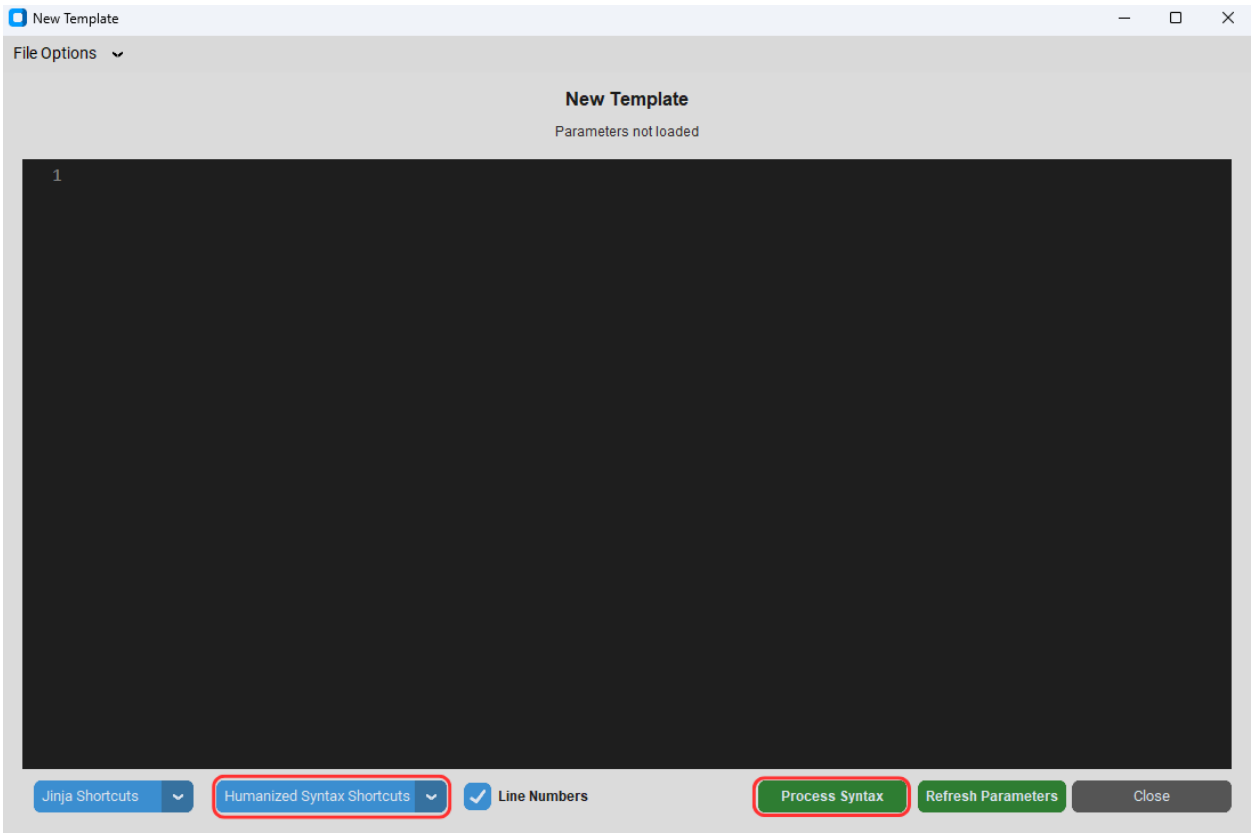


Figura 3.11: Editor Avanzado (Humanized Syntax Template Editor)

Este módulo amplía las capacidades del editor estándar integrando el motor de traducción DSL, permitiendo al usuario focalizarse únicamente en la lógica de pruebas, abstrayéndose de la compleja sintaxis de ciertas herramientas de certificación de pruebas.

El flujo de trabajo es facilitado por las siguientes funcionalidades:

- **Humanized Syntax Shortcuts:** Menú desplegable interactivo que mapea las instrucciones del DSL. Al seleccionar una instrucción, se abre un formulario que solicita los parámetros necesarios y, de forma transparente, inyecta la sintaxis correcta en el editor, permitiendo reducir la curva de aprendizaje asociada al DSL.
- **Process Syntax:** Permite compilar y validar la plantilla en tiempo real. Primero, el motor Jinja2 evalúa el código bajo la condición de lanzar una excepción visible si detecta variables no declaradas. Si la validación es exitosa, aplica la traducción DSL y muestra el *script* final en un visor de solo lectura, permitiendo su almacenamiento si el usuario aprueba el resultado.

3.4.2.3. Visor de Archivos Procesados (Processed File Viewer)

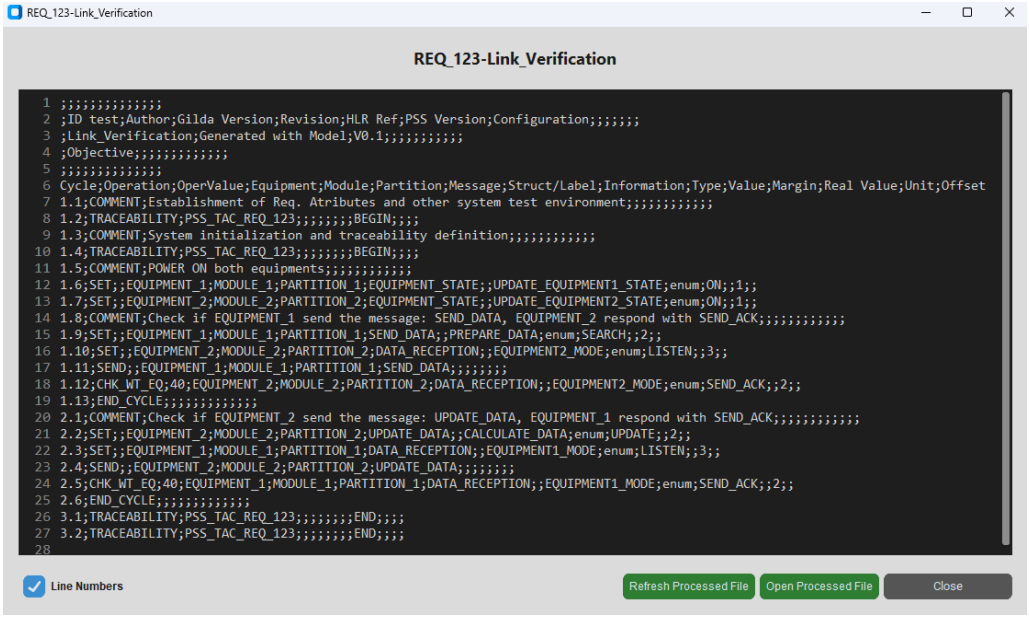


Figura 3.12: Visualizador de scripts finales (Processed File Viewer)

Un módulo complementario de solo lectura diseñado para la fase de revisión, permitiendo al ingeniero cargar cualquier archivo ya generado (*script* de pruebas), a través del botón *Open Processed File*, para validar su estructura y formato antes de proceder a la importación definitiva en la herramienta de validación.

3.4.3. Asistente Inteligente (Gem de Gemini)

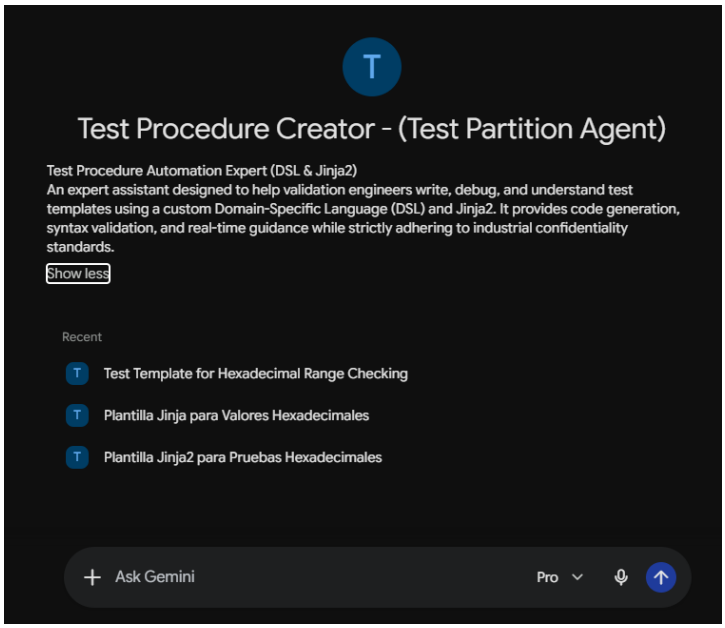


Figura 3.13: Interfaz del Asistente Inteligente de Gemini

Para reducir la curva de aprendizaje asociada a la herramienta y sus flujos de trabajo, se ha incorporado un Asistente Inteligente basado en el modelo Gemini. Este asistente actúa como un asistente de desarrollo, optimizando el proceso de generación de nuevas baterías de pruebas.

En el flujo normal de trabajo del ingeniero de validación, existen tres tipos de operaciones que le ayudan a reducir significativamente el tiempo de generación de pruebas, que son:

- **Resolución de dudas:** El asistente al estar instruido con la gramática del DSL y tener sólidos conocimientos sobre el motor de Jinja2 y su lenguaje, el usuario puede consultar al asistente, de forma que este le devuelva una explicación clara o un ejemplo de uso, evitando que el usuario se quede atascado ante un problema.

- **Generación de estructuras base:** El usuario puede solicitar la creación de plantillas a partir de descripciones en lenguaje natural. Por ejemplo, el ingeniero puede pedir al asistente que “redacte la lógica para probar que mandando una señal por el equipo X, el equipo Y la recibe”, y el asistente generará el código base con la sintaxis humanizada y aplicando bloques o parametrizaciones con Jinja2.
- **Validación y depuración:** El asistente puede analizar fallos y sugerir correcciones que el ingeniero deberá aplicar en las plantillas.

En conjunto, este módulo transforma la experiencia de usuario, siendo capaz de proporcionar asistencia durante todo el proceso de redacción de pruebas.

En cuanto a su despliegue y accesibilidad, el asistente se despliega de forma paralela a la aplicación en la nube utilizando el ecosistema corporativo empresarial (Google Workspace Enterprise). Para cumplir con las estrictas políticas de seguridad de la compañía, en lugar de operar con una API directamente en la herramienta, el Gem se publica de forma restringida dentro del dominio de Airbus Helicopters.

Este enfoque permite proporcionar acceso únicamente a los usuarios y departamentos que lo requieran para sus tareas de certificación. De este modo, se integra una asistencia al usuario blindando a su vez los datos corporativos. Toda la información sobre cómo configurar este asistente se encuentra en el archivo README.md del proyecto[2].

3.5. Descripción de la implementación

En esta sección se describe la arquitectura interna de la aplicación, desglosando los principales componentes que permiten su funcionamiento. La implementación se basa en una arquitectura modular desarrollada en Python, donde la capa de presentación (frontend) y la capa de lógica (backend) operan de forma integrada.

[2] https://github.com/daviidmartnez03/Test-Procedure-Creator/blob/main/README.md

De este modo, aunque cada capa se enfoca en su función, mantienen una interconexión que asegura la correcta ejecución de los procesos y la experiencia de usuario.

A continuación, se detallan los elementos estructurales más relevantes de la aplicación y cómo se integran entre sí para ofrecer una experiencia interactiva y fluida.

3.5.1. Estructura general del sistema

El sistema desarrollado está estructurado en tres niveles que interactúan de forma secuencial y coordinada para llevar a cabo la generación de pruebas:

- **Capa de Presentación (*Frontend*):** Responsable de la interacción directa con el usuario. Alberga la interfaz principal, los editores visuales y los editores de texto entre otros componentes. Su función es recopilar las entradas del usuario y gestionar la visualización de los resultados.
- **Capa de Control de Eventos (*Event Handler*):** Actúa como puente o enrutador entre la interfaz visual y el *backend*. De manera que, al producirse un evento en el *frontend* (por ejemplo, pulsar el botón *Process Files* o *Process Syntax*), el controlador entra en acción, empaqueta el estado actual de la aplicación (por ejemplo, los directorios seleccionados, el contenido del editor o el tipo de flujo elegido) y transfiere los datos al *backend* para su procesamiento correspondiente. También es el responsable de capturar los mensajes devueltos por el *backend* y traducirlos en ventanas modales para el usuario, de manera que el usuario pueda estar al corriente de las acciones realizadas y su estado.
- **Capa de Lógica de Negocio (*Backend*):** Núcleo computacional de la herramienta y responsable de, entre otras tareas, la transformación de los datos en pruebas finales. Recibe los datos capturados por la capa de control de eventos y realiza el correspondiente procesamiento, devolviendo el resultado para informar al usuario de las acciones llevadas a cabo y del estado de estas.

3.5.2. Flujos de procesamiento de plantillas

Una vez que la capa de control de eventos transfiere los datos al *backend*, el procesamiento de estos datos se adapta según las necesidades del escenario y del nivel de abstracción que requiera la herramienta de validación destino. Dado que este proceso constituye el método más crítico del *backend*, al ser el responsable de la generación de los *scripts* de pruebas finales, se detallará su funcionamiento en profundidad, pudiendo seguir dos flujos distintos, dependiendo de la metodología seleccionada por el usuario.

Ambos flujos comparten una fase inicial común: al invocar la acción *Process File* en el panel principal (ver Figura 9), la herramienta inicia un procesamiento secuencial de cada uno de los archivos de parámetros seleccionados, extrayendo las variables respetando la configuración sintáctica definida en el *Model Syntax* (ver Figura 8). Tras la fase inicial, el flujo difiere según la naturaleza del proyecto.

3.5.2.1. Flujo Estándar (Renderizado Jinja2 Directo)

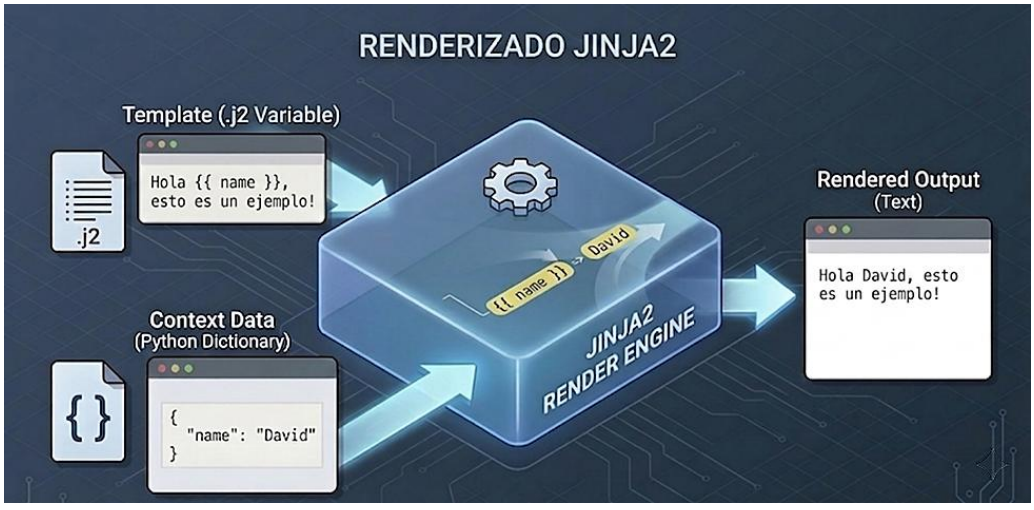


Figura 3.14: Esquema de Flujo Estándar de Renderizado Jinja2

Este modo de ejecución está orientado a la generación de pruebas para herramientas de validación que cuentan con un intérprete semántico integrado o cuyo formato de entrada estándar es naturalmente legible por el ingeniero. Se divide en las siguientes fases:

- **Fase de Procesamiento:** El motor de renderizado Jinja2 actúa como único procesador. Recibe la estructura de la plantilla elaborada en el *Template Editor* y el diccionario extraído del *Parameter File*.
- **Evaluación y Seguridad:** Las etiquetas de control condicional, los bucles iterativos ({% for %}) y las expresiones de inyección de variables ({{ variable_1 }}) se resuelven secuencialmente. Durante esta etapa, el sistema aplica políticas de validación estrictas, de forma que, si se detecta una divergencia entre las variables de la plantilla y los datos proporcionados, el proceso se aborta de forma segura para prevenir la generación de un *script* de pruebas inconsistente.
- **Salida de Datos:** Tras la evaluación, el motor compila un documento final de texto plano que se almacena automáticamente en el directorio de salida configurado (*Output Directory*). Este archivo es un *script* de pruebas terminado, listo para su importación en el entorno de pruebas correspondiente.

3.5.2.2. Flujo Avanzado (Renderizado Jinja2 + Traducción DSL)

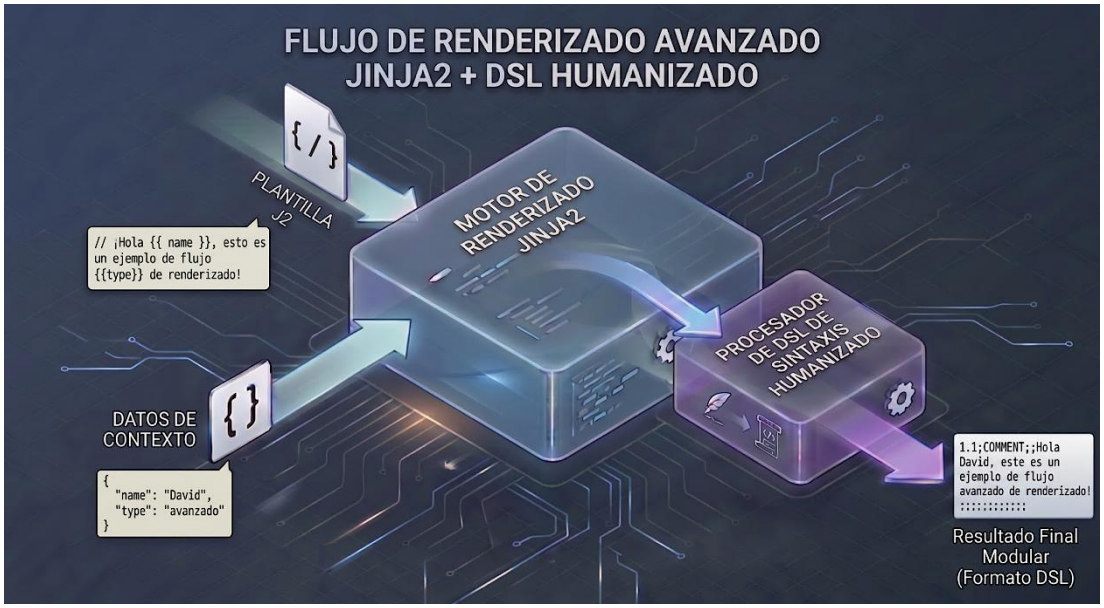


Figura 3.15: Esquema de Flujo Avanzado de Renderizado Jinja2

Este flujo ha sido diseñado estratégicamente para interactuar con sistemas de certificación cuyo lenguaje dificulta la legibilidad humana. Su ejecución se realiza en dos etapas:

- **Etapa 1: Procesamiento Jinja2 (Intermedio):** El motor Jinja2 ejecuta su ciclo habitual inyectando los parámetros en la plantilla. No obstante, en este flujo la plantilla está redactada en el Lenguaje Específico de Dominio (DSL) "humanizado" desarrollado internamente con la herramienta. El resultado de esta etapa es un documento transitorio en memoria: una versión expandida del test donde todos los bucles han sido desenrollados y los parámetros sustituidos, pero manteniendo la sintaxis humanizada.
- **Etapa 2: Traducción Semántica (textX) y Resolución de Rutas:** Inmediatamente después del renderizado inicial, el analizador léxico-sintáctico impulsado por *textX* toma el control, iniciando el proceso de traducción que se descompone en cuatro fases:

    1. **Análisis Léxico e Instanciación del Grafo de Objetos:** El analizador léxico de textX lee el documento renderizado en memoria línea por línea, lo contrasta con el metamodelo definido y valida la gramática e instancia un Árbol de Sintaxis Abstracta (AST) en la memoria de Python, donde cada instrucción de la prueba se convierte en un nodo con atributos manejables.
    2. **Resolución de Rutas:** El módulo de parseo XML extrae y mapea las rutas absolutas en el archivo de configuración corporativo, definiendo las rutas correspondientes a los mensajes y señales introducidos.
    3. **Mapeo y Traducción Semántica:** Cada tipo de nodo instanciado en el AST produce una traducción específica en el *backend*. El *backend* procesa el tipo de instrucción introducida, tomando los valores definidos por el usuario (equipos, mensajes, señales) y los inyecta dentro de las estructuras rígidas que componen el lenguaje final.
    4. **Ensamblaje Final:** El resultado del procesamiento línea por línea de forma iterativa es el ensamblaje del *script* de pruebas final listo para su importación en la herramienta de pruebas. El valor de este flujo radica en que el ingeniero de validación nunca interactúa con la sintaxis compleja final, sino que diseña la lógica en lenguaje natural (apoyado

por el *Humanized Syntax Template Editor*), siendo el motor quien asume la carga de la traducción técnica.

A continuación, se muestra un ejemplo de transformación de una instrucción definida en el metamodelo:

Instrucción definida por el usuario (DSL):

SEND: SEND_DATA

Instrucción traducida (Lenguaje Final):

1.1;SEND;;EQUIPMENT_1;MODULE_1;PARTITION_1;SEND_DATA;;;;;;;;

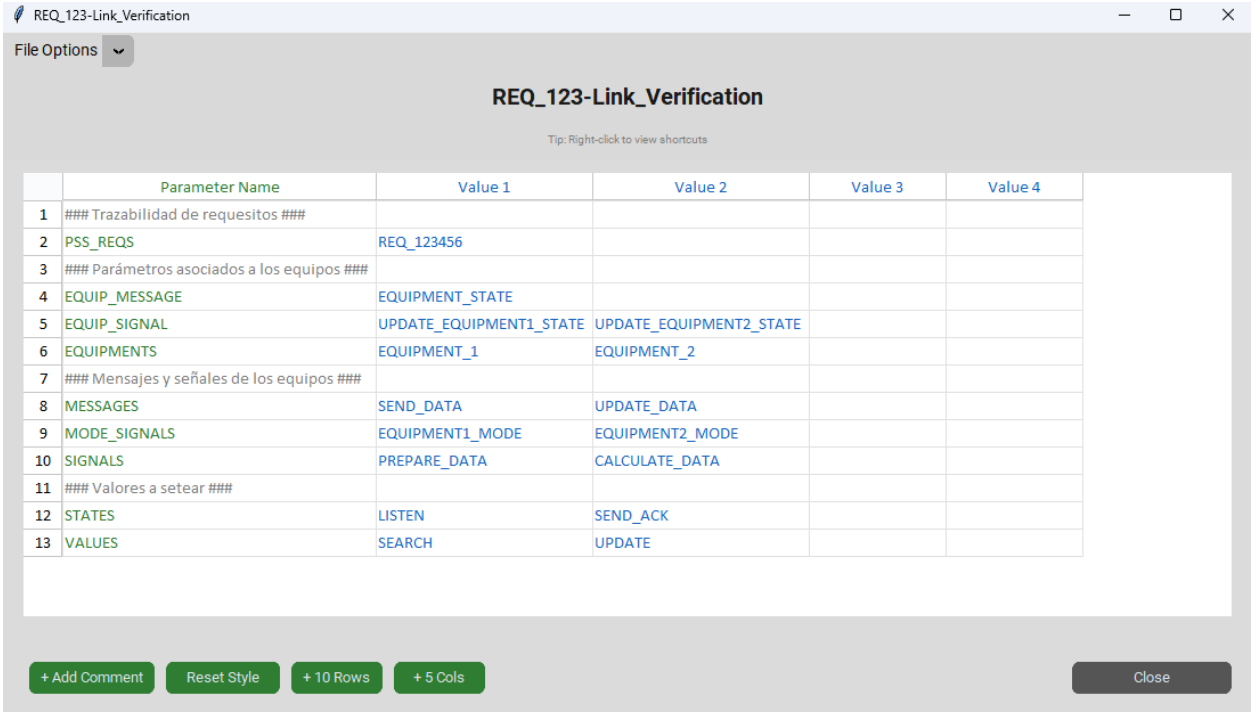
3.6. Ejemplo de uso

A continuación, se expone un caso de uso que ilustra un posible ciclo de trabajo de un ingeniero de validación utilizando el flujo de ejecución avanzado (Jinja2 + DSL). Este ejemplo proporciona una demostración de los pasos a seguir, y de cómo el usuario interactúa con la herramienta y con el asistente inteligente para crear un *script* de pruebas abstrayéndose del lenguaje requerido por la plataforma de certificación.

Para este ejemplo, se asume como punto de partida que el usuario ya ha completado la fase de **Preparación y configuración del entorno** (véase *Preparación y configuración del entorno*).

Paso 1: Creación del fichero de parámetros

Se comienza con la definición de los datos específicos del escenario de prueba. De manera que, el usuario accede al módulo **Parameter File Editor** desde el menú principal y, utilizando la interfaz de cuadrícula, introduce los nombres de las variables en la primera columna (Parameter Name) y sus valores asociados en las columnas adyacentes.



	Parameter Name	Value 1	Value 2	Value 3	Value 4
1	### Trazabilidad de requesitos ###				
2	PSS_REQS	REQ_123456			
3	### Parámetros asociados a los equipos ###				
4	EQUIP_MESSAGE	EQUIPMENT_STATE			
5	EQUIP_SIGNAL	UPDATE_EQUIPMENT1_STATE	UPDATE_EQUIPMENT2_STATE		
6	EQUIPMENTS	EQUIPMENT_1	EQUIPMENT_2		
7	### Mensajes y señales de los equipos ###				
8	MESSAGES	SEND_DATA	UPDATE_DATA		
9	MODE_SIGNALS	EQUIPMENT1_MODE	EQUIPMENT2_MODE		
10	SIGNALS	PREPARE_DATA	CALCULATE_DATA		
11	### Valores a setear ###				
12	STATES	LISTEN	SEND_ACK		
13	VALUES	SEARCH	UPDATE		

Figura 3.16: Ejemplo de fichero de parámetros real

Una vez finalizada la inserción de datos, el ingeniero pulsa *Save* o *Save as*. Automáticamente, el sistema aplica las reglas del *Model Syntax* y genera el *preview* del fichero de parámetros, el cual el ingeniero procede a guardar en caso de estar conforme con el resultado.

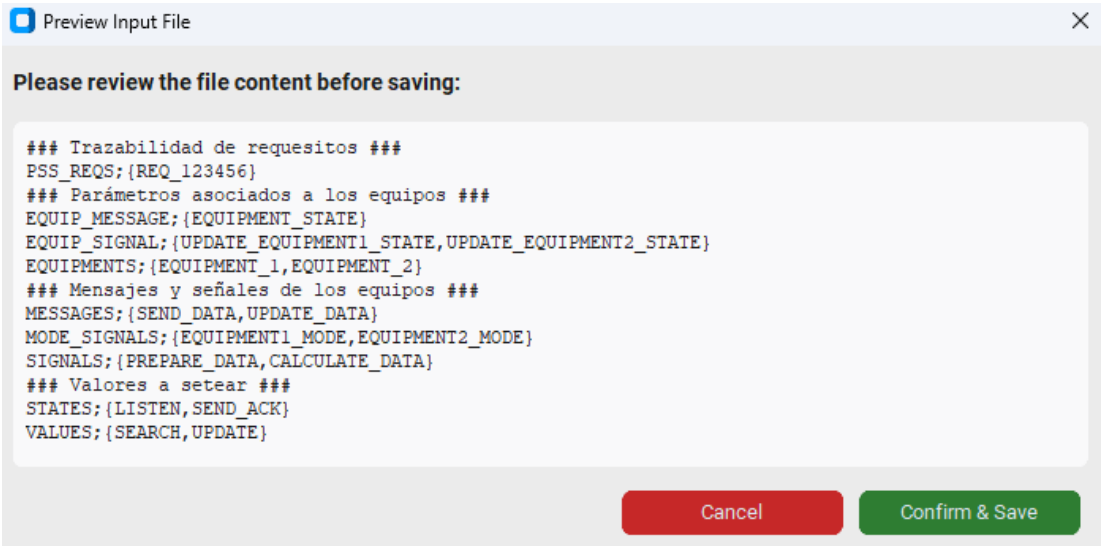


Figura 3.17: Preview del fichero de parámetros creado

Paso 2: Definición de la lógica de pruebas

Una vez finalizado y guardado el fichero de parámetros, se procede a definir la lógica del test. Para ello, se emplea el **Humanized Syntax Template Editor**, cargando previamente el fichero de parámetros recién creado, para activar el sistema de autocompletado de variables en tiempo real.

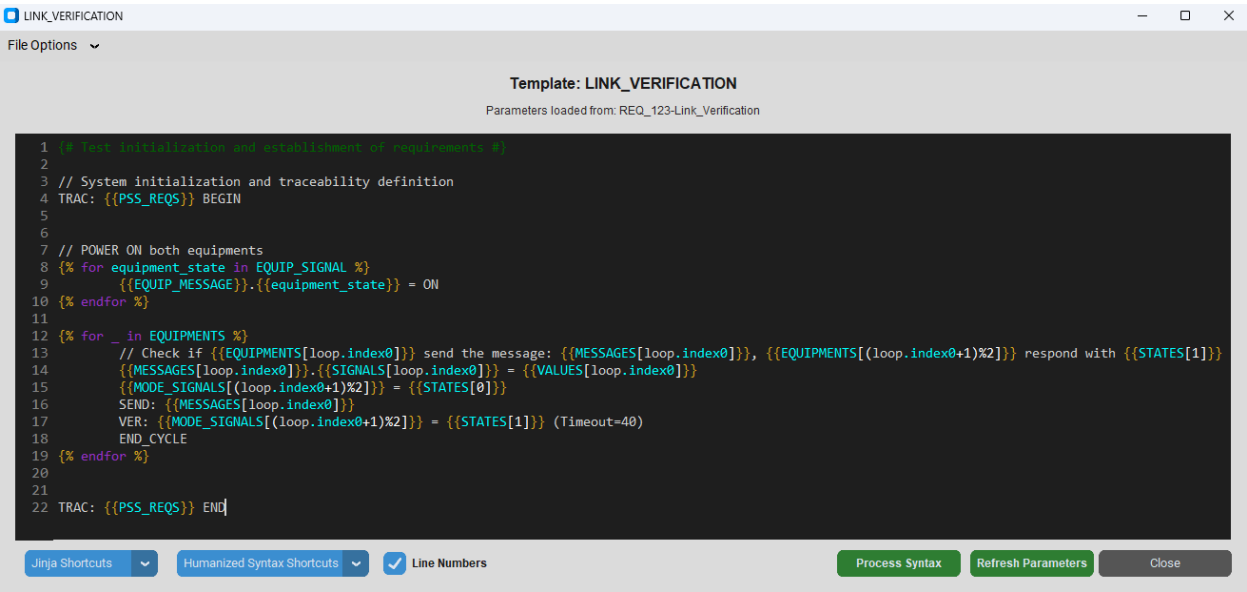


Figura 3.18: Ejemplo de template real

Paso 3: Validación y previsualización en el editor

Una vez definida la lógica de test, el ingeniero debe asegurarse que la lógica es correcta y que no existen errores gramaticales. Para ello, pulsa el botón **Process Syntax** situado en el panel inferior del editor.

De manera transparente al usuario, el sistema ejecuta las Etapas 1 y 2 del Flujo Avanzado de procesamiento explicado previamente (véase *Flujo Avanzado*). Si no existen variables sin declarar ni errores de formato, el sistema despliega un *preview* mostrando el *script* de pruebas resultante. Esta previsualización permite al usuario confirmar que su lógica definida con la sintaxis humanizada se ha traducido correctamente al formato técnico esperado por la herramienta de certificación y guardar el *script* final de pruebas ya en el lenguaje requerido.

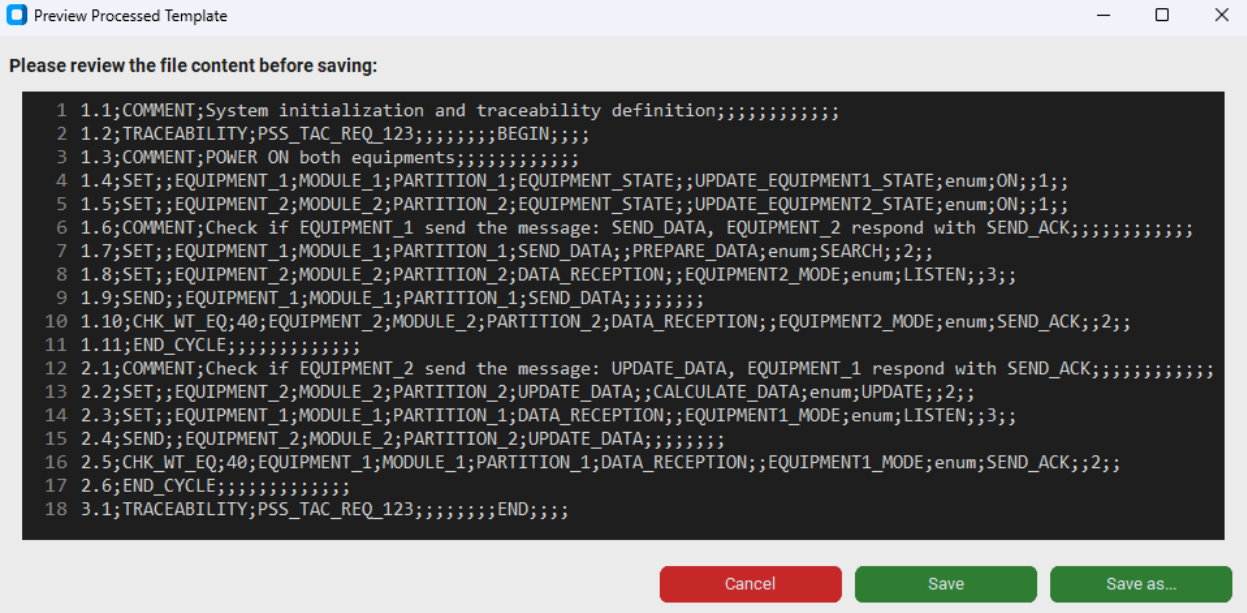


Figura 3.19: Preview del template procesado con el fichero de parámetros

Paso 4: Generación del *script* de pruebas

Tras validar que la lógica definida en la plantilla es correcta, el ingeniero regresa a la interfaz principal de la aplicación para ejecutar la generación definitiva. Aunque el paso anterior permite guardar el *script* generado individualmente, el uso de la interfaz principal resulta fundamental porque habilita el procesamiento masivo de múltiples ficheros de parámetros de forma simultánea. Siguiendo el flujo guiado descrito previamente (véase *Procesamiento de plantillas*), el sistema genera finalmente el *script* (o conjuntos de *scripts*) de pruebas listo para ser importado en la herramienta de certificación de pruebas destino.

Intervención y Soporte del Asistente Inteligente durante la generación del caso de prueba

Cabe destacar que el asistente de Inteligencia Artificial estuvo presente durante todo el proceso de generación del ejemplo de uso. Primeramente, a partir de descripciones en lenguaje natural, el modelo interpretó los requerimientos proporcionados por el usuario y generó el esqueleto con la lógica de pruebas inicial.

Posteriormente, la interacción se desarrolló de manera dinámica: el ingeniero de validación formuló consultas al asistente a través de *prompts* específicos sobre el ajuste de la lógica y su parametrización. Mediante este proceso, el usuario fue ajustando la plantilla hasta alcanzar el resultado deseado. Finalmente, se utilizó al asistente para validar la sintaxis humanizada de la plantilla previa a su compilación.

Esta asistencia supuso la reducción de la carga cognitiva del usuario y del tiempo necesitado para la redacción del modelo, ayudando además a garantizar la calidad estructural del código desde el primer paso.

Capítulo 4

Desarrollo del proyecto

Este capítulo detalla las etapas de desarrollo, las decisiones técnicas adoptadas y los desafíos superados durante la implementación del sistema. Partiendo de la arquitectura inicial y de los requerimientos establecidos por parte de Airbus Helicopters, el desarrollo del sistema se llevó a cabo mediante una implementación incremental. Si bien en la práctica, no se ejecutó bajo los principios de metodologías ágiles, para la explicación del desarrollo se ha adoptado una división basada en *sprints*. El uso de estos no representa bloques temporales de desarrollo exactos, sino que actúa como un recurso que permite explicar y estructurar los diferentes hitos del proyecto. Esta estrategia facilita la comprensión del avance progresivo del sistema: partiendo desde la exploración técnica y reestructuración de la arquitectura inicial hasta la evolución de la interfaz gráfica, la creación de un Lenguaje de Dominio Específico (DSL) y la integración de Inteligencia Artificial, que en conjunto dieron lugar a la implementación de la solución final.

4.1. Sprint 1: Análisis del sistema base y diseño de la nueva generación

Objetivo

El propósito de esta primera fase fue realizar una inmersión técnica en el software proporcionado inicialmente por Airbus Helicopters, con la finalidad de entender la funcionalidad y el código fuente del proyecto, para poder identificar cuellos de botella y limitaciones arquitectónicas y posteriormente diseñar la nueva generación del sistema.

Actividades

- **Revisión del código fuente:** Revisión de los módulos desarrollados en Python, entre los que se encontraban:
    - El configurador de directorios por defecto.
    - El configurador de sintaxis.
    - La gestión de eventos de la interfaz principal.
    - Los distintos editores.
    - El procesamiento de plantillas.
    - La llamada directa al motor Jinja2.
- **Identificación de limitaciones:** Entre las que se encontraron:
    - **Alta curva de aprendizaje:** Al carecer de herramientas visuales de asistencia como resaltado de sintaxis, autocompletado o atajos interactivos, el editor de plantillas inicial actuaba como un simple "bloc de notas". Esta situación obligaba al ingeniero de validación a memorizar la compleja sintaxis de programación. Además, ante la ausencia de validación en tiempo real, los fallos, como por ejemplo las variables mal escritas, solo se descubrían al final, ralentizando enormemente el proceso.
    - **Falta de legibilidad en los *scripts* de pruebas:** Al redactar las pruebas directamente en el lenguaje técnico y poco intuitivo que exigen algunas herramientas de certificación, se dificultaba en gran medida el entendimiento de las plantillas creadas.
    - **Alta dependencia de la configuración manual de rutas:** El usuario tenía que introducir y actualizar manualmente las rutas de los mensajes y señales, lo que hacía que la aplicación fuera muy vulnerable a errores humanos o a fallos si la estructura de rutas cambiaba.

- **Diseño de la nueva arquitectura:** Trazado de diagramas de flujo para planificar el nuevo entorno y los requisitos a implementar, entre los que se definieron:
    - **Entorno de edición optimizado:** Proporcionando asistencia en tiempo real, incluyendo resaltado de sintaxis, menús dinámicos para la inserción de código y autocompletado de variables basado en la precarga de ficheros de parámetros.
    - **Lenguaje de Dominio Específico (DSL):** Permitiendo al usuario diseñar pruebas utilizando una sintaxis "humanizada" y comprensible, para luego traducirla automáticamente al lenguaje requerido por la herramienta de certificación final.
    - **Flujo de procesamiento adaptativo:** Añadiendo un flujo de ejecución adicional que combine el renderizado Jinja2 con el procesamiento y traducción del DSL.
    - **Resolución de rutas dinámicas:** Incorporación de un módulo capaz de analizar y extraer información de ficheros XML internos corporativos, con el fin de resolver de forma dinámica las rutas de los mensajes y señales introducidos en las pruebas, minimizando la intervención manual.
    - **Asistente basado en IA:** Integración de un módulo de inteligencia artificial generativa, capaz de generar código, resolver dudas de sintaxis y guiar al usuario en la creación de plantillas.
- **Configuración del entorno local:** Preparación del entorno de desarrollo mediante la creación de un entorno virtual, instalando en él las dependencias base (Tkinter, CustomTkinter, tksheet) y las nuevas librerías requeridas para las siguientes fases (Pygments, textX).

Prototipo generado

Mapa arquitectónico definido y un flujo de desarrollo estructurado. El código heredado se asentó, refactorizó y limpió para separar estrictamente las clases de la interfaz visual de las funciones de procesamiento lógico, estableciendo así las bases para el desarrollo del nuevo entorno de generación de pruebas.

Lecciones Aprendidas

- **Evaluación del software preexistente:** Comprender en su totalidad las limitaciones del software base, permitió enfocar el esfuerzo de desarrollo en la experiencia de usuario y en la mitigación de errores sintácticos.
- **Desacoplamiento esencial:** La separación temprana de la lógica de presentación y la de negocio es crítico para evitar fallos en cascada al introducir nuevas tecnologías.
- **Experiencia del usuario:** Es importante tenerla en cuenta desde el principio, antes de profundizar en la estructuración del nuevo software.

4.2. Sprint 2: Evolución del Frontend

Objetivo

Transformar las interfaces de usuario básicas en un entorno de trabajo mejorado, incorporando capacidades interactivas y de asistencia visual, que mitiguen la curva de aprendizaje, optimicen el flujo de creación de plantillas y reduzcan la tasa de errores humanos.

Actividades

- **Refactorización del Parameter File Editor:** Se implementó lógica visual, renderizando la primera columna (*Parameter Name*) en verde para identificar las claves, y el resto (*Values*) en azul. Adicionalmente, se desarrollaron diversas funcionalidades, entre las que se encuentran:
    - Inserción de comentarios inyectando delimitadores (###) y aplicando estilos en gris oscuro (*+ Add Comment*).
    - Expansión del área de trabajo dinámicamente (*+10 rows*, *+5 cols*).
    - Formateo de estilos de celda (*Reset Style*).

- **Integración de Análisis Léxico:** Se integró la librería Pygments en el *Template Editor*, aplicando un analizador que escanea el texto en tiempo real y aplica resaltado de sintaxis sobre las etiquetas de código Jinja2, variables y texto plano.
- **Mejoras de ergonomía:** Se incluyeron ayudas visuales, como un visor de números de líneas (*Line Numbers*) para facilitar la depuración, y un botón de refresco (*Refresh Parameters*) para actualizar dinámicamente las variables en memoria, si el fichero de parámetros se modifica en paralelo.
- **Desarrollo del menú *Jinja Shortcuts*: Se** implementó un sistema de inserción de código Jinja2, proporcionando menús desplegables y botones que inyectan de forma automática la sintaxis correcta para bucles, variables y comentarios en la posición del cursor, abstrayendo la complejidad al usuario.
- **Sistema de precarga y autocompletado: Se** creó un módulo que permite al editor de plantillas leer, antes de su apertura, el fichero de parámetros y extraer un diccionario de variables disponibles.

Prototipo generado

Dos módulos de edición funcionales y mejorados, que dotan al ingeniero de validación de un entorno tabular con estilos optimizados y con acciones dinámicas, y un editor de plantillas estándar (*Template Editor*) dotado de herramientas de asistencia, resaltado sintáctico y menús de inyección rápida.

Lecciones aprendidas

- **Interfaz visual:** En herramientas de uso intensivo industrial, la codificación por colores en las tablas y el resaltado sintáctico, reducen drásticamente el desgaste cognitivo y los errores humanos.

- **Curva de aprendizaje reducida:** La creación de atajos interactivos (*Shortcuts*) consiguen abstraer la sintaxis de programación Jinja2 reduciendo el nivel de dificultad inicial para el ingeniero de validación y minimizando errores de sintaxis.

4.3. Sprint 3: Diseño e implementación del DSL

Objetivo

Diseñar y construir una gramática formal mediante el metalenguaje textX que permita definir instrucciones de pruebas legibles por humanos, sentando así las bases para su posterior traducción al lenguaje esperado por la herramienta de validación final.

Actividades

- **Análisis léxico-sintáctico del lenguaje máquina objetivo:** Estudio del lenguaje destino, para la definición y estructuración de la sintaxis "humanizada", que empleará el ingeniero de validación.
- **Definición del metamodelo:** Redacción de las reglas gramaticales en el formato de textX, creando un modelo que muestra cómo cada instrucción escrita en lenguaje humanizado (palabras clave, parámetros, delimitadores) se descompone en *tokens* sintácticos que el analizador puede entender.

```
// Definición de la estructura semántica para la instrucción SEND con // la forma breve
Send_ShortForm:
    'SEND:' message_name=ValueTypes
;

// Tipos permitidos como valor del mensaje
ValueTypes:
    NUMBER | STRING
;
```

- **Programación de las reglas de traducción:** Gracias a las reglas gramaticales, cuando el ingeniero introduce una instrucción que sigue una regla gramatical (por ejemplo, SEND: mensaje_de_prueba), el analizador de textX la verifica e instancia un objeto de Python en memoria. En este caso, se instanciaría un objeto Send_ShortForm con el atributo message_name, que contendría el valor "mensaje_de_prueba".

```
# Representación conceptual del objeto instanciado
objeto_ast = Send_ShortForm()
objeto_ast.message_name = "mensaje_de_prueba"
```

Este comportamiento resulta fundamental, ya que permite traducir las instrucciones definidas por el usuario en lenguaje humanizado en grafos de objetos trazables dentro de un Árbol de Sintaxis Abstracta (AST).

Posteriormente, estos se convierten en la entrada principal para la fase de traducción al lenguaje de la herramienta de pruebas, garantizando una conversión robusta y libre de errores.

```
# Procesamiento de la instrucción en el backend
match instruccion.__class__.__name__:
 case "Send_ShortForm":
          # El sistema identifica la clase y se accede a sus atributos específicos
          generar_codigo_maquina(instrucción.message_name)
```

Prototipo generado

Un motor de traducción semántica robusto operando en segundo plano, en el que el DSL actúa como traductor universal dentro del contexto del proyecto, asegurando que la intención del ingeniero se traduzca de forma inequívoca al formato certificado.

Lecciones aprendidas

- **Profundidad de diseño de un DSL:** La creación de un DSL efectivo requiere un profundo entendimiento de la metodología de validación y del lenguaje destino, asegurando que las instrucciones humanizadas resulten intuitivas y reflejen los procesos de validación reales.
- **Retorno de inversión temporal:** Aunque el diseño y la definición inicial de una gramática formal exigen un esfuerzo analítico y de desarrollo considerable, la abstracción de la complejidad y la consiguiente reducción de errores humanos lograron una disminución drástica en los tiempos de generación de cada nueva batería de pruebas.

4.4. Sprint 4: Reestructuración del Backend

Objetivo

Rediseñar la arquitectura de procesamiento de la herramienta para soportar ejecuciones adaptativas (Jinja2 o Jinja2 + DSL) e incorporar un sistema autónomo de resolución de rutas y dependencias mediante el análisis de un fichero XML que contiene la información.

Actividades

- **Arquitectura de procesamiento multi-flujo:** Modificación de la arquitectura original, basada únicamente en Jinja2, permitiendo un flujo de ejecución adicional:
    - **Etapa 1 - Renderizado intermedio:** El motor Jinja2 resuelve lógicas de programación e inyecta los diccionarios de datos del *Parameter File* en la plantilla. El resultado es un documento transitorio en memoria, redactado en sintaxis humanizada.
    - **Etapa 2 – Traducción al lenguaje final:** El documento intermedio es inyectado en el analizador de textX, que valida la gramática y compila el *script* de pruebas definitivo.

- **Desarrollo del analizador XML:** Creación de un módulo de procesamiento de archivos XML para extraer, mapear y resolver de forma dinámica las rutas de mensajes y señales introducidos en las instrucciones del *script* de pruebas. Integrado en la Etapa 2 anteriormente mencionada.

Prototipo generado

Un *backend* de procesamiento mejorado y adaptivo. El sistema pregunta al usuario para aplicar el flujo necesario, apoyado por una capacidad de auto-resolución de dependencias a través del *parseo* del archivo XML.

Lecciones aprendidas

- **Eficiencia de la arquitectura:** Dividir la transformación en etapas (renderizado Jinja2 → traducción semántica), facilita enormemente la depuración de los *scripts* de pruebas, aislando los fallos de renderizado de los fallos gramaticales.
- **Autonomía del sistema:** La integración de la resolución de rutas dinámicas otorga una gran robustez a la aplicación, reduciendo la necesidad de que el usuario configure manualmente las rutas de mensajes y señales, previniendo errores de escritura.

4.5. Sprint 5: Creación del Humanized Syntax Template Editor

Objetivo

Desarrollar un nuevo editor que abstraiga la complejidad sintáctica de las herramientas de validación finales, permitiendo al ingeniero diseñar *scripts* de pruebas a través de una sintaxis “humanizada” mediante instrucciones más legibles y contando con asistencia de menús guiados.

Actividades

- **Desarrollo de un nuevo editor:** Tomando como base el *Template Editor*, ampliando su estructura para soportar nuevas capas de lógica, entre las que se encuentran la gramática del DSL desarrollado.
- **Implementación del menú Humanized Syntax Shortcuts:** Desarrollo de un menú que proporciona el catálogo de instrucciones definidas en la gramática del DSL, de tal forma que, al seleccionar una instrucción, el sistema despliega un formulario en el que el usuario rellena las variables y la herramienta procede a inyectar automáticamente la instrucción humanizada con la estructura correcta en el editor.
- **Desarrollo del motor de validación (*Process Syntax*):** Implementación de una función de compilación en tiempo real, definida en la **Fase 4**. Esta rutina procede a:
    - **Primer paso: Renderizado Jinja2:** Se renderiza la plantilla con el motor Jinja2, capturando cualquier excepción derivada de variables no declaradas o errores de tipado, almacenando el resultado en memoria.
    - **Segundo paso: Procesamiento DSL:** Se aplica un procesamiento DSL al resultado almacenado en memoria, siguiendo la estricta gramática definida en el Lenguaje de Dominio Específico (DSL), capturando cualquier excepción derivada del procesamiento.
    - **Tercer paso: Previsualización de la plantilla:** Creación de un visor de solo lectura posterior al primer y segundo paso, que muestra el *script* resultante, ofreciendo al usuario la opción de guardarlo o descartarlo.
- **Integración del sistema de pre-visualización:** Comentado en el **Tercer paso** del desarrollo del motor de validación.

Prototipo generado

Un nuevo editor mejorado, que funciona como puente visual entre el diseño lógico del ingeniero y el motor de traducción subyacente. Permite al usuario construir lógicas de prueba complejas a través de una interfaz amigable, sin necesidad de conocer o memorizar la sintaxis final requerida.

Lecciones aprendidas

- **Curva de aprendizaje reducida:** Al generar código a través de menús interactivos y formularios guiados, se reduce significativamente la curva de aprendizaje del DSL y del lenguaje requerido, haciendo más fácil la creación de pruebas complejas.
- **Validación por fases:** El hecho de capturar cualquier excepción durante las etapas de renderizado Jinja2 y procesamiento DSL, previene errores silenciosos en la generación de *scripts* de prueba finales, ahorrando tiempo de depuración en fases posteriores al ciclo de pruebas.

4.6. Sprint 6: Configuración del Asistente Inteligente

Objetivo

Incorporar un modelo de Inteligencia Artificial Generativa (Gemini), como asistente durante el proceso de creación de plantillas, para proporcionar al ingeniero un soporte técnico, reduciendo la curva de aprendizaje del DSL y de Jinja2, sin comprometer las políticas de ciberseguridad industrial de Airbus Helicopters.

Actividades

- **Configuración del Gem de Gemini:** Para garantizar el cumplimiento de las normativas de seguridad, se descartó la integración directa mediante API dentro del código de la

herramienta. En su lugar, se optó por configurar un Gem (Asistente Inteligente) alojado en la plataforma de Gemini, al cual el usuario accede de forma paralela a la aplicación.

- **Entrenamiento del Gem:** Se instruyó al asistente con el diccionario de la gramática del DSL (desarrollado con textX), las reglas de Jinja2 y con ejemplos sintéticos de ficheros de parámetros y templates.
- **Definición del flujo de trabajo paralelo:** El usuario opera con la herramienta en su entorno local y de forma independiente, puede consultar al Gem para que le genere bloques de código, le autocomplete lógicas o le explique errores de compilación semántica.
- **Incorporación de protocolos de seguridad:** Al operar de forma desacoplada se elimina el riesgo de fuga de información por parte del asistente inteligente. Adicionalmente, el asistente fue instruido para trabajar con lógicas de programación abstractas y variables sintéticas, asegurando que el sistema no requiera información confidencial de la empresa.

Prototipo generado

Un Asistente Inteligente funcional y especializado, publicado de forma segura en el entorno corporativo de Gemini para Airbus Helicopters. Actúa como soporte y permite al usuario solicitar esqueletos de prueba complejos o resolver dudas de sintaxis en cuestión de segundos, acelerando la fase de diseño y manteniendo los datos reales aislados en el entorno local. Al implementarse bajo un plan empresarial, permite dar acceso a cualquier miembro del personal de Airbus Helicopters, pudiendo proporcionar el asistente a cualquier ingeniero de validación requerido.

Lecciones aprendidas

- **La importancia del contexto en los Asistentes de IA:** La inyección del contexto de la aplicación y del meta-modelo de textX como conocimiento del asistente, supuso una transformación de un asistente generalista en un especialista absoluto de la herramienta.
- **Seguridad integral mediante aislamiento:** Se demostró que operar con el Asistente en una plataforma paralela, permite aprovechar toda la potencia de la Inteligencia Artificial

en entornos altamente regulados, ya que se elimina de raíz el riesgo de exposición involuntaria de datos corporativos críticos.

Capítulo 5

Pruebas y validación

Una vez concluida la fase de desarrollo, resulta esencial validar la viabilidad y funcionalidad de la herramienta. Con este propósito, se ha diseñado un experimento práctico centrado en evaluar el nuevo sistema.

El objetivo principal de este experimento es validar el nuevo entorno de generación de pruebas. Se busca evaluar tanto el rendimiento funcional del sistema como la experiencia de usuario frente al uso del Lenguaje de Domino Específico, a los menús de asistencia interactivos y al asistente inteligente. Todo ello durante todo el flujo completo de generación de una casuística real de pruebas con datos sintéticos.

5.1. Diseño del experimento

Se han diseñado cinco tareas básicas, cubriendo un posible flujo de uso del sistema:

1. Configurar los directorios de ficheros de parámetros, plantillas y *scripts* de pruebas finales.
2. Crear un fichero de parámetros.
3. Generar una plantilla empleando menús interactivos.
4. Generar una plantilla mediante el asistente de IA.
5. Procesar la plantilla y validar el resultado generado.

Los participantes han sido observados mientras realizaban estas tareas. Se midieron los tiempos de ejecución, se registraron errores y se recopilaron opiniones finales. También se evaluó la tasa de éxito y el grado de satisfacción general con el sistema.

5.2. Metodología

Las pruebas se llevaron a cabo en sesiones individuales. A cada usuario se le explicó brevemente el funcionamiento del nuevo entorno de generación de pruebas, pero se minimizó la explicación técnica para poder evaluar la curva de aprendizaje natural de la herramienta. Además, se midió el tiempo empleado durante las cinco tareas descritas, registrando los obstáculos encontrados y, al finalizar, se recopilaron las opiniones del sistema y su nivel de satisfacción.

5.3. Descripción de los participantes

A continuación, se definen los perfiles de los participantes involucrados en la evaluación, pertenecientes al equipo de Airbus Helicopters. Estos representan el espectro de futuros usuarios finales del sistema de generación de pruebas desarrollado:

- **Usuario tipo A (Perfil Junior):** Usuarios de reciente incorporación, sin experiencia previa en la herramienta original ni familiaridad profunda con el lenguaje final de las plataformas de certificación.
- **Usuario tipo B (Perfil Senior):** Usuarios de validación veteranos, con experiencia previa en la herramienta original, pero sin experiencia con el Lenguaje de Dominio Específico (DSL) ni con asistentes de Inteligencia Artificial de Gemini.
- **Usuario tipo C (Perfil Experto):** Usuario experto en *testing*, familiarizado con la herramienta original y participante activo en la validación continua del proyecto, pero con experiencia limitada con la nueva herramienta.

5.4. Resultados cuantitativos

A continuación, se presentan los resultados obtenidos por los cinco participantes: dos usuarios junior, dos senior y uno experto, en cada una de las tareas planteadas. Se midió el tiempo en

minutos desde que el usuario iniciaba la tarea hasta que se completaba correctamente. Los datos que se muestran son la media por tipo de usuario.

Tarea	Usuario A	Usuario B	Usuario C
Configurar directorios	3min	1min	1min
Crear diccionario de datos	13min	10min	5min
Generar plantilla con menús	21min	15min	11min
Generar plantilla con asistente de IA	12min	9min	7min
Procesar y validar resultado	11min	5min	3min

Tabla 5.1: Tiempo medio por tarea

5.4.1. Éxito por tarea

- **Configurar directorios:** Todos los usuarios completaron esta tarea sin dificultad, demostrando que la interacción con el menú de configuración de directorios es intuitiva y funciona correctamente.
- **Crear diccionario de datos:** Nuevamente, todos los usuarios completaron esta tarea, asegurando la practicidad del editor de ficheros de parámetros, gracias a su similitud con aplicaciones como Excel, resultó asequible a usuarios sin experiencia previa.
- **Generar plantilla con menús:** Tres de los cinco usuarios lograron crear la plantilla utilizando los menús interactivos disponibles en el editor. Los usuarios tipo A experimentaron dificultades al no tener tanto conocimiento de la sintaxis Jinja2 y de las instrucciones disponibles para crear la lógica de pruebas, sugiriendo la necesidad de enlazar el editor con la documentación de Jinja2 y de las instrucciones, de manera que se tuviera acceso a su explicación, sus diferentes formatos de inserción y su funcionamiento.
- **Generar plantilla con asistente de IA:** Gracias al asistente de Inteligencia Artificial, los usuarios tipo A pudieron resolver las dudas mencionadas en el apartado anterior, lo que facilitó que los cinco participantes lograran completar la tarea correctamente.

- **Procesar y validar resultado:** Tres de los cinco participantes lograron completar la tarea exitosamente. Sin embargo, debido al conocimiento limitado de la herramienta de certificación final, los usuarios tipo A experimentaron dificultades para validar la lógica y poder certificar que la prueba era válida para el requisito definido, sugiriendo enlazar además de la documentación, ejemplos de lógicas de pruebas simples explicadas paso a paso, para aportar una introducción a usuarios sin experiencia.

5.4.2. Media de tiempo por tarea

- Configurar directorios: 1.7min
- Crear diccionario de datos: 9.3min
- Generar plantilla con menús: 15.7min
- Generar plantilla con asistente de IA: 9.3min
- Procesar y validar resultado: 6.3min

5.5. Resultados cualitativos y opiniones

Usuarios tipo A: Manifiestan dificultades con la definición de la lógica de pruebas y problemas con las primeras interacciones con los menús interactivos en el editor de templates. Consideran que se debe tener un conocimiento previo de las instrucciones para entender los distintos formatos de inserción y la utilidad de cada instrucción. Les gustaría tener acceso a una documentación de la sintaxis Jinja2 y de las instrucciones para tener un curso introductorio antes de utilizar la aplicación.

Usuarios tipo B: Les gusta la idea general, pero tuvieron algunos problemas con los formatos de inserción de instrucciones de los menús interactivos al no saber cuál tener que usar. Su feedback fue similar al de los usuarios tipo A, proponiendo un manual introductorio con las instrucciones definidas con la sintaxis del DSL y con sus diferentes formas de introducirlas.

Usuario tipo C: Experiencia fluida. Al tener un conocimiento general de la nueva herramienta, le resultó fácil e intuitivo llevar a cabo todas las tareas con soltura. Aun así, como mejora sugiere entrenar al asistente de Inteligencia Artificial con más casos para mejorar su rendimiento, de cara a asistir a los nuevos usuarios.

5.6. Limitaciones detectadas

- Dependencia del criterio del ingeniero. Aunque el asistente genera esqueletos de prueba sintácticamente correctos, se necesita el criterio del ingeniero de validación para la evaluación y corrección de la lógica de pruebas generada, no pudiendo automatizar el 100% de la tarea.
- Periodo de aclimatación inicial necesario. Para los usuarios nuevos, la interacción con los múltiples editores y los módulos de configuración puede resultar abrumadora los primeros minutos de uso.
- Necesidad de experiencia en la validación del resultado final. Como se evidenció con los usuarios tipo A, el desconocimiento en profundidad de la herramienta final de certificación de pruebas dificultó que los usuarios primerizos pudieran certificar visualmente que la lógica de pruebas cumple con el requisito especificado.
- Rigidez sintáctica del metalenguaje. A pesar de permitir definir la lógica de pruebas de una manera más sencilla e intuitiva, el analizador léxico no admite tolerancia a fallos, por lo que cualquier desviación gramatical o error tipográfico al definir una instrucción interrumpirá el proceso de traducción.
- Ausencia de un entorno de simulación. El ingeniero puede validar el *script* final generado y la lógica definida, pero para probar el funcionamiento real y certificar la prueba, sigue siendo necesario importar el *script* en la plataforma de certificación final para su ejecución.

5.7. Satisfacción general

Se preguntó a los usuarios su grado de satisfacción en una escala de 1 a 5 respecto a los siguientes aspectos:

Aspecto	A	B	C	Media por aspecto
Facilidad de uso	3	4	4	3.7
Interfaz visual	2	3	3	2.7
Utilidad de la sintaxis (DSL)	3	5	5	4.3
Soporte del asistente (Gemini)	5	4	4	4.3
Experiencia general	3	4	4	3.7
Media por usuario	3.2	4	4	**Media Usuarios = 3.7**

Tabla 5.2: Grado de satisfacción (1=muy baja, 5=muy alta)

Del análisis de la tabla de grado de satisfacción, se observa que los aspectos mejor valorados por los usuarios son la **utilidad de la sintaxis** (media de 4.3) y el **soporte del asistente** (media de 4.3), seguidos muy cerca de la **facilidad de uso** (media 3.7). Estos resultados sugieren que el sistema ha logrado proporcionar una experiencia satisfactoria, destacando la utilidad de definir instrucciones con el lenguaje humanizado en vez de utilizar el lenguaje final, y la capacidad de tener un asistente que ayude durante el proceso.

Por otro lado, el aspecto con menor puntuación es la interfaz visual, con una media de 2.7, siendo coherente con las limitaciones identificadas previamente respecto a la sobrecarga de información inicial.

No obstante, al observar la media global por usuario (3.7 sobre 5) y la valoración de la experiencia general (3.7 de 5), se confirma una experiencia positiva y una adopción exitosa de la herramienta.

5.8. Visualización de resultados

5.8.1. Tiempo medio por tarea

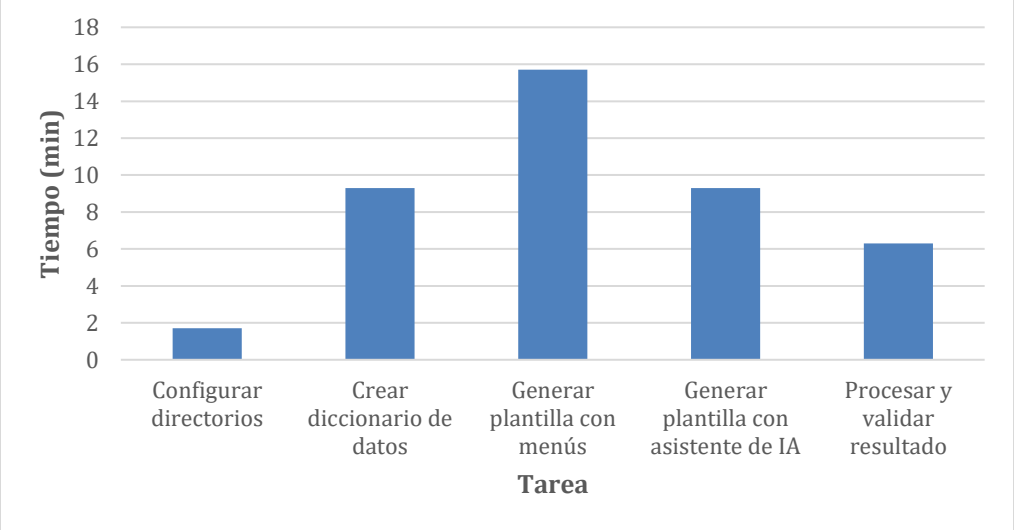


Figura 5.1: Tiempo medio de ejecución por tarea

5.8.2. Grado de satisfacción por aspecto

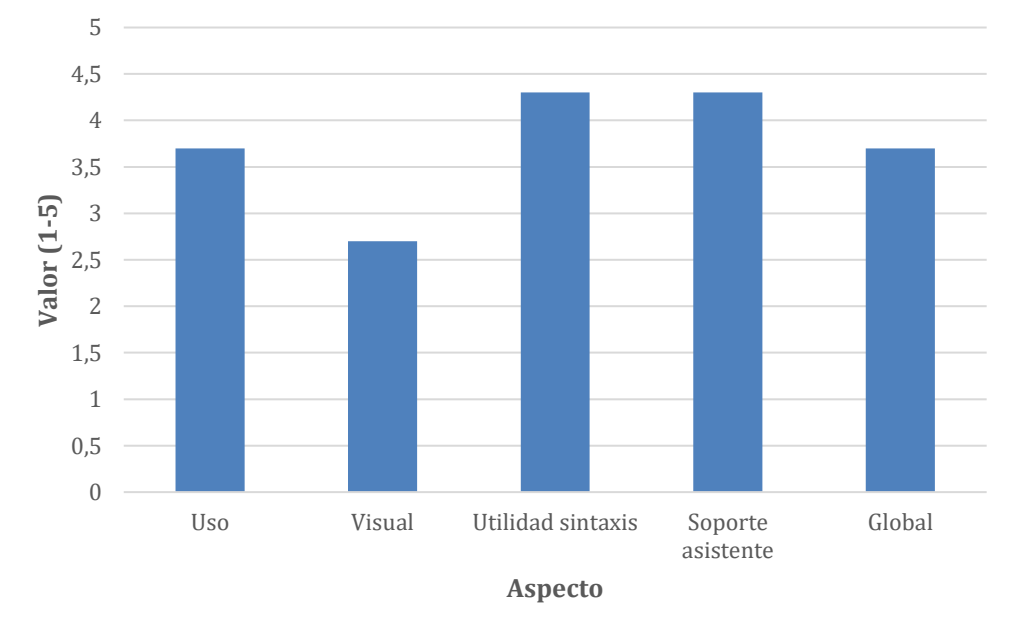


Figura 5.2: Promedio de satisfacción por aspecto

5.9. Conclusiones del experimento

El sistema ha demostrado ser funcional, demostrando ser capaz de reducir en gran medida los tiempos de desarrollo de las pruebas de certificación (acercándose a la franja de 1 hora por flujo completo de creación del test en producción), superando las limitaciones operativas de la aplicación inicial (con una franja de 2-3h por flujo de test).

La combinación del Lenguaje de Dominio Específico (DSL) junto con los menús interactivos ha logrado abstraer la complejidad técnica del lenguaje de la herramienta de certificación final, facilitando el proceso de diseño de pruebas. Asimismo, el experimento valida la hipótesis de que la Inteligencia Artificial, desplegada de forma segura y cumpliendo las normativas de seguridad, es un soporte que reduce la curva de aprendizaje, permitiendo que incluso ingenieros sin experiencia previa puedan familiarizarse con la herramienta en apenas una hora de uso.

Capítulo 6

Conclusiones

6.1. Resumen general

El presente Trabajo de Fin de Grado ha tenido como objetivo **dar respuesta a las limitaciones operativas detectadas en la herramienta base preexistente**. El resultado ha sido una solución funcional, robusta y escalable que integra tecnologías emergentes como el procesamiento de Lenguajes de Dominio Específico (DSL) y la Inteligencia Artificial Generativa, permitiendo al ingeniero de validación diseñar de manera natural, intuitiva y segura lógicas de prueba complejas.

Más allá de la funcionalidad concreta puesta en marcha, este proyecto representa un primer paso hacia un cambio de paradigma en el desarrollo de pruebas. Se ha logrado transicionar de una aplicación inicial limitada y con una elevada curva de aprendizaje, a una nueva generación que permite un desarrollo asistido. La combinación del motor de plantillas, el analizador léxico-sintáctico, la resolución dinámica de rutas y el soporte continuo del agente inteligente eliminan las barreras técnicas del estado inicial de la herramienta.

El sistema implementado no sólo permite abstraer la complejidad técnica del lenguaje final requerido por la herramienta de certificación de pruebas, sino que ofrece también un entorno con capacidades propias de un IDE (resaltado de sintaxis, autocompletado, validación previa a la compilación). En definitiva, el trabajo evidencia una mejora importante de la productividad, optimizando los tiempos del ciclo de certificación y minimizando el margen de error humano.

6.2. Esfuerzo y recursos dedicados

El desarrollo de este proyecto ha requerido una inversión significativa de tiempo, esfuerzo y recursos tanto técnicos como personales. A continuación, se presenta una estimación detallada del tiempo invertido por fases y tareas:

6.2.1. Distribución temporal por sprint

Sprint	Tareas principales	Horas Estimadas	Semanas Estimadas
Sprint 1	Análisis del sistema base y diseño de la nueva generación.	30 h	2 semanas
Sprint 2	Evolución del Frontend	30 h	1 semana
Sprint 3	Diseño e implementación del Lenguaje de Dominio Específico (DSL) con textX	40 h	2 semanas
Sprint 4	Reestructuración del Backend: Multi-flujo y procesamiento XML	40 h	2 semana
Sprint 5	Creación del Editor Avanzado (Humanized Syntax Template Editor)	40 h	1 semana
Sprint 6	Configuración e Integración del asistente inteligente (Gemini)	30 h	2 semanas
Total Desarrollo		**210 h**	**10 semanas**

Tabla 6.1: Distribución temporal por sprint

6.2.2. Tiempo de aprendizaje previo y formación autodidacta

Previo al inicio de la ejecución activa del proyecto se dedicaron aproximadamente 30 horas al estudio y experimentación con tecnologías clave como: textX, para la implementación del Lenguaje de Dominio Específico (DSL), y la tecnología Gem de Gemini, para el despliegue seguro del asistente de Inteligencia Artificial dentro del ecosistema empresarial corporativo.

6.2.3. Preparación de la memoria

La redacción, edición y organización de la memoria del proyecto requirió aproximadamente 70 horas, incluyendo la recopilación de referencias, generación de diagramas y revisión del formato académico. Durante este proceso, se emplearon de forma puntual herramientas de inteligencia artificial como soporte para la estructuración y revisión de estilo de fragmentos concretos del texto.

6.2.4. Estimación total de dedicación

El esfuerzo total invertido en el proyecto asciende a 310 horas, ejecutadas a lo largo de 16 semanas de trabajo. En la siguiente tabla se muestra la distribución temporal asignada a cada fase:

Fase del Proyecto	Dedicación Estimada (Horas)	Duración Estimada (Semanas)
Aprendizaje y pruebas previas	30	2
Tiempo de desarrollo técnico	210	10
Redacción de la memoria	70	4
Total estimado	**310 horas**	**16 semanas**

Tabla 6.2: Estimación total de horas dedicadas

6.3. Lecciones aprendidas y principales problemas resueltos

A lo largo de la evolución del proyecto, se han extraído una serie de aprendizajes fundamentales que no sólo han facilitado la consecución de los objetivos propuestos, sino que también han potenciado las competencias técnicas y organizativas del autor.

6.3.1. Experiencias documentadas

- **Uso estratégico de macros en plantillas:** Se aprendió que la implementación de macros dentro de las plantillas facilita la reutilización de patrones de código recurrentes, lo que permite optimizar en gran medida la mantenibilidad de las lógicas de prueba a largo plazo.
- **Resolución de ambigüedades mediante formatos de entrada flexibles:** Se constató que, debido a que una misma señal podía provenir de distintos equipos, dotar a cada instrucción del DSL de diferentes formatos de entrada permitía al usuario especificar rutas de forma flexible y resolver ambigüedades.
- **Reducción de curva de aprendizaje y menús interactivos:** La integración de menús interactivos para la inyección de código facilitó la adopción del sistema para ingenieros sin experiencia previa en la sintaxis de Jinja2 o del DSL.
- **Contextualización de la IA mediante casos sintéticos:** Se comprobó que el asistente inteligente era optimizable mediante su instrucción con casos sintéticos. Esto mejoró en gran medida su rendimiento.
- **Gestión de errores y tolerancia a fallos:** Se detectaron algunas incidencias durante la integración de la herramienta. La implementación de validaciones estrictas en las fases previas a la compilación evitó que los errores se propagasen hasta la ejecución final de las pruebas.
- **Equilibrio entre funcionalidad y experiencia de usuario:** Se aprendió que una buena experiencia no se basa únicamente en ofrecer muchas capacidades, sino en garantizar exponerlas a través de una interfaz interactiva e intuitiva.
- **Importancia del testeo continuo:** Realizar pruebas frecuentes y resolver errores en fases tempranas facilitó el mantenimiento, redujo problemas acumulativos y permitió iterar de forma ágil en cada sprint.

6.3.2. Valoración de funcionalidades implementadas

El resultado es un sistema funcional que ha demostrado que es posible transformar un flujo de trabajo manual y propenso a errores en un ecosistema asistido con capacidades propias de un IDE.

La separación de responsabilidades entre los distintos editores otorga a la herramienta una notable versatilidad, permitiendo a los ingenieros elegir el nivel de abstracción adecuado según la herramienta de certificación de pruebas final. Además, el procesamiento por lotes habilitado en la interfaz principal para procesar múltiples ficheros de parámetros de manera simultánea reduce de forma sustancial el tiempo dedicado a la generación de *scripts* de pruebas.

En definitiva, la unificación del motor de renderizado Jinja2 con una interfaz interactiva y la abstracción sintáctica del DSL han definido una herramienta segura, escalable y altamente eficiente para el departamento.

6.4. Impacto de asignaturas cursadas en la carrera

El desarrollo de este TFG ha sido posible gracias a la base de conocimientos adquirida en múltiples asignaturas, aunque también ha revelado algunas carencias en el plan formativo actual:

6.4.1. Conocimientos útiles aplicados

- **Lenguajes de programación y motores de plantillas:** Los conocimientos de programación en Python y el manejo de herramientas como Jinja2, aprendidos en asignaturas como Desarrollo de Aplicaciones Telemáticas (donde se empleaban *frameworks* como Django que mezclaban ambos elementos), fueron esenciales y fácilmente extrapolables al proyecto.
- **Estructuras de datos y arquitectura software:** Aunque el proyecto no requiere estructuras de datos complejas, los principios de diseño modular y patrones de organización han sido fundamentales.

- **Metodologías:** El enfoque basado en el desarrollo iterativo e incremental fue facilitado por la experiencia obtenida en prácticas y asignaturas de ingeniería en tecnologías de la telecomunicación.

6.4.2. Conocimientos que se han tenido que adquirir

- **Meta-programación y Gramáticas Formales:** El diseño y desarrollo de un Lenguaje de Dominio Específico desde cero no formaba parte del conjunto de conocimientos adquiridos en el grado, lo que requirió una fase de autoformación específica.
- **Ingeniería de *Prompts*:** La integración de LLMs como Gemini en entornos corporativos. Esto implicó comprender y especializar al asistente mediante ingeniería de *prompts*, aportándole las instrucciones y el rol a seguir, además del contexto de la gramática del DSL, garantizando el aislamiento de los datos y protegiendo la confidencialidad industrial.

6.5. Resumen de lo conseguido y comparación con otros sistemas

El sistema desarrollado constituye una solución que automatiza la generación de pruebas. A través del uso de editores avanzados, flujos de trabajo y un asistente de Inteligencia Artificial, se estructuró un entorno que permite al ingeniero diseñar lógicas de prueba de forma ágil e intuitiva, delegando en el sistema el trabajo de procesamiento para generar los *scripts* de pruebas finales.

Lejos de ser únicamente un prototipo académico, la solución se encuentra actualmente en producción en Airbus Helicopters. Su uso por parte de la compañía ya está dando un valor real, logrando una reducción directa en los tiempos de desarrollo de los ingenieros de validación y previniendo la introducción de errores en los ciclos de certificación. Además de los resultados positivos demostrados en Airbus Helicopters, la arquitectura de la herramienta ha sido diseñada con un alto grado de versatilidad, permitiendo que el sistema sea fácilmente integrable en los procesos de verificación y *testing* de cualquier otra organización o sector industrial. De este modo, la solución posee el potencial de aportar valor operativo significativo en múltiples entornos corporativos.

6.5.1. Funcionalidades logradas

- Implementación de un DSL propio que simplifica y humaniza la sintaxis de programación de pruebas.
- Desarrollo de editores con resaltado de sintaxis, menús interactivos y autocompletado en tiempo real.
- Integración segura de un asistente de IA Generativa para la generación de código y resolución de dudas.
- Resolución dinámica de rutas complejas mediante análisis de ficheros XML.
- Retrocompatibilidad con la versión anterior asegurando el mantenimiento de pruebas anteriores.

6.5.2. Hitos conseguidos

A nivel productivo, el nuevo flujo de trabajo ha generado resultados medibles de alto impacto en un corto periodo de tiempo:

- **Abstracción y clarificación del lenguaje:** La transición de la sintaxis rígida a un formato humanizado ha permitido a los ingenieros visualizar la lógica de pruebas con más claridad. Esta simplificación a nivel visual ha permitido identificar patrones de pruebas que resultaban muy difíciles de ver bajo el formato rígido anterior.
- **Optimización de código:** Gracias a la mejora en la legibilidad, y a la identificación de patrones repetitivos, se ha logrado fusionar cierta casuística de pruebas en plantillas reutilizables. Recientemente, utilizando la nueva herramienta, se han validado con éxito **30 pruebas de requisitos** (gestionadas a través de 30 ficheros de parámetros independientes), utilizando únicamente **3 plantillas base**. Anteriormente, cubrir este mismo alcance requería mantener y gestionar 6 plantillas distintas, lo que supone una reducción del 50% en el mantenimiento de código.

- **Escalabilidad y preparación futura:** Gracias a la optimización de código y la abstracción del lenguaje, se ha revelado que la herramienta es capaz de absorber un volumen de lógica de pruebas aún mayor, permitiendo cubrir más casuística en cada una de las plantillas base mencionadas en el punto anterior, aunque todavía no se han presentado requisitos definidos para probar esa lógica, la arquitectura ha quedado configurada y preparada para su validación en un futuro.
- **Soporte continuo del Asistente Inteligente:** El agente integrado está operando con un alto grado de precisión en sus respuestas, demostrando ser un pilar fundamental para guiar a los ingenieros en la redacción de nuevas pruebas, en la corrección de errores e incluso en la identificación de nuevos patrones, agilizando todo el ciclo de diseño.

6.5.3. Comparación con otros sistemas

Para dimensionar el valor de la herramienta desarrollada, compararemos los resultados obtenidos en la validación de los 30 requisitos mencionados frente a las metodologías alternativas tradicionales en la compañía:

- **Generación manual en la herramienta de certificación de pruebas final:** Para probar los 30 requisitos mencionados en la herramienta de pruebas final, se tendrían que haber configurado manualmente 30 scripts de prueba individuales. Este método tradicional habría conllevado un consumo de tiempo inviable para los plazos del departamento, una nula reutilización de código y una tasa de error humano extremadamente alta debido a la introducción manual de rutas y variables.
- **Uso de la aplicación inicial proporcionada por Airbus Helicopters:** Evaluando el escenario frente a la herramienta de automatización preexistente, la certificación de los 30 requisitos habría requerido aproximadamente el **doble de tiempo de desarrollo**. Así mismo, la ausencia de una capa de abstracción del lenguaje y la falta del soporte del asistente dificultaba enormemente la definición de las lógicas, requiriendo el doble de plantillas (6 frente a 3 actuales) para certificar las mismas pruebas.

6.6. Desarrollo temporal (Diagrama de GANTT resumido)

Tarea	Mar	Abr	May	Jun	Jul
Estudio de tecnologías					
Desarrollo por sprints					
Pruebas y validación					
Documentación					
Revisión y mejoras					

6.7. Trabajos futuros

- **Despliegue a nivel internacional:** Implementar la herramienta en los equipos de validación de ingeniería en Alemania con el fin de iniciar su explotación a nivel internacional.
- **Talleres formativos:** Realizar sesiones de entrenamiento con los equipos de validación para reducir la curva de aprendizaje y acelerar la adopción de la herramienta.
- **Ampliación y documentación de plantillas:** Migrar progresivamente el 100% de las pruebas creadas con la herramienta anterior al nuevo formato y generar un repositorio de documentación técnica de las plantillas creadas.
- **Validación en tiempo real:** Incorporar un sistema de validación sintáctica dentro del editor para detectar errores sintácticos o variables no declaradas, reduciendo el tiempo dedicado a corrección de errores.
- **Mejoras en la interfaz de usuario:** Rediseñar ciertos aspectos de la interfaz para reducir la carga de información inicial detectada durante la validación empírica.
- **Expansión de la gramática:** Añadir progresivamente nuevas instrucciones al DSL para cubrir lógicas de test más complejas.

- **Entorno de simulación local:** Implementar una conexión con la herramienta de certificación de pruebas final para probar el funcionamiento de los *scripts* generados sin necesidad de exportarlos previamente a la plataforma de certificación final.

6.8. Materiales adicionales

Para complementar la documentación técnica de esta memoria, se ha habilitado una página web del proyecto que centraliza el acceso a los recursos asociados a la herramienta[3].

En este portal se encuentra disponible una demostración en video que ilustra el funcionamiento y flujo de trabajo de la aplicación. Asimismo, la página proporciona acceso al repositorio[4] con el código de la herramienta y a la versión digital de este documento.

[3] URL del Trabajo Fin de Grado: https://daviidmartnez03.github.io/tfg-site/
[4] Repositorio del código: https://github.com/daviidmartnez03/Test-Procedure-Creator

Bibliografía

[1] Python Software Foundation, "tkinter — Python interface to Tcl/Tk," Python 3 Documentation. [En línea]. Disponible en: https://docs.python.org/3/library/tkinter.html.

[2] T. Schimansky, "CustomTkinter Documentation," CustomTkinter. [En línea]. Disponible en: https://customtkinter.tomschimansky.com/documentation/.

[3] "tksheet," PyPI. [En línea]. Disponible en: https://pypi.org/project/tksheet/.

[4] "Quickstart," Pygments. [En línea]. Disponible en: https://pygments.org/docs/quickstart/.

[5] "Introduction," Jinja Documentation. [En línea]. Disponible en: https://jinja.palletsprojects.com/en/stable/intro/.

[6] M. Mernik, J. Heering, y A. M. Sloane, "When and how to develop domain-specific languages," ACM Computing Surveys (CSUR), vol. 37, no. 4, pp. 316–344, Dic. 2005. doi: 10.1145/1118890.1118892.

[7] Dejanović et al., "textX - A Python tool for Domain-Specific Languages," textX. [En línea]. Disponible en: https://textx.github.io/textX/.

[8] Python Software Foundation, "xml.etree.ElementTree — The ElementTree XML API," Python 3 Documentation. [En línea]. Disponible en: https://docs.python.org/3/library/xml.etree.elementtree.html.

[9] M. Imran y N. Almusharraf, "Google Gemini as a next generation AI educational tool: a review of emerging educational technology," *Smart Learning Environments*, vol. 11, nº 1, art. no. 22, dic. 2024, doi: 10.1186/s40561-024-00310-z.

[10] D. Rubio, "Jinja Templates in Django," en *Beginning Django*, Berkeley, CA, EE. UU.: Apress, 2017. DOI: 10.1007/978-1-4842-2787-9_4.

[11] I. Dejanović y M. Dejanović, "Developing a Domain-Specific Language for Cognitive Tests Using TextX," en *2025 29th International Conference on Information Technology (IT)*, Zabljak, Montenegro, feb. 2025. Disponible en: https://ieeexplore.ieee.org/document/10930310

[12] T. M. King, J. Arbon, D. Santiago, D. Adamo, W. Chin, y R. Shanmugam, "AI for Testing Today and Tomorrow: Industry Perspectives," en *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*, Newark, CA, EE. UU., abr. 2019. Disponible en: https://ieeexplore.ieee.org/document/8718229

[13] Meng, B., et al. (2019). "Automated Test Generation for DO-178C Compliance". En 2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC). San Diego, CA, USA: IEEE. doi: 10.1109/DASC43569.2019.9081726.