Autor

Jordi André Ramírez Vecino

Dirigido por

Marc Sánchez Artigas

TRABAJO DE FIN DE GRADO

Evaluación del rendimiento de PyScript y Pyodide para la ejecución de flujos de trabajo en python al navegador

GRADO DE INGENIERÍA INFORMÁTICA

UNIVERSITAT ROVIRA I VIRGILI

Tarragona, Tarragona

2025

Abstract

In the current context of web development, mainly dominated by JavaScript, emerging technologies such as PyScript and Pyodide have emerged, introducing new possibilities for executing Python code directly in the browser. The main objective of this thesis is to evaluate the viability and performance of these technologies in comparison with JavaScript, considered the standard in web environments.

On the one hand, JavaScript has been for decades the language par excellence in client-side development, and the foundation on which numerous applications, websites and frameworks ecosystems have been built. On the other hand, Python, traditionally used in server-side development, data analysis and artificial intelligence, is beginning to expand to the client side thanks to tools such as Pyodide, a port of CPython compiled to WebAssembly that allows it to run in browsers. This initiative is joined by PyScript, a tool that simplifies the integration of Python in web environments through customized HTML components or files with a .py extension, which facilitates the development of Python applications directly in the browser.

To carry out the comparison, a series of specific tests have been designed and implemented to evaluate key aspects such as execution time, memory consumption and integration with external libraries. These tests range from numerical calculations

Resum

En el context actual del desenvolupament web, dominat principalment per JavaScript, han sorgit tecnologies emergents com PyScript i Pyodide, que introdueixen noves possibilitats per a l'execució de codi Python directament en el navegador. Aquest Treball de Fi de Grau té com a objectiu principal avaluar la viabilitat i el rendiment d'aquestes tecnologies en comparació amb JavaScript, considerat l'estàndard en entorns web.

D'una banda, JavaScript ha estat durant dècades el llenguatge per excel·lència en el desenvolupament del costat del client, i la base sobre la qual s'han construït nombroses aplicacions, llocs web i ecosistemes de frameworks. D'altra banda, Python, tradicionalment emprat en el desenvolupament del costat del servidor, en anàlisi de dades i en intel·ligència artificial, comença a expandir-se a l'àmbit del client gràcies a eines com Pyodide, un port de CPython compilada a WebAssembly que permet la seva execució en navegadors. A aquesta iniciativa se suma PyScript, una eina que simplifica la integració de Python en entorns web mitjançant components HTML personalitzats o arxius amb extensió .py, la qual cosa facilita el desenvolupament d'aplicacions en Python directament en el navegador.

Per a dur a terme la comparativa, s'han dissenyat i implementat una sèrie de proves específiques que permeten avaluar aspectes clau com el temps d'execució, el consum de memòria i la integració amb biblioteques externes. Aquestes proves inclouen des de càlculs numèrics intensius i visualització de dades, fins a tasques relacionades amb intel·ligència artificial i la gestió de múltiples sol·licituds concurrents.

Resumen

En el contexto actual del desarrollo web, dominado principalmente por JavaScript, han surgido tecnologías emergentes como PyScript y Pyodide, que introducen nuevas posibilidades para la ejecución de código Python directamente en el navegador. Este Trabajo de Fin de Grado tiene como objetivo principal evaluar la viabilidad y el rendimiento de estas tecnologías en comparación con JavaScript, considerado el estándar en entornos web.

Por un lado, JavaScript ha sido durante décadas el lenguaje por excelencia en el desarrollo del lado del cliente, y la base sobre la que se han construido numerosas aplicaciones, sitios web y ecosistemas de frameworks. Por otra parte, Python, tradicionalmente empleado en el desarrollo del lado del servidor, en análisis de datos y en inteligencia artificial, comienza a expandirse al ámbito del cliente gracias a herramientas como Pyodide, un puerto de CPython compilada a WebAssembly que permite su ejecución en navegadores. A esta iniciativa se suma PyScript, una herramienta que simplifica la integración de Python en entornos web mediante componentes HTML personalizados o archivos con extensión .py, lo que facilita el desarrollo de aplicaciones en Python directamente en el navegador.

Para llevar a cabo la comparativa, se han diseñado e implementado una serie de pruebas específicas que permiten evaluar aspectos clave como el tiempo de ejecución, el consumo de memoria y la integración con bibliotecas externas. Estas pruebas incluyen desde cálculos numéricos intensivos y visualización de datos, hasta tareas relacionadas con inteligencia artificial y la gestión de múltiples solicitudes concurrentes.

Índice

Índice de Tablas

Índice de código

Índice de Figuras

Índice de observaciones

1. Introducción

El desarrollo web ha evolucionado de forma constante, impulsado por nuevos lenguajes y tecnologías que buscan simplificar y optimizar la creación de aplicaciones. A pesar de esta diversidad de herramientas, JavaScript [1] se ha consolidado como el lenguaje de programación dominante en el desarrollo web, gracias a su versatilidad y su integración nativa con los navegadores. Esto ha generado un ecosistema robusto con herramientas como React [2], Vue [3] y Angular [4], que han facilitado la creación de aplicaciones dinámicas e interactivas.

En este contexto, Python [5], ampliamente utilizado en ciencia de datos, automatización e inteligencia artificial, comienza a expandirse hacia el desarrollo web gracias a tecnologías como PyScript [6] y Pyodide [7] permitiendo ejecutar código Python directamente en el navegador utilizando WebAssembly (Wasm) [8], una tecnología que permite ejecutar lenguajes como Python, C [9] o C++ [10] en la web de forma eficiente. Esta innovación ofrece nuevas oportunidades para los desarrolladores que prefieren trabajar con Python sin abandonar el entorno del navegador, pero también plantea dudas sobre su viabilidad frente al ecosistema establecido de JavaScript.

Esta situación genera preguntas clave como ¿Pueden PyScript y Pyodide ofrecer un rendimiento comparable al de JavaScript en el navegador? ¿Hasta qué punto pueden integrarse con otras librerías y manejar tareas complejas como visualización, cálculos intensivos o concurrencia? ¿Es Python una alternativa real para el desarrollo del lado del cliente, o su uso sigue limitado a nichos concretos?

Este trabajo busca responder a estas cuestiones mediante un análisis comparativo riguroso. A través de una serie de pruebas y benchmarks [11], se evaluarán aspectos como el rendimiento, el uso de memoria y la capacidad de integración con bibliotecas. Además, se estudiará cómo estas tecnologías afectan a la experiencia de desarrollo, permitiendo ofrecer una visión crítica sobre sus ventajas, limitaciones y potencial de adopción en distintos escenarios del desarrollo web moderno.

El enlace al proyecto es público, se puede visitar mediante este enlace.

1.1. Motivación

La motivación principal de este trabajo surge del interés personal por comprender en profundidad nuevas tecnologías que permiten ejecutar otros lenguajes, no únicamente Python, directamente en el navegador, una idea que, hasta hace poco, parecía reservada exclusivamente a lenguajes como JavaScript. Herramientas como PyScript y Pyodide representan un enfoque novedoso que desafía el paradigma tradicional del desarrollo web, y su estudio ofrece una oportunidad única para analizar no solo su rendimiento, sino también su aplicabilidad real en distintos escenarios. Desde un punto de vista formativo, este proyecto permite aprender el funcionamiento interno de estas tecnologías, identificar sus puntos fuertes y sus limitaciones, y evaluar en qué contextos pueden llegar a ser alternativas viables a JavaScript.

Además, este trabajo responde a la inquietud de explorar caminos distintos en el desarrollo web, ampliando el horizonte más allá de las soluciones convencionales. En este sentido, PyScript no solo es interesante como tecnología, sino también como propuesta conceptual: permitir a desarrolladores acostumbrados al ecosistema Python trasladar sus conocimientos al navegador sin necesidad de cambiar de lenguaje. Esto plantea interrogantes que merecen ser estudiados, como hasta qué punto es práctica esta transición o si realmente se consigue una experiencia de desarrollo comparable. Con esto, podemos lograr beneficios en el mejor de los casos, como más facilidad para desarrollar en caso de estar acostumbrado a Python.

Por otro lado, este proyecto también busca ser una oportunidad de aprendizaje en cuanto a la realización de estudios comparativos rigurosos, mediante la creación de pruebas controladas (benchmarks) que permitan analizar con criterio técnico el comportamiento de cada tecnología. Asimismo, se ha aprovechado el desarrollo del trabajo para aprender y aplicar Astro [12], un framework [13] moderno orientado a la creación de sitios web rápidos y modulares, que ha sido empleado como base para presentar los resultados obtenidos. Todo ello hace que este proyecto no solo tenga una dimensión tecnológica, sino también investigadora y formativa.

1.2. Objetivos

1.2.1. Técnicos

El principal objetivo de este Trabajo de Fin de Grado es realizar un análisis comparativo entre PyScript, Pyodide y JavaScript en el contexto del desarrollo web. Para ello, se pretende comparar su rendimiento en términos de tiempo de ejecución, uso de memoria y eficiencia computacional en función de la prueba, así como evaluar su capacidad de integración con bibliotecas externas del ecosistema científico de Python como NumPy **[14]**, pandas **[15]** y Matplotlib **[16]**. El estudio incluye el diseño y la implementación de una serie de pruebas representativas de tareas comunes en el desarrollo web, tales como cálculos numéricos intensivos, visualización de datos, manipulación del DOM **[17]** y entre otras A partir de los resultados obtenidos, se identificarán las ventajas y limitaciones de cada tecnología y se elaborarán recomendaciones sobre los escenarios en los que resulte

más conveniente utilizar PyScript frente a JavaScript. Finalmente, se desarrollará una plataforma web con Astro que facilite la ejecución, la visualización y la comparación de los benchmarks de forma accesible y reproducible.

El estudio también tiene como objetivo analizar la facilidad de uso de estas tecnologías, sus ventajas y desventajas respecto a la otra herramienta y su potencial adopción en escenarios reales. A partir de los resultados obtenidos, se pretende identificar las respuestas a estos planteamientos de cada herramienta, y ofrecer recomendaciones sobre cuándo puede ser más adecuado utilizar PyScript frente a JavaScript. Finalmente, este proyecto contempla la exploración de herramientas modernas como Astro para estructurar y presentar los benchmarks desarrollados, facilitando así su interpretación y reutilización en futuros trabajos.

1.2.2. Formativos

Este proyecto también tiene como finalidad el desarrollo de competencias personales y académicas. Uno de los objetivos principales es profundizar en el uso de Python, especialmente en un entorno poco habitual como es el navegador web, explorando tecnologías como PyScript y Pyodide, y comprendiendo su arquitectura y funcionamiento interno. A su vez, se busca adquirir experiencia práctica en la planificación y ejecución de estudios comparativos de rendimiento, aprendiendo a diseñar benchmarks rigurosos y a analizar sus resultados con criterio técnico.

Además, se pretende aprender a trabajar con Astro, un framework moderno para el desarrollo web, con el objetivo de construir una interfaz clara e interactiva para visualizar y ejecutar las pruebas desarrolladas. También se abordará el aprendizaje claramente de JavaScript, tanto para su comparación como para entender su papel en el desarrollo web moderno. Por último, este trabajo contribuirá a reforzar habilidades clave como la documentación técnica, la organización de proyectos y la presentación clara de resultados, tanto de forma escrita como visual.

2. Planificación y metodología

2.1. Planificación temporal

La planificación temporal de este Trabajo de Fin de Grado se ha estructurado a lo largo de todo el cuatrimestre, abarcando desde finales de enero hasta la primera semana de junio, fecha prevista para la entrega. Para visualizar esta planificación, se ha elaborado un diagrama de Gantt [18] , el cual permite representar de forma clara y ordenada las diferentes tareas que componen el proyecto, así como su duración y solapamiento en el tiempo.

El diagrama de Gantt está dividido en cuatro grandes fases que representan los bloques principales del trabajo. Estas fases están formadas a su vez por subapartados que detallan tareas específicas. A continuación, se describe brevemente cada una de estas fases principales:

1. Definición y diseño: Esta fase inicial comprende las tareas necesarias para sentar las bases del proyecto. Incluye el estudio preliminar del estado del arte, la identificación de los ámbitos temáticos a evaluar, el diseño general de los benchmarks que se utilizarán en la comparativa y la definición de las métricas que permitirán medir el rendimiento de las tecnologías analizadas.

2. Desarrollo local: Durante esta fase se desarrolla localmente el conjunto de pruebas correspondientes a cada ámbito temático identificado previamente. Cada subapartado representa el desarrollo y validación de una prueba específica en ambos lenguajes (PyScript/Python y JavaScript), asegurando que las tareas son comparables y funcionales.

3. Integración y desarrollo web con Astro: Una vez completadas las pruebas locales, esta fase se centra en la implementación del entorno web. Se diseña e implementa una interfaz en Astro que permite ejecutar y visualizar los resultados de las pruebas directamente desde el navegador. Esta etapa también incluye la integración del código adaptado al formato del proyecto Astro.

4. Análisis final: En la etapa final del proyecto, se realiza un análisis comparativo de los resultados obtenidos, se extraen conclusiones sobre la viabilidad y el rendimiento de cada tecnología, y se redacta y revisa el documento final del TFG.

El diagrama de Gantt ha sido representado con un sistema de colores para facilitar la interpretación visual:

- Color morado: Indica la duración de cada fase principal, proporcionando una vista global de los grandes bloques del proyecto.
- Color azul: Representa las tareas específicas (por ejemplo, 2.1, 3.2, 4.3), detallando el tiempo asignado a cada una de ellas dentro del marco general de su fase correspondiente.

Descripción	Inicio	Duración	Fin
1.Definición y diseño	20/1/25	21	10/2/25
1. 1 Estudio preliminar	20/1/25	7	27/1/25
1. 2 Identificacion de ambitos	27/1/25	7	03/2/25
1. 3 Diseño de benchmarks	03/2/25	7	10/2/25
1. 4 Definicion de metricas de cada ambito	03/2/25	7	10/2/25
2.Desarrollo local	10/2/25	56	07/4/25
2. 1 Desarrollo de pruebas para el 1r ambito	10/2/25	14	24/2/25
2. 2 Desarrollo de pruebas para el 2ndo ambito	24/2/25	14	10/3/25
2. 3 Desarrollo de pruebas para el 3r ambito	10/3/25	7	17/3/25
2. 4 Desarrollo de pruebas para el 4to ambito	17/3/25	7	24/3/25
2. 5 Desarrollo de pruebas para el 5to ambito	24/3/25	7	31/3/25
2. 6 Desarrollo de pruebas para el 6to ambito	31/3/25	7	07/4/25
3.Integración y desarrollo web con Astro	7/4/25	35	12/5/25
3. 1 Diseño del frontend con Astro	7/4/25	7	14/4/25
3. 2 Implementacion de las diferentes vistas y componentes de la web	14/4/25	7	21/4/25
3. 3 Integracion y adaptacion del codigo para cada prueba	21/4/25	14	05/5/25
3. 4 Retoques finales del codigo	05/05/25	7	12/5/25
4.Analisis final	12/5/25	24	05/6/25
4. 1 Analisis comparativo en base a resultados	12/5/25	7	19/5/25
4. 2 Conclusiones	19/5/25	3	22/5/25
4. 3 Redaccion y revision del TFG	22/5/25	14	05/6/25

Figura 1. Diagrama de Gantt 1. Primera parte del diagrama de Gantt.

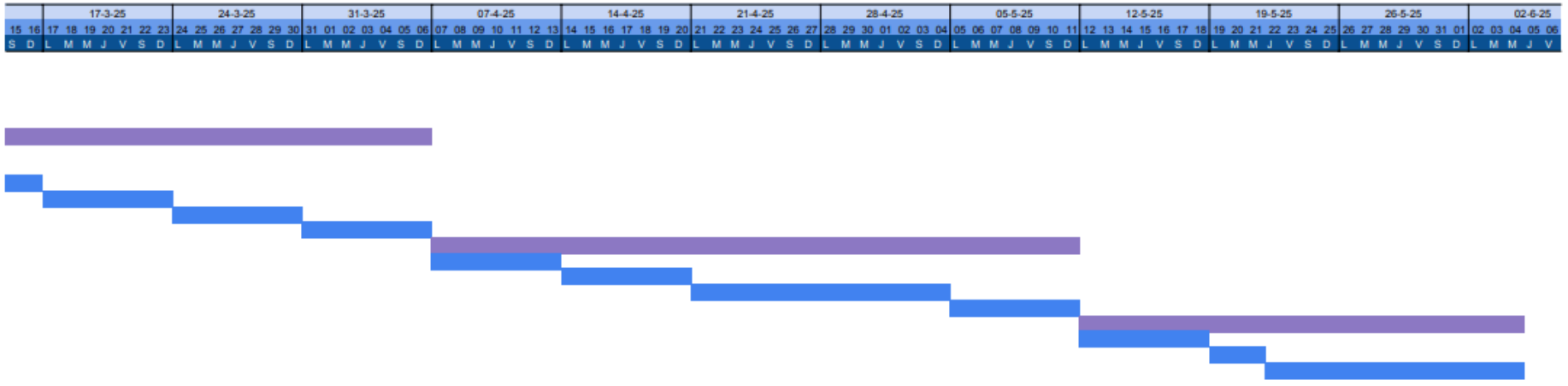

Figura 2. Diagrama de Gantt 2. Segunda parte del diagrama de Gantt.

2.2. Metodología general de desarrollo e investigación

La metodología seguida en este Trabajo de Fin de Grado se ha estructurado en varias etapas que permiten abordar de manera sistemática la comparación entre PyScript y JavaScript. El primer paso consistió en definir los ámbitos o temas más relevantes y representativos del desarrollo web con enfoque científico, seleccionando áreas como la inteligencia artificial, la criptografía, etc. La elección de estos ámbitos no solo responde a su importancia en la computación moderna, sino también a su creciente aplicación dentro del navegador, lo que los convierte en candidatos ideales para evaluar la viabilidad de tecnologías emergentes como PyScript.

A partir de estos ámbitos se diseñaron una serie de pruebas específicas para cada uno, estableciendo una jerarquía clara: el ámbito o tema define la categoría general (por ejemplo, cálculos matemáticos), la prueba concreta la tarea que se va a realizar (por ejemplo, multiplicación de matrices), y la versión representa una implementación particular de esa prueba, con ligeras variaciones en el enfoque o los recursos utilizados. Esta estructura jerárquica permite organizar y comparar de forma sistemática distintos enfoques de resolución de un mismo problema. Las versiones son el nivel más técnico de esta jerarquía, ya que contienen el código fuente concreto que se ejecuta. Y a nivel de métricas en los benchmarks, van variando así mismo como la estructura de código para tener diferentes puntos de vista.

Cada prueba fue desarrollada inicialmente en entorno local, tanto en su variante Python como en su variante JavaScript, asegurando que cada versión fuera funcional y comparable en términos de lógica y objetivos. Por ejemplo, en el caso de la multiplicación de matrices, se implementaron dos versiones: una utilizando estructuras de datos nativas y otra con estructuras optimizadas.

Una vez completada esta fase, se procedió al desarrollo del entorno web utilizando el framework Astro. En esta etapa se diseñó una interfaz capaz de integrar todas las pruebas previamente implementadas, permitiendo su ejecución y visualización directamente desde el navegador. El código desarrollado localmente fue adaptado y estructurado dentro del proyecto Astro, de manera que cada prueba pudiera ejecutarse tanto de forma aislada como desde el entorno web completo. Esto garantiza una experiencia de usuario unificada y facilita la comparación directa entre las distintas tecnologías en un contexto realista.

Es importante destacar que, en la organización final del proyecto, el repositorio queda estructurado en dos carpetas principales: una carpeta de benchmarks, que contiene las versiones locales de cada prueba acompañadas de un archivo index básico para su ejecución independiente; y una segunda carpeta que corresponde al proyecto desarrollado en Astro, el cual incluye tanto el código de la página web como las pruebas adaptadas específicamente a su estructura interna. Esta separación permite al usuario elegir entre ejecutar los benchmarks de forma autónoma o mediante la interfaz web, según su decisión.

Por otro lado, nuestro proyecto se centra principalmente en el uso de PyScript para comparar su rendimiento con JavaScript, por ello, se realizó el siguiente diagrama de flujo para facilitar el entendimiento del desarrollo del flujo de la ejecución de un código PyScript, en especial, enfocado a nuestro proyecto, ya sea en la parte del Framework de Astro, o en la parte de local.

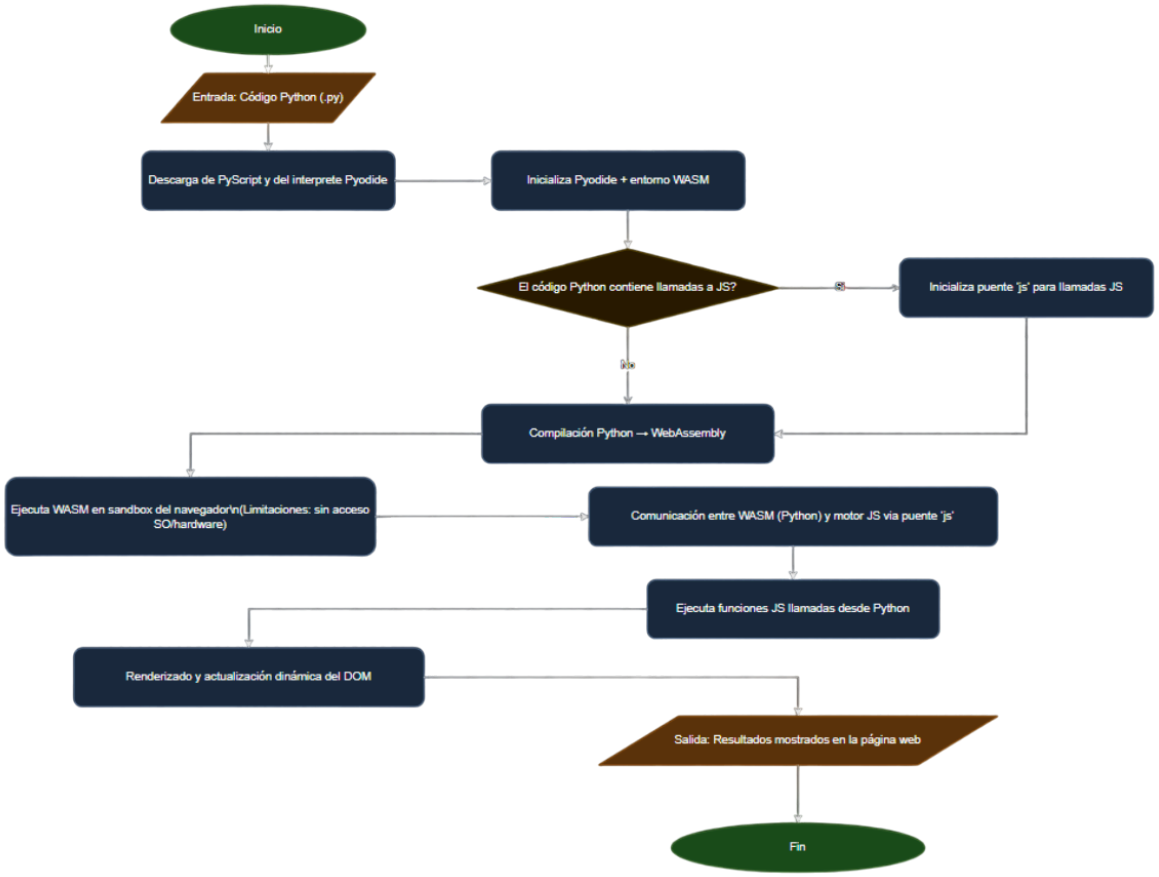

Figura 3. Diagrama de flujo de PyScript..

En este diagrama de flujo se describen los pasos que sigue el navegador para ejecutar código Python utilizando PyScript. A continuación, explicamos el proceso en detalle. Más adelante, en el Marco Teórico, se retomará este flujo con una explicación más profunda sobre las tecnologías implicadas.

1. El usuario escribe su código en Python, que puede incluirse de dos formas:
    a. Mediante una etiqueta *<py-script>*, que permite incrustar código directamente en el HTML.
    b. O bien, como se hace en este proyecto para mayor organización, incluyendo archivos .py externos mediante una etiqueta *<script type="py" src="archivo.py">*.
    c. En caso de que se necesite una configuración adicional de PyScript (como paquetes o librerías), se agrega un atributo *config="archivo.json"* o *config="archivo.toml"* que PyScript interpretará para personalizar el entorno.

2. Para que PyScript funcione correctamente, se deben incluir sus recursos en el HTML. Esto se hace con una etiqueta <*link rel="stylesheet"*> y un <*script src="..."*> apuntando al CDN oficial de PyScript.

   Cuando se carga la página, el navegador hace peticiones HTTP para descargar:

   a. PyScript
   b. Pyodide

3. Una vez descargado, el navegador carga el archivo .wasm de Pyodide, lo que inicializa una instancia de CPython dentro del entorno sandboxed del navegador. Este sandbox aísla completamente el entorno Python del sistema operativo del usuario, impidiendo el acceso directo al sistema de archivos, red local o hardware.

   Este entorno está completamente aislado del sistema operativo del usuario por razones de seguridad, siguiendo las restricciones del sandbox del navegador.

4. Pyodide analiza si el código Python contiene llamadas a objetos o funciones del entorno JavaScript:
   a. Si las hay, se crean automáticamente proxies que permiten al código Python interactuar con objetos globales de JavaScript (como document, window, console.log) a través del objeto especial js.
   b. Si no, el código Python se ejecuta directamente

5. Pyodide compila el código Python a bytecode de CPython (el mismo formato intermedio usado por un intérprete tradicional). Este bytecode es interpretado por el propio intérprete CPython embebido en Pyodide, el cual ya ha sido compilado previamente a WebAssembly y ejecuta eficientemente las instrucciones dentro del navegador.

6. El código Python se ejecuta dentro del entorno WebAssembly sandbox, lo que implica:
   a. Sin acceso a archivos locales reales ni a hardware.
   b. Cualquier operación que interactúe con el entorno web (DOM, red, eventos) debe realizarse a través del puente JavaScript.

7. A medida que se ejecuta el código Python, pueden suceder varias acciones:
   a. Llamadas a funciones JavaScript desde Python usando js.funcionJS(). Estas llamadas se traducen y ejecutan en el motor JS del navegador.
   b.
   c. Mutaciones en el DOM, ya sea mediante bibliotecas como PyScript UI o directamente manipulando elementos del árbol DOM desde Python.

8. Finalmente, se retornan los resultados al contexto JS y se finaliza la ejecución del código Python.

2.3. Metodología para los benchmarks

La definición de los ámbitos de los benchmarks se fundamentó en la identificación de áreas clave donde el rendimiento en el navegador resulta crítico tanto en el desarrollo web moderno como en aplicaciones de cómputo científico. Para ello, se realizó un análisis exploratorio con el objetivo de seleccionar casos representativos y relevantes que permitieran comparar el comportamiento JavaScript y PyScript.

Dicho análisis tuvo en cuenta los siguientes criterios:

- Relevancia en el desarrollo web actual
    - Se consideraron tareas comunes y desafiantes que aparecen habitualmente en aplicaciones web reales, incluyendo interacción con el usuario, procesamiento de datos en el navegador, generación de gráficos y manejo de múltiples solicitudes.
- Importancia en aplicaciones científicas y técnicas
    - Se seleccionaron áreas donde el uso de bibliotecas numéricas, análisis de datos, etc. Tienen un peso significativo.

- Capacidad de implementación en distintas tecnologías
    - Se priorizaron pruebas que pudieran implementarse de manera equivalente en JavaScript, WebAssembly y PyScript, permitiendo una comparación justa entre tecnologías bajo un mismo escenario funcional.

Además, se consideraron las limitaciones técnicas propias de cada tecnología, especialmente en el caso de WebAssembly y PyScript. A pesar de permitir la ejecución de código nativo en el navegador, existen restricciones como la compatibilidad limitada con ciertas bibliotecas de Python o el acceso restringido a características del sistema operativo.

En base a esto, se definieron los siguientes Benchmarks, de los cuales saldrán pruebas referentes a esos temas y tendrán sus versiones correspondientes:

- 1. Cálculos matemáticos intensivos
    - Pruebas que cubrirán típicas operaciones matemáticas de alto costo que se podrían usar comúnmente en un ámbito científico.
- 2. Procesamiento de grandes volúmenes de datos
    - Se definirán grandes conjuntos de volúmenes para simular típicos casos de datos de gran tamaño sobre los cuales se quiere trabajar sobre ellos.
- 3. Carga y representación de gráficos complejos
    - Se definirá un conjunto de datos para a continuación, realizar un gráfico sobre ellos.
- 4. Manejo de múltiples solicitudes concurrentes
    - Pruebas que analizaron el rendimiento del envío y respuesta de las peticiones a un servidor.

- 5. Cálculo y verificación de integridad en datos sensibles
    - Se realizarán pruebas que abarquen funciones o procedimientos criptográficos para simular la eficiencia de las librerías de integridad de datos.
- 6. Reconocimiento de patrones y clasificación de datos
    - Usar modelos y entrenarlos para ver qué tan eficientes son en ambas tecnologías.

Cada uno de estos ámbitos contiene diferentes pruebas que abordan el problema desde múltiples enfoques. Para lograr un análisis más completo, se desarrollaron versiones alternativas de cada prueba, explorando variaciones como:

- Uso o no de bibliotecas externas
- Ejecución con o sin Web Workers
- Implementaciones optimizadas frente a versiones más simples

3. Marco teórico

Para llevar a cabo este Trabajo de Fin de Grado, es esencial entender bien las tecnologías que vamos a analizar y el contexto en el que se usan en el desarrollo web actual. En esta sección, vamos a explicar de manera clara y sencilla los lenguajes, herramientas y conceptos clave que forman parte de este estudio.

Comenzaremos hablando del desarrollo web en general y después de JavaScript, el lenguaje que ha sido el rey del desarrollo web durante años. Veremos sus características principales, cómo funciona de forma nativa en los navegadores y por qué es tan importante para crear aplicaciones web interactivas y dinámicas.

Posteriormente, nos adentraremos en WebAssembly (Wasm), una tecnología que ha revolucionado la forma en que podemos usar lenguajes de programación como Python directamente en el navegador. Gracias a WebAssembly, han surgido herramientas como PyScript y Pyodide, que permiten ejecutar código Python en la web sin necesidad de un servidor backend [19]. En esta sección, explicaremos cómo funcionan estas herramientas, para qué sirven y qué bibliotecas científicas son compatibles con ellas.

Por último, hablaremos de las bibliotecas y conceptos que son clave para este estudio, como NumPy, pandas y Matplotlib, que usaremos en las pruebas de rendimiento. También daremos un contexto sobre por qué es importante que Python pueda integrarse en el desarrollo web y cómo estas tecnologías podrían cambiar la forma en que construimos aplicaciones en el futuro.

3.1. Desarrollo web

El desarrollo web es el proceso de creación, construcción y mantenimiento de sitios y aplicaciones accesibles a través de Internet. Este proceso abarca desde la programación de la lógica y funcionalidades en el lado del servidor hasta el diseño de la interfaz y la experiencia de usuario en el lado del cliente (frontend) [20]. El desarrollo web moderno busca crear aplicaciones eficientes, accesibles y optimizadas para una variedad de dispositivos, mejorando la interacción y la experiencia del usuario.

Tradicionalmente, el desarrollo web se estructura en tres componentes clave:

1. Frontend: La parte visual e interactiva que el usuario final ve y con la que interactúa, construida principalmente con HTML [21], CSS [22] y JavaScript.
2. Backend: La lógica detrás de escena, que incluye la gestión de bases de datos, autenticación y la comunicación entre el servidor y el cliente, desarrollada con lenguajes como Python o Node.js [23].
3. Full Stack [24]: Un enfoque que combina tanto el desarrollo frontend como backend, permitiendo a los desarrolladores trabajar en todas las capas de la aplicación.

El objetivo final del desarrollo web es crear aplicaciones eficientes, accesibles y optimizadas para una variedad de dispositivos, mejorando la interacción y experiencia del usuario.

3.1.1 Evolución del desarrollo web

El desarrollo web comenzó en los 90 con páginas estáticas basadas en HTML y CSS simples. La llegada de JavaScript permite añadir interactividad en el navegador, revolucionando las aplicaciones web. Más tarde, frameworks como React facilitaron la creación de aplicaciones más rápidas y mantenibles, especialmente con el auge de los dispositivos móviles. Sin embargo, JavaScript seguía siendo el único lenguaje nativo en navegadores, limitando a desarrolladores que preferían otros lenguajes. WebAssembly surgió para ejecutar código compilado (como Python o C++) en el navegador con alto rendimiento, ampliando las posibilidades del desarrollo web.

3.2. JavaScript

JavaScript es un lenguaje de programación interpretado, lo que significa que su código se ejecuta directamente en el navegador sin necesidad de una compilación previa. Esto permite una respuesta rápida a las interacciones del usuario, siendo fundamental para el desarrollo de aplicaciones web dinámicas e interactivas.

Una de las características clave de JavaScript es que cuenta con funciones de primera clase, lo que implica que las funciones pueden ser tratadas como cualquier otra variable: pueden asignarse a otras variables, pasarse como argumentos a otras funciones o incluso devolverse como resultado.

Aunque su popularidad se ha consolidado principalmente en el desarrollo web del lado del cliente, JavaScript también se utiliza ampliamente fuera del navegador. Entornos como Node.js permiten su uso en el backend, y también está presente en aplicaciones de escritorio.

JavaScript es un lenguaje versátil que ofrece diversas características, entre las que destacan:

- Prototipos: JavaScript utiliza un modelo basado en prototipos para heredar propiedades y métodos, lo que permite una forma flexible de reutilizar código sin necesidad de clases tradicionales.
- Multiparadigma: Soporta varios estilos de programación, incluyendo la programación orientada a objetos (OOP) [25], funcional e imperativa. Esta versatilidad permite a los desarrolladores adoptar el enfoque que mejor se ajuste a sus necesidades.
- Tipado dinámico: No requiere declarar el tipo de las variables, ya que este se determina en tiempo de ejecución. Esto facilita un desarrollo rápido, aunque también puede provocar errores difíciles de detectar si no se gestiona correctamente.

3.3. WebAssembly

WebAssembly (Wasm) es una tecnología que permite ejecutar código binario de bajo nivel en navegadores con un rendimiento cercano al nativo, superando en ciertos casos la eficiencia de JavaScript. Permite compilar lenguajes como C o C++ para su ejecución directa en el navegador, ampliando las posibilidades del desarrollo web, especialmente

para aplicaciones que requieren gran potencia de cálculo o gráficos complejos. Sin embargo, al funcionar dentro de un entorno seguro y limitado (sandbox del navegador) [26], no puede acceder directamente a recursos del sistema operativo o hardware, sino que debe interactuar con ellos a través de las APIs [27] disponibles en JavaScript, que actúan como intermediarios.

WebAssembly no reemplaza a JavaScript, sino que lo complementa, combinando sus fortalezas para crear aplicaciones web más potentes y eficientes. Mientras Wasm maneja las tareas intensivas en rendimiento, JavaScript se encarga de la lógica, la interacción con el usuario y la manipulación del DOM. Esta cooperación permite desarrollar aplicaciones complejas, como videojuegos o software científico, con velocidades casi nativas, sin perder la flexibilidad y dinamismo que ofrece JavaScript.

Figura 4. Logotipo de WebAssembly

Un ejemplo destacado del uso de WebAssembly son las aplicaciones de edición de video en línea, como Photoshop. Estas aplicaciones, que suelen requerir un gran volumen de código y están desarrolladas en lenguajes como C para garantizar alto rendimiento, tradicionalmente implicaban sacrificar parte de su eficiencia al ser portadas a la web mediante JavaScript. Sin embargo, con WebAssembly, ahora es posible ejecutar un editor de video completo, escrito en C, directamente en el navegador, con un rendimiento casi idéntico al de su versión de escritorio.

3.3.1. Flujo de trabajo

WebAssembly convierte el navegador en una máquina virtual capaz de ejecutar código de alto rendimiento escrito en lenguajes como C, C++, Rust o Python, sin depender exclusivamente de JavaScript. El flujo de trabajo básico es el siguiente:

1. Escritura del código fuente: El desarrollador crea el programa en un lenguaje compatible con WebAssembly (por ejemplo, C, C++, Rust o Python).
2. Compilación a bytecode [28]: El código se compila a un archivo .wasm usando herramientas como Emscripten (para C/C++) [29] o Pyodide (para Python), optimizando el código para su ejecución eficiente en el navegador.
3. Carga en el navegador: El archivo .wasm se importa en la aplicación web mediante JavaScript, usando las APIs específicas de WebAssembly.
4. Traducción a código máquina: El motor del navegador traduce el bytecode a código máquina nativo y lo ejecuta dentro del navegador.

5. Interoperabilidad con JavaScript: Debido a que WebAssembly no puede acceder directamente al DOM ni otras APIs del navegador, utiliza un "glue code" en JavaScript para comunicar y coordinar ambas tecnologías, permitiendo llamar funciones WebAssembly desde JavaScript y mostrar resultados en la interfaz.

Por ejemplo, si tenemos una función en C que realiza un cálculo, como sumar dos números, podemos compilarla a WebAssembly mediante Emscripten y usar el glue code para llamar a esta función desde JavaScript. Luego, el resultado puede mostrarse en el DOM, algo que WebAssembly no puede hacer por sí solo.

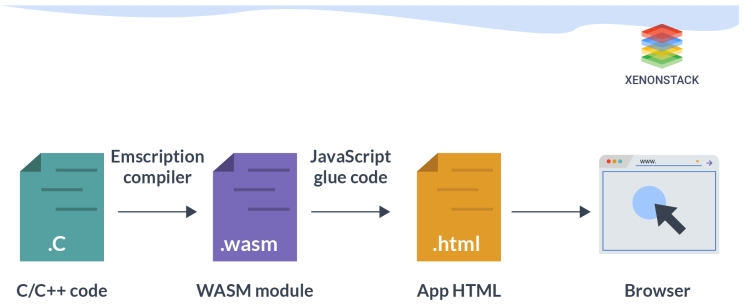

Figura 5. Flujo de funcionamiento de WebAssembly

3.4. Pyodide

En relación con WebAssembly, Pyodide es un puerto de CPython [30] a WebAssembly/Emscripten, lo que permite ejecutar código Python directamente en el navegador sin necesidad de un servidor backend. Pyodide hace posible instalar y utilizar paquetes de Python en el entorno web, permitiendo que cualquier paquete escrito en Python puro sea compatible.

Dado que Python se usa ampliamente en el ámbito científico, muchas de sus bibliotecas requieren extensiones en C para lograr un rendimiento óptimo. Pyodide ha abordado este desafío portando no sólo paquetes escritos en Python, sino también bibliotecas científicas con componentes en C, como NumPy, Pandas, Matplotlib y Scikit-learn [31]. Además, incluye paquetes de propósito general como RegEx [32], ampliando aún más sus capacidades.

Pyodide nace con el objetivo de ejecutar Python y parte de su ecosistema científico directamente en navegadores web. Para conseguirlo, Pyodide se basa en Emscripten, un compilador que convierte código de C o C++ en WebAssembly, y aprovecha que la implementación de Python (CPython) está escrita principalmente en C. De esta manera, Pyodide genera un binario WebAssembly que incluye el intérprete de Python, así como un subconjunto de bibliotecas nativas compiladas.

El funcionamiento de Pyodide se puede resumir en los siguientes pasos:

1. El desarrollador escribe código en Python, ya sea una aplicación completa o módulos específicos.
2. Pyodide utiliza Emscripten para compilar CPython a WebAssembly. Esto incluye no sólo el intérprete de Python, sino también las bibliotecas estándar y las extensiones en C portadas por Pyodide.
3. El resultado de la compilación es un archivo .wasm que contiene el intérprete de Python y las bibliotecas necesarias.
4. El archivo .wasm se carga en una aplicación web mediante JavaScript, utilizando las APIs de WebAssembly. Esto permite ejecutar el intérprete de Python directamente en el navegador.
5. Una vez cargado, el intérprete de Python ejecuta el código escrito por el desarrollador.

3.5. CPython

CPython es la implementación oficial y más popular del lenguaje Python, escrita en C. Funciona como un intérprete que traduce el código Python a un formato intermedio llamado bytecode, que luego es ejecutado por la Máquina Virtual de Python (PVM) [33]. Este enfoque permite que el mismo bytecode sea portable y pueda ejecutarse en cualquier sistema con un intérprete compatible, sin necesidad de recompilar.

Aunque existen otras implementaciones de Python, CPython es la referencia principal debido a su cumplimiento estricto de la especificación del lenguaje, su estabilidad y la compatibilidad con la mayoría de las bibliotecas disponibles. Cuando un usuario instala Python en su sistema, generalmente está instalando CPython, el software que realmente interpreta y ejecuta el código Python, destacándose por su amplio ecosistema y soporte.

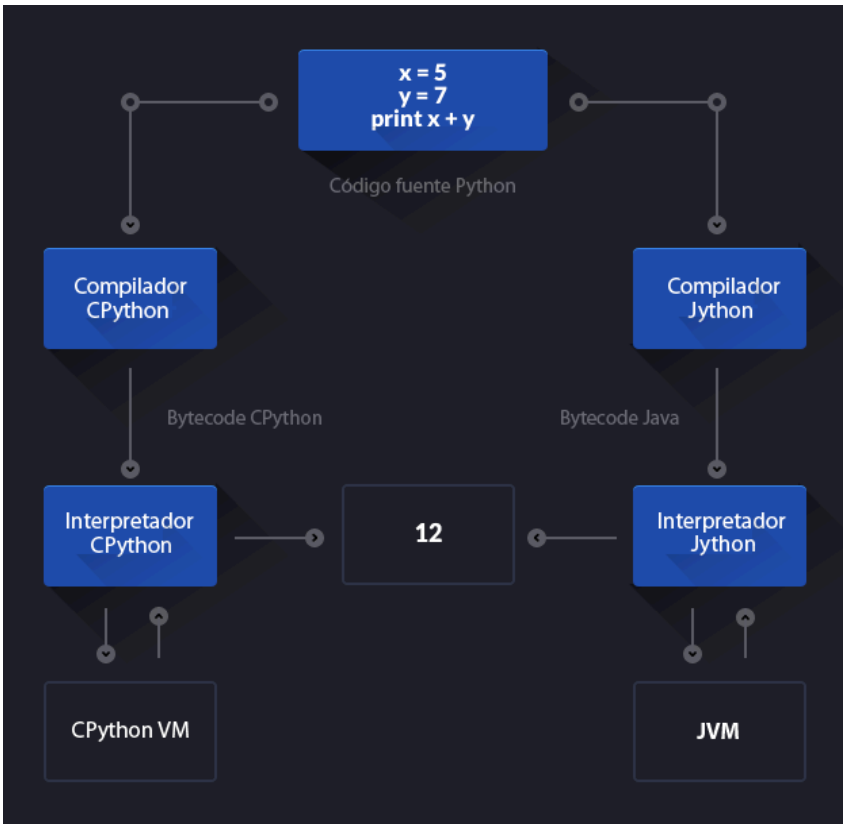

Figura 6. Flujo de funcionamiento de CPython

3.6. Python

Python es un lenguaje de programación de alto nivel, conocido por su sintaxis clara y sencilla, que facilita el desarrollo tanto para principiantes como para expertos. Su extensa biblioteca estándar y la activa comunidad han generado numerosas librerías para diversas áreas como inteligencia artificial, análisis de datos, automatización y desarrollo web. Python destaca por su flexibilidad, rapidez en el desarrollo gracias a su naturaleza interpretada, y su capacidad para manejar desde proyectos simples hasta aplicaciones complejas.

Aunque tradicionalmente Python se ha usado en servidores, tecnologías como Pyodide y PyScript, que usan WebAssembly, permiten ahora ejecutar Python directamente en el navegador. Esto abre nuevas posibilidades para usar Python en aplicaciones web del lado cliente, compitiendo con JavaScript y permitiendo ejecutar aplicaciones complejas sin necesidad de backend. Un ejemplo destacado es la ejecución del clásico juego *Doom* directamente en la barra de direcciones del navegador, mostrando cómo Python, junto con Pyodide y WebAssembly, está revolucionando el desarrollo web y ampliando sus capacidades.

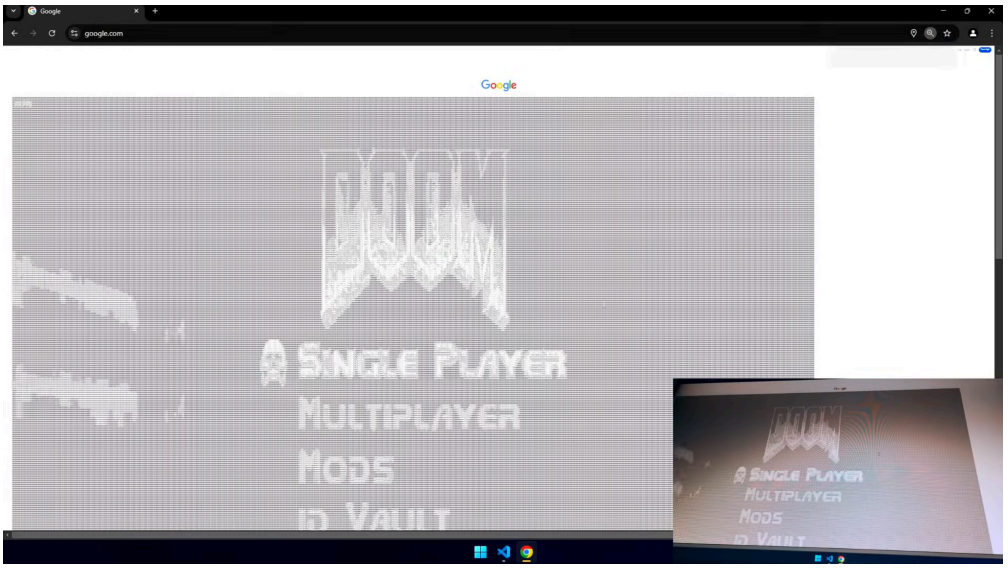

Figura 7. Ejemplo de Python y Pyodide: Imagen de la ejecución del juego Doom en el navegador utilizando Pyodide.

3.6.1. Ecosistema de Python

Python se ha convertido en uno de los lenguajes de programación más versátiles y utilizados en el mundo gracias a su extenso ecosistema de herramientas y bibliotecas. Su comunidad activa ha desarrollado una gran cantidad de librerías que cubren múltiples áreas, desde el análisis de datos y la inteligencia artificial hasta el desarrollo web y la automatización.

Procesamiento numérico y cálculo matemático

Uno de los grandes puntos fuertes de Python es su capacidad para manejar grandes volúmenes de datos de manera eficiente. Sin embargo, las estructuras de datos nativas de Python pueden volverse lentas cuando se procesan grandes conjuntos de datos o se realizan cálculos matemáticos intensivos. Para abordar esta limitación, existen varias bibliotecas optimizadas que mejoran significativamente el rendimiento.

- **NumPy**: Es la biblioteca fundamental para cálculos numéricos en Python. Ofrece soporte para arrays multidimensionales y operaciones vectorizadas que superan en rendimiento a las listas nativas de Python, permitiendo cálculos hasta 50 veces más rápidos. Además, es ampliamente utilizada en álgebra lineal, estadística y simulaciones científicas.
- **SciPy** [34]: Extiende las capacidades de NumPy proporcionando módulos adicionales para optimización, integración, procesamiento de señales y álgebra lineal avanzada. Su eficiencia y compatibilidad con NumPy la convierten en una herramienta clave en el ámbito científico y técnico.

Análisis y manipulación de datos

El análisis y la manipulación de datos son esenciales en múltiples campos, desde la ciencia de datos hasta el desarrollo web. Python destaca en este ámbito gracias a bibliotecas optimizadas que permiten trabajar con grandes volúmenes de datos de manera eficiente. Estas herramientas facilitan la transformación, limpieza y estructuración de la información, lo que resulta clave para aplicaciones como visualización de datos en tiempo real y procesamiento en el navegador.

La biblioteca que más destaca para realizar esto es *pandas*. Es la biblioteca más utilizada para la manipulación y análisis de datos en Python. Ofrece estructuras de datos como DataFrames [35], que permiten gestionar grandes volúmenes de información de forma eficiente. Su integración con NumPy y su capacidad para manejar formatos como CSV [36], JSON [37] y bases de datos la convierten en una herramienta fundamental en proyectos de ciencia de datos y visualización.

Visualización de datos y gráficos interactivos

Python se ha convertido en una herramienta esencial para el análisis y la visualización de datos, mediante sus librerías es posible crear gráficos de alta calidad que van desde simples diagramas hasta complejas visualizaciones interactivas. Entre las bibliotecas más populares para la visualización de datos en Python destacan Matplotlib y Plotly [38].

Matplotlib es la biblioteca fundamental para la creación de gráficos en Python. Ofrece un control granular sobre cada aspecto de un gráfico, lo que permite personalizar las visualizaciones de manera exhaustiva. Con Matplotlib, puedes crear una amplia variedad de gráficos, desde simples diagramas de dispersión hasta complejos gráficos 3D.

Plotly, por su parte, es una biblioteca de visualización de datos interactiva que va más allá de Matplotlib. Permite crear gráficos dinámicos y atractivos que se pueden explorar de manera intuitiva. Con Plotly, puedes crear gráficos 3D, mapas, diagramas de red y más, todo ello con una interfaz más sencilla y con una mayor capacidad para crear visualizaciones interactivas.

Desarrollo web

Python también es un lenguaje muy popular en el ámbito del desarrollo web y ahora más que nunca con la llegada de PyScript, gracias a su simplicidad, legibilidad y la amplia variedad de herramientas que facilita este tipo de proyectos. Python cuenta con diversos frameworks que simplifican la creación de aplicaciones web, tanto para desarrolladores principiantes como para expertos. Entre los más destacados se encuentra Django [39].

Asincronía y Concurrencia

La asincronía [40] y concurrencia [41] son conceptos clave para el desarrollo de aplicaciones eficientes, especialmente cuando se trata de manejar múltiples tareas al mismo tiempo, como en el caso de servidores web, aplicaciones en tiempo real o sistemas distribuidos. Python, aunque es conocido por su simplicidad, también ofrece un ecosistema

robusto para implementar programas concurrentes y asíncronos, ayudando a mejorar el rendimiento y la escalabilidad de las aplicaciones.

Una de estas librerías es Asyncio [42], esta forma parte de la biblioteca estándar de Python y es fundamental para implementar programación asíncrona. Asyncio permite escribir código concurrente utilizando la sintaxis *async* y *await*, facilitando la ejecución de tareas que pueden esperar (como peticiones HTTP, acceso a bases de datos o lectura/escritura de archivos) sin bloquear el hilo principal del programa. Es ideal para aplicaciones que manejan muchas operaciones de entrada y salida, como servidores web asíncronos o aplicaciones de chat en tiempo real.

Seguridad y encriptación de datos

La seguridad y la encriptación de datos son aspectos fundamentales en el desarrollo de aplicaciones que manejan información sensible. En un mundo donde la protección de datos es una prioridad, Python destaca por su amplio ecosistema de bibliotecas especializadas que ofrecen herramientas poderosas para garantizar la confidencialidad, integridad y autenticidad de la información.

PyCryptodome [43] es una biblioteca de Python de código abierto que proporciona una amplia gama de herramientas criptográficas para asegurar la información de manera confiable. Esta biblioteca es ampliamente utilizada en el desarrollo de aplicaciones que requieren un alto nivel de seguridad, como sistemas de autenticación, protección de datos sensibles y comunicaciones seguras. Entre sus funcionalidades más destacadas se incluyen

- Cifrado simétrico y asimétrico
- Generación de números aleatorios seguros
- Firma digital

Aprendizaje Automático y Machine Learning en Python

El aprendizaje automático (Machine Learning, ML) [44] es uno de los campos más dinámicos y relevantes en la ciencia de datos y la inteligencia artificial. Python ha emergido como uno de los lenguajes de programación más utilizados en este ámbito debido a su rica comunidad y a la disponibilidad de bibliotecas poderosas y bien documentadas.

Scikit-learn es una de las bibliotecas más completas y populares para la implementación de algoritmos de aprendizaje automático en Python. Ofrece una amplia gama de herramientas y técnicas para tareas como la clasificación, regresión, clustering, reducción de dimensionalidad y evaluación de modelos. Su diseño modular permite que los usuarios construyan, ajusten y validen modelos de manera intuitiva y eficiente, convirtiéndola en una herramienta esencial tanto para investigadores como para desarrolladores.

3.7 PyScript

PyScript es un framework que permite ejecutar código Python directamente en el navegador web, aprovechando la capacidad de WebAssembly (Wasm) y Pyodide para ejecutar código nativo en el entorno del navegador de manera eficiente. A través de PyScript, los desarrolladores pueden escribir y ejecutar aplicaciones de Python en la web

sin necesidad de un servidor backend, lo que abre nuevas posibilidades para el desarrollo web interactivo utilizando Python en lugar de JavaScript.

A diferencia de soluciones tradicionales basadas en JavaScript, PyScript permite que los desarrolladores escriban sus aplicaciones web en Python, sin necesidad de aprender o integrar JavaScript. Esto hace que el proceso de creación de aplicaciones web sea más accesible para los programadores que ya están familiarizados con Python, y al mismo tiempo, mantiene la compatibilidad con las tecnologías web estándar.

Estas son sus características más destacables:

1. Ejecución local en el navegador
    a. Usa Pyodide y WebAssembly para ejecutar Python directamente en el cliente, sin servidores intermedios.
2. Integración con HTML y CSS
    a. Permite combinar código Python con etiquetas HTML y estilos CSS para construir interfaces visuales ricas e interactivas.
3. Soporte para bibliotecas de Python
    a. Compatible con bibliotecas populares como NumPy, Matplotlib, Pandas, Plotly y scikit-learn, ideales para análisis de datos y aprendizaje automático.
4. Flujo de trabajo simplificado:
    a. Elimina la necesidad de servidores, entornos múltiples o lenguajes adicionales, lo que agiliza el desarrollo para quienes ya manejan Python.

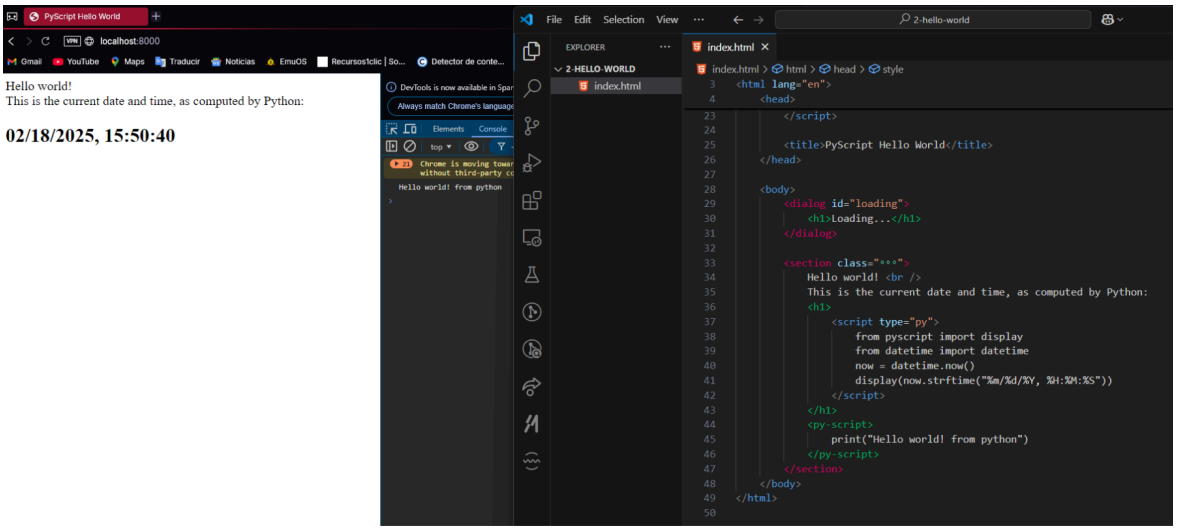

Figura 8. “Hello world” utilizando PyScript.

4. Diseño de la solución

En este apartado se describe en detalle la arquitectura general y el diseño implementado para el desarrollo del proyecto, abordando tanto los aspectos técnicos como las decisiones clave que guiaron su construcción.

El diseño de la solución no se limita únicamente a la estructura técnica, sino que también incluye la justificación de las decisiones tomadas en relación con la interfaz de usuario, el enfoque modular del sistema, el uso de tecnologías específicas y la organización de los benchmarks por ámbitos funcionales. Cada benchmark fue diseñado con un propósito concreto, permitiendo evaluar distintas capacidades de las tecnologías bajo estudio, como rendimiento computacional, capacidad de manejo de datos o paralelismo.

A su vez, se detalla la manera en que el sistema se ha estructurado en componentes claramente diferenciados, tales como el frontend, el backend, y los workers [45], asegurando así la escalabilidad, mantenibilidad y flexibilidad del entorno.

4.1. Arquitectura general

Para comenzar, el proyecto o sistema final se divide en dos carpetas dentro del repositorio:

- Carpeta de benchmarks
  - Aquí se encuentra el código de cada ámbito de prueba estructurado en la jerarquía ámbito → prueba → versión. Cada versión es un bloque de código autónomo que puede ejecutarse de forma aislada en local, facilitando la recolección de métricas en condiciones controladas.
  - Se diseñó en cada versión, que se ejecutará un script ejecutando todo lo necesario para cada prueba, con ello, se facilita la ejecución.
  - Además, la página index.html contiene únicamente la parte básica de código justa y necesaria para mostrar los resultados.
- Proyecto Astro
  - Esta página web integra las mismas pruebas adaptadas a la estructura de Astro y Tailwind CSS [46]. Al iniciar cualquier operación, aparece una pantalla de carga que informa al usuario de que el benchmark está en ejecución y muestra las diferentes métricas al finalizar la ejecución. La interfaz dispone de controles para seleccionar el ámbito, la prueba y la versión, así como de un área de resultados donde se exponen los diferentes resultados en base a sus propias métricas definidas.

Cabe destacar que algunas versiones no emplean Web Workers intencionadamente, de modo que ejecutan toda la carga en el hilo principal. Esto permite medir el impacto del bloqueo de la interfaz antes de introducir versiones paralelas que sí utilizan workers. De este modo el usuario es consciente del rendimiento de la herramienta ejecutándose. Aunque esto sucede solo en el proyecto Astro, ya que se quiere dar a entender ese FeedBack al usuario, pero en local e internamente, solo son los primeros ámbitos los que se ejecutan en el hilo principal. En las siguientes, se usan web workers.

Por otro lado, la solución incluye un backend mínimo implementado como endpoints de Astro, cuya función es servir archivos estáticos (modelos, datos de prueba) y proporcionar, cuando sea necesario, conexiones de WebSocket para simular múltiples solicitudes concurrentes. Aparte de proporcionar contenido, variables, y funciones en tiempo dinámico para optimizar el uso de componentes y de páginas. De tal manera se reutiliza código y es escalable.

4.2. Esquema del proyecto

En este apartado se presenta la estructura del proyecto, tanto en su versión local como en el entorno desarrollado con Astro. El objetivo es ofrecer una representación gráfica que facilite la comprensión del funcionamiento general del sistema, conforme a lo descrito hasta ahora.

En el primer gráfico incluido, se muestra un árbol jerárquico de carpetas que parte desde la raíz del proyecto, en este caso la carpeta “Local-TFG”. Desde esta carpeta principal se ramifican los seis entornos de benchmarks definidos, cada uno correspondiente a un ámbito temático distinto. A su vez, cada entorno contiene entre dos y tres pruebas específicas, centradas en diferentes algoritmos o procesos, con el fin de evaluar el comportamiento y rendimiento de las tecnologías involucradas.

Cada una de estas pruebas puede contar con varias versiones de implementación. Sin embargo, hay casos en los que solo existe una única versión, por lo que no se genera una subcarpeta adicional. Para diferenciar visualmente los elementos del esquema, se utilizan dos colores: azul claro para representar las carpetas (es decir, estructuras organizativas del proyecto) y verde pistacho para los contenedores finales que incluyen el código funcional de cada implementación. En total, el sistema recopila 18 implementaciones de código distribuidas entre los distintos benchmarks.

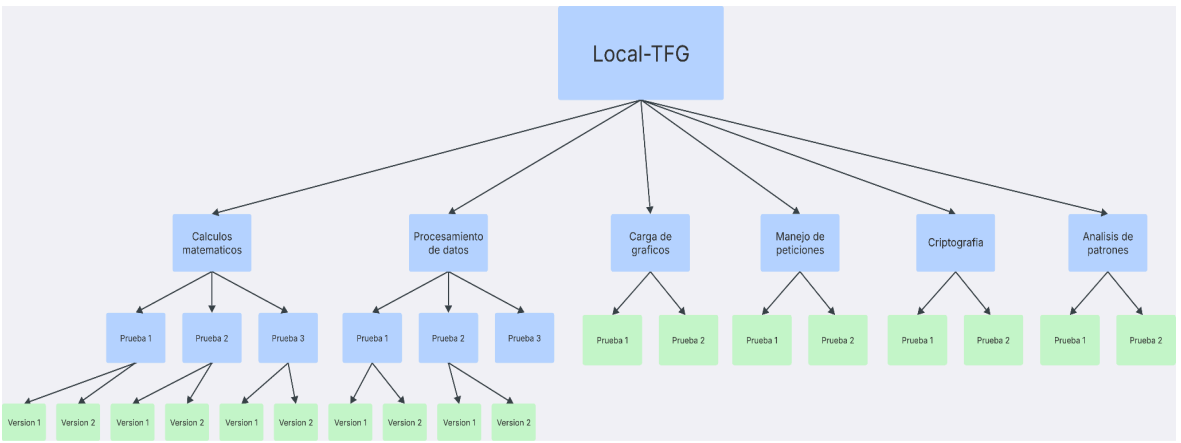

Figura 9. Esquema del proyecto en local.

En el segundo gráfico, se observa una estructura más compleja que la anterior, ya que incluye una serie de carpetas esenciales para su funcionamiento. Las más importantes son *src* y *public*. La carpeta *public* contiene todos los archivos estáticos necesarios para el

cliente, como imágenes, *scripts* y, sobre todo, los propios archivos de los benchmarks, que deben estar aquí para poder ser ejecutados desde el navegador. Por su parte, *src* agrupa el código fuente del proyecto.

Dentro de src se encuentran carpetas clave como *scripts*, donde se incluyen funciones reutilizables para las vistas; *pages*, que organiza la estructura de navegación según su jerarquía de carpetas y archivos, permitiendo generar automáticamente las URLs correspondientes; *layout*, que contiene las plantillas generales como el encabezado y pie de página; y *components*, donde se agrupan elementos visuales reutilizables, como tarjetas de contenido y otros bloques que ayudan a estructurar cada vista. En el interior de *pages*, hay archivos especiales como *[test]* y *[versión]*, que actúan como rutas dinámicas generadas automáticamente según los datos definidos en la carpeta data.

Además, el proyecto incluye otras carpetas relevantes generadas o gestionadas por Astro. La carpeta *.astro* almacena archivos temporales necesarios para la compilación del proyecto. *dist* contiene la versión final lista para producción, es decir, el sitio web tal como se desplegará. Y *node_modules* agrupa todas las dependencias externas del proyecto, instaladas mediante el gestor de paquetes de Node.js.

En cuanto a la codificación visual del gráfico, se ha utilizado un sistema de colores para facilitar su comprensión. El color malva representa las carpetas estructurales principales que organizan el proyecto. El azul agua señala las implementaciones concretas de los benchmarks. El púrpura indica carpetas relacionadas con estilos o archivos estáticos. El amarillo se ha asignado a los archivos y carpetas de JavaScript. El turquesa destaca los componentes y elementos visuales empleados en la interfaz. Y, por último, el color salmón identifica la carpeta correspondiente a la API.

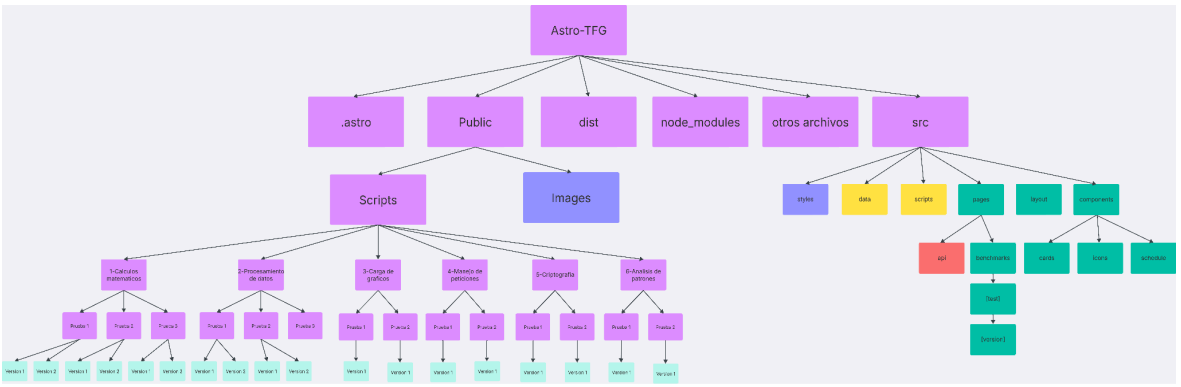

Figura 10. Esquema del proyecto Astro.

Como parte complementaria al esquema general del proyecto desarrollado con Astro, a continuación se presenta el diagrama de la arquitectura software (Figura 11), en el cual se representa el diseño general del sistema Astro, incluyendo los principales componentes implicados en la ejecución de las pruebas. En los siguientes apartados se detalla de forma más profunda la implementación de cada uno de estos elementos, pero a modo de resumen, se describen a continuación:

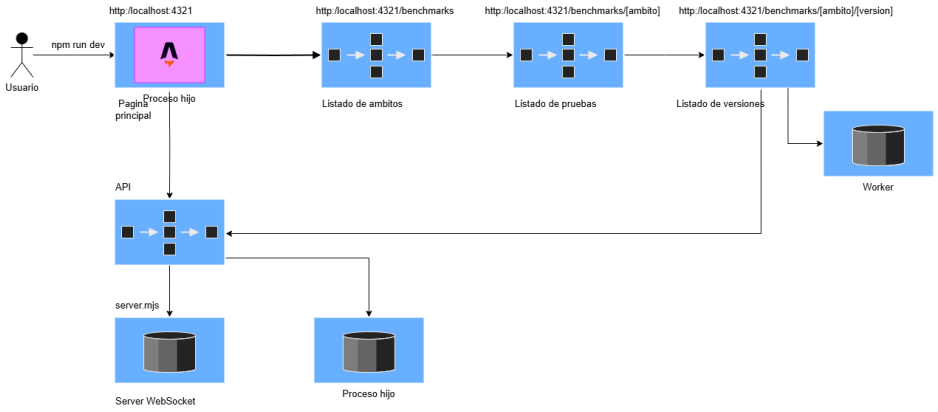

Figura 11. Diagrama de la arquitectura de software.

Para desplegar el proyecto Astro, el usuario debe ejecutar el comando npm run dev. Esto lanzará el servidor de desarrollo local, accesible a través del navegador mediante la URL *http://localhost:4321*. De forma simultánea, se inicia un segundo servidor en segundo plano que habilita el canal de comunicación vía WebSockets, necesario para ciertas pruebas interactivas.

Desde el navegador, el recorrido principal del sistema sigue una estructura jerárquica de rutas que permite acceder progresivamente a los diferentes benchmarks. Primero, a través de la ruta */benchmarks*, se muestran los distintos ámbitos temáticos disponibles. Posteriormente, accediendo a */benchmarks/[ámbito]*, se listan las distintas pruebas y versiones disponibles dentro del ámbito seleccionado. Finalmente, se accede a la página específica de ejecución mediante la ruta */benchmarks/[ámbito]/[versión]*, donde el usuario puede interactuar con la versión concreta de la prueba.

Esta última vista activa diversos componentes, en función del tipo de benchmark seleccionado. Las posibles situaciones incluyen:

- Ejecución de procesos hijos en el servidor: Utilizada para pruebas a través de la API que requieren medir el rendimiento fuera del navegador. En este caso, se lanza un proceso hijo desde el backend de Astro que ejecuta scripts externos (en Python o Node.js).

- Inicialización de Web Workers: Permite ejecutar código en hilos secundarios dentro del navegador, separando la carga de trabajo del hilo principal y analizando así el rendimiento en entornos multihilo del lado cliente.

- Invocación de funciones de la API: Se realizan peticiones HTTP a funciones alojadas en los endpoints del backend de Astro. La más destacada es la conexión WebSocket, cuyo servidor se inicia al lanzar el proyecto. A través de esta API se obtiene la dirección del servidor WebSocket, permitiendo establecer la conexión y ejecutar la prueba que requiere comunicación con el servidor.

La página principal del proyecto está estructurada en distintos apartados que se organizan de forma clara y jerárquica para facilitar la navegación. En primer lugar, se presenta un encabezado (header) compuesto por dos imágenes situadas en los extremos, un título central, una breve descripción introductoria y cuatro botones que permiten acceder rápidamente a las secciones principales: Introducción, Sobre el proyecto, Benchmarks y Resultados. A continuación, cada sección se dispone en orden descendente. La sección de Introducción incluye una descripción general del propósito del sitio. Le sigue la sección Sobre el Proyecto, que contiene cuatro desplegables con información detallada y diferenciada. Posteriormente, se encuentra la sección Benchmarks, que incluye un desplegable con una descripción por cada prueba, acompañado de un icono a la derecha que redirige a la vista dedicada a los benchmarks. Finalmente, la sección de Resultados resume la información obtenida tras la ejecución de las pruebas. Finalmente concluye con un footer.

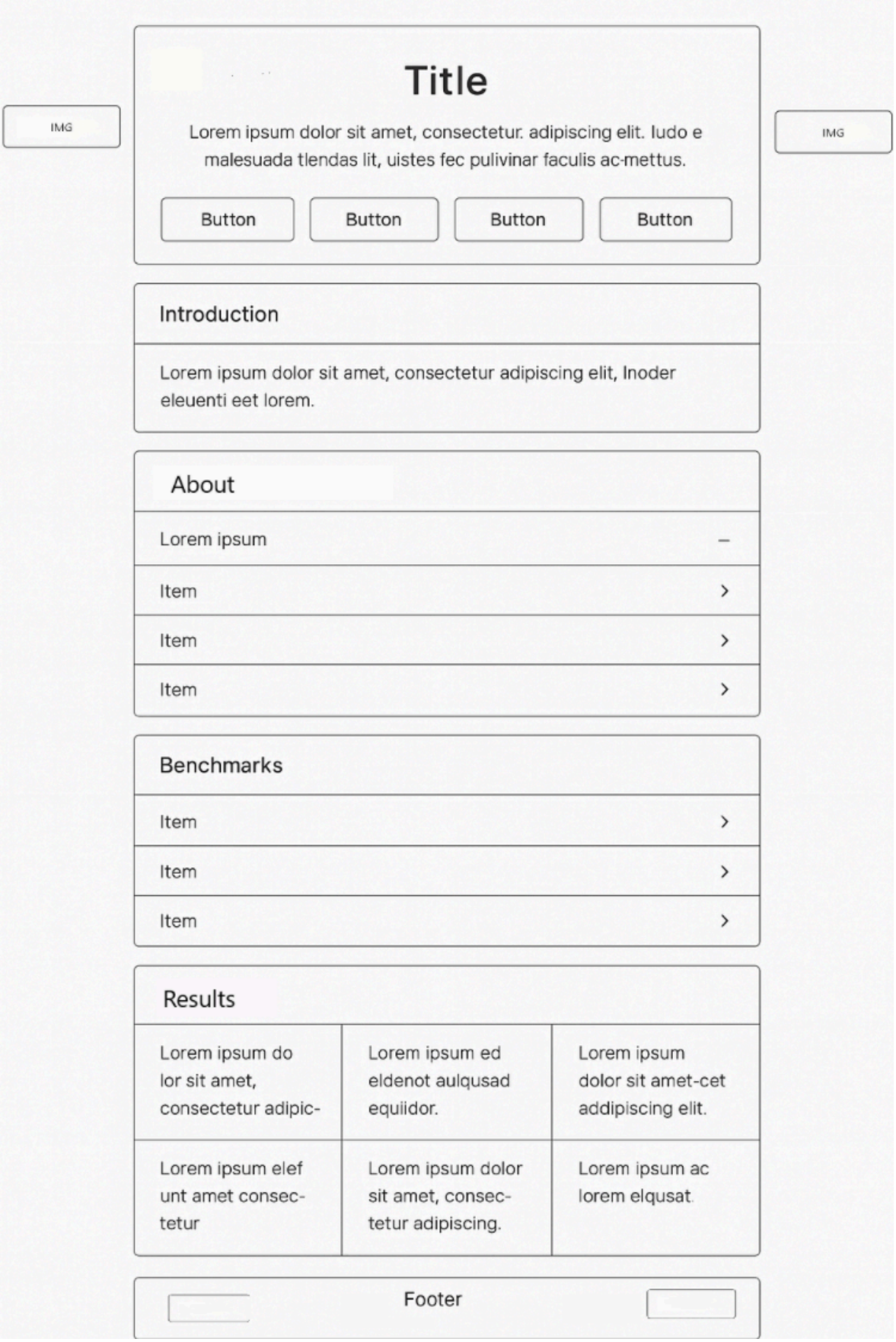

Figura 12. Esquema de la vista principal del proyecto Astro.

La vista dedicada a la ejecución de un benchmark presenta una estructura sencilla y funcional. En la parte superior se encuentra el encabezado (header), compuesto por un título principal, una breve descripción y dos botones que forman parte de la cabecera, uno se dirige al inicio y otro a los diferentes benchmarks. Después, se muestra el contenido central del benchmark. Este inicia con el título correspondiente a la versión de la prueba, seguido por una matriz de entradas (inputs), que suele disponerse en una cuadrícula de 2x2, aunque en algunos casos puede variar a 1x2 o a ninguno, dependiendo si la prueba necesita inputs o no. Bajo los campos de entrada, se sitúan los botones de ejecución, que pueden ser dos o cuatro, siempre organizados en una única fila horizontal. Seguidamente, se presenta una tabla de resultados que refleja las métricas obtenidas. El tamaño de esta tabla varía entre 2x2 y 2x4, dependiendo de la prueba. Finalmente, la página se cierra con el mismo footer.

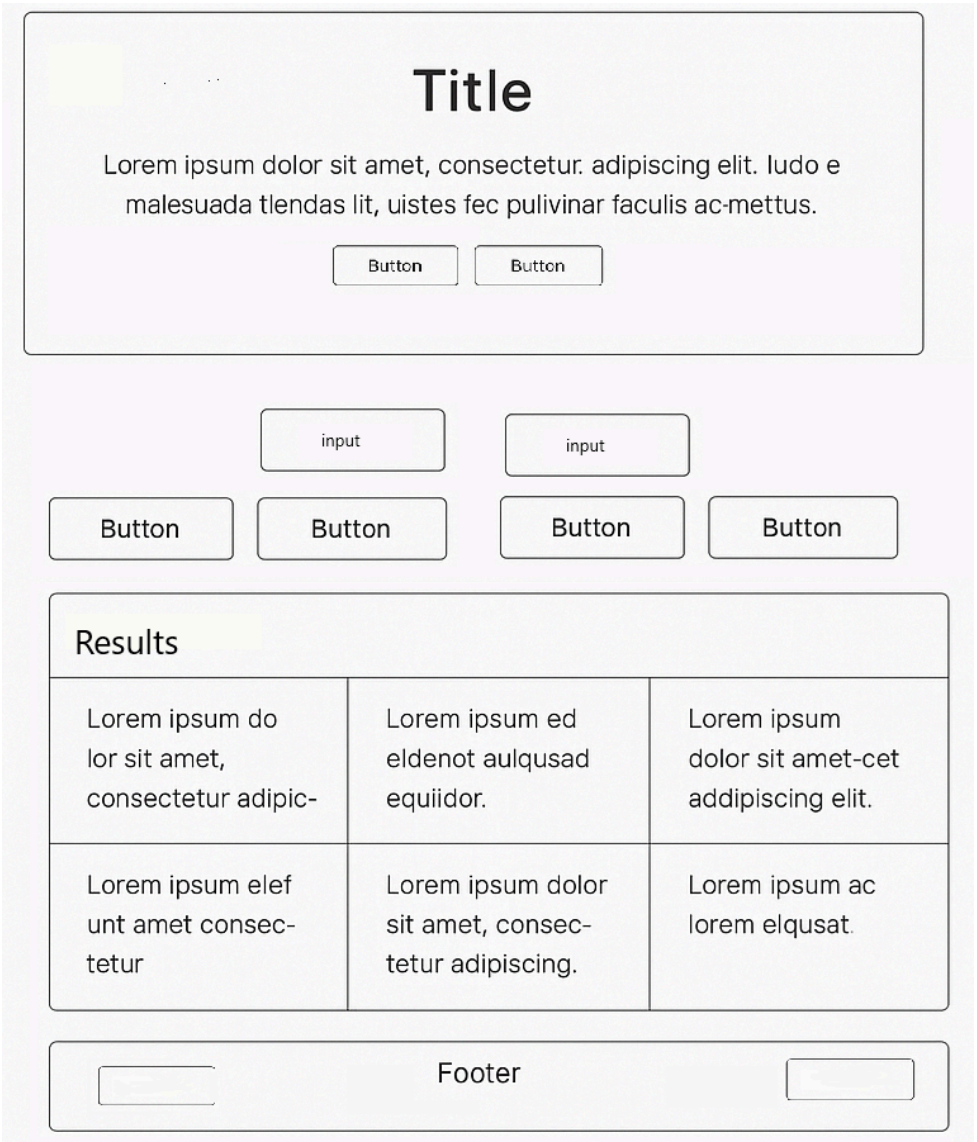

Figura 13. Esquema de la vista del benchmark del proyecto Astro.

4.4. Decisiones de diseño relevantes

Una de las primeras decisiones importantes en el desarrollo del proyecto fue mantener una separación clara entre el entorno local y el entorno implementado con Astro.

Por un lado, ejecutar los benchmarks en local permite replicar un entorno más próximo al que se utilizará durante el desarrollo real de aplicaciones, donde los procesos están aislados y controlados de forma manual. Esta configuración facilita trabajar directamente con los servidores y acceder a los recursos del sistema sin limitaciones impuestas por un entorno de navegador. Además, permite medir de forma más precisa el rendimiento puro del código, sin interferencias externas como la carga del entorno web, el renderizado del DOM o la ejecución de scripts adicionales. Al tener instancias separadas de Node.js y Python, se puede garantizar que ambos lenguajes reciban las mismas peticiones de prueba y devuelvan métricas equivalentes bajo condiciones controladas, lo cual es esencial para una comparación justa. Por este motivo, los resultados de los Benchmarks principalmente se tomaron desde el entorno local y no de Astro, para evitar cualquier posible interferencia en los resultados.

Por otro lado, el entorno con Astro está diseñado con fines de presentación, estructura y despliegue web. Astro permite construir una interfaz atractiva y bien organizada, accesible desde cualquier navegador. Además, al integrar las pruebas dentro de una arquitectura de componentes reutilizables y rutas dinámicas, se facilita la navegación y la escalabilidad del proyecto. Esto permite centralizar datos, separar responsabilidades (como estilos, lógica y estructura), y mantener una experiencia de usuario coherente y mantenible.

La siguiente decisión importante fue la elección de Astro como framework principal para la interfaz del proyecto, esto responde a varias ventajas clave que ofrece respecto a otras opciones. Una de las más destacadas es que Astro permite generar páginas completamente estáticas sin necesidad de cargar JavaScript en el cliente, a menos que se especifique explícitamente. Esto no solo mejora considerablemente los tiempos de carga, sino que también reduce el consumo de recursos en el navegador, lo que resulta especialmente útil en un proyecto como este, enfocado en medir rendimientos y tiempos de ejecución de distintas tecnologías. Además, Astro facilita la integración de componentes reutilizables y una estructura de rutas clara basada en el sistema de archivos, lo cual simplifica el mantenimiento y escalado del sitio.

4.5. Benchmarks

Cada benchmark fue implementado de forma independiente para facilitar su ejecución en local. Para ello, se creó un archivo index.html sencillo como punto de entrada y un script denominado start_servers.py, encargado de levantar, según el benchmark correspondiente, dos servidores locales: uno para ejecutar el código en Node.js y otro para hacerlo en CPython. Ambos servidores están preparados para recibir exactamente las mismas solicitudes, ejecutar la lógica de la prueba y devolver las métricas resultantes. Además, un

tercer servidor se encarga de servir la interfaz de la página web, mostrando el archivo index.html.

En cuanto al diseño del código, se ha procurado mantener una estructura coherente y paralela entre lenguajes, respetando la nomenclatura característica de cada uno. Las funciones compartidas entre ambos lenguajes se escriben directamente en el archivo index.html, mientras que las funciones específicas de cada uno se desarrollan en sus respectivos entornos, utilizando siempre herramientas propias del lenguaje salvo que se indique lo contrario.Los resultados generados por las pruebas se presentan en una tabla HTML, organizada por celdas, y siempre se incluye el tiempo de carga del entorno PyScript, identificado como Page Load Time (PLT).

Para mantener una organización clara, el código de cada lenguaje se ubica en una carpeta con el nombre del mismo. A su vez, el proyecto sigue una jerarquía estructurada en tres niveles: los ámbitos o temas que agrupan las pruebas relacionadas en base a los conceptos planteados, las pruebas que definen cada benchmark de forma concreta y, por último, las versiones, que representan distintas implementaciones del mismo benchmark. Las versiones numeradas como 2 tienden a enfocarse en aplicaciones más cercanas a problemas del mundo real, mientras que las versiones 1 suelen centrarse en algoritmos básicos o académicos. En aquellos casos donde solo existe una versión, esta corresponde directamente a una aplicación de carácter práctico.

Por último, cabe destacar que en todas las pruebas se mide el PLT, es decir, el tiempo transcurrido hasta que PyScript está completamente cargado y disponible en la página. Este valor varía en función del tamaño del bundle y de las dependencias del entorno, y se puede observar tanto en las pruebas locales como en la versión final del proyecto desplegado con Astro.

4.5.1. Cálculos matemáticos intensivos.

En el ámbito de los cálculos matemáticos intensivos se diseñaron tres pruebas principales donde se midieron los siguientes parámetros:

- El tiempo de ejecución (en milisegundos)
- El consumo de memoria (en megabytes)

De forma paralela, el mismo código de cada prueba también da la opción a ejecutar la prueba mediante solicitudes HTTP a los servidores de Node.js y CPython. En estos entornos servidor se registran:

- El tiempo de respuesta (en milisegundos)
- El uso de memoria (en megabytes)
- El porcentaje de uso de CPU

Además, al ejecutar las pruebas con PyScript en el navegador se introduce la métrica “Py + WA”, que agrupa el tiempo total de descarga y compilación del binario de Python (WebAssembly) más el tiempo de ejecución de la prueba. Este valor ofrece una visión global del coste combinado de inicialización y procesamiento en el cliente.

Un aspecto a tener en cuenta es la configuración de PyScript, que se define en un archivo de Python mediante distintos formatos (por ejemplo, TOML [47] o JSON). Esta configuración varía según los requisitos de cada prueba y permite especificar parámetros como paquetes a cargar o el uso de Web Workers. Aunque no existe una razón técnica estricta que obligue a usar un formato u otro, se han empleado ambos para ilustrar la flexibilidad de PyScript a la hora de gestionar la configuración y mostrar las múltiples opciones que ofrece.

4.5.2. Procesamiento de grandes volúmenes de datos

En este tema y en adelante, se decidió centrar el análisis exclusivamente en la ejecución en el navegador, omitiendo las mediciones en servidores externos y descartando la métrica “Py + WA” al comprobar que su impacto en el tiempo total era insignificante. Por lo tanto, solo se evaluaron PyScript y JavaScript, registrando las siguientes métricas:

- Tiempo total de ejecución
- Consumo total de memoria RAM
- Tiempo de cada operación individual
- Uso de memoria por operación individual

A partir de la segunda versión de la prueba, se introducen los Web Workers. En este benchmark su uso persigue dos objetivos: relegar la carga de trabajo al hilo secundario para no bloquear la interfaz principal y distribuir la tarea en paralelo cuando se emplea más de un worker, con el objetivo de observar su escalabilidad. La mecánica de comunicación entre el hilo principal y los workers se realiza mediante mensajes JSON; los workers se encargan exclusivamente de ejecutar los cálculos definidos por la prueba.

Para poder usar SharedArrayBuffer, necesario para compartir memoria entre el hilo principal y los workers, fue preciso servir el entorno local a través de HTTPS y configurar las cabeceras de aislamiento correspondientes. De este modo, los workers pueden operar sobre buffers compartidos de forma segura y eficiente.

Por último, también cambiamos de usar TOML a JSON, para demostrar que hay diversas opciones para configurar PyScript.

4.5.3. Carga y representación de gráficos complejos

En este tema se buscó probar el rendimiento de la inicialización del worker y principalmente del rendimiento para generar gráficos no interactivos, que internamente sean únicamente un PNG. Y en la siguiente prueba, será un gráfico interactivo.

En general, las métricas medidas fueron:

- Tiempo promedio de inicialización del worker
- Tiempo promedio generación de datos: Tiempo en generar los datos.
- Tiempo de renderizado: Tiempo en generar el PNG o el Gráfico
- Uso promedio de memoria
- Tiempo total

Es de mencionar, que ahora los resultados se harán en media, con un input que permitirá obtener un resultado estadístico en base a la cantidad de repeticiones de una ejecución de las operaciones de la prueba. También aparecerán nuevos inputs para realizar pruebas con diferentes datos. También es de destacar que a nivel de local, se añade un cronómetro en adelante para observar el tiempo transcurrido de las operaciones, en el proyecto de Astro no se incluyó ya que no se consideraba útil teniendo en cuenta la pantalla de carga.

4.5.4. Manejo de múltiples solicitudes concurrentes

Este benchmark, al igual que los posteriores, contará únicamente con una única versión de implementación. A diferencia de benchmarks anteriores que ofrecían tanto una versión simple como una optimizada, en este caso se ha optado por centrarse exclusivamente en la versión optimizada. La razón de esta decisión radica en que, si bien las versiones simples permiten comparar el rendimiento de operaciones básicas y funciones nativas, en escenarios reales siempre se priorizará el enfoque que brinde el mejor rendimiento posible.

En este benchmark se utilizarán dos servidores distintos. Al igual que en el primer benchmark, el script se encargará de levantar ambos servidores localmente, permitiendo así procesar las peticiones de forma controlada y reproducible. Además de permitir colocar un número de peticiones y un delay para simular un proceso de ejecución largo en el servidor.

Las métricas evaluadas en este benchmark serán:

- Tiempo de inicialización del worker
- Tiempo promedio por petición
- Tiempo total de ejecución
- Total de peticiones
- Último valor recibido

4.5.5. Cálculo y verificación de integridad en datos sensibles

Este tema tiene el objetivo de recopilar el rendimiento de las librerías criptográficas en ambos lenguajes, esto nos ayudará a tener en cuenta su eficacia en ámbitos como la ciberseguridad. Para ello, se tuvo en cuenta las siguientes métricas evaluadas:

- Tiempo de inicialización del worker
- Tamaño del archivo/Tamaño del mensaje
- Número de simulaciones
- Tiempo promedio de Hash [48]/encriptado
- Tiempo promedio de verificado/desencriptado
- Tiempo total de ambas operaciones
- Último valor recibido
- Tiempo total de ejecución

4.5.6. Reconocimiento de patrones y clasificación de datos

En la actualidad, la inteligencia artificial se ha convertido en un tema de gran relevancia y presencia constante en múltiples ámbitos. Por este motivo, se optó por desarrollar un benchmark centrado en este entorno, con el objetivo de comparar distintas herramientas disponibles y analizar su rendimiento. Para ello, se utilizó un modelo previamente diseñado, que fue entrenado manualmente empleando el mismo conjunto de datos que se evaluaría durante las pruebas.

Las métricas usadas fueron:

- Tiempo de inicialización del worker
- Tiempo de entrenamiento del modelo
- Tiempo de interferencia (Tiempo en hacer las predicciones)
- Precisión
- Tamaño del modelo
- Tiempo total de ejecución

5. Marco comparativo

Este capítulo presenta las especificaciones usadas y las hipótesis iniciales para comparar PyScript, Pyodide y JavaScript.

Antes de realizar los experimentos, se establecerán hipótesis iniciales. Estas hipótesis se validan o ajustan mediante las propias pruebas, lo que permitirá obtener conclusiones fundamentadas sobre las capacidades de PyScript, Pyodide y JavaScript.

Para garantizar una evaluación precisa, los experimentos se podrán ejecutar como antes mencionado, tanto en local como mediante el framework de Astro, aunque para poder lograr la máxima precisión sin interferencias, se priorizo usar el entorno local.

Especificaciones entorno local

- **Hardware**
  - Procesador (CPU) [49] → 13th Gen Intel(R) Core(TM) i5-13400F 2.50 GHz
    - Núcleos: 10
    - Threads: 16
  - Memoria RAM [50] → 32,0 GB 6000 MHz CL 36 - DDR5
  - Almacenamiento → 2TB NVMe PCIe Gen4 - Disco Duro M.2 SSD
  - Tarjeta gráfica (GPU) → RTX 4070 Windforce OC 12GB GDDR6X DLSS3
- **Software**
  - Sistema operativo → Windows 11 Home ARM-64 bits
  - Intérprete de Python → Python 3.12.4
  - Node.js → v22.1.0
  - Navegador → Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/131.0.0.0 Safari/537.36 OPR/116.0.0.0 (Edition std-1)
  - Editor de código → Visual Studio Code
  - Astro → 5.7.9
  - Tailwind CSS → 4.1.4
  - PyScript → 2024.5.1

Ámbito	PyScript	JavaScript
Cálculos matemáticos intensivos	WebAssembly proporciona aceleración, pero sigue siendo menos eficiente que la ejecución nativa en JS.	Las optimizaciones del motor V8 [51] y la compilación JIT [52] permiten una ejecución más rápida y directa.
Procesamiento de grandes volúmenes de datos	Bibliotecas como *pandas* y *NumPy* ofrecen estructuras muy optimizadas para análisis de datos.	Manejo eficiente menor de las operaciones básicas sobre gran volumen de datos..
Carga y representación de gráficos complejos	*Matplotlib* y *Plotly* pueden ser potentes, pero menos eficientes en el navegador.	*Plotly* y *Canvas* son altamente optimizados para la visualización en la web.
Manejo de múltiples solicitudes concurrentes	*Con fetch + asyncio.gather,* se pueden realizar solicitudes concurrentes eficientemente.	*Promise.all* y los *Web Workers* ofrecen una concurrencia más nativa y eficiente en entornos web.
Cálculo y verificación de integridad en datos sensibles	*PyCryptodome* permite cálculos de hash y cifrado con el mismo rendimiento que JavaScript	*Web Crypto API* es eficiente y está optimizada para navegadores.
Reconocimiento de patrones y clasificación de datos	*sklearn* puede funcionar igual que en JS	*TensorFlow.js* ejecuta modelos optimizados dentro del navegador mejor que PyScript.

Tabla 1. Hipótesis inicial PyScript vs JavaScript.

Las hipótesis están basadas en el comportamiento esperado de cada tecnología dentro del entorno del navegador. PyScript permite ejecutar código Python directamente en la web mediante WebAssembly, lo que posibilita el uso de bibliotecas científicas populares como NumPy, pandas o scikit-learn. Muchas de estas bibliotecas están escritas internamente en C, lo que significa que, pese a la sobrecarga inicial de cargar el entorno de ejecución, ciertas operaciones pueden alcanzar un rendimiento muy alto, en algunos casos comparable o incluso superior al de sus equivalentes en JavaScript, especialmente en tareas matemáticamente intensivas o bien vectorizadas.

Sin embargo, esta ventaja potencial suele verse mitigada por la sobrecarga que implica cargar e inicializar el entorno de ejecución de Python en el navegador. PyScript utiliza WebAssembly para ejecutar intérpretes como Pyodide, lo que requiere descargar varios megabytes de archivos .wasm y módulos de Python precompilados. Este proceso no solo

incrementa significativamente el tiempo de carga inicial, sino que también consume más memoria y ralentiza la respuesta inicial de la aplicación. A diferencia de JavaScript, PyScript necesita establecer una capa adicional de interpretación que introduce latencias antes de poder ejecutar cualquier lógica de usuario.

Por su parte, JavaScript ha sido diseñado específicamente para la web, y su ecosistema está fuertemente optimizado para este entorno. Motores como V8 aprovechan años de mejoras en compilación JIT, y herramientas como Promise.all, Web Workers o la Web Crypto API permiten una ejecución rápida, asincrónica y segura. Además, en visualización, bibliotecas como Plotly.js, D3 o Canvas están estrechamente integradas con la plataforma del navegador, lo que proporciona una experiencia más fluida que las versiones adaptadas desde Python.

En general, aunque PyScript permite acercar herramientas científicas al entorno web, JavaScript sigue siendo más eficiente en la mayoría de casos por su integración nativa y optimización directa para navegadores.

5.1. Herramientas comparativas

Durante el desarrollo se llevaron a cabo mediciones de distintas métricas, seleccionadas en función de los objetivos específicos de cada prueba. En los siguientes apartados se explicará de forma breve qué herramientas se utilizaron para realizar dichas mediciones y cómo funcionan.

5.1.1. PyScript / Python

gc.collect

En las implementaciones con PyScript y Python, realizamos el uso de una función llamada *gc.collect(),* que forma parte de su sistema de "recolección de basura" (garbage collection). Básicamente, Python a veces guarda en memoria cosas que ya no se están usando, como variables antiguas u objetos que ya no necesitamos. Esto puede afectar a los resultados del benchmark, ya que podría parecer que se está usando más memoria o que el código es más lento de lo que realmente es.

Para evitar eso, justo antes de comenzar cada prueba, usamos gc.collect() para decirle a Python: “Limpia todo lo que ya no se está usando”. De esta forma, empezamos cada medición desde cero, con la memoria lo más limpia posible, y así los resultados son más precisos y reales.

time.perf_counter()

Para medir los tiempos, utilizamos una función llamada time.perf_counter(), que nos sirve para medir el tiempo con mucha precisión. Es un cronómetro muy exacto que nos dice cuánto tiempo pasa desde que empezamos a contar hasta que terminamos una tarea. Es

ideal para estas pruebas porque mide el tiempo real transcurrido y tiene una resolución muy alta, lo que significa que puede detectar incluso cambios muy pequeños en el tiempo. Así, podemos obtener resultados fiables y detallados para nuestro benchmark.

Además cabe destacar, que esta función tiene como descripción la siguiente:

“Performance counter for benchmarking.”

tracemalloc

Para medir el uso de memoria en nuestro benchmark usamos *tracemalloc*, una herramienta de Python que nos ayuda a saber cuánta memoria está utilizando nuestro programa en un momento dado. De este módulo, utilizamos las siguientes funciones:

- **tracemalloc.start()**: Es el inicio para que el contador empiece a rastrear cuánta memoria se está usando
- **tracemalloc.get_traced_memory():** Nos proporciona una tupla [53] de dos números: la cantidad de memoria que se está usando en ese instante y la cantidad máxima que se ha usado desde que arrancamos el contador.
- **tracemalloc.stop()** apaga el contador cuando ya no necesitamos medir más.

Esta herramienta es ideal para PyScript porque nos da una forma sencilla y directa de medir la memoria en un entorno donde no tenemos acceso a herramientas externas o al sistema operativo.

En cambio, en el servidor Python usamos la librería *psutil*, que es una herramienta mucho más potente y completa para monitorear recursos del sistema operativo. Con *psutil* podemos acceder directamente a la memoria usada por el proceso de Python desde el punto de vista del sistema operativo, incluyendo información precisa sobre el uso de memoria física y virtual.

Por eso, para medir memoria en PyScript usamos *tracemalloc* (porque es lo que tenemos disponible y funciona bien en el navegador) y en el servidor usamos *psutil* (porque podemos acceder a datos más completos y reales del sistema operativo).

psutil

En el servidor, para medir el uso de memoria y CPU de nuestro programa Python, utilizamos la librería *psutil*, que proporciona acceso directo a información detallada del sistema operativo sobre los procesos en ejecución.

Para la memoria, usamos la función *process.memory_info().rss*, que devuelve la cantidad de memoria física (RAM) que está utilizando el proceso Python en bytes. Esta métrica refleja la memoria residente real en el sistema, es decir, la memoria que el proceso ocupa efectivamente en RAM, lo que permite conocer con precisión el consumo de recursos del programa.

Para medir el uso de CPU, empleamos *process.cpu_percent(interval=None)*. Esta función devuelve el porcentaje de CPU que el proceso ha utilizado desde la última vez que se llamó a esta función. Por eso, para obtener una medición precisa y estable del uso de CPU, es recomendable llamar primero a *process.cpu_percent(interval=None)* para inicializar la

medición, luego esperar un intervalo de tiempo (por ejemplo, 0.1 o 0.5 segundos) y finalmente llamar de nuevo a *process.cpu_percent(interval=<segundos>)* para obtener el valor real durante ese intervalo. Esto se debe a que si se llama con interval=None por primera vez, devuelve 0.0 porque no hay datos previos para calcular la diferencia.

En resumen, el flujo correcto para medir el uso de CPU con psutil es:

- Inicializar la medición con process.cpu_percent(interval=None).
- Esperar un breve intervalo de tiempo (usando time.sleep() o realizando operaciones durante ese tiempo).
- Llamar a process.cpu_percent(interval = <*segundos*>) (o el tiempo que se considere) para obtener el porcentaje de CPU utilizado en ese intervalo.

5.1.2. JavaScript / Node.js

En JavaScript no existe un equivalente directo a *gc.collect()* que permita forzar explícitamente la recolección de basura. Esto es debido a que a diferencia de PyScript y Python, la recolección de JavaScript es automática y gestionada por el motor del propio lenguaje. Esto es gracias a las optimizaciones de los motores nativos de JavaScript (Sucede tanto en el navegador como en Node.js).

Performance.now()

Para medir con precisión el tiempo de ejecución de diversas operaciones se utilizó la función *performance.now()*, disponible tanto en navegadores como en entornos de servidor con Node.js (a través del módulo *perf_hooks*).

performance.now() devuelve el tiempo transcurrido en milisegundos con alta precisión desde un punto fijo de referencia, conocido como "tiempo de navegación" en navegadores o "origen de tiempo monotónico" en Node.js. Esto permite medir intervalos de tiempo de manera mucho más exacta que métodos tradicionales como *Date.now()*.

Mientras que *Date.now()* solo ofrece resolución en milisegundos enteros, *performance.now()* puede medir hasta fracciones de milisegundo (microsegundos), lo cual es esencial para benchmarks donde cada décima de milisegundo cuenta.

Performance.memory.usedJSHeapSize

Para medir el uso de memoria en JavaScript, especialmente en el navegador, se usó *Performance.memory.usedJSHeapSize*. Esta propiedad indica la cantidad de memoria en bytes que está siendo utilizada actualmente en el heap de JavaScript, es decir, la memoria que el motor JS emplea para almacenar objetos y datos en tiempo de ejecución.

mo el valor está en bytes, para facilitar su comprensión se convierte a megabytes dividiendo entre 1024 dos veces (1024 * 1024), ya que 1 megabyte equivale a 1,048,576 bytes.

En el código se comprueba primero si la propiedad performance.memory está disponible, porque no todos los navegadores o entornos la soportan. Si está disponible, devuelve el valor convertido a megabytes; si no, devuelve un valor indicativo (-1) que señala que la medición no se pudo realizar.

process.memoryUsage().heapUsed

En el entorno de Node.js, es posible acceder a métricas detalladas sobre el uso de memoria del proceso actual a través de la función process.memoryUsage(). Esta función devuelve un objeto con varias propiedades relacionadas con el consumo de memoria en diferentes áreas del motor de ejecución de JavaScript (V8). Esta función en PyScript para hacernos una idea, es el equivalente a *psutil.memory_info().rss,* la función que utilizamos en el servidor de Python

El valor de *heapUsed* representa la cantidad de memoria (en bytes) que actualmente está siendo utilizada activamente por los objetos JavaScript del programa, es decir, aquellos que están dentro del heap gestionado por el recolector de basura de V8. Para convertir este valor a megabytes, simplemente se divide entre 1024 * 1024.

cpu.usage()

Para obtener el uso de CPU en el servidor cuando se ejecuta un benchmark con Node.js, se utiliza la librería externa *node-os-utils*, que proporciona acceso a métricas del sistema operativo, como el consumo de CPU de una forma sencilla. En concreto, la función *cpu.usage()* nos permite conocer el porcentaje de uso total de la CPU en el momento de realizar la medición.

Este valor se captura una vez finalizada la operación principal del benchmark al igual *process.memoryUsage().heapUsed*, como puede ser la multiplicación de matrices, con el fin de evaluar cuánta carga de CPU ha generado el cálculo. La elección de medir al final (en lugar de de forma continua) se hace para obtener una referencia sencilla y práctica del impacto de la operación en el rendimiento general del servidor.

5.2. Resultados previos

Antes de entrar en el detalle de cada benchmark, estas son las conclusiones clave de la comparación entre JavaScript y PyScript en navegador:

- Rendimiento general: JavaScript (V8+JIT) es el más rápido en algoritmos y gráficos, mientras que PyScript (Pyodide/WebAssembly) tiene un arranque más lento y no accede a aceleración nativa.
- Procesamiento de datos: PyScript sobresale en tareas de datos masivos gracias a NumPy/Pandas, superando a JavaScript cuando la carga se distribuye (Web Workers), pese a su inicialización costosa.
- Gráficos y visualización: JavaScript domina al usar *Canvas, SVG y WebGL* [54] directamente; PyScript depende de llamadas intermedias a JS, lo que penaliza su fluidez e independencia.
- Concurrencia: En *fetch/WebSocket* [55] ambos son comparables; JS aprovecha el event loop y Web Workers con bajo overhead, PyScript coopera vía Pyodide+JS, competitivo sólo en cargas ligeras.
- Criptografía y clasificación: JavaScript usa *SubtleCrypto* y *ml-knn* en WASM con optimizaciones hardware, por lo que supera ampliamente a PyScript en hashes, cifrado y KNN en cargas elevadas.

Por otra parte, en base a los resultados, se llegó a la conclusión de que PyScript resulta útil únicamente en dos entornos muy concretos:

- Sistemas con recursos de memoria muy limitados donde se pueda tolerar un mayor tiempo de ejecución a cambio de un consumo más moderado de RAM.
- Procesamiento intensivo de grandes volúmenes de datos, siempre que la carga inicial del intérprete Pyodide sea asumible y se aprovechen las estructuras vectorizadas de NumPy/Pandas para obtener un rendimiento global eficiente.

Estas conclusiones rápidas sirven como guía para entender, de un vistazo, las fortalezas y limitaciones de cada tecnología antes de revisar los resultados detallados de cada prueba.

6. Implementación y Benchmarking

En este capítulo se presenta, de forma resumida y con apoyo visual mediante capturas de pantalla, la implementación de la página web desarrollada. Asimismo, se describen los experimentos diseñados para comparar el rendimiento de PyScript y Pyodide frente a JavaScript en distintos escenarios representativos del desarrollo web. A través de estos benchmarks, se evalúan aspectos clave como la velocidad de ejecución y el comportamiento en diferentes entornos de ejecución.

6.1. Página Web

Para el desarrollo de la interfaz que centraliza y presenta los distintos benchmarks realizados, se ha optado por utilizar el framework Astro, junto con el sistema de estilos Tailwind CSS. Esta combinación tecnológica ha sido seleccionada tras valorar las necesidades específicas del proyecto, como el rendimiento, la modularidad, la facilidad de mantenimiento y la rapidez de desarrollo.

Astro es un framework emergente orientado a la creación de sitios web estáticos, que pone un énfasis especial en la optimización del rendimiento y la experiencia del usuario. A diferencia de otros frameworks tradicionales que envían una gran cantidad de JavaScript al cliente para gestionar la interactividad, Astro adopta un enfoque radicalmente distinto: por defecto, elimina todo el JavaScript innecesario en el cliente. Solo carga en el navegador el código estrictamente necesario para aquellas partes de la página que requieren interactividad, lo que resulta en sitios web mucho más rápidos y livianos. En nuestro caso, queremos minimizar el uso de este JavaScript innecesario y solo tener en cuenta lo esencial para la interactividad de los benchmarks y de la propia página web

6.1.1 Tailwind CSS: eficiencia y flexibilidad en el diseño

Para la capa visual, se ha decidido usar Tailwind CSS, un framework CSS basado en clases utilitarias que facilita la creación rápida de interfaces limpias y consistentes. A diferencia de los sistemas tradicionales basados en hojas de estilo CSS extensas y personalizadas, Tailwind permite construir componentes visuales directamente en el HTML o archivos .astro mediante clases predefinidas, que controlan propiedades como colores, márgenes, paddings, tipografía, y mucho más.

El uso de Tailwind ha supuesto una importante ventaja para el desarrollo, se poseía experiencia previa con esta herramienta, lo que ha acelerado considerablemente la fase de diseño y prototipado visual. Además, la naturaleza modular de Tailwind ayuda a mantener el código CSS final reducido y limpio, lo que contribuye a mejorar el rendimiento general del sitio.

```
<a class="focus: inline-flex scale-90 cursor-pointer flex-row
items-center justify-center gap-x-2 rounded-lg px-5 py-2.5 text-center
font-medium text-white opacity-70 transition-all duration-200 ease-in-out
```

```
outline-none hover:scale-110 hover:bg-slate-700 hover:text-purple-400
hover:opacity-100 hover:shadow-lg focus:ring focus:ring-purple-400"
    href={ref}
>
    <slot />
</a>
```

Código 3. Ejemplo de uso de Tailwind CSS.

Astro facilita la incorporación de Tailwind a través de su sistema de configuraciones y plugins, permitiendo que el proceso de construcción genere estilos optimizados y aplicados sólo cuando son realmente utilizados en el proyecto. Esto asegura que la cantidad de CSS enviada al cliente sea mínima, mejorando los tiempos de carga.

Por otro lado, Astro al tratar las páginas como componentes y permitir la reutilización de fragmentos de código (como layouts, headers, y footers), facilita aplicar de manera coherente y homogénea los estilos definidos con Tailwind en toda la aplicación.

6.1.2 Estructuración

El proyecto web se estructura siguiendo la filosofía de Astro, donde cada página se representa mediante un archivo .astro dentro de la carpeta *src/pages*. Astro mapea automáticamente estos archivos a rutas accesibles por el usuario, por lo que, por ejemplo, un archivo *src/pages/benchmarks.astro* se podrá consultar desde /benchmarks en el navegador sin necesidad de configuración adicional. Esto simplifica la gestión del enrutamiento y permite añadir o modificar páginas con solo crear o editar archivos.

Además, Astro permite definir rutas dinámicas mediante nombres de archivo con corchetes, como [id].astro, que permiten renderizar páginas con parámetros variables, muy útil para mostrar detalles de benchmarks o pruebas específicas sin duplicar código, lo cual se ha aplicado tanto para ver las diferentes pruebas de los benchmarks, como para ver sus diferentes implementaciones, si agregar un archivo a cada uno.

Para mantener el proyecto organizado y escalable, se han creado componentes reutilizables para elementos comunes de la interfaz, como cabeceras (Header.astro), tarjetas de contenido (CardHeader.astro), botones, y menús de navegación. Estos componentes permiten centralizar la lógica visual y funcional, facilitando el mantenimiento y la coherencia estética. Esto teniendo en cuenta que sigue el mismo método de funcionamiento que Frameworks como React, donde se define el componente en *src/components/...* y se exporta, de ese modo en nuestras páginas podemos importarlo y utilizarlo.

Por otro lado, Astro nos proporciona la reutilización mediante *props* y *slots,* el primero son variables o datos que nos permitirán cambiar dinámicamente el funcionamiento del componente, en el caso más fácil implementado, fue la reutilización de un enlace, donde el link fue introducido mediante los props. Por otro lado, los *slots* son aquellas etiquetas propias de HTML que podemos introducir en el componente, de esa forma, es posible

enseñar elementos dentro de los componentes dinámicamente. En el ejemplo del código anterior, se enseña esta etiqueta *slot* que representa la primera etiqueta heredada.

```
<NavegationButton ref={versionLink }>
      <slot name="run"> Ir a la versión</slot>
</NavegationButton>
```

Código 4. Ejemplo de uso de los slots..

Gracias a la naturaleza estática y optimizada de Astro, el sitio web resultante puede ser desplegado fácilmente en cualquier servicio de hosting estático, como Netlify, Vercel, GitHub Pages, o incluso servidores propios. El proceso de construcción (astro build) genera archivos HTML, CSS y JavaScript minimalistas, preparados para servir con máxima rapidez.

Adicionalmente, el sitio puede integrar funcionalidades más dinámicas o interactivas usando componentes que cargan JavaScript solo cuando es necesario. Esto es ideal para los benchmarks que requieren mostrar gráficos, formularios o ejecutar código en el navegador sin sacrificar el rendimiento inicial.

Astro posee diversas formas de definir la configuración para un proyecto, adaptándose a diferentes necesidades y escenarios de desarrollo. En nuestro caso, se ha optado por una configuración que permite aprovechar al máximo las características modernas del framework, al mismo tiempo que se habilitan funcionalidades clave para nuestro proyecto, como el uso de Web Workers, el soporte para peticiones WebSocket y el estilo con Tailwind CSS.

Para ello, se utiliza un archivo de configuración basado en la API moderna de Astro, en el que se define la salida del proyecto como un servidor *(output: 'server')* y se emplea el adaptador oficial para Node.js *(@astrojs/node)* con el modo standalone. Esta elección responde a la necesidad de ejecutar un servidor backend junto con la aplicación, lo que nos permite habilitar las peticiones del benchmark 4, especialmente las conexiones WebSocket.

Además, se ha añadido un middleware personalizado que modifica las cabeceras HTTP de las respuestas, configurando explícitamente las políticas de seguridad y acceso necesarias para habilitar ciertas funcionalidades avanzadas del navegador, concretamente:

- Se establecen cabeceras como *Cross-Origin-Opener-Policy* y *Cross-Origin-Embedder-Policy* para cumplir con los requisitos de seguridad que permiten el uso de *SharedArrayBuffer*, la caracteristica para manejar Web Workers que permite que compartan memoria en el contexto del navegador. Sin esta configuración, el uso de Web Workers para ejecutar código Python (PyScript) y el manejo paralelo de benchmarks podría verse restringido y no funcionar correctamente.
- Se habilita *CORS (Cross-Origin Resource Sharing)* permitiendo el acceso desde cualquier origen, lo cual es requerido para que el frontend pueda comunicarse sin problemas con el backend, que en este caso también se ejecuta dentro del mismo

proyecto Astro pero en un entorno servidor Node.js independiente. Esto asegura que las peticiones para WebSockets y APIs no se vean bloqueadas por políticas de seguridad del navegador, garantizando la interoperabilidad necesaria para el benchmarking.

Para complementar esta configuración, el proyecto incluye un archivo *server.mjs* que se ejecuta como servidor backend Node.js. Este servidor es responsable de gestionar las conexiones WebSocket que permiten la comunicación en tiempo real entre el cliente y el servidor, La integración de este servidor backend con la aplicación Astro se realiza de forma coordinada a través de los scripts definidos en el package.json, de tal manera que al ejecutar *npm run dev*, se mandará a ejecutar a su vez dicho servidor.

6.1.3 Enrutamiento

Como se indicó anteriormente, el sistema de enrutamiento en Astro permite tanto rutas estáticas como dinámicas. Dado que el proyecto incluye múltiples ámbitos y versiones de pruebas, se optó por un enfoque dinámico para esta sección, con el objetivo de optimizar el uso de recursos y reutilizar componentes, evitando así la creación de archivos redundantes.

El proyecto se compone de un total de cuatro vistas principales. En todas ellas se emplea un componente de encabezado (Header), que puede variar o mantenerse igual según la vista, y un pie de página (Footer) que se mantiene constante en toda la aplicación. Cada vista reutiliza diversos componentes previamente implementados, lo que favorece la escalabilidad y el mantenimiento del proyecto a futuro.

La primera vista corresponde a la página de inicio (index). Esta ha sido diseñada con un encabezado y pie de página adecuados para ofrecer una presentación clara desde el primer acceso. En esta sección se expone la información general del proyecto, incluyendo una introducción, una descripción del proyecto, un resumen de los benchmarks realizados y una presentación breve de los resultados y conclusiones obtenidos.

Figura 14. Imagen de la página principal del proyecto Astro.

La siguiente vista implementada corresponde a la página que agrupa los enlaces a los distintos benchmarks. Esta vista no presenta elementos destacables, ya que se limita a reutilizar uno de los componentes presentes en la página de inicio. A diferencia de la vista principal, está únicamente incluye dicho componente específico, acompañado del encabezado (Header) y el pie de página (Footer).

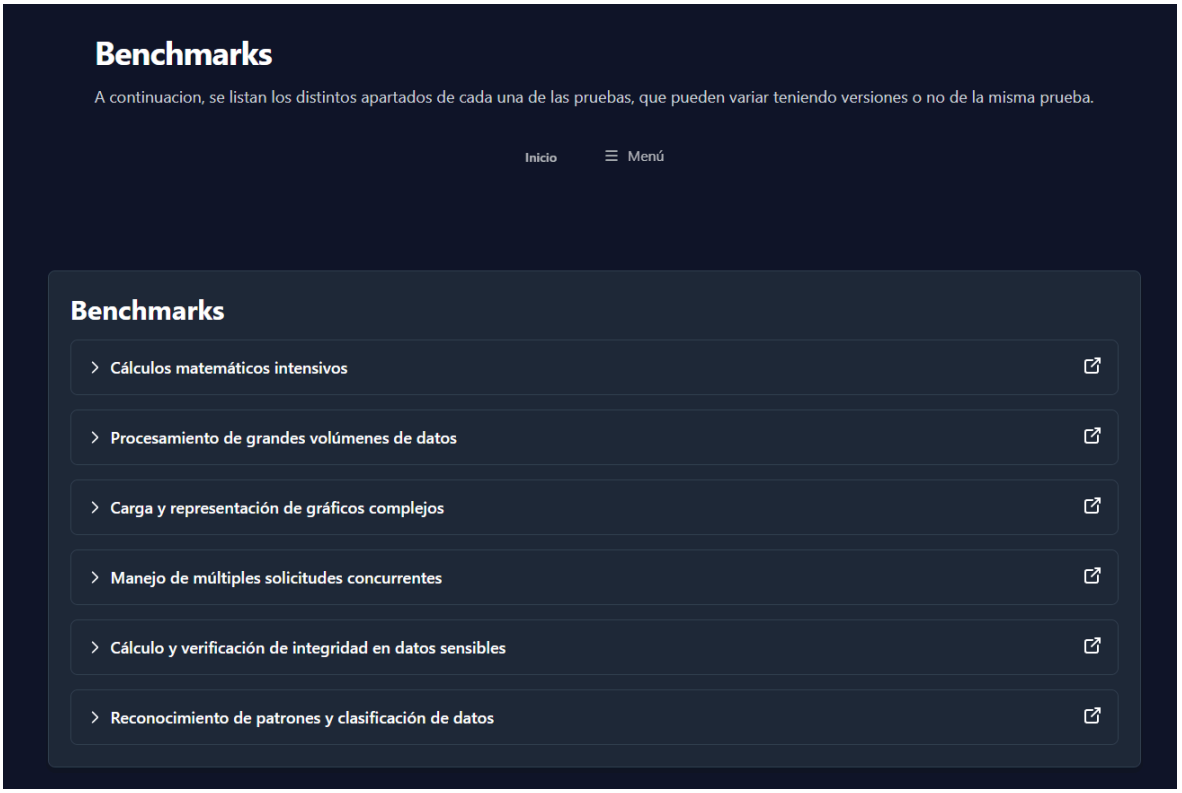

Figura 15. Imagen de la página de benchmarks del proyecto Astro.

La penúltima vista a destacar es la que muestra el listado de las distintas pruebas y sus respectivas implementaciones dentro de un ámbito específico. En esta vista se presentan los títulos, descripciones y explicaciones tanto del ámbito seleccionado como de cada prueba y sus versiones. Además, se detallan las diferencias entre las distintas versiones de una misma prueba, proporcionando al usuario una visión clara y comparativa.

Figura 16. Imagen de la página del listado de pruebas del primer ámbito del proyecto Astro.

Finalmente, la página principal del proyecto es aquella en la que se ejecutan las distintas implementaciones de todas las pruebas desarrolladas. Esta vista destaca por su comportamiento dinámico, adaptándose automáticamente según el diseño de cada prueba: pueden mostrarse u ocultarse botones, campos de entrada, filas y columnas en la tabla de resultados, e incluso gráficos, en función de los requisitos específicos de cada versión. Todo esto es posible gracias a la flexibilidad que ofrece Astro para construir sus vistas dinámicas y personalizadas.

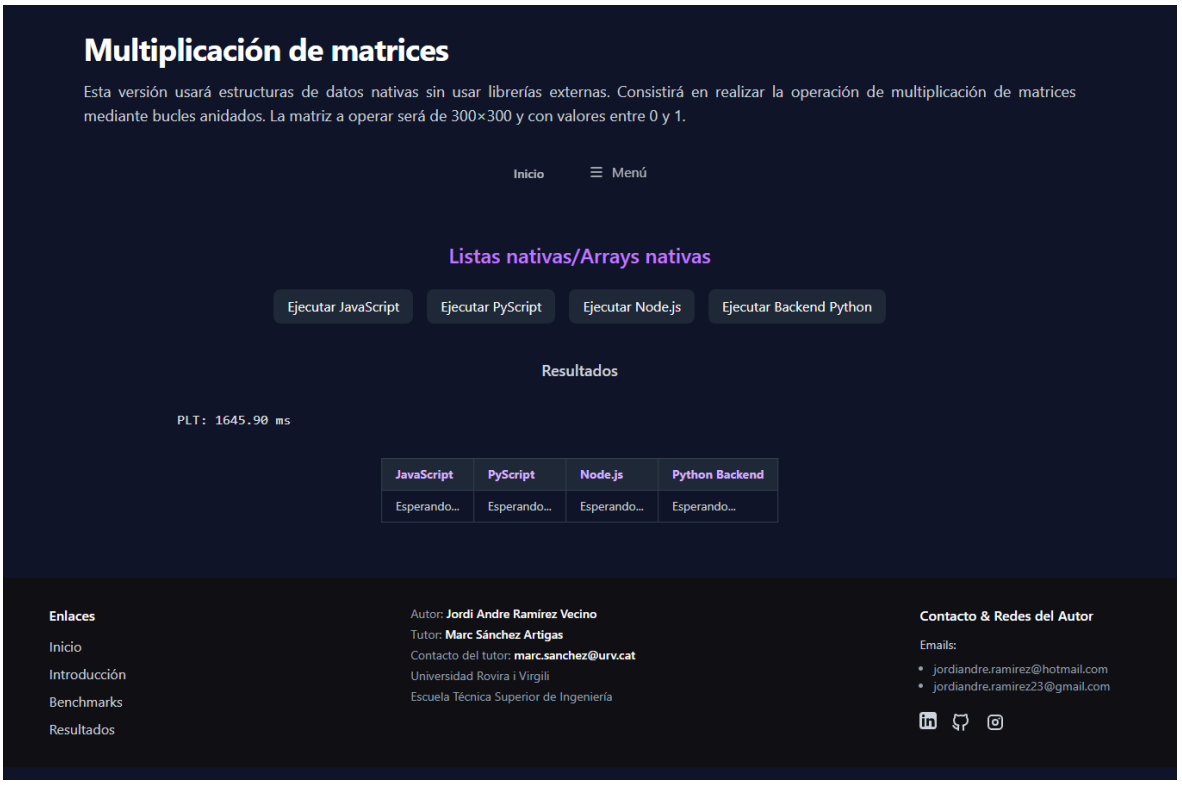

Figura 17. Imagen de la página principal de la implementación de la prueba.

6.1.3 API

Uno de los elementos clave en el diseño del portal para los benchmarks ha sido la implementación de una API propia dentro del entorno de Astro. Esta API no solo actúa como interfaz entre el frontend y el backend, sino que además permite simular condiciones específicas de prueba, gestionar conexiones WebSocket y ejecutar procesos en segundo plano, todos ellos para la evaluación del rendimiento de los distintos lenguajes y tecnologías.

6.1.3.1. Simulación de peticiones asincrónicas: /4.1.1/[delay].ts

La primera parte de la API está pensada para simular peticiones asincrónicas que introducen una latencia artificial controlada. Este comportamiento se implementa en el endpoint *api/4.1.1/[delay].ts*, que permite al usuario configurar, a través de la URL, el retardo que debe simular el servidor antes de responder.

El propósito principal de este endpoint es servir como backend para las pruebas relacionadas con el benchmark 4.1, en el cual se evalúa la capacidad del sistema cliente (ya sea en PyScript o JavaScript) de gestionar múltiples peticiones concurrentes con distintos niveles de latencia. Cada llamada a esta ruta incrementa un contador de solicitudes activas, que si es voluntad del usuario, permite mostrar las peticiones concurrentes posibles, en este caso, nuestro proyecto máximo permite 6 peticiones concurrentes. Además respecto al código, el *[delay]* es lo que comentamos previamente, un parámetro en la URL que nos

permite cambiar el valor del delay, una vez obtenido, ejecuta un setTimeout por el tiempo indicado y luego responde con un pequeño paquete de datos aleatorios, simulando una respuesta real de un servidor bajo carga.

6.1.3.2. Gestión segura de WebSocket: /4.2.1/socket.ts

Para el benchmark 4.2, centrado en la evaluación del rendimiento de comunicaciones persistentes en tiempo real mediante WebSockets, se ha desarrollado un segundo endpoint: *api/4.2.1/socket.ts*.

Este archivo actúa como un intermediario o "proxy seguro" entre el cliente y el verdadero servidor WebSocket (*server.mjs*). En lugar de exponer directamente las rutas y parámetros internos del servidor WebSocket, este endpoint genera dinámicamente una URL con parámetros codificados (como delay o id) y la devuelve al cliente. De esta forma, se abstrae la lógica del backend, ofreciendo una interfaz más sencilla, modular y segura para establecer conexiones. Permitiendo tener mantenimiento y escalabilidad para el servidor.

6.1.3.3. Ejecución de scripts como procesos hijo: /api/run-backend.ts

Una de las pruebas del sistema es la ejecución de scripts externos (en Python o Node.js) como si fueran servicios independientes o pequeños servidores. Esta funcionalidad se encapsula en el endpoint */api/run-backend.ts*, el cual permite lanzar procesos hijo desde el servidor Astro.

Al recibir una petición POST, este endpoint extrae el tipo de script (Python o Node), localiza su ruta dentro del proyecto y lo ejecuta usando el módulo *child_process* de Node.js. La salida del script es recogida, analizada y devuelta al cliente como JSON. Este método resulta ideal para simular los benchmarks en los servidores, para que de esta forma sea posible simular su rendimiento en un backend profesional.

Además, al separar esta lógica en procesos hijo, se evita bloquear el hilo principal del servidor, se mejora la seguridad y se garantiza la posibilidad de escalar a múltiples ejecuciones concurrentes sin afectar al rendimiento general del sistema.

Cada experimento estará orientado a un ámbito específico y medirá distintas métricas relevantes dentro de ese contexto. Cabe destacar que, en todos los experimentos, se priorizará maximizar la similitud entre el código escrito en Python (utilizando PyScript o Pyodide) y el código en JavaScript. Esto significa que se buscará escribir el mismo código y de la misma forma en ambos lenguajes, de tal manera que las diferencias observadas se deban únicamente a las características intrínsecas de cada tecnología y no a discrepancias en la implementación.

Además, es importante considerar que, debido a la compilación de los archivos Python a WebAssembly, siempre existirá un tiempo inicial de carga y análisis del código antes de que las pruebas puedan comenzar. Aunque este tiempo no forma parte de las métricas principales de rendimiento, es un factor relevante al evaluar la experiencia del usuario y la eficiencia en entornos donde el tiempo de inicio es crítico.

Para los experimentos, se utilizaron las siguientes métricas:

- ET (Execution Time): Tiempo de ejecución del algoritmo en milisegundos.
- PLT (Page Load Time): Tiempo en milisegundos que tarda en cargarse la página por primera vez debido a PyScript.
- RAM (Random Access Memory): Pico de memoria RAM utilizado durante la ejecución del algoritmo.
- CPU (Central Processing Unit): Porcentaje de uso de la CPU durante la ejecución del algoritmo.
- ET (Py + WA): Tiempo total de la llamada y ejecución de Python, sumada con el tiempo adicional de WebAssembly
- N º E: Número de ejecución.

6.2. Cálculos matemáticos intensivos

Estas pruebas medirán la eficiencia de cada tecnología al ejecutar operaciones matemáticas de alto costo computacional, como transformaciones numéricas, factorizaciones, cálculos algebraicos, entre otros. Con el objetivo principal de evaluar el rendimiento en operaciones típicas que se usarían en el ámbito científico del desarrollo web.

En este primer apartado se definieron 3 principales pruebas las cuales tendrán 2 versiones cada uno, la primera será una implementación realmente no optimizada y la segunda se tratara de implementar con las opciones disponibles para simular un código optimizado en el mundo real. Por otro lado, las implementaciones a parte de ejecutarse en el hilo principal del navegador, se buscará ejecutar en 2 servidores propios, uno de Node.js y otro de Python. De esta forma, podremos tener una vista de ambos lenguajes en diferentes entornos y ver la variación de uso de recursos tanto en el servidor como en el cliente o el navegador.

6.2.1. Prueba 1 - Multiplicación de matrices

Esta prueba consistió en ejecutar multiplicaciones de matrices directamente en el hilo principal, utilizando ambos lenguajes en archivos separados. A lo largo del proyecto se siguió una nomenclatura coherente para los nombres de archivo:

- main.extensión: archivo principal ejecutable desde el navegador/cliente.
- server.extensión: código ejecutado como servidor local.
- app.extensión: servidor de peticiones HTTP o WebSocket.
- worker.extensión: El código que emplean los workers para su ejecución en paralelo.

En la primera versión de esta prueba, se realizaron multiplicaciones de matrices de tamaño 300×300 mediante un algoritmo tradicional basado en bucles for. Los elementos de las matrices eran valores flotantes generados aleatoriamente en el intervalo [0.0, 1.0). En ambos lenguajes, estos valores se almacenan internamente el estándar de doble precisión, es decir, con 1 bit para el signo, 11 bits para el exponente y 52 bits para la fracción.

Se emplearon únicamente estructuras nativas y bibliotecas estándar con tal de hacer un uso mínimo, de esta forma se usaron para obtener los datos aleatorios: en Python se utilizó el módulo random, y en JavaScript, la función Math.random().

En la segunda versión de la prueba se emplearon herramientas optimizadas para maximizar el rendimiento de las operaciones. En el caso de Python, se utilizó NumPy tanto para la generación como para la multiplicación de matrices, aprovechando su capacidad de vectorización y ejecución eficiente en bajo nivel. Por su parte, en JavaScript se utilizó TensorFlow.js, una librería que permite realizar operaciones tensoriales de forma eficiente y que, además, puede beneficiarse de aceleración por hardware cuando está disponible.

Asimismo, se modificó el tamaño de las matrices, pasando de un único caso de 300×300 a tres nuevos tamaños más exigentes: 500×500, 1000×1000 y 2000×2000, con el objetivo de observar el comportamiento de las herramientas bajo distintas cargas. A pesar de estos cambios, se mantuvo el mismo rango de valores en las matrices, generando números de punto flotante en el intervalo [0.0, 1.0).

Versión 1: Listas nativas/Arrays nativas

Nº	Lenguaje	Entorno	PLT	RAM (MB)	ET (ms)	CPU (%)
1	Python	Navegador	1739.60 ms	5.19 MB	13455.0 ms WA:13523.60 ms	
	Python	Servidor		10.52 MB	2134.79 ms	0%
	JavaScript	Navegador		109.92 MB	48.9 ms	
	JavaScript	Servidor		0.83 MB	1150.51 ms	1.43 %
2	Python	Navegador	2285.80 ms	5.19 MB	15490.0 ms WA:15556.00 ms	
	Python	Servidor		10.5 MB	1212.42 ms	0%
	JavaScript	Navegador		105.63 MB	43.8 ms	
	JavaScript	Servidor		0.82 MB	1060.12 ms	1.26%
3	Python	Navegador	1520.70 ms	5.19 MB	13126.0 ms WA:13197.80 ms	
	Python	Servidor		6.07 MB	1233.04 ms	0%
	JavaScript	Navegador		140.34 MB	47.5 ms	

	JavaScript	Servidor		0.44 MB	1049.44 ms	2.82 %

Tabla 2. Resultados cálculos matemáticos: prueba 1-1.

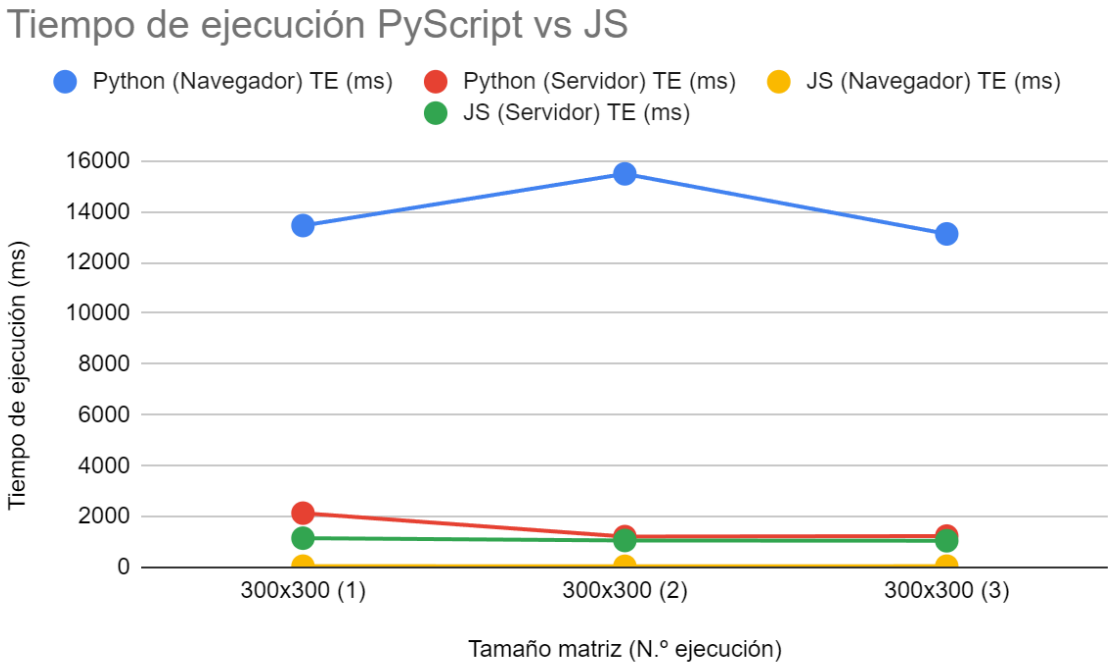

Figura 18. Diagrama de TE de cálculos matemáticos: prueba 1-1.

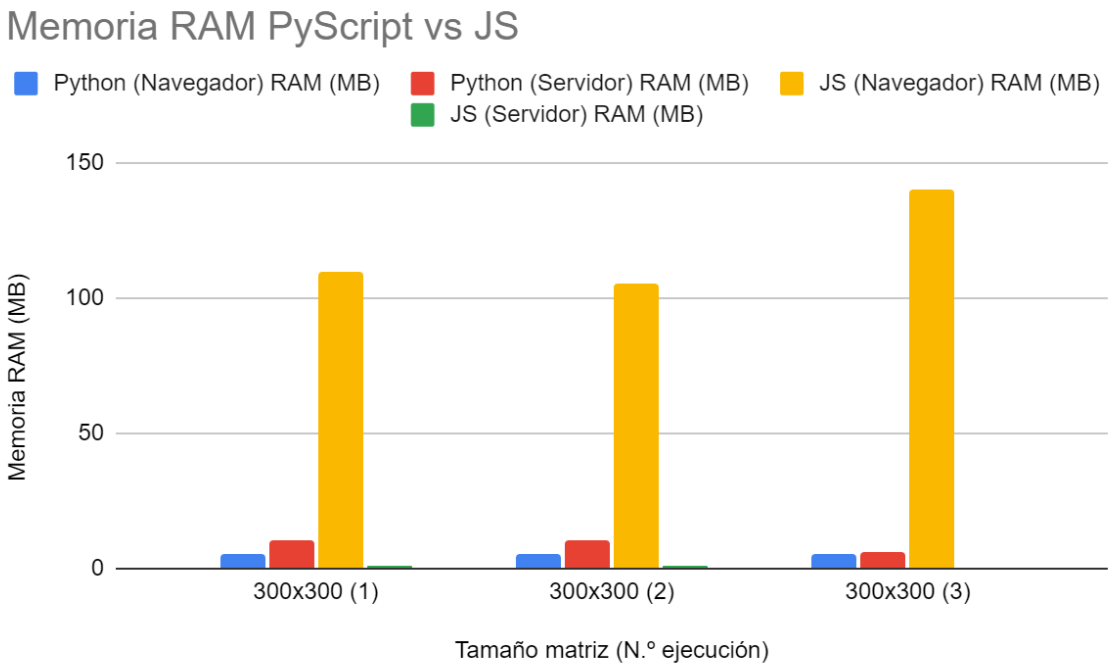

Figura 19. Diagrama de memoria de cálculos matemáticos: prueba 1-1.

Observación 1.
JavaScript es el entorno más rápido para la ejecución en navegador, logrando entre 200 y 300 veces mejor rendimiento que PyScript debido a la optimización de motores JIT como V8.

Observación 2.
PyScript presenta una alta latencia inicial (PLT) por la carga y configuración del entorno WebAssembly, lo que impacta negativamente su tiempo total de ejecución.

Observación 3.
Python ejecutado en servidor ofrece un rendimiento competitivo frente a PyScript, siendo una opción más eficiente para operaciones matemáticas complejas, aunque sigue siendo más lento que JavaScript en servidor.

Observación 4.
El uso de memoria es mayor en JavaScript navegador debido a la asignación dinámica y optimizaciones internas, mientras que Python en navegador mantiene un consumo constante y más bajo.

Versión 2: Numpy/TensorFlow

500x500

Lenguaje	Entorno	PLT	RAM (MB)	ET (ms)	CPU (%)
Python	Navegador	4500.30 ms	5.72 MB	68.0 ms WA:73.30 ms	
Python	Servidor		9.46 MB	5.28 ms	0.2 %
JavaScript	Navegador		207.54 MB	11.8 ms	
JavaScript	Servidor		1.24 MB	1526.36 ms	1.65 %

Tabla 3. Resultados cálculos matemáticos: prueba 1-2-1.

1000x1000

Lenguaje	Entorno	PLT	RAM (MB)	TE (ms)	CPU

					(%)
Python	Navegador	4340.56 ms	22.89 MB	657.0 ms WA:745.00 ms	
Python	Servidor		23.99 MB	42.12 ms	0.6 %
JavaScript	Navegador		190.95 MB	32 ms	
JavaScript	Servidor		1.01 MB	4539.10 ms	1.67 %

Tabla 4. Resultados cálculos matemáticos: prueba 1-2-2.

2000x2000

Lenguaje	Entorno	PLT	RAM (MB)	TE (ms)	CPU (%)
Python	Navegador	4113.48 ms	91.56 MB	22232.0 ms WA:23038.00 ms	
Python	Servidor		93.56 MB	224.6 ms	0.8 %
JavaScript	Navegador		232.94 MB	108.9 ms	
JavaScript	Servidor		1.01 MB	27184.45 ms	6.86 %

Tabla 5. Resultados cálculos matemáticos: prueba 1-2-3.

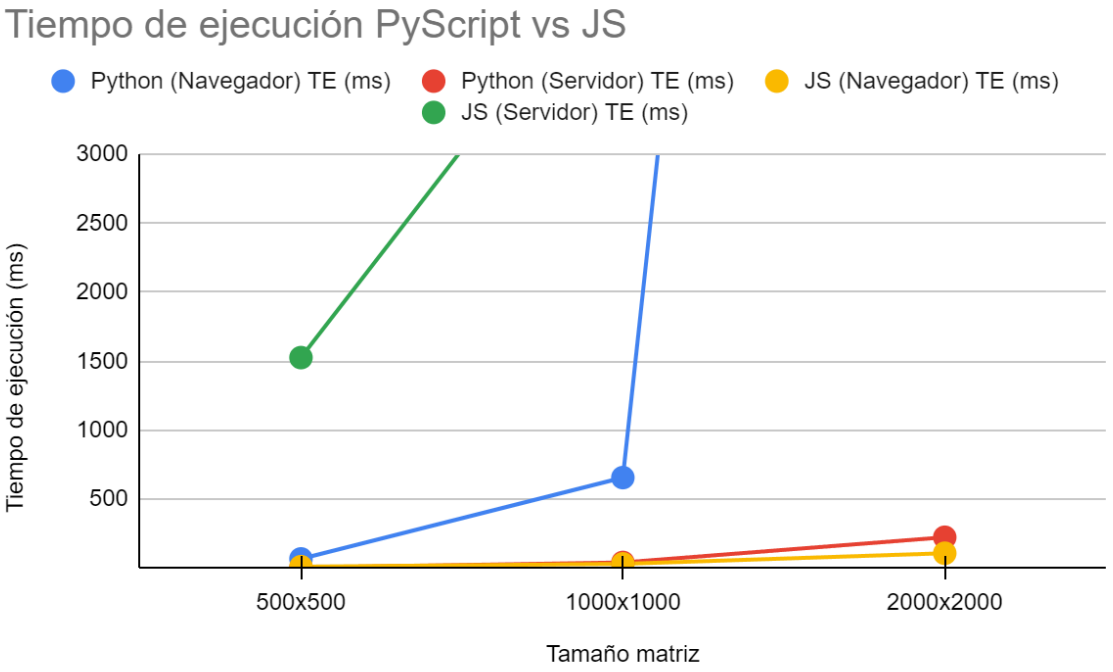

Figura 20. Diagrama de TE de cálculos matemáticos: prueba 1-2.

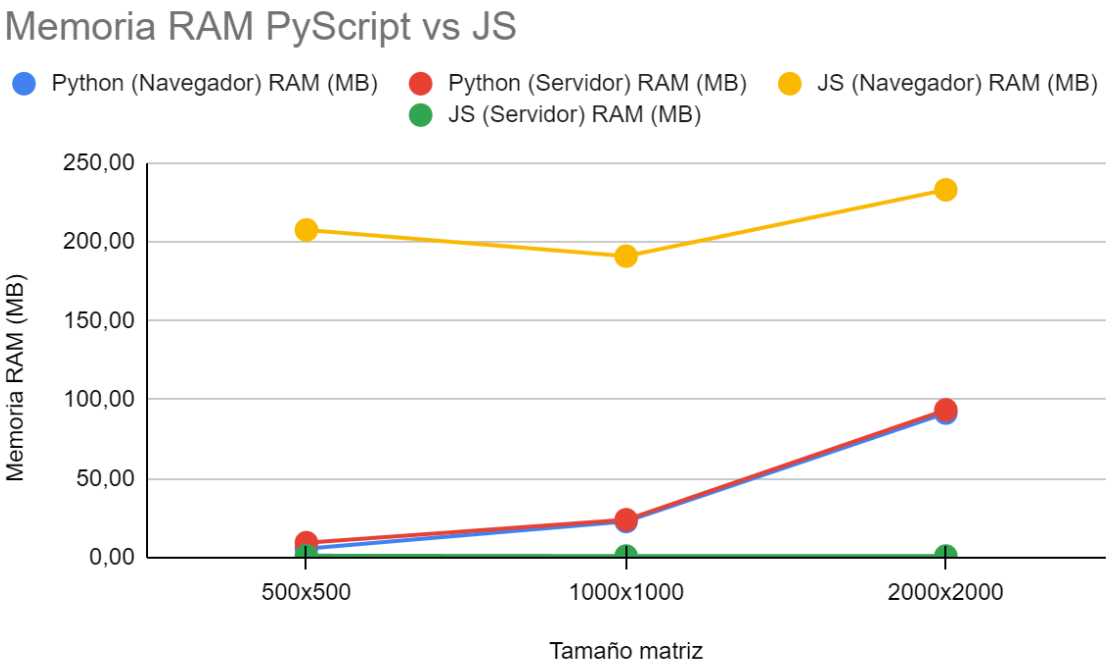

Figura 21. Diagrama de memoria de cálculos matemáticos: prueba 1-2.

Los resultados obtenidos confirman que JavaScript en navegador mantiene un rendimiento significativamente superior frente a PyScript, gracias a su capacidad para aprovechar la compilación Just-In-Time y las optimizaciones avanzadas del motor V8. Esto se refleja en tiempos de ejecución mucho más bajos, incluso al trabajar con matrices de gran tamaño,

aunque con un consumo de memoria considerablemente mayor debido a la complejidad de sus estructuras internas y la gestión dinámica de recursos para acelerar el procesamiento. Por otro lado, Python ejecutado en servidor es el único entorno capaz de ofrecer un rendimiento competitivo frente a JavaScript en navegador, con tiempos de ejecución similares y un uso de memoria más moderado. En comparación, JavaScript en servidor muestra un rendimiento inferior, con tiempos considerablemente más altos y un consumo de memoria muy bajo, lo que indica que, para operaciones intensivas, el entorno servidor de Python resulta más eficiente y equilibrado que su contraparte en JavaScript.

6.2.2. Prueba 2 - Detección de Números Primos

Esta prueba tiene un alto nivel de importancia en el ámbito científico y computacional, ya que el cálculo de números primos es una operación fundamental en áreas como la criptografía, la teoría de números y el análisis algorítmico. Determinar la eficiencia con la que un entorno de ejecución maneja esta tarea es crucial para evaluar su capacidad para soportar cálculos matemáticos intensivos y aplicaciones científicas. En este caso, se utiliza para la primera versión un algoritmo clásico y poco optimizado para la detección de primos mediante divisiones limitadas hasta la raíz cuadrada del número, lo que permite un equilibrio entre precisión y eficiencia. De tal manera esta versión se encargará de calcular los números primos del 1 hasta 10^6.

Para la segunda versión de la prueba incorporamos técnicas modernas y optimizadas como el uso del criba de Eratóstenes [56] implementada mediante la biblioteca SymPy, junto con el manejo eficiente de datos a través de NumPy, y en JavaScript el uso de TypedArrays. Esta combinación permite un procesamiento mucho más rápido y eficiente que el enfoque tradicional basado en pruebas de divisibilidad individual. Además, se añade la repetición de 1000 veces el cálculo, de tal modo que los resultados serán basados en un número estadístico de ejecuciones, pero sin embargo, reduciendo el número hasta 10^4.

Por otro lado, la criba de Eratóstenes es un algoritmo eficiente y sencillo para encontrar todos los números primos menores a un número dado. Funciona eliminando iterativamente los múltiplos de cada número primo encontrado, dejando solo los primos sin eliminar. Es uno de los métodos clásicos más utilizados para la generación de números primos en matemáticas y computación.

Versión 1: Algoritmos Convencionales junto a estructuras nativas

En esta variante, se implementó el algoritmo clásico para la detección de números primos, el cual consiste en comprobar si un número es divisible por cualquier entero menor que él. Sin embargo, la ejecución de este algoritmo resultó ser extremadamente lenta para el rango objetivo de esta prueba (hasta 10^7), lo que hizo necesario introducir una serie de optimizaciones para mejorar su eficiencia, la cual es la que se mencionó anteriormente.

```
def is_prime(n):
    for k in range(2, n):
```

```
        if n % k == 0:
            return False
    return True



def primes_to_n(n):
    primes = []

    for i in range(n):
        if is_prime(i):
            primes.append(i)
```

Código 1. Ejemplo del algoritmo clásico: prueba 1-2.

El algoritmo inicial para detectar si un número es primo se basa en la comprobación de divisibilidad por todos los enteros menores que él. Este enfoque, aunque sencillo, es computacionalmente costoso, especialmente para rangos grandes, ya que realiza un número excesivo de operaciones de división.

Para mejorar el rendimiento del algoritmo, se aplicaron las siguientes optimizaciones:

- Exclusión de Números Pares:
  - Todos los números primos mayores que 2 son impares. Por lo tanto, se puede reducir el número de comprobaciones recorriendo solo los números impares a partir de 3.
- Comprobación de Divisibilidad hasta la Raíz Cuadrada:
  - Si un número n no es primo, al menos uno de sus divisores debe ser menor o igual a su raíz cuadrada. Esto reduce significativamente el número de divisiones necesarias.
- Exclusión de Múltiplos de 2 y 3:
  - Se comprueba si el número es divisible por 2 o 3 antes de realizar cualquier otra operación, ya que estos son los divisores más comunes.
- Incremento en Pasos de 2:
  - En el bucle de comprobación de divisibilidad, se incrementa el paso en 2 para evitar comprobar números pares.

El algoritmo optimizado queda de la siguiente manera:

```
def is_prime(n):
    if n < 2:
        return False
    if n in (2, 3):
        return True
    if n % 2 == 0 or n % 3 == 0:
```

```
        return False

    for k in range(5, int(n**0.5) + 1, 2):
        if n % k == 0:
            return False
    return True

def primes_to_n(n):
    primes = []
    if n > 2:
        primes.append(2)

    for i in range(3, n, 2):
        if is_prime(i):
            primes.append(i)
```

Código 2. Ejemplo del algoritmo optimizado: prueba 1-2.

Nº	Lenguaje	Entorno	PLT	RAM (MB)	TE (ms)	CPU (%)
1	Python	Navegador	2795.60 ms	1.5 MB	12728.0 ms WA:12729.40 ms	
	Python	Servidor		1.79 MB	1259.25 ms	1.4 %
	JavaScript	Navegador		109.10 MB	56.1 ms	
	JavaScript	Servidor		1.94 MB	1082.70 ms	1.43 %
2	Python	Navegador	2727.50 ms	1.5 MB	13301.0 ms WA:13304.20 ms	
	Python	Servidor		0.06 MB	1137.66 ms	0.8 %
	JavaScript	Navegador		92.83 MB	58.4 ms	
	JavaScript	Servidor		1.94 MB	1076.73 ms	1.73 %
	Python	Navegador		1.5 MB	13906.0 ms	

3			2672.30 ms		WA:13913.70 ms	
	Python	Servidor		0.06 MB	1189.49 ms	0.9 %
	JavaScript	Navegador		105.01 MB	55.7 ms	
	JavaScript	Servidor		1.82 MB	1076.16 ms	2.43 %

Tabla 6. Resultados cálculos matemáticos: prueba 2-1.

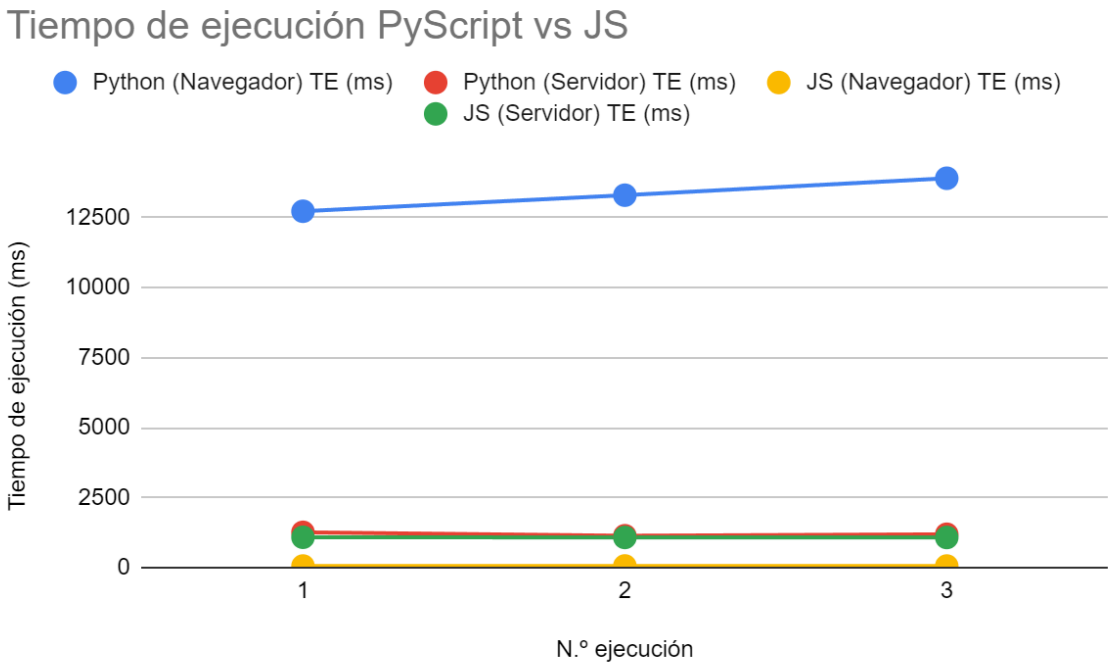

Figura 22. Diagrama de TE de cálculos matemáticos: prueba 2-1.

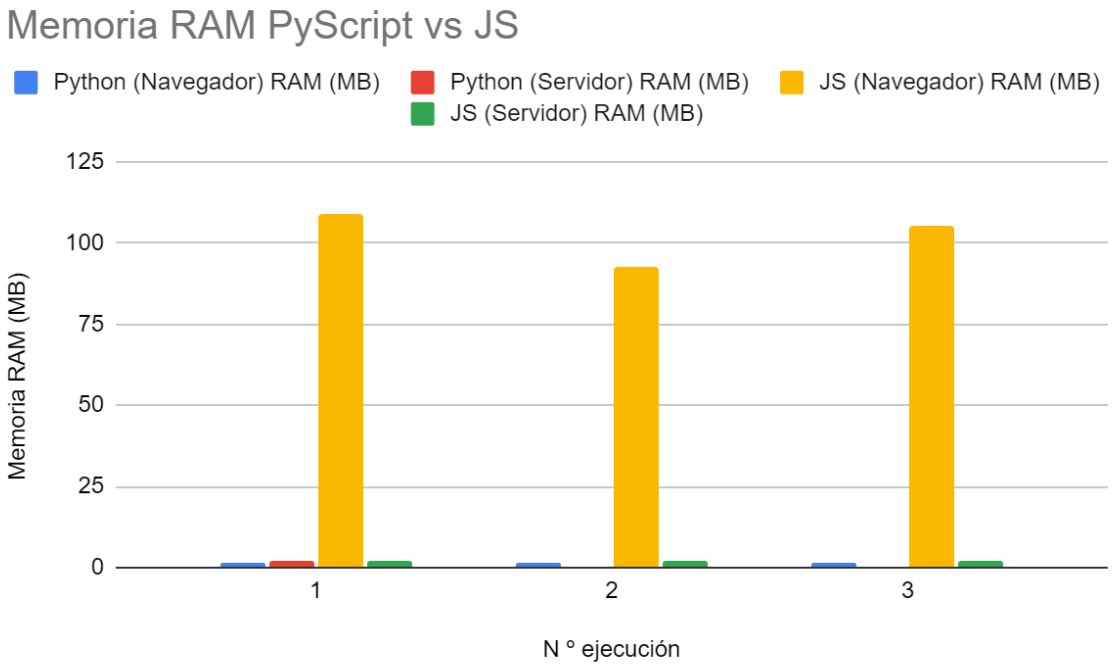

Figura 23. Diagrama de memoria de cálculos matemáticos: prueba 2-1.

Observación 5. JavaScript sigue siendo el entorno más rápido, pero a costa de ser quien usa mas memoria RAM

Versión 2: Algoritmos Optimizados con Librerías Especializadas

A nivel de implementación, el algoritmo en Node.js requirió un tiempo de ejecución considerablemente alto, lo que hizo que su participación fuera poco relevante para el análisis de resultados. Por esta razón, se optó por realizar 50 repeticiones en lugar de las 1000 inicialmente planteadas. Debido a esto, los resultados obtenidos fueron considerados poco significativos y, en consecuencia, no se tomaron en cuenta para el análisis final.

Lenguaje	Entorno	PLT	RAM (MB)	TE (ms)	CPU (%)
Python	Navegador	1825.00 ms	(avg, 1000x): 0.02 MB	Total ET (1000x): 28311.0 ms WA:28451.00 ms ET (avg, 1000x): 28.22 ms	
Python	Servidor		(avg, 1000x): 0 MB	Total ET (1000x): 1969.72 ms ET (avg, 1000x): 1.93 ms	CPU (avg, 1000x): 1.06 %
JavaScript	Navegador		(avg, 1000x): 93.92 MB	Total ET (1000x): 517.90 ms ET (avg, 1000x): 0.50 ms	

Tabla 7. Resultados cálculos matemáticos: prueba 2-2.

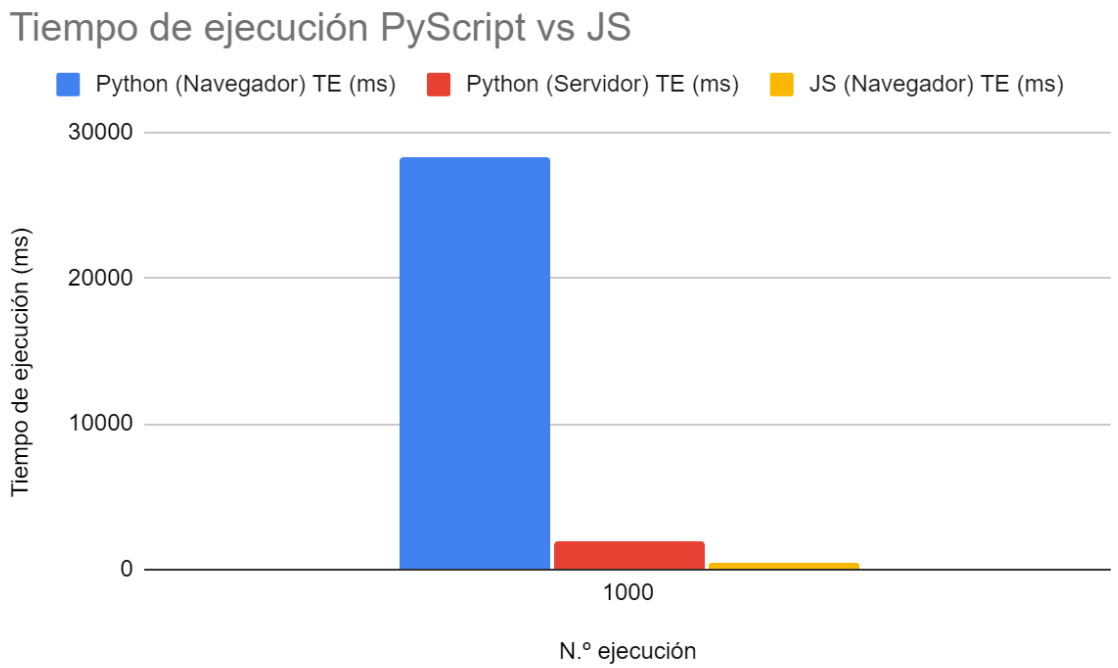

Figura 24. Diagrama de ET de cálculos matemáticos: prueba 2-2.

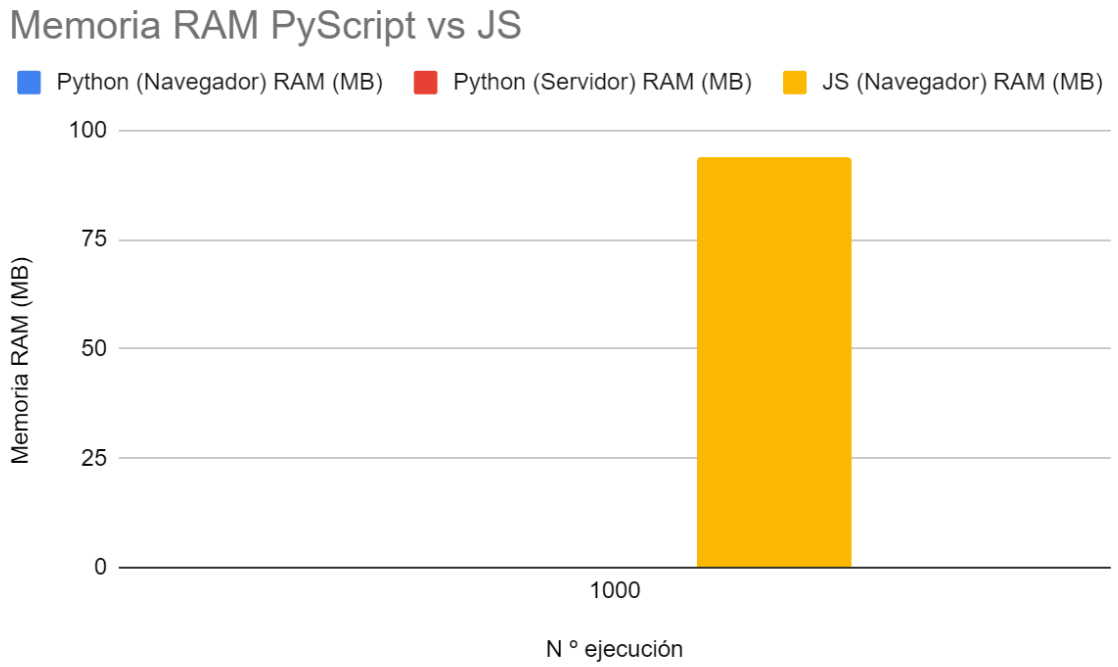

Figura 25. Diagrama de memoria de cálculos matemáticos: prueba 2-2.

Observación 6.
PyScript tiene un tiempo de ejecución aproximadamente 50 veces mayor que JavaScript en navegador. Destacando por su bajo consumo de memoria gracias al uso de librerías optimizadas como NumPy y SymPy, pero su ejecución sin completo acceso a sus correspondientes optimizaciones, limita su tiempo de ejecución.

Observación 7.
JavaScript, ejecutándose en el motor V8 con compilación Just-In-Time, ofrece un rendimiento mucho más rápido, pero con un consumo de memoria considerablemente mayor debido a sus optimizaciones internas y gestión dinámica de recursos que permiten usar la RAM para aprovechar un muy bajo tiempo de ejecución.

6.2.3. Prueba 3 - Cálculo de dígitos de π

En esta prueba se ha decidido implementar el cálculo de dígitos de π, utilizando distintas versiones del algoritmo en cada implementación para ofrecer una perspectiva comparativa sobre su rendimiento. El objetivo es evaluar cómo afectan el uso de estructuras de datos nativas y optimizadas, así como el manejo de precisión arbitraria, al rendimiento en tiempo de ejecución y consumo de memoria.

En la primera versión, se emplea el algoritmo Bailey–Borwein–Plouffe (BBP). En este caso, se calculan 3 dígitos de π, repitiendo el proceso también 10^3 veces. Esta implementación se realiza de forma manual y con estructuras nativas del lenguaje, sin el

uso de librerías externas. El algoritmo BBP permite calcular directamente cualquier dígito hexadecimal de π sin necesidad de conocer los anteriores, lo que lo convierte en una fórmula ideal para el paralelismo. No obstante, aquí se implementa de forma secuencial para evaluar el rendimiento nativo puro de los lenguajes utilizados.

En la segunda versión, se reduce el número de repeticiones a 10, pero se incrementa el número de dígitos calculados a 10^4 En este caso, se introduce el uso de librerías especializadas en precisión arbitraria: mpmath en Python/PyScript, y math.js en JavaScript. La precisión arbitraria hace referencia a la capacidad de representar y operar con números con un número de dígitos mucho mayor al límite de precisión estándar (como los 64 bits del tipo float). Esto permite realizar cálculos con una exactitud muy superior, a costa de mayor consumo de recursos.

En esta versión también se implementa el algoritmo Gauss–Legendre [57], un método iterativo que utiliza promedios aritméticos y geométricos para aproximar el valor de π. Su principal ventaja es su rápida convergencia, ya que el número de dígitos correctos se duplica en cada iteración, lo que lo convierte en una opción eficiente para cálculos de alta precisión.

Versión 1: Bailey-Borwein-Plouffe (BBP con Estructuras nativas) sin precisión arbitraria

Antes de mostrar los resultados, cabe destacar que al igual que en la anterior prueba, JavaScript en el servidor tardó más de lo esperado y no se tuvo en cuenta. La implementación esta vez, se dejó exactamente igual, y en los resultados no lo contemplaremos.

Lenguaje	Entorno	PLT	RAM (MB)	TE (ms)	CPU (%)
Python	Navegador	1825.00 ms	(avg, 1000x): 0.0 MB	Total ET (1000x): 52081.90 ms WA:52090.30 ms ET (avg, 1000x): 42.65 ms	
Python	Servidor		(avg, 1000x): 0 MB	Total ET (1000x): 2681.05 ms ET (avg, 1000x): 1.58 ms	CPU (avg, 1000x): 0.91 %
JavaScript	Navegador		(avg, 1000x): 201.54 MB	Total ET (1000x): 51.70 ms ET (avg, 1000x): 0.04 ms	

Tabla 8. Resultados cálculos matemáticos: prueba 3-1.

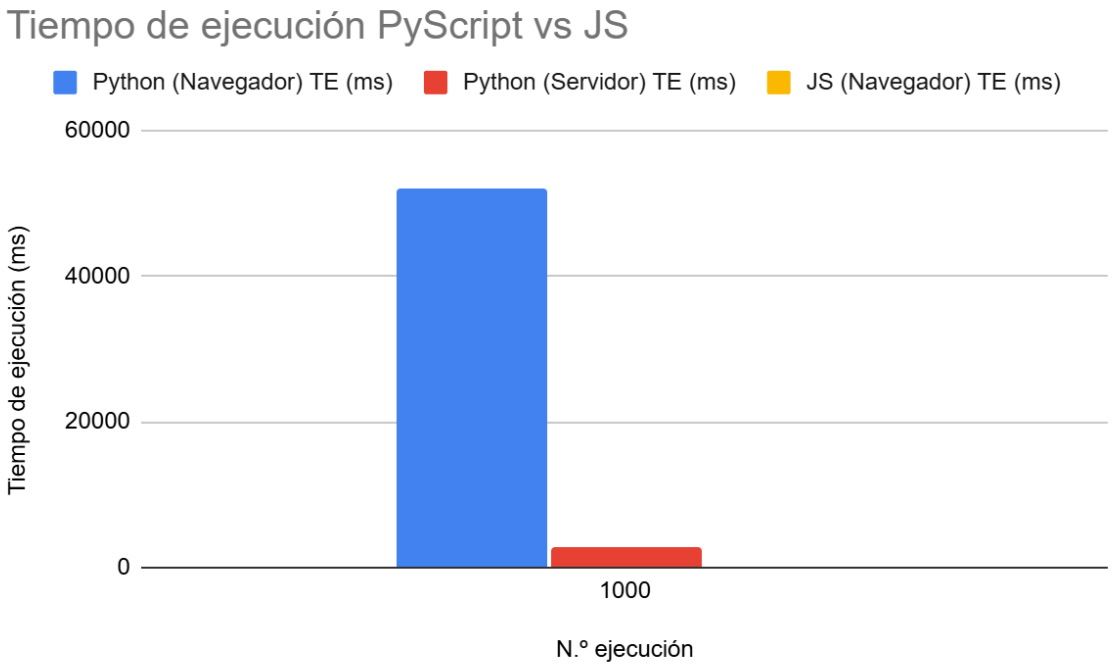

Figura 26. Diagrama de TE de cálculos matemáticos: prueba 3-1.

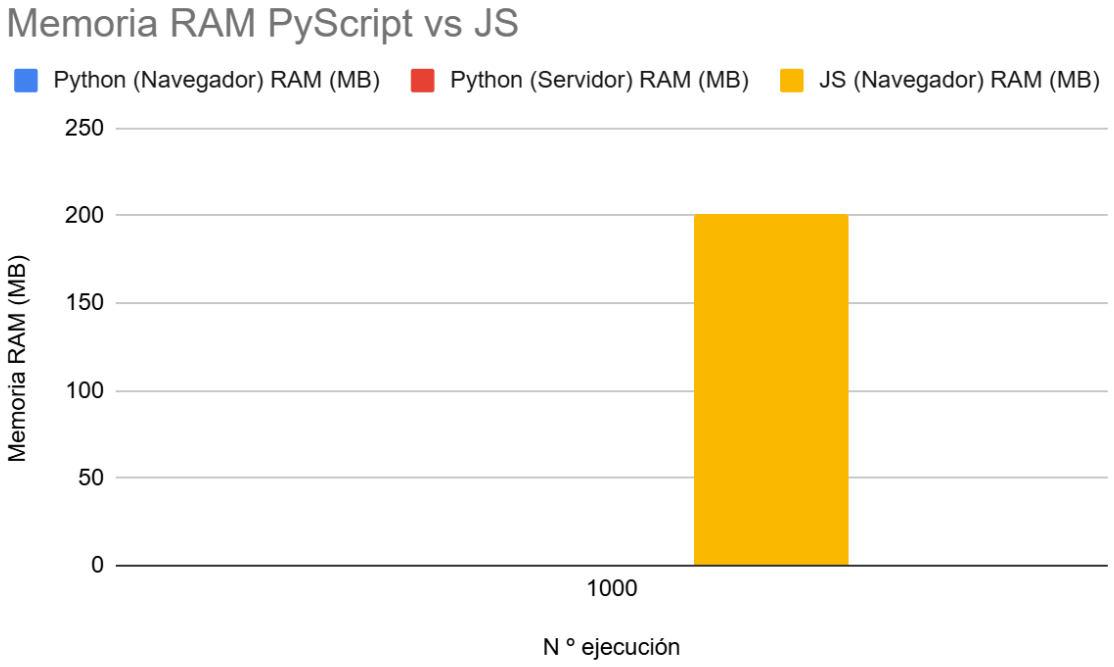

Figura 27. Diagrama de memoria de cálculos matemáticos: prueba 3-1.

Versión 2: Gauss-Legendre con precisión arbitraria

Lenguaje	Entorno	PLT	RAM (MB)	TE (ms)	CPU (%)
Python	Navegador	8064.50 ms	(avg, 10x): 0.11 MB	Total ET (10x): 4454.56 ms WA: 4466.74 ms ET (avg, 10x): 435.58 ms	
Python	Servidor		(avg, 10x): 0.1 MB	Total ET (10x): 5638.48 ms ET (avg, 10x): 563.85 ms	(avg, 10x): 18.94 %
JavaScript	Navegador		(avg, 10x): 297.76 MB	Total ET (10x): 57685.23 ms ET (avg, 10x): 5768.45 ms	
JavaScript	Servidor		(avg, 10x): 8.00 MB	Total ET (10x): 44802.69 ms ET (avg, 10x): 4480.24 ms	(avg, 10x): 0.24 %

Tabla 9. Resultados cálculos matemáticos: prueba 3-2.

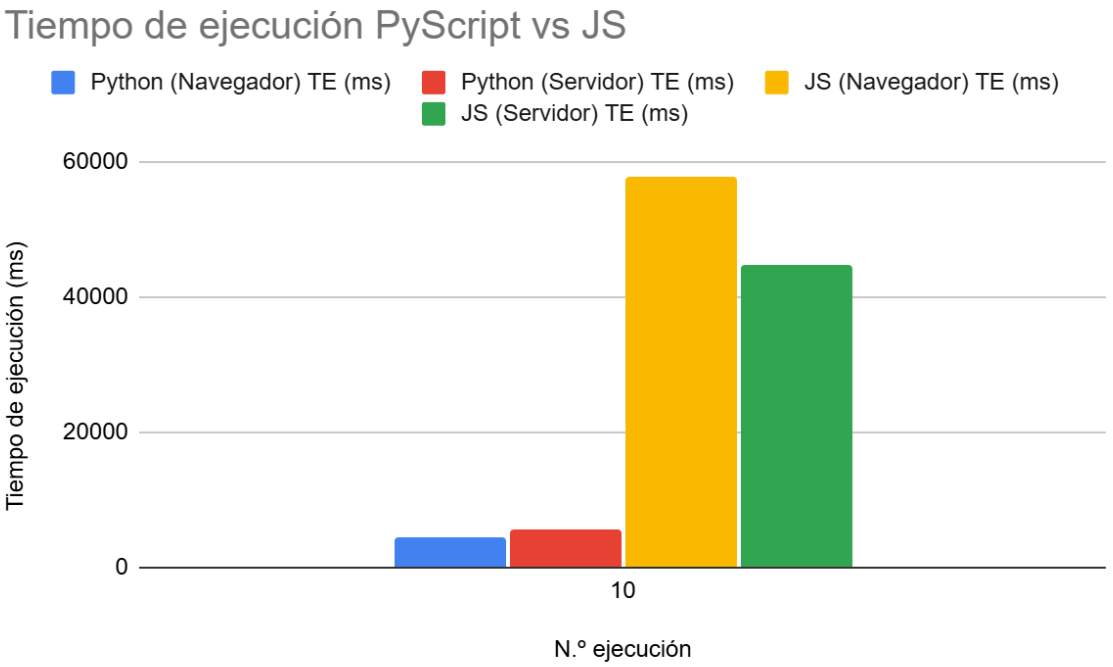

Figura 28. Diagrama de TE de cálculos matemáticos: prueba 3-2.

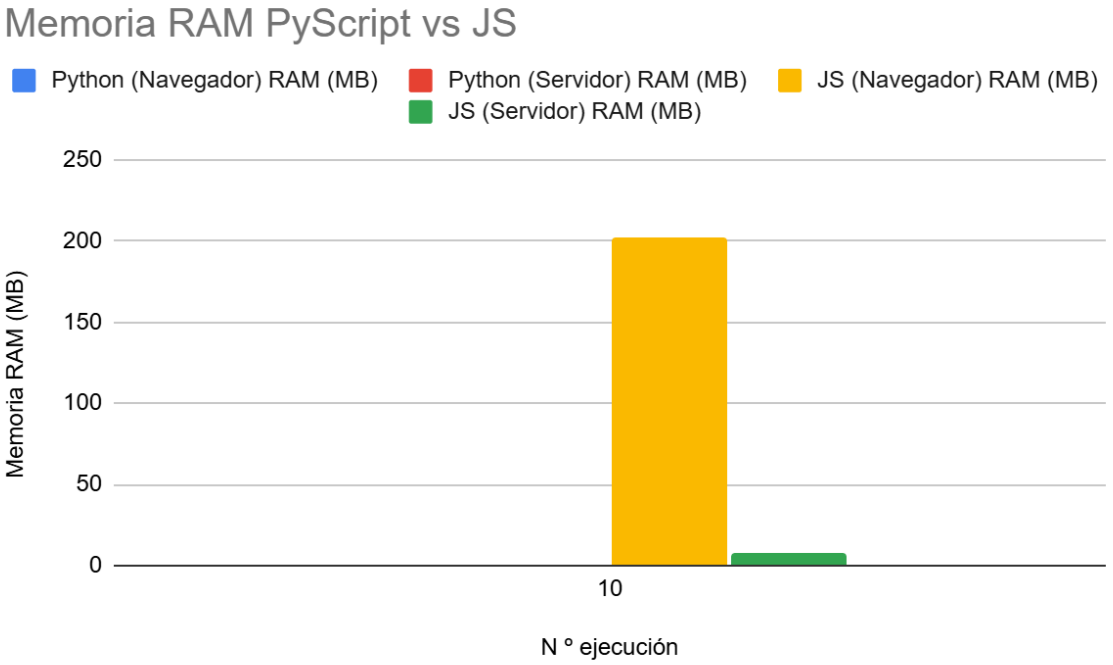

Figura 29. Diagrama de memoria de cálculos matemáticos: prueba 3-2.

Observación 8.
En cálculos de precisión arbitraria, PyScript supera significativamente a JavaScript en navegador, con tiempos de ejecución más de 10 veces menores y un consumo de memoria muchísimo más eficiente.

Observación 9.
Esta ventaja se debe a que PyScript usa la biblioteca mpmath optimizada en Python y ejecutada sobre WebAssembly, mientras que JavaScript depende de mathjs y BigNumber [58], menos eficientes para operaciones matemáticas complejas.

Observación 10.
La superioridad de PyScript en este caso es específica para cálculos de precisión arbitraria; en otros tipos de operaciones numéricas, como las que usan NumPy o sympy, JavaScript puede ser más rápido debido a optimizaciones nativas en el motor del navegador.

6.3. Procesamiento de grandes volúmenes de datos

Estas pruebas evaluarán la eficiencia de cada tecnología al ejecutar operaciones típicas en ámbitos estadísticos y científicos sobre grandes conjuntos de datos. Se contemplarán diversas tareas, desde transformaciones, búsquedas y filtrados, hasta cálculos como suma, media y desviación estándar.

Se definieron tres pruebas principales, donde las dos primeras cuentan con dos versiones cada una: una implementación sencilla y otra optimizada, enfocada en casos prácticos del día a día.

- Primera prueba: Se trabajará con un conjunto grande de datos con valores entre 0 y 1000, aplicando los siguientes algoritmos o procedimientos: creación, transformación, ordenamiento, búsqueda, filtrado y eliminación.

- Segunda prueba: Similar a la primera, pero con un enfoque en operaciones matemáticas típicas del ámbito científico, como la suma total, el cálculo de la media y la desviación estándar, dejando de lado algoritmos de manipulación como ordenar o filtrar.

- Tercera prueba: Esta prueba incluirá dos entradas en la interfaz web para permitir configurar el número de ejecuciones y la cantidad de workers que realizarán la operación. De este modo, el cálculo no se ejecutará en el hilo principal, sino mediante Web Workers. Las operaciones y el rango de valores serán los mismos que en la segunda prueba, pero utilizando diferentes bibliotecas: en Python se

emplearán *Pandas* y *NumPy* , mientras que en JavaScript se utilizarán *Math* y *danfo.js*, de modo que ambos trabajen con estructuras de datos tipo DataFrame.

6.3.1. Prueba 1 - Realizar operaciones de carga, transformación y procesamiento sobre un gran conjunto de datos.

Esta prueba, tal como se mencionó anteriormente, tiene como objetivo ejecutar una serie de algoritmos sobre un conjunto de datos de tamaño 10^7, con valores comprendidos entre 0 y 1000, incluyendo ambos extremos.

Las operaciones a realizar incluyen:

- crear la estructura de datos (una lista tanto en Python como en JavaScript),
- transformar el conjunto aplicando una operación matemática sencilla,
- ordenar la estructura,
- filtrar los datos según un umbral determinado,
- y eliminar un elemento de la estructura.

La segunda versión de esta prueba difiere únicamente en la implementación de los algoritmos y las estructuras de datos empleadas. En PyScript se utilizará NumPy para las operaciones y estructuras, mientras que en JavaScript se hará uso de TypedArrays, Arrays, Sets y funciones matemáticas nativas (Math).

Versión 1: Estructuras de Datos nativas con implementaciones no optimizadas

PLT: 2000ms - 5000ms

CREATE

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	155.86 MB	31698.00 ms
JavaScript	Navegador	1273.45 MB	154.20 ms

Tabla 10. Resultados procesamiento de datos: CREATE prueba 1-1.

TRANSFORM

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	155.86 MB	26126.00 ms
JavaScript	Navegador	163.36 MB	685.60 ms

Tabla 11. Resultados procesamiento de datos: TRANSFORM prueba 1-1.

SORT

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	57.20 MB	1703.00 ms
JavaScript	Navegador	262.44 MB	2053.00 ms

Tabla 12. Resultados procesamiento de datos: SORT prueba 1-1.

SEARCH

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	0.00 MB	131.00 ms
JavaScript	Navegador	0.00 MB	0.60 ms

Tabla 13. Resultados procesamiento de datos: SEARCH prueba 1-1.

FILTER

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	20.96 MB	2552.00 ms
JavaScript	Navegador	32.28 MB	77.70 ms

Tabla 14. Resultados procesamiento de datos: FILTER prueba 1-1.

DELETE

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	0.00 MB	26797.00 ms
JavaScript	Navegador	0.27 MB	16239.00 ms

Tabla 15. Resultados procesamiento de datos: DELETE prueba 1-1.

TOTAL

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	190.73 MB	89213.00 ms
JavaScript	Navegador	1440.28 MB	19016.40 ms

Tabla 16. Resultados procesamiento de datos: TOTAL prueba 1-1.

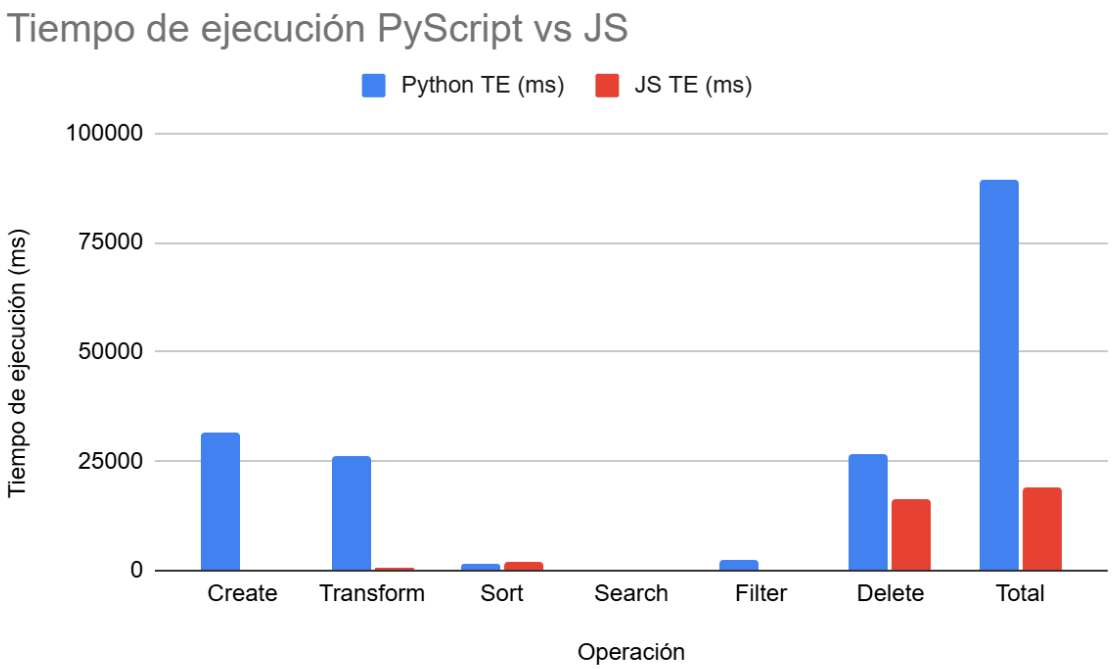

Figura 30. Diagrama de TE de procesamiento de datos: prueba 1-1.

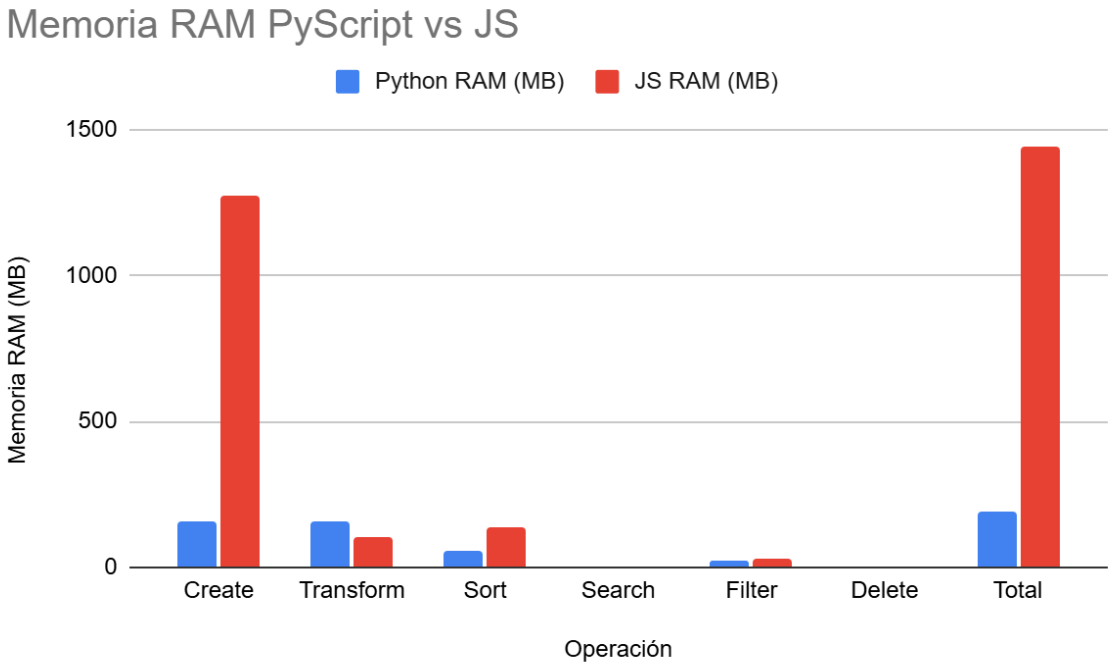

Figura 31. Diagrama de memoria de procesamiento de datos: prueba 1-1.

Observación 11.
JavaScript supera ampliamente a PyScript en eficiencia para crear y transformar estructuras de datos, logrando tiempos de ejecución decenas o incluso cientos de veces menores, aunque PyScript iguala el rendimiento en la ordenación gracias al algoritmo Timsort.

Observación 12.
La gestión de memoria y las estrategias internas del runtime influyen mucho en el rendimiento: JavaScript usa más memoria en ciertas operaciones, pero su manejo optimizado permite procesos como búsqueda y filtrado mucho más rápidos y eficientes que PyScript.

Versión 2: Estructuras de Datos nativas con implementaciones no optimizadas

En esta versión, lo más destacable es que las estructuras de datos las cambiamos como anteriormente mencionamos, y que además, las funciones en el caso de que sean posibles, se realizaron mediante librerías externas.

PLT: 1500ms - 5000ms

CREATE

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	0 MB	68.27 ms
JavaScript	Navegador	687.85MB	135.30 ms

Tabla 17. Resultados procesamiento de datos: CREATE prueba 1-2.

TRANSFORM

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	228.94 MB	118.66 ms
JavaScript	Navegador	76,99 MB	124.98 ms

Tabla 18. Resultados procesamiento de datos: TRANSFORM prueba 1-2.

SORT

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	76.30 MB	427.36 ms |
JavaScript	Navegador	17.20 MB	1779.34 ms

Tabla 19. Resultados procesamiento de datos: SORT prueba 1-2.

SEARCH

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	9.54 MB	7.21 ms
JavaScript	Navegador	0.00 MB	0.05 ms

Tabla 20. Resultados procesamiento de datos: SEARCH prueba 1-2.

FILTER

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	47.63 MB	75.76 ms
JavaScript	Navegador	27.53 MB	261.80 ms

Tabla 21. Resultados procesamiento de datos: FILTER prueba 1-2.

DELETE

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	85.75 MB	17.59 ms
JavaScript	Navegador	30.80 MB	156.04 ms

Tabla 22. Resultados procesamiento de datos: DELETE prueba 1-2.

TOTAL

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	228.94 MB	757.86 ms
JavaScript	Navegador	785.32 MB	2457.68 ms

Tabla 23. Resultados procesamiento de datos: TOTAL prueba 1-2.

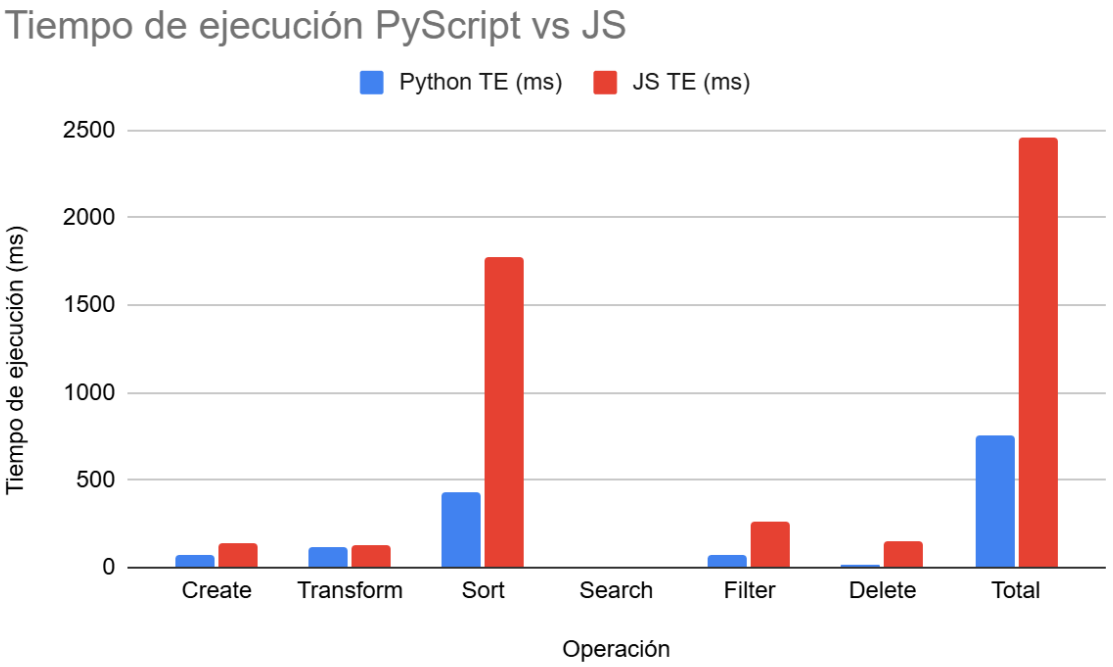

Figura 32. Diagrama de TE de procesamiento de datos: prueba 1-2.

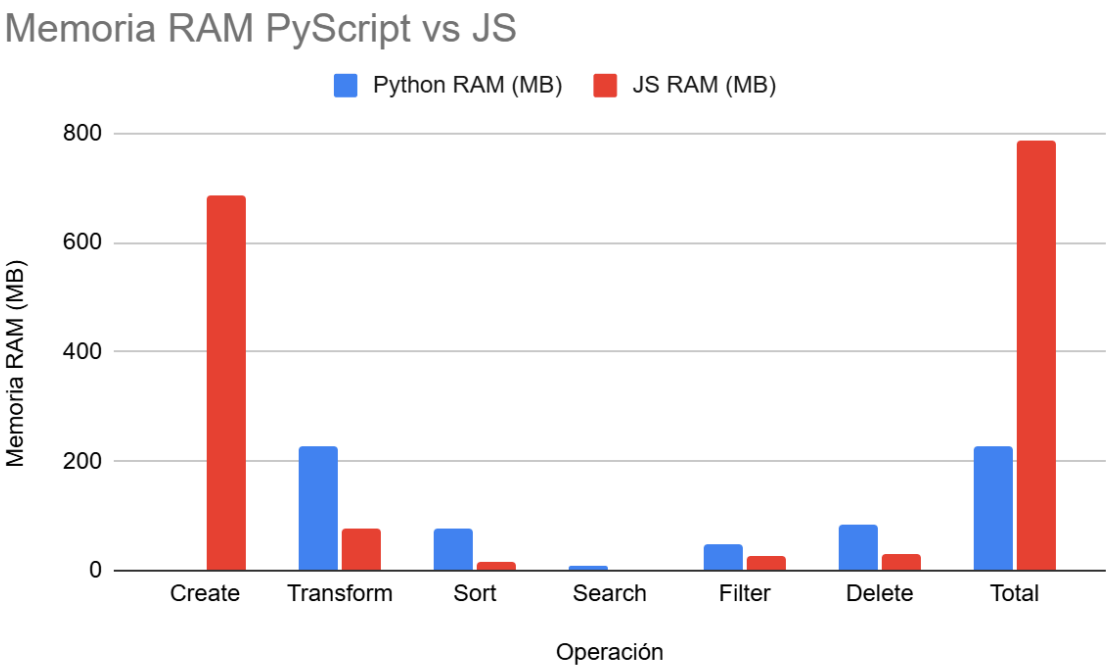

Figura 33. Diagrama de memoria de procesamiento de datos: prueba 1-2.

Observación 13.
PyScript con NumPy procesa la creación y transformación de datos numéricos en el navegador más rápido, pero usando más memoria que JavaScript en esta última. Porque Numpy crea estructuras de datos en memoria para optimizar las operaciones.

Observación 14.
Python es casi cuatro veces más rápido en ordenar datos, pero consume más memoria por la misma razón que la anterior observación.

Observación 15.
JavaScript es más rápido en búsquedas y usa muy poca memoria, pero en filtrado y borrado, PyScript sigue siendo más veloz aunque con mayor consumo de recursos por las copias intermedias que emplea para optimizar el tiempo.

Observación 16.
El rendimiento en general de PyScript es mejor en esta en el navegador, gracias a que NumPy está optimizado en C para manejar grandes volúmenes de datos con algoritmos avanzados, mientras que JavaScript es más generalista y menos eficiente para estos cálculos específicos mediante sus librerías.

6.3.2. Prueba 2 - Análisis Estadístico y Paralelización en Grandes Volúmenes de Datos.

Para esta prueba se utilizó la organización y estructura del código, modificando únicamente el tipo de operaciones realizadas. En lugar de aplicar algoritmos complejos, se trabajan ahora operaciones matemáticas básicas sobre un mismo conjunto de datos, manteniendo su tamaño. Específicamente, se calcula la suma, la media y la desviación estándar del conjunto.

En la segunda versión de la prueba se realizan las mismas operaciones, pero utilizando un sistema en paralelo. El usuario puede configurar el número de ejecuciones que se realizarán sobre un conjunto de 100.000 datos, así como el número de workers que procesarán dichas ejecuciones. De este modo, el total de ejecuciones se divide equitativamente entre los workers, permitiendo evaluar el impacto de la paralelización en el rendimiento.

Versión 1: Estructuras de Datos nativas

PLT: 2000ms - 12000ms

CREATE

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	155.87 MB	34169.00 ms
JavaScript	Navegador	1046.49 MB	166.50 ms

Tabla 24. Resultados procesamiento de datos: CREATE prueba 2-1.

SUM

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	0.00 MB	1588.00 ms
JavaScript	Navegador	1.24 MB	60.90 ms

Tabla 25. Resultados procesamiento de datos: SUM prueba 2-1.

MEAN

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	0.00 MB	1408.00 ms
JavaScript	Navegador	6.75 MB	52.50 ms

Tabla 26. Resultados procesamiento de datos: MEAN prueba 2-1.

STD

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	0.00 MB	4208.00 ms
JavaScript	Navegador	2.24 MB	102.70 ms

Tabla 27. Resultados procesamiento de datos: STD prueba 2-1.

TOTAL

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	155.87 MB	41616.00 ms
JavaScript	Navegador	1046.49 MB	361.80 ms

Tabla 28. Resultados procesamiento de datos: TOTAL prueba 2-1.

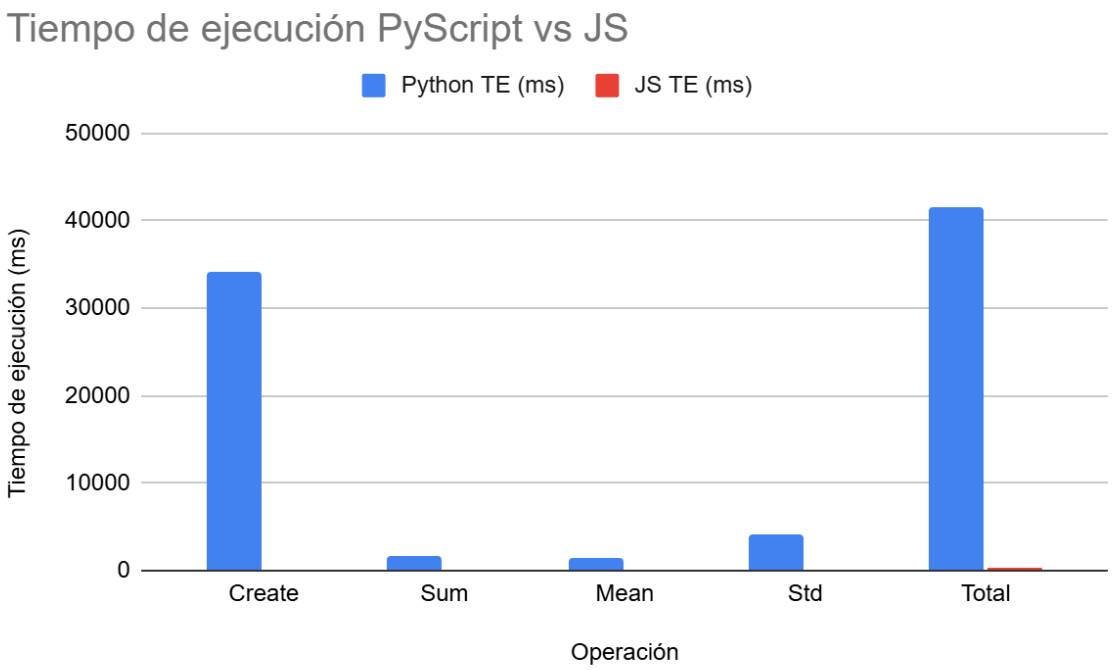

Figura 34. Diagrama de TE de procesamiento de datos: prueba 2-1.

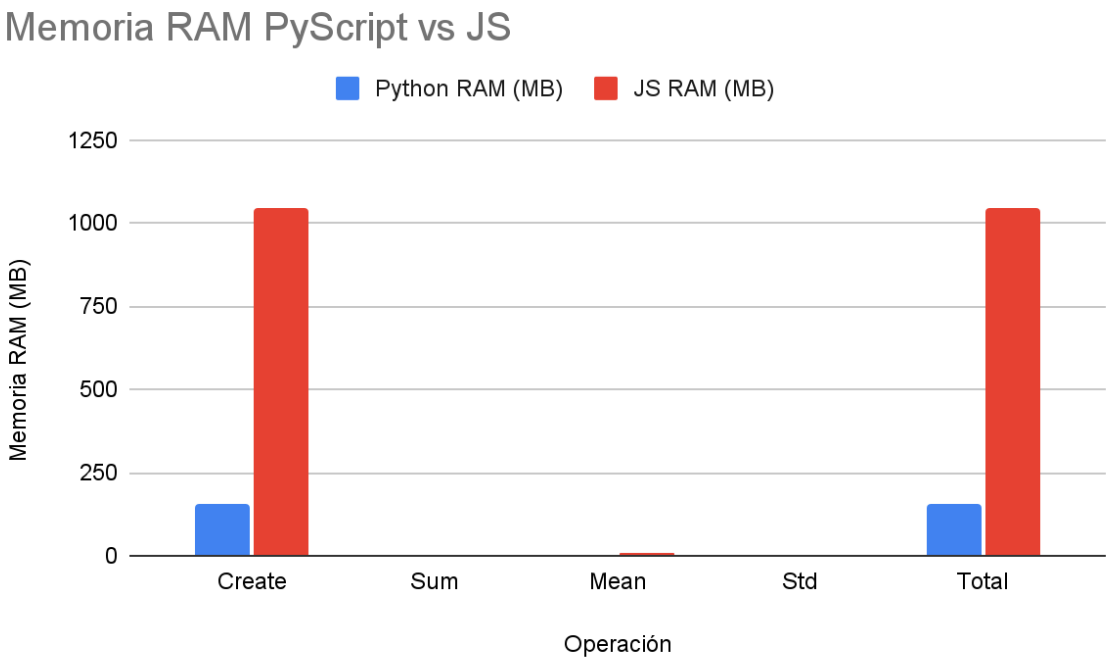

Figura 35. Diagrama de memoria de procesamiento de datos: prueba 2-1.

Observación 17. JavaScript ejecuta operaciones numéricas sobre arrays planos y datos primitivos de manera casi directa en el motor del navegador, logrando tiempos muy bajos gracias a su optimización nativa y bajo overhead.

Versión 2: Estructuras optimizadas con paralelismo

En esta versión, además de implementar operaciones de análisis estadístico mediante estructuras y funciones optimizadas, se incorpora la paralelización para mejorar el rendimiento.

Es importante destacar varios aspectos clave en este proceso:

- Medición del rendimiento: Para obtener resultados precisos, los tiempos de ejecución se han considerado únicamente a partir de la segunda ejecución de los experimentos. Esto se debe a que, cuando se cambia el número de *workers*, es necesario crearlos o eliminarlos, lo que introduce una latencia adicional que no se refleja en el tiempo total de las operaciones.

- Requisitos del entorno: Para permitir el uso de workers en PyScript, ha sido necesario utilizar un servidor HTTPS con los certificados correspondientes. De lo

contrario, PyScript no podría ejecutar *workers* en paralelo debido a restricciones de seguridad del navegador.

- Paralelización: Los *workers* se ejecutan en paralelo, distribuyendo las tareas asignadas entre ellos. Sin embargo, dentro de cada *worker*, las operaciones se realizan de manera secuencial.

- Sincronización: Una vez distribuidas las tareas entre los workers, se emplea un mecanismo de sincronización (basado en la espera de la finalización de todas las promesas en JavaScript y *asyncio.gather* en PyScript) para asegurarse de que todas las tareas han concluido antes de continuar con el procesamiento de los resultados.

100 Repeticiones, 1 workers

PLT: 2000ms - 5000ms

CREATE

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	0,38 MB	0.76 ms
JavaScript	Navegador	38.15 MB	76.72 ms

Tabla 29. Resultados procesamiento de datos: CREATE prueba 2-2.

SUM

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	0 MB	0.05 ms
JavaScript	Navegador	0 MB	70.60 ms

Tabla 30. Resultados procesamiento de datos: SUM prueba 2-2.

MEAN

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	0 MB	0.12 ms
JavaScript	Navegador	0 MB	68.71 ms

Tabla 31. Resultados procesamiento de datos: MEAN prueba 2-2.

STD

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	0 MB	0.38 ms
JavaScript	Navegador	0MB	12.29 ms

Tabla 32. Resultados procesamiento de datos: STD prueba 2-2.

TOTAL

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	0,38 MB	1.39 ms | 735.83 ms
JavaScript	Navegador	38.15 MB	228.35 ms | 22842.87 ms

Tabla 33. Resultados procesamiento de datos: TOTAL prueba 2-2.

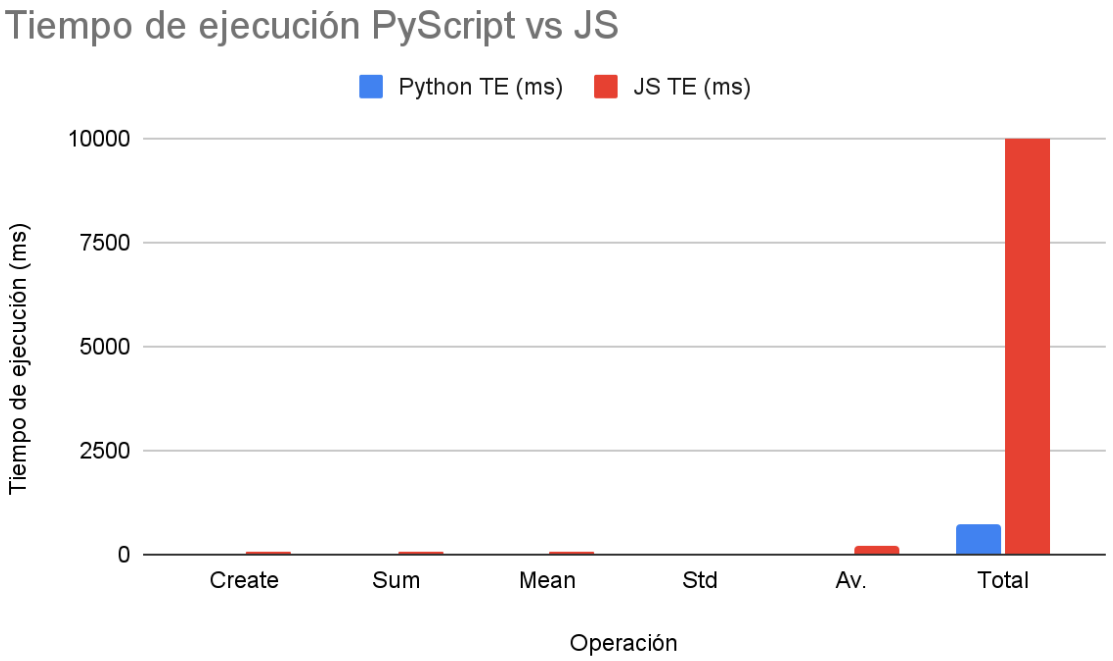

Figura 36. Diagrama de TE de procesamiento de datos: prueba 2-2.

1000 Repeticiones, 10 workers

PLT: 2000ms - 5000ms

CREATE

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	0,38 MB	1,75 ms
JavaScript	Navegador	38.15 MB	151,06 ms

Tabla 34. Resultados procesamiento de datos: CREATE prueba 2-2.

SUM

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	0 MB	0,14 ms
JavaScript	Navegador	0 MB	135,52 ms

Tabla 35. Resultados procesamiento de datos: SUM prueba 2-2.

MEAN

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	0 MB	0,29 ms
JavaScript	Navegador	0 MB	135,39 ms

Tabla 36. Resultados procesamiento de datos: MEAN prueba 2-2.

STD

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	0 MB	0,93 ms
JavaScript	Navegador	0 MB	25,44 ms

Tabla 37. Resultados procesamiento de datos: STD prueba 2-2.

TOTAL

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	0 MB	3,25 ms | 2921,53 ms
JavaScript	Navegador	0 MB	447,47 ms | 46523,15 ms

Tabla 38. Resultados procesamiento de datos: TOTAL prueba 2-2.

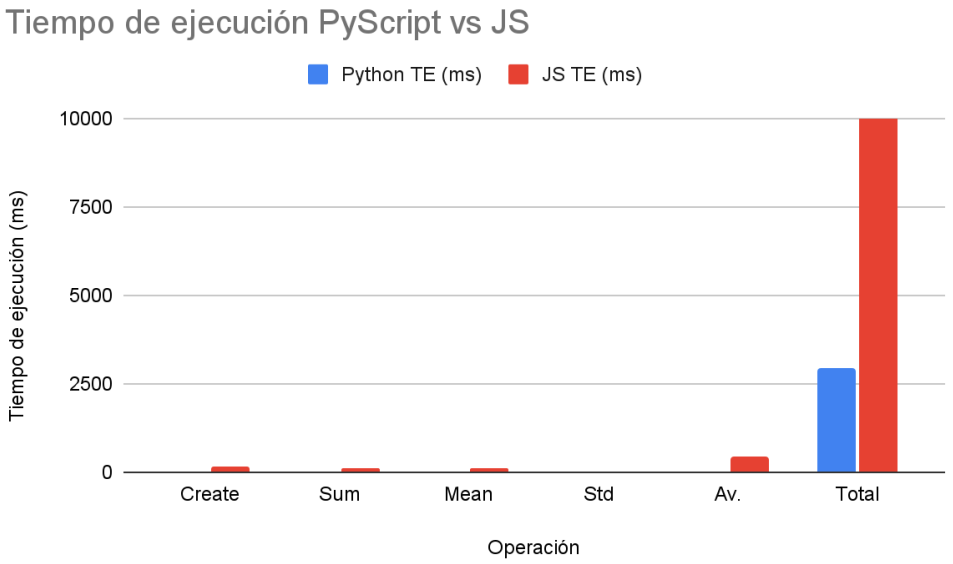

Figura 37. Diagrama de TE de procesamiento de datos: prueba 2-2.

Observación 18.
Con 1 worker y 100 ejecuciones, PyScript es más rápido y consume menos memoria que JavaScript gracias a NumPy, que usa funciones vectorizadas en C compiladas a WebAssembly, superando el rendimiento de Typed Arrays en JavaScript.

Observación 19.
Al aumentar a 10 workers y 1000 repeticiones, PyScript mantiene y mejora su ventaja relativa en tiempo y memoria, demostrando mejor escalabilidad en este tipo de operaciones básicas.

Observación 20.
Para operaciones numéricas básicas en navegador, PyScript con NumPy representa la opción más eficiente y recomendada frente a JavaScript.

6.3.3. Prueba 3 - Misma versión que la anterior, pero usando Pando y Danfo.js.

Para esta prueba, se diseñó únicamente una versión de la implementación. A partir de este punto, todas las pruebas seguirán esta misma lógica: utilizar una única versión optimizada que represente una situación real y cotidiana, tal como se esperaría en un entorno laboral profesional. El objetivo es reflejar un enfoque práctico y eficiente, priorizando la calidad del código por encima de la comparación estructural entre lenguajes.

A diferencia de las pruebas anteriores, esta implementación introduce una nueva estructura de datos: los dataframes. Los dataframes permiten organizar la información de forma tabular, similar a una hoja de cálculo, lo que facilita el análisis, filtrado y manipulación de grandes volúmenes de datos. En el contexto de Python, esto se implementa comúnmente

mediante la biblioteca *pandas*, que proporciona una interfaz poderosa y expresiva para trabajar con datos estructurados. Esta elección se alinea con prácticas comunes en análisis de datos y ciencia aplicada, donde los dataframes son una herramienta estándar por su rendimiento y flexibilidad. Por otro lado, esto se implementa en JavaScript con *Danfo.js.*

1000 Repeticiones, 10 workers

PLT: 2500ms - 7000ms

CREATE

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	0,38 MB	2,07 ms
JavaScript	Navegador	0,38 MB	9,71 ms

Tabla 39. Resultados procesamiento de datos: CREATE prueba 3-1.

SUM

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	0,38 MB	0,81 ms
JavaScript	Navegador	0,38 MB	11,74 ms

Tabla 40. Resultados procesamiento de datos: SUM prueba 3-1.

MEAN

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	0,38 MB	0,96 ms
JavaScript	Navegador	0,38 MB	10,29 ms

Tabla 41. Resultados procesamiento de datos: MEAN prueba 3-1.

STD

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	0,38 MB	1,51 ms
JavaScript	Navegador	0,38 MB	10,03 ms

Tabla 42. Resultados procesamiento de datos: STD prueba 3-1.

TOTAL

Lenguaje	Entorno	RAM (MB)	TE (ms)
Python	Navegador	0,38 MB	7,13 ms | 862,08 ms
JavaScript	Navegador	0,38 MB	41,78 ms | 4245,10 ms

Tabla 43. Resultados procesamiento de datos: TOTAL prueba 3-1.

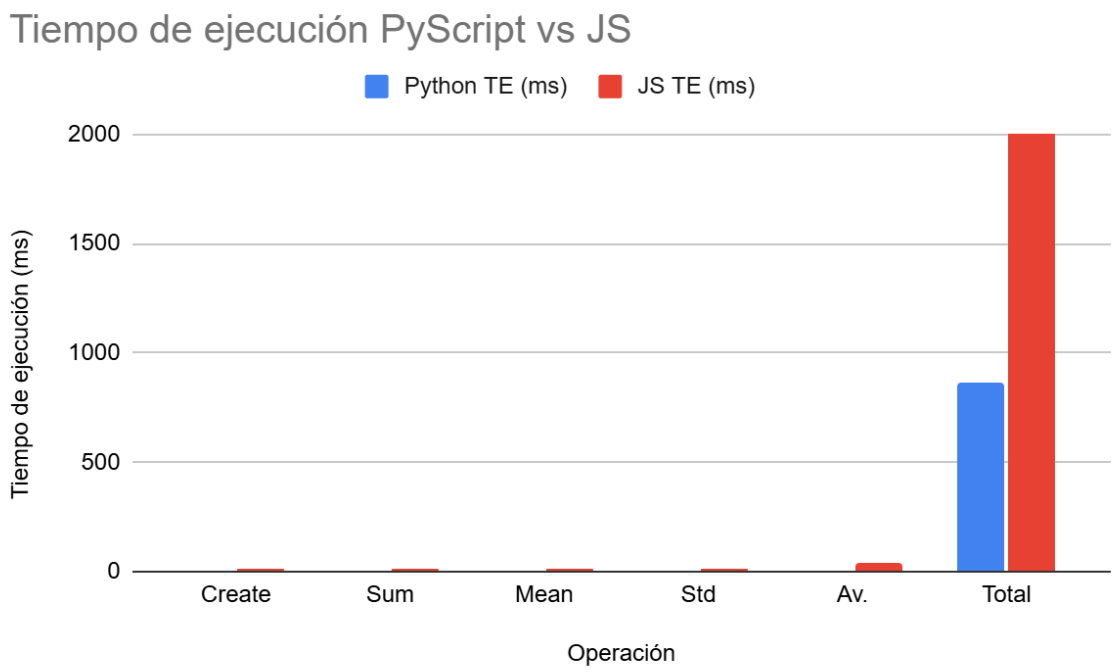

Figura 38. Diagrama de TE de procesamiento de datos: prueba 1-3.

Observación 21.
La principal ventaja de PyScript radica en la ejecución vectorizada y optimizada de NumPy, mientras que danfo.js procesa datos con bucles interpretados que no escalan bien, provocando una penalización significativa en tiempo de ejecución.

Observación 22.
JavaScript, aunque corre directamente en el motor del navegador con compilación JIT, usa danfo.js que está escrito enteramente en JavaScript, lo que limita su eficiencia en cálculos sobre grandes datos.

6.4. Carga de gráficos complejos

En estas pruebas evaluaremos el rendimiento de ambas tecnologías al generar gráficos complejos. La primera prueba se centrará en los gráficos de dispersión, utilizando para ello

un conjunto de datos de 100,000 puntos generado con NumPy en PyScript y TypedArrays junto con funciones nativas de Math en JavaScript. Para la representación visual, en el entorno de JavaScript se emplea la API nativa del navegador mediante el elemento <canvas>, sin utilizar librerías externas, lo que permite evaluar el rendimiento base del entorno. En cambio, en PyScript se usará la conocida librería *matplotlib*, en combinación con NumPy, para producir un gráfico de dispersión que se exportará como una imagen en formato PNG. En esta fase, mediremos el tiempo de inicialización del worker, la generación de datos, el renderizado final, y el promedio de tiempo por ejecución. Además, ambas implementaciones permiten especificar el número de ejecuciones, lo que facilita obtener métricas representativas bajo distintas cargas.

En la segunda prueba, el enfoque se traslada hacia la creación de gráficos más complejos, específicamente gráficos de series temporales con múltiples líneas. Aquí, en lugar de generar imágenes estáticas, se optará por gráficos interactivos. Tanto en JavaScript como en PyScript se utilizará la librería *Plotly*, una herramienta potente para visualización avanzada, aunque en cada entorno se invoca de forma distinta: directamente desde JavaScript en ambos casos, de esa forma podemos demostrar cómo ambas tecnologías trabajan entre sí. La generación de datos se seguirá realizando con TypedArrays y *Math* en JavaScript, y con NumPy en PyScript. Además, en esta prueba se podrá configurar dinámicamente tanto la cantidad de series como el número de puntos por serie, permitiendo analizar cómo se comportan ambas tecnologías frente a una carga creciente de datos y elementos visuales.

6.4.1. Prueba 1 - Renderizado de gráficos de dispersión masivos

100 Repeticiones.

PLT: 2500ms - 5000ms

Operation	PyScript	JavaScript
Worker	0,00 ms	0,05 ms
Data	2,78 ms	1,65 ms
Rendering	527,08 ms	123,13 ms
Memory	1,53 MB	1,53 MB
Total	65426,93 ms	12580,08 ms

Tabla 44. Resultados carga de gráficos complejos: prueba 1-1.

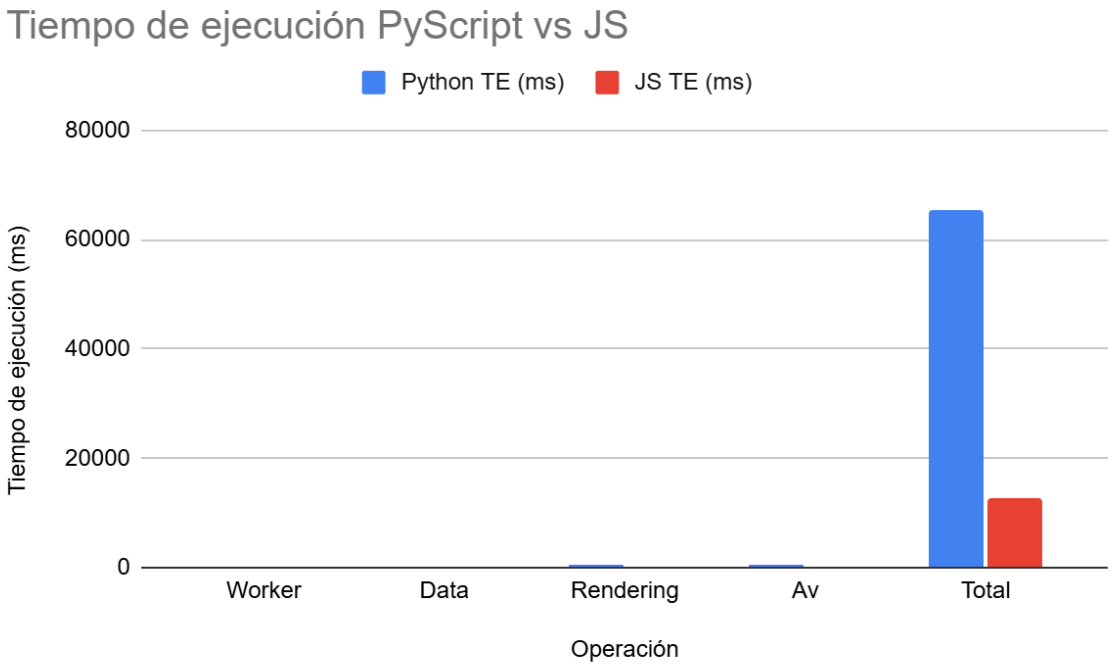

Figura 39. Diagrama de TE de carga de gráficos complejos: prueba 1-1.

Observación 23.
JavaScript es mucho más rápido en renderizado gráfico en tiempo real (12.5 s vs. 65 s) gracias a *OffscreenCanvas*, mientras que PyScript con matplotlib sufre un gran overhead por su proceso pesado de generación y conversión de imágenes.

Observación 24.
El consumo de memoria es similar en ambos, ya que manejan estructuras de datos parecidas en 64 bits; la diferencia está en la eficiencia del procesamiento y renderizado.

Observación 25.
JavaScript inició Web Workers casi instantáneamente, mientras que PyScript tiene un retraso significativo en la creación debido a la arquitectura de Pyodide y WebAssembly.

6.4.2. Prueba 2 - Visualización Interactiva de Series Temporales Complejas

En esta prueba, se destacan dos aspectos clave: por un lado, la medición de FPS se realiza durante los últimos 3000 ms (3 segundos) desde que se muestra el gráfico en pantalla; por otro, el tiempo total de ejecución refleja exclusivamente la duración de la operación principal, sin incluir tiempos adicionales como la inicialización del worker.

Series: 5

Puntos: 10.000

PLT: 3000ms - 10000ms

Operation	PyScript	JavaScript
Worker	2954.54 ms	7.70 ms
Data	28.00 ms	4,50 ms
Rendering	242.80 ms	0,20 ms
Memory	0.46 MB	0.46 MB
FPS	232.02 FPS	207.13 FPS
Total	270.80 ms	4.70 ms

Tabla 45. Resultados carga de gráficos complejos: prueba 2-1.

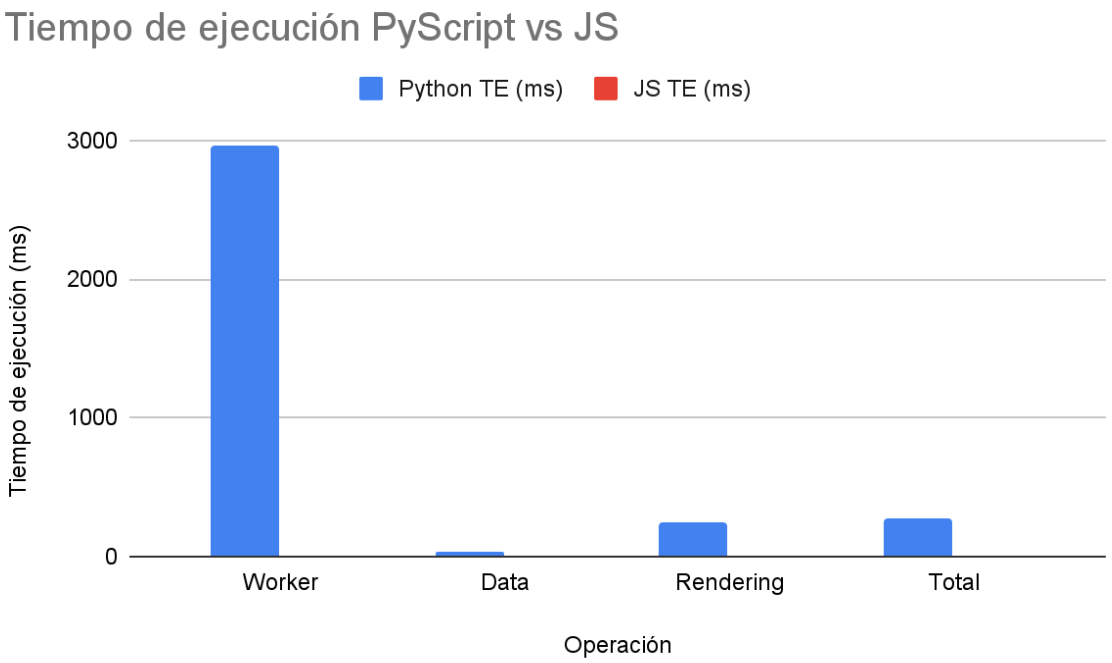

Figura 40. Diagrama de TE de carga de gráficos complejos: prueba 2-1.

Observación 26.
JavaScript supera ampliamente a PyScript en todos los aspectos de rendimiento. La operación completa en JavaScript toma apenas 4.7 ms, mientras que en PyScript se dispara a 270 ms, aunque se usen librerías optimizadas en PyScript, sigue sin ser tan optimizada como JavaScript en el ámbito de la generación de gráficos complejos.

6.5. Manejo de múltiples solicitudes concurrentes

En esta sección nos enfocamos en evaluar el rendimiento de envío y recepción de múltiples solicitudes de forma concurrente. El objetivo es analizar qué tan eficientemente se comunican ambos entornos, JavaScript y PyScript, con un servidor, simulando el comportamiento típico de una API.

El usuario podrá configurar dos parámetros clave en ambas implementaciones: el número de solicitudes a realizar y un valor de delay. Este delay representa una carga simulada del servidor, como si estuviera realizando un trabajo intensivo, emulando así un entorno más cercano al mundo real.

En la primera prueba, las solicitudes se realizan mediante peticiones HTTP convencionales, y la respuesta del servidor consistirá en un pequeño JSON con 10 números aleatorios entre 0 y 1, junto con una ID de referencia. En la segunda prueba, el mismo tipo de respuesta será entregado, pero utilizando un canal WebSocket, permitiendo mantener una conexión persistente y bidireccional con el servidor. Esto nos permitirá comparar no solo la eficiencia de cada lenguaje, sino también el impacto del protocolo de comunicación utilizado.

6.5.1. Prueba 1 - Solicitudes concurrentes con Fetch/Promise.all (JavaScript) vs Asyncio/Fetch (PyScript)

En la primera prueba, utilizaremos peticiones concurrentes a un backend sencillo, al que le enviaremos el número de peticiones y el delay indicado por el usuario. Lo más destacable es que el número máximo de peticiones concurrentes es de 6, tanto en la versión local como en el framework, y esto se determinó mediante una variable en la API.

Solicitudes: 50

Delay: 1000

PLT: 2000ms - 8000ms

Operation	PyScript	JavaScript
Worker	2752,97 ms	0,07 ms
Avg time	4756.40 ms	4827.93 ms
Total time	9103.01 ms	9185.15 ms

Tabla 46. Resultados peticiones concurrentes: prueba 1-1.

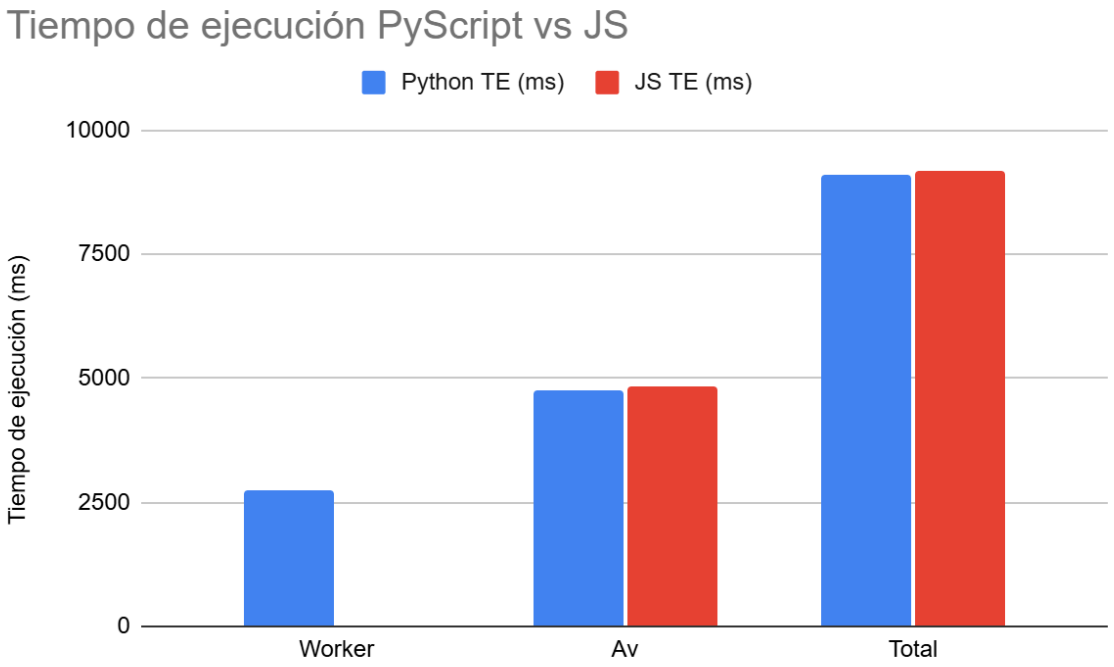

Figura 41. Diagrama de TE de solicitudes concurrentes: prueba 1-1.

Observación 27.
La mayor diferencia está en el tiempo de inicialización del worker: JavaScript es casi instantáneo (0,07 ms), mientras que PyScript tarda mucho más (2750 ms) debido a la carga adicional de Pyodide y WebAssembly.

Observación 28.
El tiempo de ejecución promedio y total de las solicitudes HTTP es similar en ambos (alrededor de 4.7-4.8 s promedio), porque ambos están limitados a 6 peticiones simultáneas.

6.5.2. Prueba 2 - Manejo de WebSockets para solicitudes concurrentes en JavaScript vs PyScript

A diferencia de la anterior prueba, esta fue realizada con WebSockets en vez de utilizar peticiones cotidianas HTTP.

Solicitudes: 500

Delay: 1000

PLT: 2000ms - 5000ms

Operation	PyScript	JavaScript
Worker	3133.36 ms	0.07 ms
Avg time	1067.68 ms	1045.04 ms
Total time	1296.60 ms	1060.76 ms

Tabla 47. Resultados peticiones concurrentes: prueba 2-1.

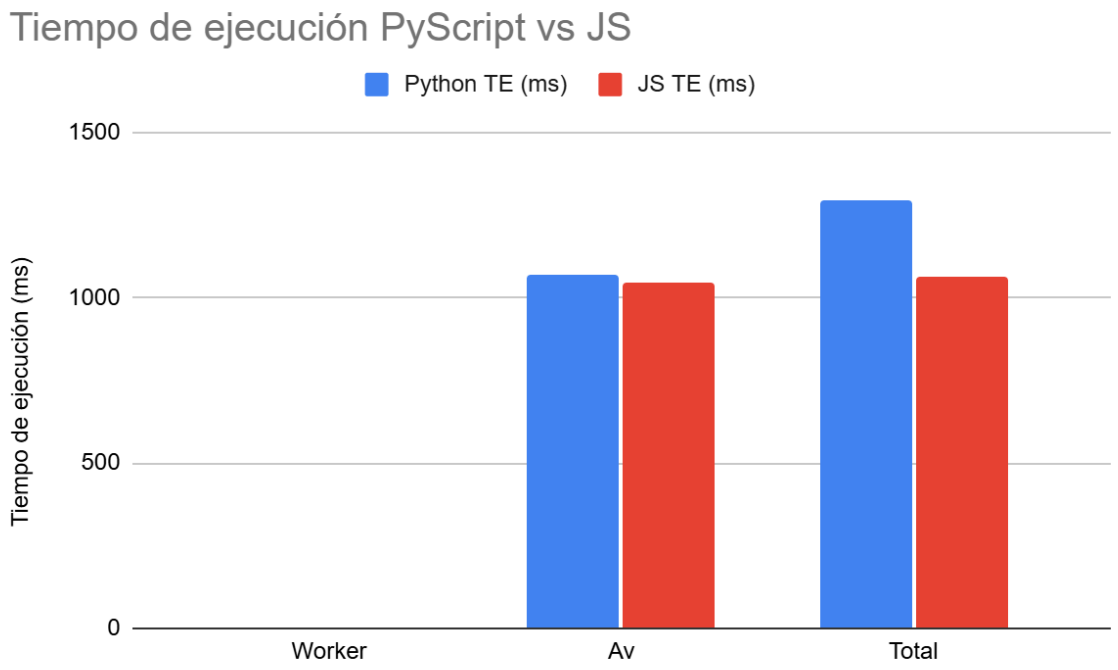

Figura 42. Diagrama de TE de solicitudes concurrentes: prueba 2-1.

Observación 29.
La ejecución de las peticiones es similar en PyScript y JavaScript, pero PyScript sufre una penalización importante por la lenta inicialización del worker, lo que afecta negativamente el rendimiento total.

6.6. Cálculo y verificación de integridad en datos sensibles

En este quinto ámbito nos adentramos en un terreno especialmente crítico dentro de muchas aplicaciones modernas: el cálculo y la verificación de integridad en datos sensibles. Aquí, más allá del simple rendimiento computacional, entra en juego la

fiabilidad y seguridad de las operaciones realizadas, aspectos fundamentales en sistemas que manejan información confidencial o verificable.

Las pruebas que componen este bloque están diseñadas para evaluar cómo se comportan JavaScript y PyScript en operaciones criptográficas comunes, particularmente en contextos de validación de integridad y cifrado simétrico. La primera prueba se enfoca en la generación y verificación de hashes criptográficos SHA-256 aplicados sobre archivos de gran tamaño, una operación esencial para asegurar que los datos no han sido alterados. La segunda, en cambio, se centra en el cifrado y descifrado con el algoritmo AES-GCM de 128 bits, verificando además la integridad del mensaje como parte del proceso. Donde ambas pruebas el usuario podrá indicar las veces a ejecutar y el tamaño del archivo/mensaje.

Ambas pruebas se apoyan en herramientas estándar de cada entorno: Web Crypto API en JavaScript y PyCryptodome en PyScript. Al analizarlas, no sólo mediremos el tiempo total y promedio de cada operación, sino también el uso de memoria y el correcto cumplimiento de la verificación de integridad. Con esto podremos valorar en qué medida cada tecnología está preparada para enfrentar escenarios donde la protección y validación de los datos no es una opción, sino un requisito indispensable.

6.6.1. Prueba 1 - Generación y Verificación de Hashes Criptográficos SHA-256

Compararemos el rendimiento de PyCryptodome y Web Crypto API al generar y verificar hashes SHA-256 para archivos grandes, evaluando su eficiencia en la validación de integridad de datos sensibles.

SHA-256 es una función de dispersión criptográfica que toma una entrada de cualquier longitud y produce una salida de 256 bits (32 bytes) altamente resistente a colisiones y preimágenes. Gracias a su diseño, incluso un cambio mínimo en los datos de origen genera un hash completamente diferente, lo que la convierte en una herramienta fundamental para verificar la integridad de archivos y comunicaciones en entornos de alta seguridad.

Para ello se realizará lo siguiente:

- Se generará un archivo simulado según el tamaño del usuario proporcionado.
- Se calculará el hash SHA-256 del contenido.
- Se volverá a calcular el hash para verificar que coincide con el valor original.
- Se repetirá el proceso múltiples veces para evaluar rendimiento promedio.

Se utilizará lo siguiente:

- En JavaScript: se usará *crypto.subtle.digest()* del *Web Crypto API*.
- En PyScript: se usará *SHA256.new()* y *.digest()* de *PyCryptodome*.

Se medirá lo siguiente:

- Tiempo total para generación y verificación de hashes.
- Tiempo promedio por operación.
- Uso de memoria durante el proceso.
- Tamaño del archivo procesado.

Repeticiones: 5

Tamaño: 10MB

PLT: 2000ms - 5000ms

Operation	PyScript	JavaScript
Avg hash time	189.01 ms	9.03 ms
Avg verify time	188.91 ms	8.82 ms
Total op time	1889.62 ms	89.25 ms
Total time	1917.90 ms	89.33 ms

Tabla 48. Resultados criptografía: prueba 1-1.

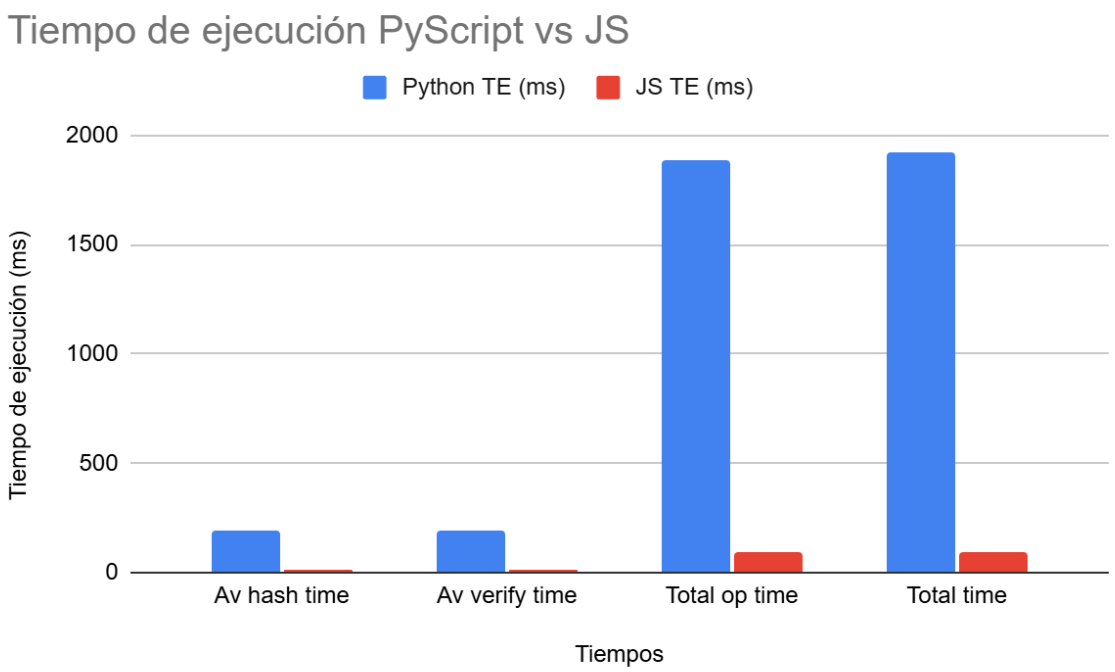

Figura 43. Diagrama de TE de criptografía: prueba 1-1.

Observación 30.
JavaScript es mucho más rápido porque usa la *Web Crypto API* nativa y acelerada por hardware (~9 ms por hash), mientras que PyScript ejecuta PyCryptodome sobre

WebAssembly, añadiendo overhead de interpretación y gestión, tardando ~189 ms. Esto muestra que para criptografía en navegador, *Web Crypto API* es claramente más eficiente.

6.6.2. Prueba 2 - Cifrado y Descifrado Simétrico con AES-GCM (128 bits)

Evaluaremos la eficiencia de ambos entornos en tareas de cifrado/descifrado simétrico con datos sensibles, utilizando AES-GCM con clave de 128 bits, verificando además la integridad del mensaje.

AES-GCM (Galois/Counter Mode) es un modo de cifrado que no solo protege la confidencialidad del mensaje, sino que también asegura su integridad. Usa una clave secreta y un número aleatorio (nonce) para cifrar los datos en bloques, y genera una etiqueta (tag) que permite detectar si el mensaje ha sido modificado. Al descifrar, se comprueba esta etiqueta: si no coincide, el mensaje se rechaza, garantizando que no fue alterado ni corrompido.

Para ello se realizará lo siguiente:

- Se generará una clave simétrica aleatoria de 128 bits.
- Se cifra un mensaje de 1 MB de texto aleatorio con AES-GCM.
- Se incluirá un nonce y una etiqueta de autenticación.
- Se descifra el mensaje y se verifica su integridad.

Se utilizará lo siguiente:

- En JavaScript: se usará *crypto.subtle.encrypt()* y *crypto.subtle.decrypt()* con AES-GCM.
- En PyScript: se usará *AES.new(..., AES.MODE_GCM)* desde PyCryptodome.

Se medirá lo siguiente:

- Tiempo total de cifrado y descifrado.
- Tiempo promedio por operación.
- Verificación de integridad: éxito o fallo.
- Uso de memoria.
- Tamaño del mensaje cifrado/descifrado.

Repeticiones: 50000

Tamaño: 0,1MB

PLT: 2000ms - 5000ms

Operation	PyScript	JavaScript
Avg encrypt time	1,62 ms	0,10 ms
Avg decrypt time	1,70 ms	0,24 ms
Integrity check	OK	OK
Integrity success	50_000/50_000	50_000/50_000
Total crypto time	165585,01 ms	17009,69 ms
Overall total time	165786, 77 ms	19187,87 ms

Tabla 49. Resultados criptografía: prueba 2-1.

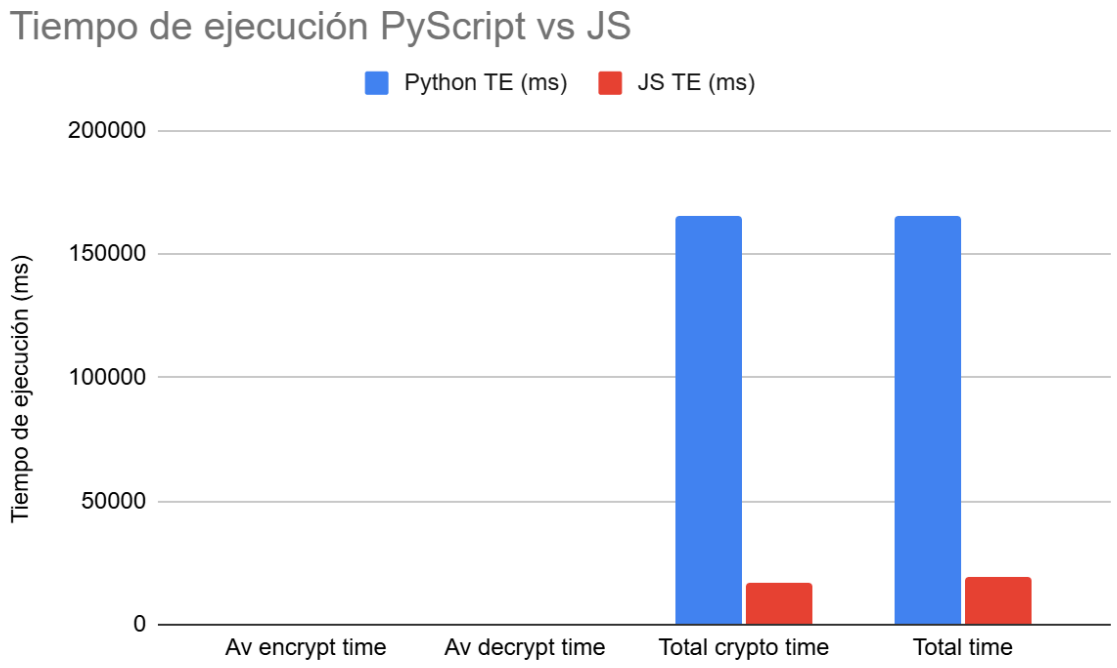

Figura 44. Diagrama de TE de criptografía: prueba 2-1.

Observación 31.
JavaScript cifra y descifra mucho más rápido (0,10 ms y 0,24 ms) porque usa la API nativa *crypto.subtle*, optimizada y sin capas intermedias. PyScript, en cambio, ejecuta PyCryptodome dentro de Pyodide (Python sobre WebAssembly), lo que añade un overhead significativo por conversiones y llamadas entre JS y WASM. Además, la

generación de números aleatorios en PyScript es menos directa, aumentando aún más el coste

6.7. Reconocimiento de patrones y clasificación de datos

En este último apartado se analiza el rendimiento de algoritmos de clasificación ejecutados en el navegador, comparando la implementación en PyScript mediante la librería TensorFlow frente a un modelo equivalente con TensorFlow.js.

Se evalúan tareas típicas de aprendizaje supervisado como el entrenamiento, la inferencia y la carga de modelos, utilizando datasets estándar y modelos ampliamente conocidos en el ámbito del machine learning. Sin embargo, al iniciar el desarrollo de este apartado, PyScript/Pyodide no tiene soporte para la librería TensorFlow de Python, por lo que finalmente se optó por utilizar scikit-learn como alternativa en el entorno Python. Por su parte, en el entorno JavaScript también se adaptaron los algoritmos para que mantuvieran una equivalencia funcional con los usados en Python. En lugar de TensorFlow.js, se optó por bibliotecas específicas que implementan los mismos modelos de aprendizaje automático disponibles en scikit-learn.

Para la primera prueba se empleó la biblioteca *ml-random-forest*, que replica la estructura de los modelos de bosque aleatorio de scikit-learn, La primera prueba que se implementa es la clasificación del dataset Iris. Este conjunto de datos contiene 150 muestras de flores clasificadas en tres especies distintas: Setosa, Versicolor y Virginica. Cada flor está representada por cuatro características numéricas: longitud y ancho del sépalo, y longitud y ancho del pétalo. En ambas implementaciones se entrena un modelo de bosque aleatorio [59] para predecir la especie a partir de estas características, utilizando la versión de RandomForestClassifier correspondiente en cada lenguaje. Los modelos se entrenan y ejecutan íntegramente en el navegador, permitiendo medir el tiempo de carga de datos, el entrenamiento, la predicción y los recursos utilizados en tiempo real.

La segunda prueba consiste en la clasificación del dataset Digits. Este conjunto de datos está formado por imágenes de dígitos manuscritos del 0 al 9, representadas como matrices de 8x8 píxeles, lo que se traduce en 64 características numéricas por muestra. Es un conjunto más complejo que Iris, con 1.797 muestras en total. En esta prueba se optó por usar el algoritmo *K-Nearest Neighbors* [60], que asigna una clase a cada nueva muestra comparándola con las muestras más cercanas del conjunto de entrenamiento. En PyScript se utilizó *KNeighborsClassifier* de scikit-learn, mientras que en JavaScript se empleó ml-knn, que ofrece una implementación funcionalmente similar. Como en la prueba anterior, los modelos fueron ejecutados directamente en el navegador, comparando el comportamiento de ambos entornos a nivel de tiempos de ejecución, eficiencia de predicción y consumo de recursos. Ambos experimentos permiten contrastar de forma efectiva el rendimiento y viabilidad del aprendizaje automático dentro del navegador utilizando PyScript frente a soluciones nativas en JavaScript.

6.7.1. Prueba 1 - Clasificación del dataset Iris

Las métricas registradas son:

- Tiempo de entrenamiento
- Tiempo de inferencia
- Precisión media (accuracy)
- Uso de memoria
- Tiempo total (carga + entrenamiento + inferencia)

Repeticiones: 10

PLT: 2000ms - 5000ms

Operation	PyScript	JavaScript
Av training time	256.05 ms	169.56 ms
Av Inference time	12.05 ms	1.89 ms
Av. Accuracy	100.00%	98.67 %
Model size	0.16 MB	0.28 MB
Total time	2717.59 ms	1714.68 ms

Tabla 50. Resultados reconocimiento de patrones: prueba 1-1.

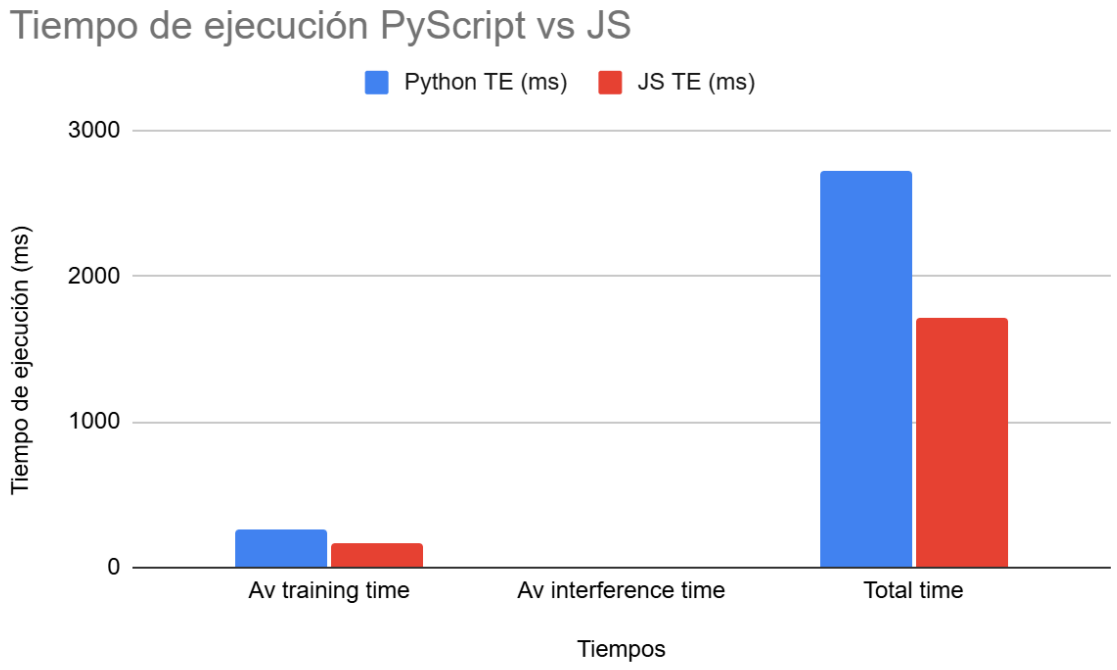

Figura 45. Diagrama de TE reconocimiento de patrones: prueba 1-1.

Observación 32.
El entrenamiento es similar en ambos, sin embargo, JavaScript consigue tener mejor rendimiento, por otra parte. El tiempo de interferencia es 10 veces mejor que PyScript.

6.7.2. Prueba 2- Clasificación del dataset Digits

Las métricas registradas son:

- Tiempo de entrenamiento
- Tiempo de inferencia
- Precisión media (accuracy)
- Uso de memoria
- Tiempo total (carga + entrenamiento + inferencia)

Repeticiones: 50

PLT: 2.000ms - 5.000 ms

Operation	PyScript	JavaScript
Av training time	1.52 ms	1.16 ms
Av Inference time	27.89 ms	527.15 ms
Av. Accuracy	98,70 %	98.48 %
Model size	0.62 MB	0.25 MB
Total time	6816,04 ms	26420.47 ms

Tabla 51. Resultados reconocimiento de patrones: prueba 2-1.

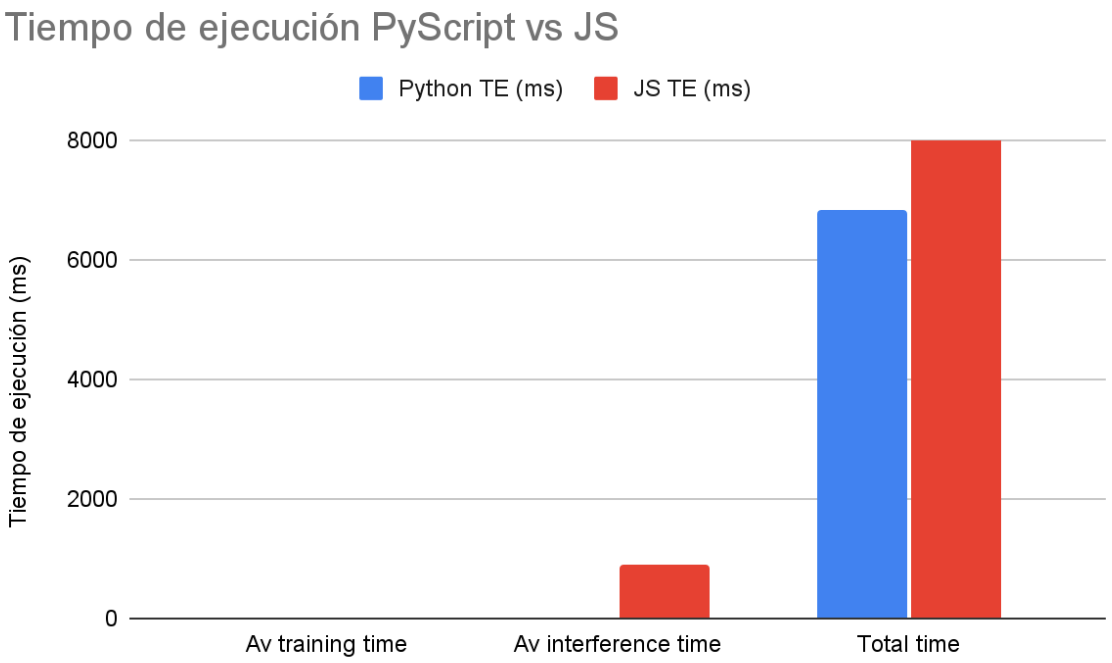

Figura 46. Diagrama de TE reconocimiento de patrones: prueba 2-1.

Observación 33.
En este caso, el modelo empleado ha favorecido a PyScript, lo que nos indica que realmente no podemos obtener un mejor rendimiento claro, si no, en base al modelo, puede que vaya mejor en una tecnología que otra.

6.8. “Glue Code”

En este apartado se presentan los trozos de código que más interactúan entre los componentes desarrollados en JavaScript y en PyScript dentro de un mismo entorno web. Este tipo de código actúa como intermediario entre ambos lenguajes, facilitando la coordinación de eventos, el intercambio de datos y la ejecución sincronizada de pruebas de rendimiento.

El uso de PyScript dentro de una aplicación web implica no solo la ejecución de código Python en el navegador, sino también su correcta integración con el entorno JavaScript, que continúa siendo responsable de muchas de las funciones nativas del navegador, como el manejo del DOM, la gestión de eventos o la interacción con APIs web.

Por tanto, este apartado recoge los fragmentos más relevantes de ese “Glue Code” que permite que ambos lenguajes trabajen en conjunto.

Page load time (PLT)

Un ejemplo representativo de este tipo de integración es el siguiente fragmento, que permite calcular el Page Load Time (PLT), es decir, el tiempo transcurrido desde que comienza a ejecutarse el script hasta que PyScript finaliza su carga e inicialización:

```
let startTime = performance.now();


document.addEventListener('py:ready', () => {
    let endTime = performance.now();
    let loadTime = endTime - startTime;
    document.getElementById('output').innerText +=
        `PLT: ${loadTime.toFixed(2)} ms`;
});
```

Código 3. Código del PLT implementado en el proyecto.

Este código se ubicaría en el index.html, y funciona de la siguiente manera:

- Se registra el tiempo actual al principio mediante *performance.now()*.
- El evento *py:ready* es lanzado automáticamente por PyScript cuando su entorno se encuentra completamente cargado en el navegador.
- Al capturar este evento desde JavaScript, se mide el tiempo total de carga del entorno Python en el navegador.
- Finalmente, el tiempo es mostrado en pantalla, lo que permite evaluar el coste de inicialización de PyScript respecto al entorno nativo de JavaScript.

Este tipo de integración no solo es útil para medir métricas de rendimiento, sino que constituye una capa de conexión esencial para pruebas comparativas como las desarrolladas en este trabajo.

Onclick y py-click

Otro ejemplo representativo de glue code lo encontramos en la forma en que se enlaza la ejecución de código tanto en JavaScript como en Python desde una misma interacción del usuario. En este caso, se trata de un botón que lanza dos funciones diferentes; una escrita en JavaScript y otra en Python, al mismo tiempo, mediante los atributos onclick y py-click:

```
<button onclick="runPyBenchmark()" py-click="run_py_benchmark">
    Run in PyScript
</button>

###

function runPyBenchmark() {
    startTimerWebAssembly = performance.now();
}

###

def run_py_benchmark(event):
    // Código
```

Código 4. Código de Onclick y Py-click de ejemplo.

La clave aquí está en la combinación de onclick (para JavaScript) y py-click (para PyScript). Gracias a esto, una sola acción del usuario, hacer clic en el botón, lanza código en dos lenguajes distintos de forma coordinada.Esto nos sirve para:

- Sincronizar medidas de rendimiento, como el tiempo exacto desde que se activa el benchmark hasta que PyScript comienza sus cálculos.
- Unificar la interfaz de usuario, permitiendo que tanto las operaciones Python como los controles JavaScript compartan los mismos elementos visuales sin duplicación de lógica.
- Comparar tiempos y comportamientos, especialmente útil en pruebas como la multiplicación de matrices, donde el rendimiento entre entornos puede variar significativamente.

Este tipo de glue code refleja cómo PyScript y JavaScript pueden convivir en una misma aplicación web, ejecutando tareas complementarias de forma sincronizada.

Llamadas mediante “js.Function”

Otra muestra clara del funcionamiento del glue code es la capacidad que ofrece PyScript para llamar funciones escritas en JavaScript desde código Python. Esto se hace posible gracias al módulo *.js* que actúa como un puente entre ambos lenguajes, permitiendo acceder a funciones y variables definidas en el contexto JavaScript del navegador. Este mismo módulo, lo podemos dar a entender que es un proxy del objeto global *window* de JavaScript.

Mediante eso, se permite que desde Python se puedan invocar funciones, acceder a variables o interactuar con objetos definidos en JavaScript.

```
<script>
    function clearCell(elementId) {
        document.getElementById(elementId).textContent = '';
    }
</script>

###

import js  # type: ignore

def run_py_benchmark(event):
    js.clearCell("pyscript-output")
```

Código 5. Código del módulo js de ejemplo.

Llamadas desde JavaScript a PyScript

Otro ejemplo del glue code en este proyecto es la invocación de una función Python desde un archivo JavaScript. En este caso, se define una función global runPy dentro del objeto window, que se puede activar desde un botón o un evento de interfaz. Su objetivo es preparar la ejecución del benchmark en PyScript, mostrando un loader de carga y midiendo el tiempo de ejecución. Es decir, de esta forma, es posible llamar las funciones del código de Python directamente desde JavaScript.

```
(window as any)["runPy"] = function () {
```

```
    (window as any)["showExecutionLoader"]();

    requestAnimationFrame(() => {
        setTimeout(() => {
            if ((window as any)["run_py_benchmark"]) {
                startTimerWebAssembly = performance.now();
                (window as any)["run_py_benchmark"](event);
            } else {
                console.error('run_py_benchmark no está definido');
                (window as any)["hideExecutionLoader"]();
            }
        }, 0);
    });
};
```

La función *run_py_benchmark*, escrita en Python, es expuesta automáticamente al contexto JavaScript por PyScript, por lo que puede invocarse directamente como *window.run_py_benchmark*(...). Esta integración directa permite que el frontend controle la ejecución del código Python sin necesidad de establecer comunicación externa mediante API o WebSocket.

Código 6. Llamadas desde JavaScript a Python.

7. Conclusiones finales

A lo largo de este TFG hemos comparado detalladamente JavaScript y PyScript en el navegador, evaluando cómputo intensivo, manejo de datos, gráficos, concurrencia, criptografía y clasificación. A continuación, presentamos las conclusiones clave de cada apartado.

7.1. Cálculos matemáticos intensivos

JavaScript destaca por su velocidad en el navegador gracias al motor V8 y la compilación Just-In-Time (JIT), lo que lo hace muy eficaz en bucles y operaciones aritméticas repetitivas, en especial en algoritmos. Esta agilidad viene acompañada de un consumo de memoria elevado, necesario para técnicas como inline caching [61] y hidden classes [62]. Por su parte, Python en servidor logra un equilibrio óptimo entre rendimiento y uso de recursos gracias a bibliotecas científicas como NumPy y SciPy, que ejecutan código nativo en C y aprovechan instrucciones SIMD [63] del hardware. PyScript, finalmente, aunque mantiene un perfil de memoria relativamente bajo, sufre un notable retraso en los cálculos pesados debido a la capa de WebAssembly y la imposibilidad de acceder directamente a aceleraciones de hardware.

- JavaScript (navegador)
  - Velocidad superior por el motor V8 y JIT.
  - Alto uso de memoria.
- Python (servidor)
  - Excelente rendimiento vs. uso de recursos.
  - NumPy/SciPy en C con aceleración hardware.
- PyScript (navegador)
  - Rendimiento inferior por la capa WebAssembly.
  - Consumo de memoria moderado, a costa de tiempos de procesamiento más altos

7.2. Procesamiento de grandes volúmenes de datos.

En este apartado, aunque JavaScript optimiza bucles y funciones repetitivas, al carecer de un ecosistema de herramientas como NumPy o pandas, el trabajo con grandes colecciones se vuelve más lento y consume más memoria.

- JavaScript
  - Desempeño competitivo en volúmenes pequeños, gracias al JIT de V8.
  - Carece de librerías especializadas para manejar estructuras complejas (arrays de gran tamaño, transformaciones masivas), por lo que su eficiencia cae notablemente cuando aumenta el volumen de datos.

Después de la fase de arranque, en la que PyScript tarda en cargar su entorno WebAssembly, las operaciones masivas sobre arrays y Data Frames se ejecutan de forma muy eficiente, compensando con interés el coste inicial y superando a JavaScript en escenarios de procesamiento distribuido de datos.

- PyScript

- Utiliza un modelo de datos columnar y operaciones vectorizadas de NumPy/pandas, lo que le permite procesar grandes volúmenes de datos con mayor fluidez.
- La inicialización de cada worker es lenta (debido a la carga de Pyodide), pero una vez arrancados distribuyen mejor la carga computacional y escalan con más consistencia.

7.3. Renderizado de gráficos complejos

Al interactuar directamente con las capacidades gráficas del navegador, JavaScript ejecuta operaciones de dibujo de forma muy eficiente y sin pasos de conversión, lo que es clave para generar imágenes estáticas y animaciones en tiempo real sin degradar la experiencia del usuario.

- JavaScript
  - Puede usar Canvas, SVG, WebGL y OffscreenCanvas sin capas intermedias.
  - OffscreenCanvas permite dibujar fuera del hilo principal, evitando bloqueos de UI y aprovechando múltiples hilos.
  - Las APIs nativas del navegador están diseñadas para alto rendimiento en gráficos, lo que se traduce en tiempos de renderizado muy bajos y gran fluidez.

Aunque PyScript puede generar gráficos mediante bibliotecas como Matplotlib o Plotly, todos los pasos de renderizado acaban delegando a JavaScript, introduciendo latencia adicional y dependencia a JavaScript.

- PyScript
  - No puede invocar Canvas o WebGL directamente; debe pasar el trabajo a JavaScript mediante el objeto js.
  - Cada llamada gráfica implica traducir datos de Python a JS, pintar en el canvas y, a veces, convertir formatos (por ejemplo, de matriz NumPy a imagen PNG).
  - Estas traducciones y el overhead de WebAssembly ralentizan el renderizado y complican el código, reduciendo la fluidez en gráficos interactivos.

7.4. Manejo de múltiples solicitudes concurrentes

Los resultados de concurrencia muestran que, aunque JavaScript conserva una ligera ventaja en velocidad media gracias a su event loop [64] y Web Workers, PyScript puede acercarse al rendimiento de JavaScript en escenarios de múltiples solicitudes concurrentes cuando la carga se centra en la gestión de llamadas más que en la computación intensiva. Gracias a la interoperabilidad de Pyodide con las APIs nativas (por ejemplo fetch),

PyScript delega gran parte del trabajo en el motor del navegador, logrando tiempos de respuesta muy similares a los de JavaScript.

- JavaScript (navegador)
    - Se beneficia de un modelo de ejecución asincrónico sólido: event loop , Promises, async/await y Web Workers con inicialización casi instantánea.
    - Gestiona múltiples fetch o conexiones WebSocket sin bloquear el hilo principal, manteniendo un control preciso de los tiempos de respuesta gracias al motor V8.
- PyScript (navegador)
    - A pesar de su capa de WebAssembly y la necesidad de convertir entre Python y JavaScript, utiliza internamente el fetch del navegador a través de Pyodide, lo que le permite manejar concurrentemente varias solicitudes con eficiencia.
    - La sobrecarga de PyScript en concurrencia ligera es mínima, aunque podría crecer en escenarios más agresivos o con procesamiento interno más pesado, ya que no puede explotar hilos nativos o E/S de bajo nivel.

7.5. Cálculo y verificación de integridad en datos sensibles

En el ámbito criptográfico, JavaScript superó ampliamente a PyScript debido al acceso directo a la API SubtleCrypto, que aprovecha optimizaciones a nivel del sistema y aceleración por hardware. Esto permite realizar operaciones como el hashing SHA-256 o el cifrado AES-GCM con gran rapidez y eficiencia. Por el contrario, PyScript depende de bibliotecas puramente implementadas en Python y ejecutadas sobre WebAssembly, lo que elimina la posibilidad de acceder a extensiones como AES-NI o instrucciones SIMD, generando un cuello de botella en términos de velocidad y rendimiento.

7.6. Reconocimiento de patrones y clasificación de datos

En las tareas de clasificación utilizando el conjunto de datos Iris, tanto JavaScript como PyScript lograron inicialmente resultados similares en precisión y consumo de recursos. No obstante, al aumentar la complejidad de las pruebas al cambiar al dataset Digits, JavaScript demostró una capacidad superior para escalar en eficiencia, gracias a la combinación de su motor V8 y librerías optimizadas. PyScript, aunque correcto en funcionalidad, sufrió limitaciones propias de su ejecución sobre WebAssembly, mostrando tiempos de respuesta más altos y menor adaptabilidad bajo carga intensiva.

7.7. Conclusión general

El proyecto demostró que JavaScript mantiene una ventaja consistente en la mayoría de los escenarios evaluados, especialmente en algoritmos y operaciones en tiempo real, tanto en implementaciones nativas como en ejecuciones interactivas en el navegador. Esta ventaja se debe en parte a su estrecha integración con el motor V8 y al soporte directo del entorno del navegador, lo que permite una ejecución optimizada y acceso a recursos nativos como los Web Workers, la aceleración por hardware y operaciones eficientes sobre memoria e hilos.

PyScript, por su parte, presenta un rendimiento notable en tareas de procesamiento de datos, donde las bibliotecas especializadas de Python como pandas o NumPy ofrecen estructuras y funciones avanzadas para el análisis eficiente de grandes volúmenes de datos. Esto lo convierte en una herramienta prometedora para ciertos tipos de cargas de trabajo, especialmente aquellas que se benefician del ecosistema científico de Python.

Sin embargo, para aprovechar realmente las capacidades de PyScript es necesario considerar dos contextos muy concretos. El primero corresponde a entornos con recursos limitados, donde puede aceptarse una ejecución más lenta a cambio de flexibilidad y simplicidad en el desarrollo. Aunque este tipo de entorno se asemeja a sistemas embebidos [65], en la práctica estos requieren respuestas inmediatas, lo que descarta a PyScript como solución viable debido a sus tiempos de inicialización. El segundo contexto es el procesamiento intensivo de datos, donde el tiempo de respuesta no necesita ser inmediato pero sí razonable. En este caso, PyScript puede ofrecer ventajas, siempre que se asuma un coste inicial elevado en la carga del entorno y los módulos.

Una de las limitaciones clave de PyScript que se ha observado a lo largo del proyecto, es su incapacidad para aprovechar directamente la aceleración por hardware o las optimizaciones específicas del navegador. Esto se debe a que PyScript funciona como una capa de interpretación de código Python sobre WebAssembly, a través de Pyodide. A diferencia de JavaScript, que está altamente optimizado y ejecutado directamente por el motor del navegador, PyScript introduce una capa adicional de abstracción que impide un acceso eficiente a recursos como optimizaciones con la memoria RAM como JavaScript, que utiliza realmente mucho la memoria para optimizar los tiempos de ejecución.

En lo que respecta al uso de Web Workers, las diferencias también fueron notables. JavaScript mostró una escalabilidad efectiva al trabajar con múltiples workers, con tiempos de inicialización mínimos. PyScript, si bien mostró una mejora considerable en escalabilidad una vez completada la fase de arranque, sufre una penalización significativa en la creación de workers, debido a la necesidad de cargar y preparar el entorno Python para cada uno de ellos.

En conclusión, PyScript es adecuado para aplicaciones donde la escalabilidad de procesos y la reutilización del ecosistema Python son esenciales, y donde se puede tolerar un mayor tiempo de carga inicial. Sin embargo, no es recomendable para escenarios que requieran alta interactividad, respuesta inmediata, acceso a operaciones paralelas de bajo nivel o aprovechamiento directo del hardware, donde JavaScript sigue siendo la opción más eficiente y práctica dentro del navegador.

Referencias

[1] JavaScript. URL: https://developer.mozilla.org/es/docs/Web/JavaScript
[2] React. URL: https://es.react.dev
[3] Vue. URL: https://vuejs.org
[4] Angular. URL: https://angular.dev
[5] Python. URL: https://www.python.org
[6] PyScript. URL: https://pyscript.net
[7] Pyodide. URL: https://pyodide.org/en/stable/
[8] WebAssembly. URL: https://webassembly.org
[9] C. URL: https://informatica.uv.es/estguia/ATD/apuntes/laboratorio/Lenguaje-C.pdf
[10] C++. URL: https://www2.eii.uva.es/fund_inf/cpp/temas/1_introduccion/introduccion.html
[11] benchmarks. URL: https://www.tableau.com/es-es/learn/articles/what-is-a-benchmark
[12] Astro. URL: https://astro.build
[13] framework. URL: https://unirfp.unir.net/revista/ingenieria-y-tecnologia/framework/
[14] NumPy. URL: https://numpy.org
[15] pandas. URL: https://pandas.pydata.org
[16] Matplotlib. URL: https://matplotlib.org
[17] DOM. URL: https://developer.mozilla.org/es/docs/Web/API/Document_Object_Model/Introduction
[18] Diagrama de Gantt. URL: https://asana.com/es/resources/gantt-chart-basics
[19] Backend. URL: https://descubrecomunicacion.com/que-es-backend-y-frontend/
[20] Frontend. URL: https://descubrecomunicacion.com/que-es-backend-y-frontend/
[21] HTML. URL: https://developer.mozilla.org/es/docs/Web/HTML
[22] CSS. URL: https://developer.mozilla.org/es/docs/Web/CSS
[23] Node.js. URL: https://nodejs.org/es
[24] Full Stack. URL: https://udit.es/actualidad/que-es-y-como-formarse-en-desarrollo-full-stack/
[25] OOP. URL: https://www.computerweekly.com/es/definicion/Programacion-orientada-a-objetos-OOP
[26] Sandbox. URL: https://www.proofpoint.com/es/threat-reference/sandbox
[27] APIs. URL: https://www.xataka.com/basics/api-que-sirve
[28] bytecode. URL: https://blog.thedojo.mx/2023/01/22/entendiendo-el-bytecode.html
[29] Emscripten. URL: https://emscripten.org/docs/introducing_emscripten/about_emscripten.html
[30] CPython. URL: https://docs.python.org/es/3.8/reference/introduction.html
[31] Scikit-learn. URL: https://scikit-learn.org
[32] RegEx. URL: https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/regex-en-python/
[33] PVM. URL: https://panutanur.medium.com/pvm-python-virtual-machine-df83ca6a79a6
[34] SciPy. URL: https://scipy.org
[35] DataFrame. URL: https://datascientest.com/es/que-es-un-dataframe
[36] CSV. URL: https://www.geeknetic.es/Archivo-CSV/que-es-y-para-que-sirve
[37] JSON. URL: https://es.wikipedia.org/wiki/JSON
[38] Plotly. URL: https://ironpdf.com/es/python/blog/python-help/plotly-python/
[39] Django. URL: https://aws.amazon.com/es/what-is/django/
[40] Asincronismo. URL: https://pycun.net/blogs/que-es-el-asincronismo/
[41] Concurrismo. URL: https://codigofacilito.com/articulos/programacion-concurrente
[42] Asyncio. URL: https://docs.python.org/es/3/library/asyncio.html
[43] PyCryptodome . URL: https://pypi.org/project/pycryptodome/
[44] Machine learning. URL: https://www.iberdrola.com/innovacion/machine-learning-aprendizaje-automatico

[45] Worker. URL: https://developer.mozilla.org/es/docs/Web/API/Web_Workers_API
[46] Tailwind CSS. URL: https://tailwindcss.com
[47] TOML. URL: https://es.wikipedia.org/wiki/TOML
[48] Hash. URL: https://latam.kaspersky.com/blog/que-es-un-hash-y-como-funciona/2806/?srsltid=AfmBOorBr0XboqtasLUCvpH2BzcPPwIDdY-4MStJu5eLIdoQ8juaTNhC
[49] CPU. URL: https://es.wikipedia.org/wiki/Unidad_central_de_procesamiento
[50] RAM. URL: https://es.wikipedia.org/wiki/Memoria_de_acceso_aleatorio
[51] V8. URL: https://www.cloudflare.com/es-es/learning/serverless/glossary/what-is-chrome-v8/
[52] JIT. URL: https://blog.thedojo.mx/2023/01/18/compilacion-just-in-time-que-es.html
[53] Tupla. URL: https://ellibrodepython.com/tuplas-python
[54] Canvas, SVG y WebGL. URL: https://www.yworks.com/blog/svg-canvas-webgl
[55] fetch/WebSocket. URL: https://ably.com/topic/websockets-vs-http
[56] Criba de Eratóstenes. URL: https://www.superprof.es/diccionario/matematicas/aritmetica/criba-eratostenes.html
[57] Gauss–Legendre. URL: https://es.wikipedia.org/wiki/Algoritmo_de_Gauss-Legendre
[58] BigNumber. URL: https://docs.ethers.org/v5/api/utils/bignumber/
[59] Bosque aleatorio. URL: https://www.ibm.com/es-es/think/topics/random-forest
[60] K-Nearest Neighbors. URL: https://www.ibm.com/mx-es/think/topics/knn
[61] Inline caching. URL: javascript.plainenglish.io/v8-engine-and-inline-caching-in-javascript-fef80054a551?gi=91c5c920479c
[62] Hidden classes. URL: https://v8.dev/docs/hidden-classes
[63] SIMD. URL: https://es.wikipedia.org/wiki/SIMD
[64] Event loop. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Execution_model
[65] Sistemas embebidos. URL: https://www.ceupe.com/blog/sistema-embebido.html