Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Simulador didáctico de arquitectura de computadores: aplicación de metodologías de integración y mantenimiento

Didactic computer architecture simulator: application of integration and maintenance methodologies

Óscar Carrasco Benítez

La Laguna, 13 de septiembre de 2022

D. **Iván Castilla Rodríguez**, con N.I.F. 78.565.451-G profesor Contratado Doctor adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor.

C E R T I F I C A (N)

Que la presente memoria titulada:

”*Simulador didáctico de arquitectura de computadores: aplicación de metodologías de integración y mantenimiento*”

ha sido realizada bajo su dirección por D. **Óscar Carrasco Benítez**,
con N.I.F. 42.240.843-D.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 13 de septiembre de 2022

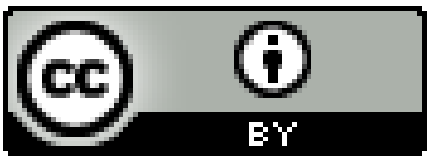
Agradecimientos

A mi tutor, Iván Castilla, por haberme acompañado hasta aquí con infinita paciencia, ayudándome a progresar semana tras semana.

A Adrián Abreu, por establecer las bases del desarrollo y contribuir a la organización y puesta en marcha del trabajo.

A Miranda, por estar ahí durante muchas de las extensas horas de desarrollo del proyecto y redacción de la memoria.

Licencia



Resumen

Uno de los mayores retos en el desarrollo moderno de software es la gestión y mantenimiento de proyectos. A lo largo de los años, los sistemas informáticos han ido adquiriendo complejidad de manera continuada y progresiva, algo que conlleva un aumento gradual de las dificultades que se deben afrontar según se van implementando nuevas características y funcionalidades.

En la actualidad, el ecosistema más popular y completo para el desarrollo de proyectos de software está consolidado por las tecnologías Web. Debido al veloz ritmo de evolución de dichas tecnologías, se requiere de un conjunto de metodologías y buenas prácticas adecuado, capaz de permitir el desarrollo y despliegue de aplicaciones con facilidad y estabilidad.

Se presenta un caso favorable para indagar en esta problemática y proyectar soluciones al respecto: un simulador Web de arquitecturas de computadores conocido como SIMDE, cuyo desarrollo ha carecido de actividad durante años y abre multitud de posibilidades a la hora de investigar y aplicar mejoras en su mantenimiento y escalabilidad para las futuras líneas de desarrollo.

El principal objetivo de este trabajo ha sido realizar mejoras sustanciales en, esencialmente, todas las facetas que conforman la gestión de un proyecto de software a través del marco de trabajo que habilita el desarrollo del SIMDE: gestión de dependencias, pruebas de software, soporte a la retrocompatibilidad, internacionalización, seguridad, integración y despliegue.

Palabras clave: Arquitectura de computadores, Tecnologías Web, Gestión de dependencias, Internacionalización, Testing, Integración continua, Despliegue continuo.

Abstract

One of the biggest challenges in modern software development is project management and maintenance. Over the years, computer systems have steadily and progressively become more complex, leading to a gradual increase in the difficulties to be faced as new features and functionalities are implemented.

Currently, the most popular and complete ecosystem for the development of software projects is consolidated by Web technologies. Due to the fast-paced evolution of these technologies, an adequate set of methodologies is required to allow the development and deployment of applications with ease and stability.

A favorable case study is presented to investigate this problem and apply solutions to it: a Web simulator of computer architectures known as SIMDE, whose development has presented minor activity for years and opens many possibilities when it comes to research and implementation of improvements regarding its maintenance and scalability for future lines of development.

The main objective of this work has been to make substantial improvements in essentially all facets of software project management, through the framework that enables the development of SIMDE: dependency management, software testing, support for backwards compatibility, internationalization, security, integration and deployment.

Keywords: Computer architecture, Web technologies, Dependency management, Internationalization, Testing, Continuous integration, Continuous deployment.

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

1.1. Motivación para el trabajo

En todo proyecto de software es de vital importancia considerar la aplicación de metodologías y flujos de trabajo que asistan, una vez las bases de código crecen lo suficiente, al mantenimiento del proyecto. Si bien gestionar estas dificultades en el entorno laboral y en el ámbito del desarrollo de software libre es algo habitual, resulta complicado adaptar el aprendizaje y la práctica de estas habilidades durante el transcurso de un grado universitario, dado que el trabajo práctico rara vez alcanza a presentar la necesidad de implementar dichas metodologías.

Un proyecto recurrente en los trabajos de fin de grado es el *SIMDE*[1], un simulador web de arquitectura de computadores. Su potencial de mejora en materias de automatización de tareas asociadas al desarrollo lo vuelve ideal para convertirse en un marco de referencia que comprenda y muestre diversas características y tareas que faciliten la integración y mantenimiento del proyecto.

1.2. Antecedentes y estado inicial

Inicialmente, el SIMDE fue concebido en el año 2004 con el nombre de *"Simulador para Planificación Dinámica y Estática"*[2], siendo este realizado como proyecto de fin de carrera de la Ingeniería Superior en Informática por el ex-alumno y actual profesor contratado doctor y tutor de este trabajo de fin de grado, Iván Castilla.

La gran utilidad a nivel educativo de la herramienta ha causado que, poco más de una década después del desarrollo inicial, varios ex-alumnos comiencen a evolucionar el proyecto hacia el mundo de las tecnologías Web (Figura 1.1).

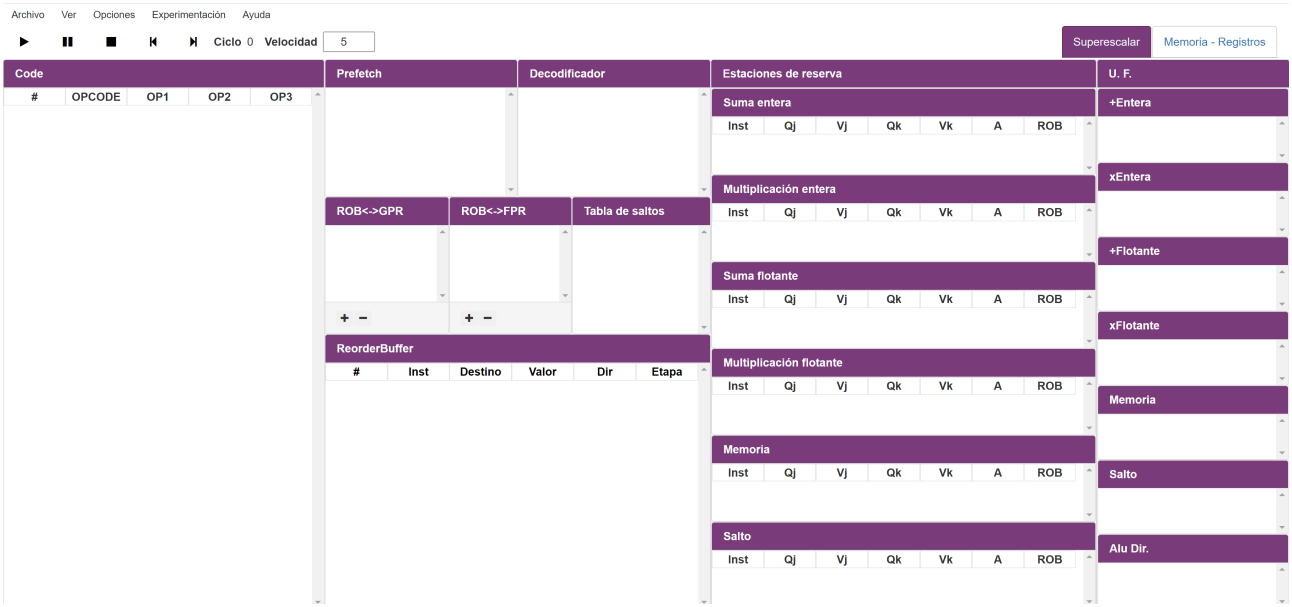


Figura 1.1: Interfaz de la máquina superescalar del SIMDE Web

Los trabajos realizados a partir de entonces son los siguientes:

- Reescritura del proyecto y adaptación a las tecnologías web en el trabajo *"Simulador didáctico de arquitectura de computadores"*[3], realizado por el ex-alumno Adrián Abreu.
- Diseño de un sistema de gamificación como refuerzo al aprendizaje en el trabajo *"Plataforma de ludificación de un simulador didáctico de arquitectura de computadores"*[4], realizado por el ex-alumno Antonio López.
- Implementación de un simulador de máquina *Very Long Instruction Word*, abreviado como VLIW[5] en el trabajo *"Simulador didáctico de una Arquitectura de planificación estática"*[6], realizado por la ex-alumna Melissa Díaz.

1.2.1. Flujo de desarrollo

Actualmente, el desarrollo del proyecto en su totalidad se encuentra en GitHub[1]. El flujo de desarrollo es bastante simple: todas las confirmaciones de cambios han sido realizadas en el mismo repositorio raíz; si bien existían algunas instancias de asuntos, o *issues* en el repositorio del proyecto, estas no poseen profundidad, careciendo de asociación a las solicitudes de incorporación, o *pull requests*. Se creaba una rama en el mismo repositorio para subir los cambios y esta era fusionada posteriormente en la rama principal, actualmente nombrada *main*.

1.2.2. Gestión de dependencias

Las dependencias del proyecto se han mantenido desactualizadas desde el año 2018. Existen ciertos casos de obsolescencia considerables, como el de *Webpack*[7], un sistema de construcción y empaquetado para el despliegue, su versionado ha progresado desde la versión 3 a la 5, lo cual implica el seguimiento de dos guías de migración, una por versión, para actualizar las configuraciones asociadas a dicha herramienta.

El sistema de bloqueo de dependencias fija de manera estricta la numeración para la versión de cada módulo. Esto reduce significativamente la posibilidad de que el proyecto se vea afectado por la inyección de código malicioso[8] en una posterior versión menor de las dependencias.

Cabe destacar que una de las dependencias, *node-sass*, requería de una versión obsoleta de *Node.js* durante el proceso de instalación. En el caso de utilizar una versión de *Node.js* que careciese de compatibilidad con la API, la instalación de dependencias resultaría fallida a la hora de compilar el módulo. Esto directamente impedía la el despliegue del proyecto en un entorno actual de desarrollo o producción, dado el extendido uso de las versiones modernas de *Node.js* que ya se encuentran disponibles los gestores de paquetería como *apt*, de *Debian* o *dnf*, de *Fedora*.

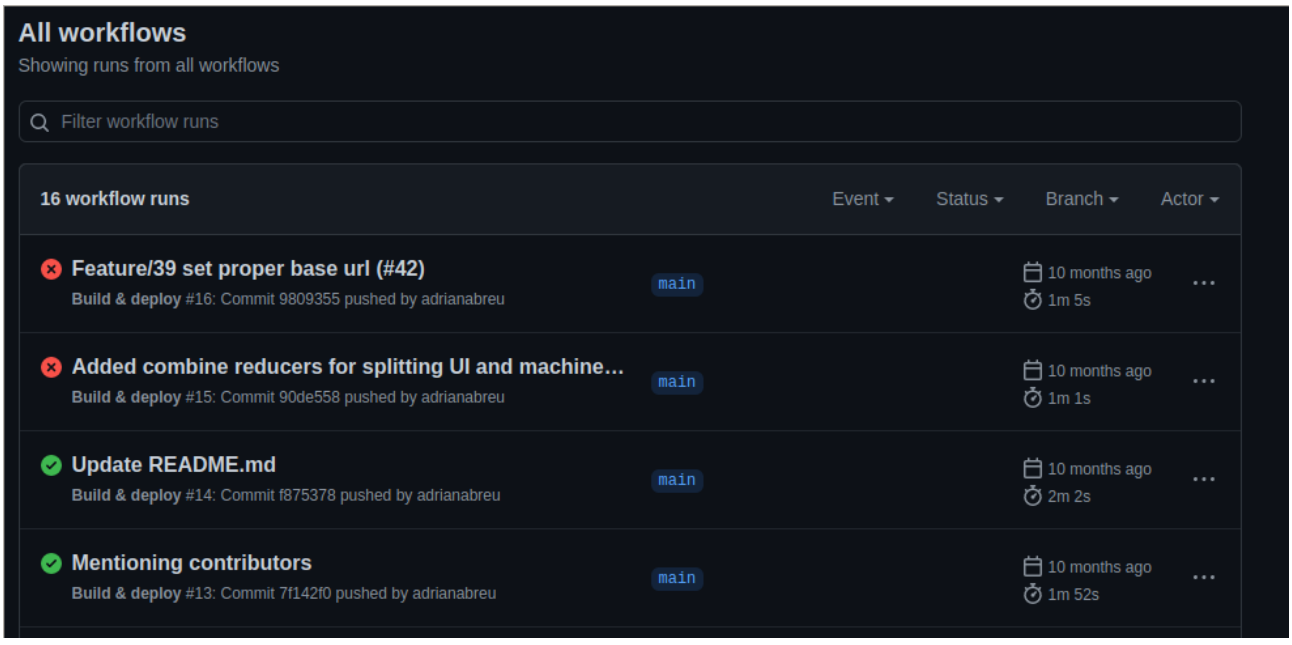


Figura 1.2: Interfaz principal de GitHub Actions

1.2.3. Integración y despliegue continuo

En el estado inicial del trabajo, se puede apreciar la existencia de una tarea para la construcción y despliegue de la aplicación establecida en los flujos de trabajo de *GitHub Actions*, un sistema propietario de GitHub para la automatización de tareas tales como ejecutar tests, y multitud de integraciones en otros sistemas de *Cloud Computing* y, en general, infraestructura de tipo *Software as a Service*.

Múltiples ejecuciones de esta tarea, la cual se realiza por cada nueva confirmación en el repositorio, son mostradas en la Figura 1.2.

Se encuentra, además, una configuración de *Docker* en un fichero *Dockerfile* localizado en la raíz del proyecto, la cual encapsula toda la base de código en una imagen con el propósito de usarla como método de instanciación de contenedores para el desarrollo.

1.3. Objetivos del trabajo

Mientras que la mayoría de objetivos de este trabajo están directamente dirigidos a implementar técnicas y configurar herramientas asociadas a la gestión de proyectos de software, también se realiza la implementación de una nueva característica en el proyecto, y se resuelven algunos fallos relacionados con la gestión del estado en el código.

Estos últimos objetivos tienen una gran importancia en relación al resto del proyecto, pues nos permiten validar el nuevo flujo de desarrollo, demostrar las mejoras en calidad y dejar registradas en el proyecto las directivas de desarrollo a seguir a partir de entonces.

En los siguientes apartados se presentan y se detallan, de manera individual, los objetivos necesarios para la elaboración del trabajo:

1.3.1. Gestión de dependencias

La gestión de dependencias plantea objetivos relacionados con los módulos esenciales para la construcción del proyecto. Es necesario comenzar con las tareas asociadas a este objetivo justo al comienzo, puesto que las dependencias deben estar en la última versión para asegurar el correcto funcionamiento de las herramientas y de la relación entre los módulos.

Este objetivo también plantea tareas de automatización para la actualización y control de las dependencias, donde se pretende utilizar *Renovate*, un servicio para la automatización de actualizaciones y notificaciones sobre el estado de las dependencias (Figura 1.3).

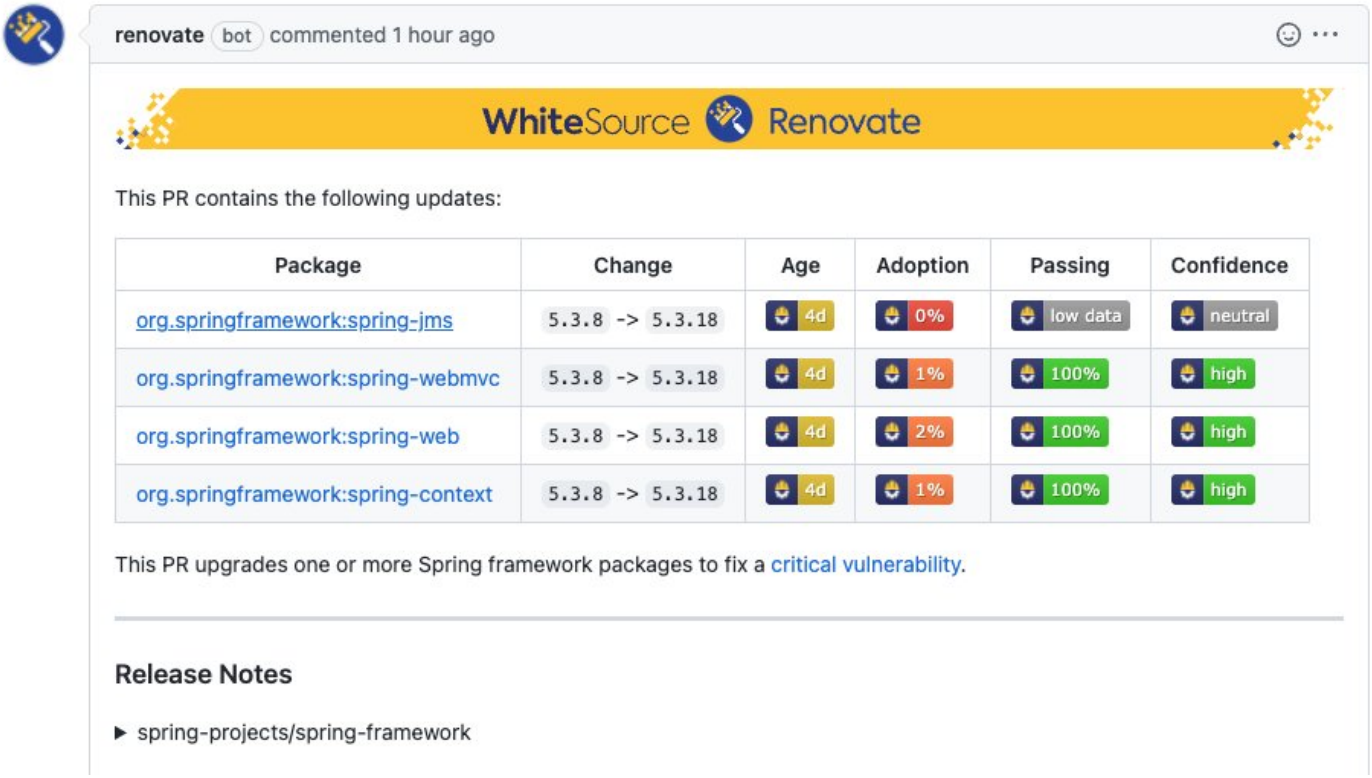


Figura 1.3: Pull request realizada por Renovate para actualizar diversas dependencias

En relación a la gestión de dependencias, también se contempla la automatización de tareas, a través de GitHub Actions, que verifiquen la seguridad de los módulos instalados.

Finalmente, se realiza una de las tareas más complejas y necesarias de todo el trabajo: refactorizar y configurar de nuevo la base de código y las herramientas que presenten cambios drásticos en su funcionamiento e integración en el proyecto.

1.3.2. Integración continua a través de GitHub Actions

Con el objetivo de mejorar la integración continua, se procede a configurar GitHub Actions para instalar y verificar las dependencias, ejecutar los tests unitarios y, finalmente, construir el proyecto. Se busca realizar configuraciones modulares con estos pasos, para posteriormente formar un flujo con ellos.

El principal motivo que justifica la modularización es la posibilidad de reutilizar dichas tareas para formar nuevos flujos en el futuro, además de que este procedimiento encapsula, abstrae y aísla el funcionamiento de cada proceso.

1.3.3. Virtualización con Docker y despliegue continuo

Es de gran interés simplificar el despliegue del proyecto SIMDE, por lo que se marca como uno de los principales objetivos la virtualización del entorno de producción. Esta actividad será realizada utilizando una herramienta de creación y gestión de *imágenes* y *containers* conocida como Docker, que nos permitirá generar una imagen del entorno para, seguidamente, formar contenedores virtuales.

Volverá a repasarse el trabajo realizado en Gitub Actions, puesto que se pretende desplegar, tanto en GitHub Pages como en el servidor del SIMDE en el IaaS, las nuevas versiones de producción una vez estén listas y aprobadas en la rama principal del repositorio.

1.3.4. Desarrollo de características y flujo de trabajo

Como objetivo final y síntesis de todo lo realizado, se procederá a desarrollar nuevas funcionalidades y características en la base de código del SIMDE, con el propósito de demostrar las cualidades y capacidades del nuevo flujo de desarrollo. La característica que se ha escogido implementar será una funcionalidad *Drag & Drop*, que facilite la interacción entre las tablas de instrucciones del simulador *VLIW*.

En adición a esto, se pretende facilitar la transparencia en la realización de cambios y especificar un procedimiento estandarizado para que el proceso de contribución resulte más sencillo, de manera que los desarrolladores puedan adaptarse con más facilidad al flujo de desarrollo del proyecto. A través de este objetivo también se comprueba el correcto funcionamiento del trabajo de adaptación y automatización.

Finalmente y, de ser necesario, se realizará una inspección de errores puntual para solventar cualquier problema que afecte significativamente al funcionamiento principal y básico de la plataforma.

1.4. Fases del desarrollo

Tomando ventaja de la situación inicial descrita previamente, se realizará un desarrollo lineal dirigido directamente hacia las necesidades de mayor importancia para la actualización y renovación del proyecto.

El desarrollo comprende las fases descritas en la Tabla 1.1.

En el siguiente capítulo, se describen las herramientas y tecnologías involucradas en el desarrollo del proyecto, algunas de ellas ya mencionadas a lo largo de esta introducción.

Tabla 1.1: Fases de desarrollo del proyecto

Fase del desarrollo	**Tarea**
Gestión de dependencias	Actualización de dependencias obsoletas
	Resolución de dependencias con módulos abandonados
	Auditoría de módulos y mitigación de vulnerabilidades
	Refactorización de configuración de herramientas y código con APIs incompatibles
Integración continua a través de GitHub Actions	Revisión de las tareas actualmente desplegadas
	Modularización de las tareas de integración
	Formación de flujos de integración y comprobación del funcionamiento
Virtualización con Docker y despliegue continuo	Construcción del entorno de producción en Docker
	Generación de una imagen para el despliegue del proyecto
	Configuración de claves secretas y variables de entorno
	Creación del flujo de trabajo para el despliegue continuo
Desarrollo de características y flujo de desarrollo	Especificación de flujo de desarrollo
	Implementación de características en la plataforma
	Revisión del proyecto y resolución de errores

En el siguiente capítulo, se describen las herramientas y tecnologías involucradas en el desarrollo del proyecto, algunas de ellas ya mencionadas a lo largo de esta introducción.

Capítulo 2

Herramientas y tecnologías

A lo largo de este capítulo se presentarán, de manera detallada, las herramientas y tecnologías que han resultado necesarias durante el desarrollo del trabajo, explicando la utilidad y funcionalidad que prestan, y el propósito que cumplen.

Se ha decidido separar las herramientas de las tecnologías a la hora de definirlas, puesto que las primeras son más relevantes en relación a los objetivos del proyecto. En el caso de las tecnologías para el desarrollo, si bien son utilizadas en una porción considerable de este trabajo, no poseen mayor importancia que demostrar los resultados de la configuración y contribuir a justificar el uso de las herramientas escogidas para el proyecto.

Cabe señalar que, a excepción de *Renovate*[9], las tecnologías y herramientas implicadas en el desarrollo se encontraban ya establecidas en el proyecto previo a la realización de este trabajo.

2.1. Herramientas

2.1.1. npm

npm[10] es el principal gestor de paquetería de Node.js. Su principal funcionalidad consiste en crear y gestionar paquetes versionados y licenciados para proyectos realizados en JavaScript, tanto para librerías como en el caso de software ejecutable. Esto lo realiza en un fichero llamado *package.json*, del que se muestra un ejemplo en la Figura 2.1.

Otra de sus principales características es la capacidad de instalar dependencias externas, o *módulos*, desde un registro público alojado en internet. Las dependencias pueden separarse en "dependencias generales", que

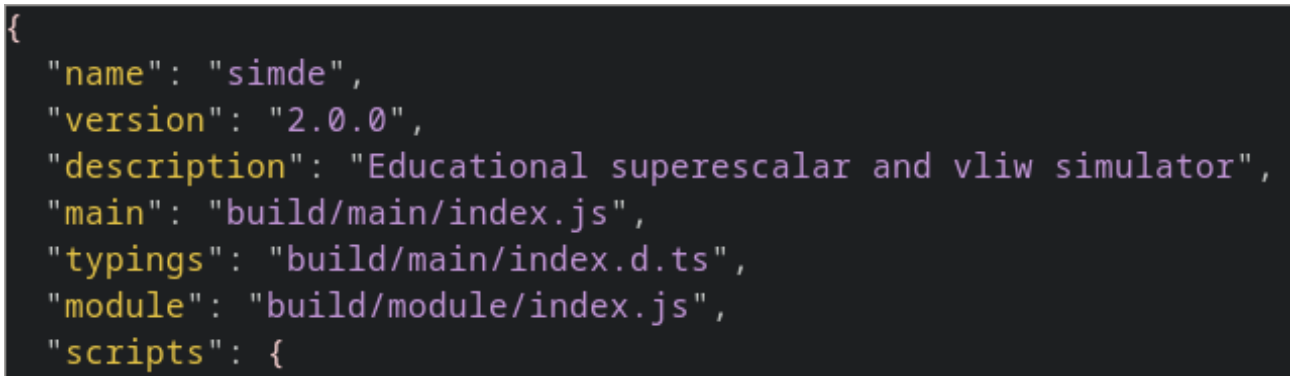


Figura 2.1: Comienzo del fichero *package.json* asociado al proyecto SIMDE

forman parte íntegra del producto, y "dependencias de desarrollo", las cuales suelen ser herramientas y utilidades exclusivamente necesarias para el entorno de desarrollo, como puede ser un *linter* o un generador de documentación.

Finalmente, otra de las utilidades principales de npm es la ejecución de *scripts*[11] configurables, lo cual resulta ideal para la automatización de tareas compuestas por múltiples pasos y para asociar las tareas generales del entorno con herramientas y parámetros específicos, potencialmente reemplazables y difíciles de recordar.

2.1.2. Webpack

Webpack[7] es una herramienta de empaquetado de módulos (Figura 2.2). En esencia, el propósito de Webpack consiste en juntar todas las fuentes y elementos multimedia de un proyecto con el fin de generar ficheros estáticos, sirviendo como resultado el producto final y útil del proyecto.

El funcionamiento de Webpack se define en uno o múltiples ficheros de código en JavaScript, generalmente nombrados de manera idéntica o similar a *"webpack.json"*. La configuración posee soporte para *plug-ins* y dispone, además, de un sistema de reglas para procesar de manera específica los diferentes ficheros de entrada posibles, dependiendo de las características y necesidades del proyecto.

2.1.3. GitHub Pages

GitHub Pages[12] es un servicio de publicación de sitios web estáticos asociado directamente con los repositorios de código alojados en GitHub.

Figura 2.2: Logo de Webpack

Una versión del SIMDE Web se encuentra actualmente alojada sobre este servicio[13].

2.1.4. Renovate

Renovate[9] es un servicio de automatización para la gestión de dependencias. Se encarga de revisar periódicamente las dependencias del proyecto, notificar en caso de actualización y, finalmente, preparar una solicitud de incorporación con los cambios pertinentes, lista para integrar al repositorio.

Es altamente configurable, permitiendo minimizar el ruido causado por las notificaciones, ajustando el periodo de ejecución a un horario definible. También permite acumular y agrupar, en una sola solicitud, múltiples actualizaciones de dependencias.

Un ejemplo de solicitud de actualización de dependencias de *Renovate* se muestra en la Figura 1.3, ubicada en el capítulo 1.

2.1.5. Docker

Docker[14] es una herramienta de creación y gestión de contenedores virtuales. Los contenedores encapsulan el entorno de ejecución del proyecto del resto del sistema en el que se hospeda, facilitando su funcionamiento y maximizando la compatibilidad, independientemente del sistema anfitrión (Figura 2.3).

La principal ventaja de usar *Docker* es la facilidad que provee la utilización de contenedores para reducir la barrera entre el desarrollo del software y el despliegue en entornos de pruebas y en producción.

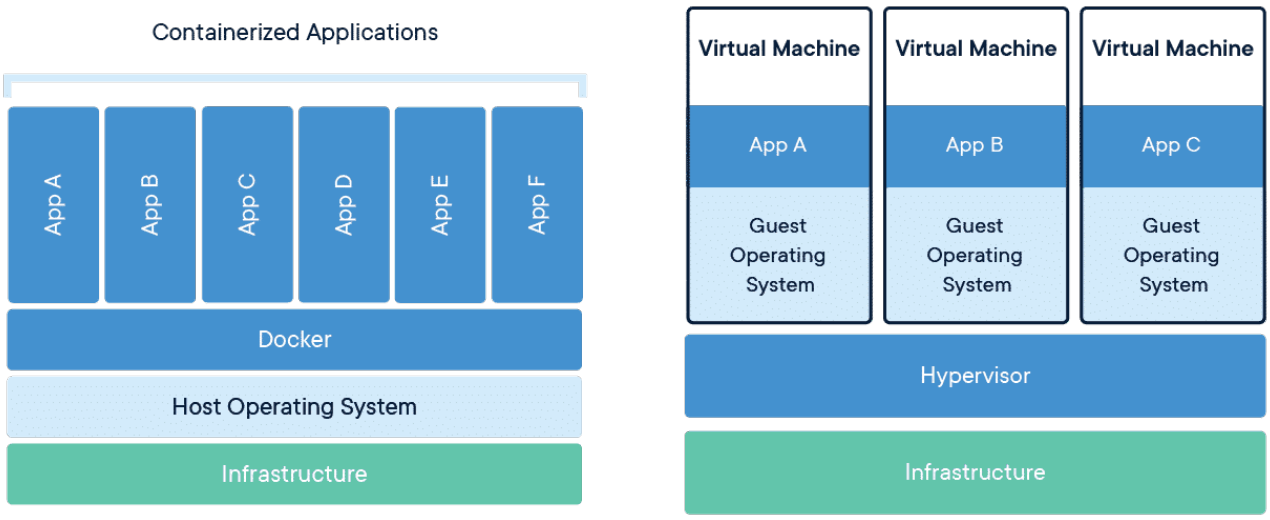


Figura 2.3: Arquitectura de Docker, por Docker, Inc.

Para generar contenedores capaces de desplegar la aplicación, primero se debe generar una *imagen* a partir de un fichero *Dockerfile*, el cual debe contener la configuración necesaria de modo que el software a desplegar pueda ejecutarse correctamente.

Respecto a la utilización de contenedores de *Docker* en un entorno de desarrollo, esto es, para realizar el desarrollo de software dentro de un contenedor, abstraeremos la utilización de *Docker* a través de herramienta que se introduce a continuación.

2.1.6. DevContainers

Los *DevContainers*[15] son una utilidad que asiste en la utilización de contenedores de Docker para el desarrollo de software (Figura 2.4). A lo largo de este proyecto, se ha utilizado la extensión del editor de texto *Visual Studio Code* conocida como *“Remote - Containers"*.

Dicha extensión facilita la creación de un entorno virtualizado de desarrollo, partiendo de la gestión de contenedores de Docker y de un fichero autogenerado y configurable, nombrado *"devcontainer.json"*. Además, está orientada a trabajar de manera nativa con el editor, de manera que diferentes herramientas, como pueden ser la terminal del contenedor y el acceso

al depurador y al sistema de ficheros resultan mucho más accesibles que si se utilizase Docker de manera directa.

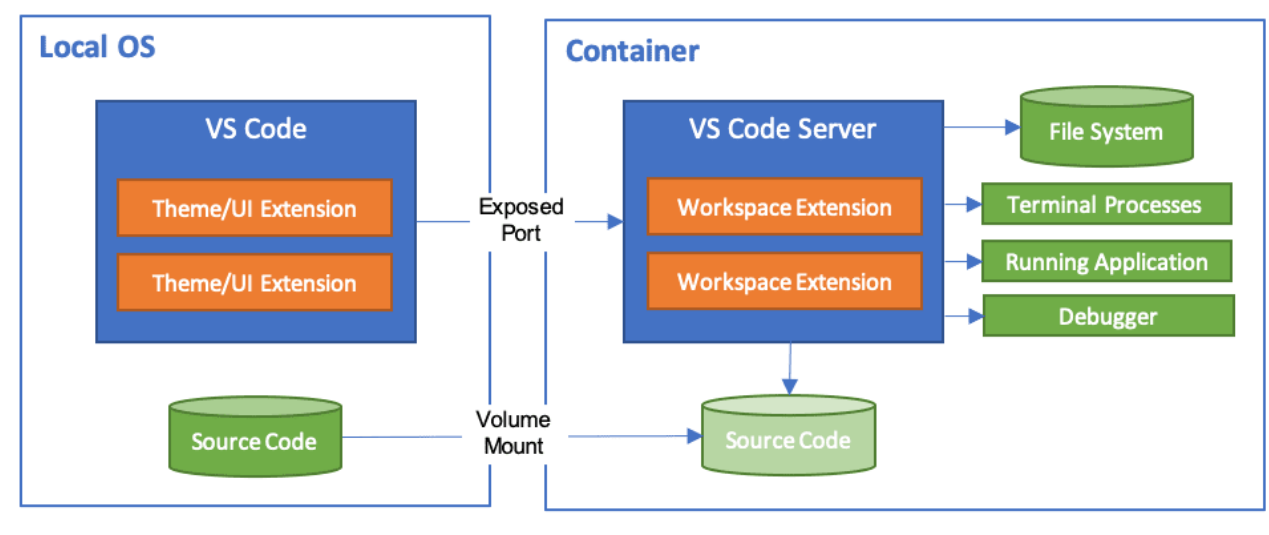


Figura 2.4: Diagrama de comunicación de un sistema DevContainer, por Microsoft

2.2. Tecnologías para el desarrollo

2.2.1. TypeScript

TypeScript[16] es un lenguaje de programación de tipado fuerte, diseñado para ser un superconjunto de JavaScript y transpilar a este.

Es recomendable utilizarlo en proyectos medianos y grandes, puesto que provee de mejores mecanismos de seguridad e integridad de datos a través de su sistema de tipos y de las mejoras respecto al paradigma de *Programación Orientada a Objetos* que implementa respecto a JavaScript.

También ofrece algunas abtracciones de alto nivel muy útiles, como pueden ser los *genéricos*, las *interfaces* y la *unión de tipos*.

2.2.2. React

React[17] es una librería de *front-end* escrita en JavaScript para la creación de interfaces de usuario interactivas a partir de bloques de construcción denominados *componentes*.

Algunos de los principales atractivos de React como librería de *User Interfaces* es la sencillez en el diseño de su API y su funcionamiento, y su

velocidad a la hora de manejar elementos del *Document Object Model*, o *"DOM"*.

Esto se debe a que React mantiene un *DOM* interno y oculto, al cual se le conoce como *"Shadow DOM"*, y que modifica hasta que algún cambio entre el *DOM* real y el *Shadow DOM* es lo suficientemente significativo como para aplicar los cambios realizados en este último sobre el *DOM* real.

2.2.3. Redux

Redux[18] es una librería de JavaScript que implementa un modelo de gestión de datos a través de la centralización y gestión del estado de una aplicación (Figura 2.5).

Redux es comúnmente utilizado junto con React a través de *React-Redux* con el propósito de proveer de estado y capacidades para manejar flujos de datos a los componentes de la popular librería de interfaces de usuario.

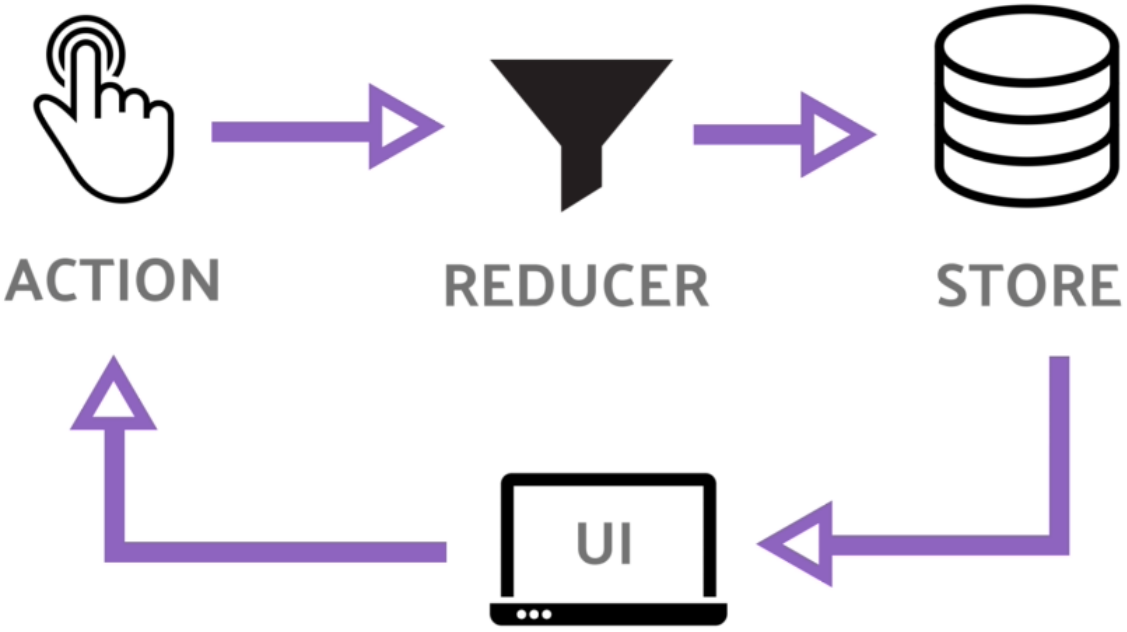


Figura 2.5: Diagrama de actividades de Redux [19]

Una vez introducidos los diferentes elementos técnicos que forman parte del desarrollo de este proyecto, se procede a explicar, en el siguiente capítulo, las bases detrás de las decisiones de diseño respecto a la integración y el mantenimiento del software.

Capítulo 3

Integración y mantenimiento de software

En este capítulo serán introducidos y discutidos los principales tópicos relevantes del proyecto sobre integración y mantenimiento de software. Las metodologías más comunes son presentadas, desarrolladas y, si resulta conveniente, comparadas, tanto para casos generales como parcialmente sobre las características específicas del proyecto *SIMDE Web*.

Son introducidos, pues, los métodos a valorar y aplicar en este proyecto, tanto en los flujos de desarrollo y de automatización de tareas como con respecto a las buenas prácticas que sostienen la calidad del desarrollo y del proyecto en general.

3.1. Entorno de desarrollo

Se presentan en esta sección las cuestiones clave que se deben tener en cuenta durante las etapas de desarrollo del software.

3.1.1. Desarrollo en contenedores

Es recomendable utilizar *DevContainers* para desarrollar el proyecto, puesto que evita instalar dependencias directamente en el equipo de desarrollo, gestionando los recursos de manera interna.

Si en adición a esto, considerasemos que *DevContainers* se encuentra configurado de manera apropiada, también aceleraría la puesta en marcha del desarrollo en sistemas nuevos y la transición entre los entornos de desarrollo y producción.

3.1.2. Dependencias de desarrollo

Resulta importante añadir ciertas dependencias de desarrollo, o *development dependencies*[20], para asistir a los desarrolladores a la hora de crear código. Entran en consideración los diferentes *linters* para la estructura, estilo y funcionalidad de la aplicación, herramientas de generación y referencia a la documentación y utilidades varias que aceleren la etapa de desarrollo de software.

3.1.3. Servidor de desarrollo

Una de las principales y más útiles funcionalidades de *Webpack* es el despliegue en desarrollo de un pequeño servidor web local capaz de presentar y actualizar, en tiempo real y de manera reactiva, los cambios realizados en el código, un concepto conocido como *hot code reloading*.

Es, con diferencia, la mejor manera existente de comprobar los cambios e interactuar con el proyecto para probar las implementaciones de funcionalidad en el momento.

3.2. Flujo de trabajo

El flujo de trabajo consiste en los pasos comprendidos entre la preparación del desarrollo de nueva funcionalidad en la base de código y la integración de dicha funcionalidad con el resto del proyecto. Por lo general, el flujo de trabajo está fuertemente asociado a los sistemas de control de versiones[21], como *Git*, *Mercurial* y *Subversion*.

Como es lógico, se han investigado flujos de trabajos relacionados con *Git*, puesto que es el sistema de control de versiones utilizado actualmente por el proyecto del simulador.

3.2.1. Git flow

Git flow[22] es uno de los flujos de trabajos más utilizados a la hora de organizar y emitir contribuciones de código a repositorios de proyectos de software.

Consiste, principalmente, en separar la naturaleza de nuevas implementaciones de manera etiquetada, generando una estructura de numerosas

ramas que divergen y convergen entre sí hacia una rama principal, comúnmente denominada *main*, o *master*.

Mientras que este sistema parece resultar bastante ordenado y limpio, la integración y fusión de diferentes ramas entre sí resulta una complicación significativa si estas son actualizadas de manera independiente y asíncrona, lo cual es bastante común. Esto puede apreciarse en la Figura 3.1.

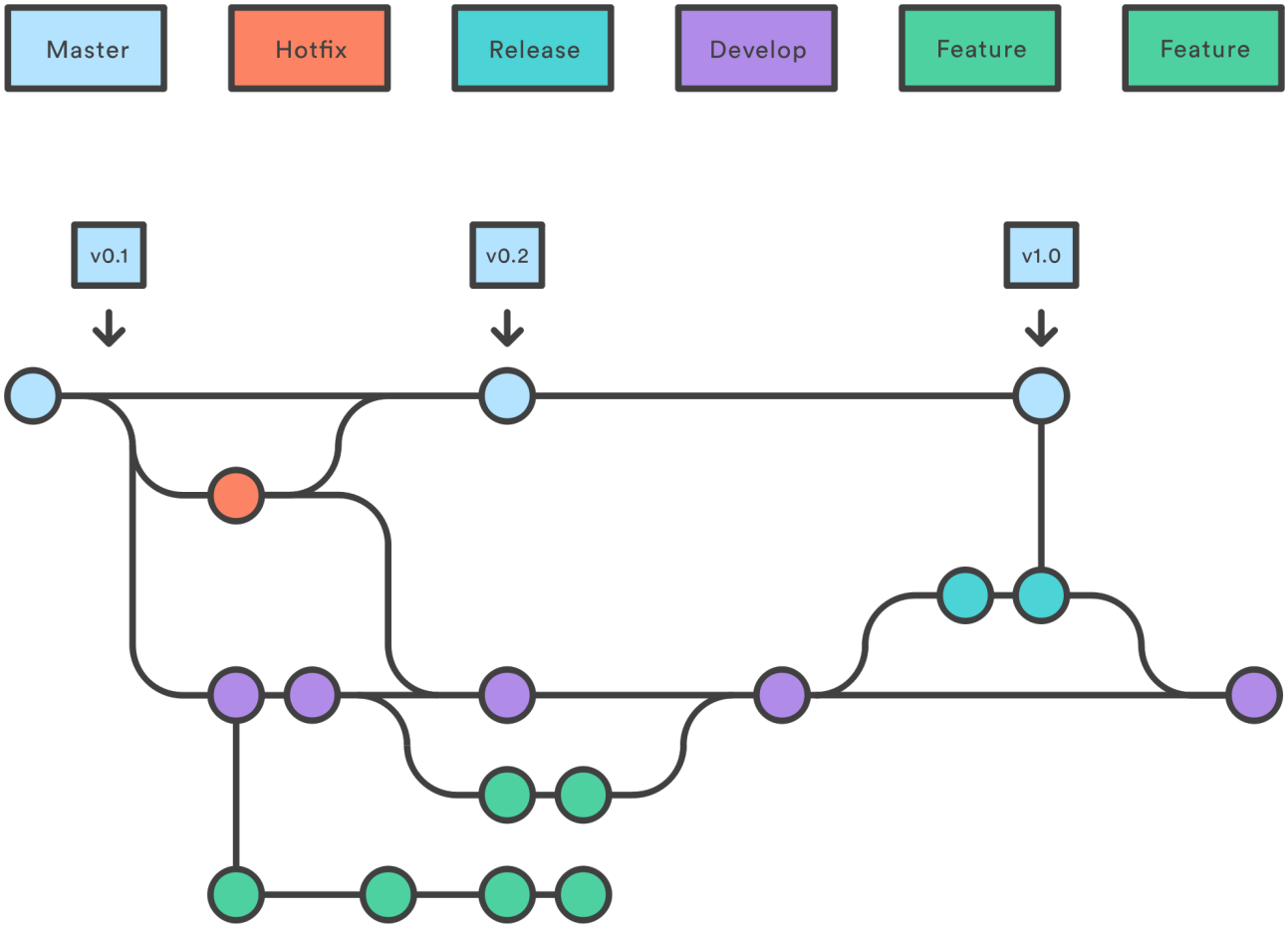


Figura 3.1: Ejemplo de diagrama de ramas de *Git flow* [23]

3.2.2. Trunk-based development

Trunk-based development[24] es una alternativa en escala y moderna a *git flow*. Consiste en realizar contribuciones al proyecto de manera directa hacia una rama principal, o a través de ramas divergentes únicamente de la principal y de corto periodo de vida.

Las ramas de publicación del proyecto de software se generan exclusivamente a partir de la rama principal, y son de sólo lectura.

3.3. Pruebas de software y test-driven development

3.3.1. Pruebas unitarias

Las *pruebas unitarias*[25] consisten en verificar el correcto funcionamiento del código ejecutando bloques básicos del código.

Por lo general, cuando se desarrolla una nueva estructura en el código, esta debe venir acompañada de pruebas que determinan el funcionamiento inicial de la nueva funcionalidad. Si algún cambio a futuro alterase de manera inadvertida el comportamiento anterior, las pruebas unitarias fallarían, advirtiendo que el funcionamiento ha cambiado de manera inesperada.

3.3.2. Pruebas end-to-end

Las pruebas *end-to-end*[26], comúnmente abreviadas con el acrónimo *E2E*, son realizadas en entornos de pruebas, y consisten en simular la interacción de un usuario real contra la interfaz de usuario de la plataforma, comprobando que dichas interacciones resultan íntegras y determinadas a lo largo de todo el flujo de actividad que esta provoca.

Por lo general, al tipo de herramienta más utilizado para realizar pruebas *end-to-end* se les conoce como *headless browser*, o navegador sin interfaz gráfica. El más popular para este propósito se le conoce como *Playwright*[27], y es desarrollado principalmente por *Microsoft*.

3.3.3. Pruebas de integración

Las *pruebas de integración*[28] comprueban el funcionamiento conjunto de los elementos que componen el software, teniendo en cuenta dependencias de sistemas externos, como bases de datos u otros servicios.

Estas pruebas son similares a las pruebas unitarias, puesto que son realizables con los mismos *frameworks* de pruebas que en el caso de estas, pero se debe tener en cuenta que sólo se realizan en entornos de pruebas o de pre-producción, con el sistema completo desplegado.

3.3.4. Pruebas de regresión

Por último, las *pruebas de regresión*[29] son un tipo de prueba general que consiste en replicar situaciones que han resultado problemáticas en el

pasado para evitar que sucedan regresiones. Este tipo de pruebas suele ser bastante efectivo a la hora de lidiar con casos esquina.

3.4. Automatización de tareas para el mantenimiento

Con respecto a la automatización de tareas de mantenimiento, el proyecto se centra en la utilización de *Renovate*, sustituyendo además, en nuestro caso, al principal servicio de gestión de dependencias por excelencia, *dependabot*[30]. Por tanto, se procede a comparar ambas herramientas:

3.4.1. Renovate

Renovate fue introducido previamente en el capítulo 2. Su principal ventaja respecto a *dependabot* recae en un control mejorado de las solicitudes de incorporación de las dependencias, pudiendo agruparlas y ajustarlas a un horario definido de manera que no causen ruido en exceso o consuman recursos de la línea de montaje a la hora de aplicar las actualizaciones. Un ejemplo de configuración de *Renovate*, programado para emitir las actualizaciones el lunes a las seis de la mañana, se muestra en la Figura 3.2

3.4.2. Dependabot

Dependabot parte de un sistema más simple y su uso ha sido más popularizado que el de *Renovate*. Esto le añade una ventaja considerable respecto a la integración con los repositorios, dado que GitHub posee, de manera nativa, soporte para añadir *dependabot* a los repositorios con prácticamente un solo click.

3.5. Containerización e integración contínua

La línea de trabajo respecto a la integración contínua depende de una buena implementación de la imagen que creará *containers* para desplegar en producción. Es por ello que se debe generar y configurar debidamente un fichero *Dockerfile* que implemente y minimice las necesidades del proyecto y genere un sistema apropiado como imagen de Docker.

Figura 3.2: Configuración de ejemplo de *Renovate*, habilitada en el proyecto SIMDE Web

Teniendo esto, la integración contínua consiste, simplemente, en comprobar que la generación o *build* de una nueva imagen de Docker a partir de los cambios resulte satisfactoria, y que la suite de pruebas unitarias no falle en ninguna expectativa de funcionalidad.

3.6. Despliegue contínuo

Por último, resultaría ideal extender la línea de montaje, de modo que el sistema despliegue en un entorno de pruebas, y se realicen las correspondientes pruebas de integración y *end-to-end*, similar a la presentada en la Figura 3.3.

Se parte del resultado de la integración contínua y, en el caso de pasar

el resto de pruebas de manera exitosa, el sistema es desplegado al sitio web estático alojado en la plataforma *GitHub Pages*.

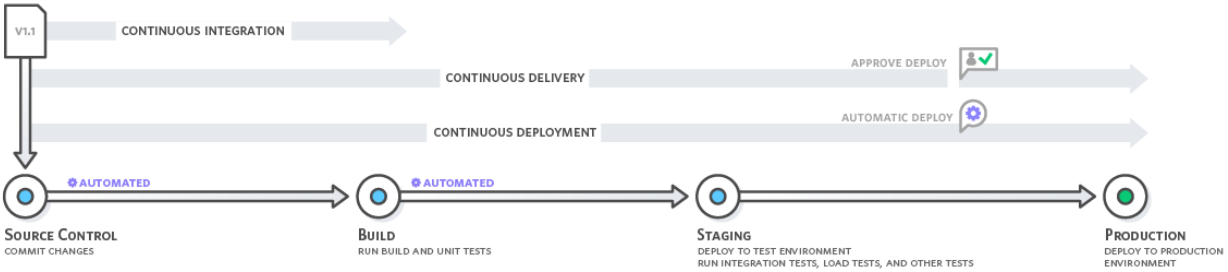


Figura 3.3: Líneas de montaje de integración, entrega y despliegue contínuos

Al finalizar este capítulo, se concluye la presentación de conceptos necesarios para comprender el contexto de este proyecto y su intención. Por tanto, está todo preparado para que, comenzando por el siguiente capítulo, se proceda a desarrollar el trabajo práctico realizado sobre el simulador.

Capítulo 4

Desarrollo del proyecto

Se expone, a lo largo de este capítulo, el proceso de realización de la parte práctica del trabajo, donde se aplican los apartados teóricos, metodologías, tecnologías y herramientas que se han introducido desde el comienzo de la memoria hasta este punto.

Mientras que mayoritariamente se introducen las fases del proyecto cronológicamente según su realización, existen ciertas excepciones que, por conveniencia, han sido añadidas en un punto diferente del desarrollo. Esto sólo ha sido aplicado en los casos que, tras considerar si dicha alteración es independiente del trabajo que la precede, exista otro grupo de fases que compartan el mismo tópico o realicen un propósito similar. Por ejemplo, mientras que la gestión de dependencias se realiza al comienzo del trabajo, no es hasta las últimas fases del proyecto que se considera la aplicación de *Renovate* como servicio en el repositorio.

4.1. Estado inicial y gestión de dependencias

El primer paso a la hora de entrar en contacto con el proyecto del simulador *SIMDE Web* ha sido la clonación del repositorio[1] en un entorno de desarrollo local, configurado directamente sobre un sistema de usuario. Con el objetivo de lanzar el proyecto en un entorno de desarrollo, se procede a intentar instalar las dependencias. Esta intención fracasa, tal y como se puede ver en la Figura 4.1.

4.1.1. Actualización de dependencias obsoletas

Se comienza eliminando la dependencia que impide la instalación del proyecto, un *plug-in* de *Sass*[31] para utilizar con *Webpack*.

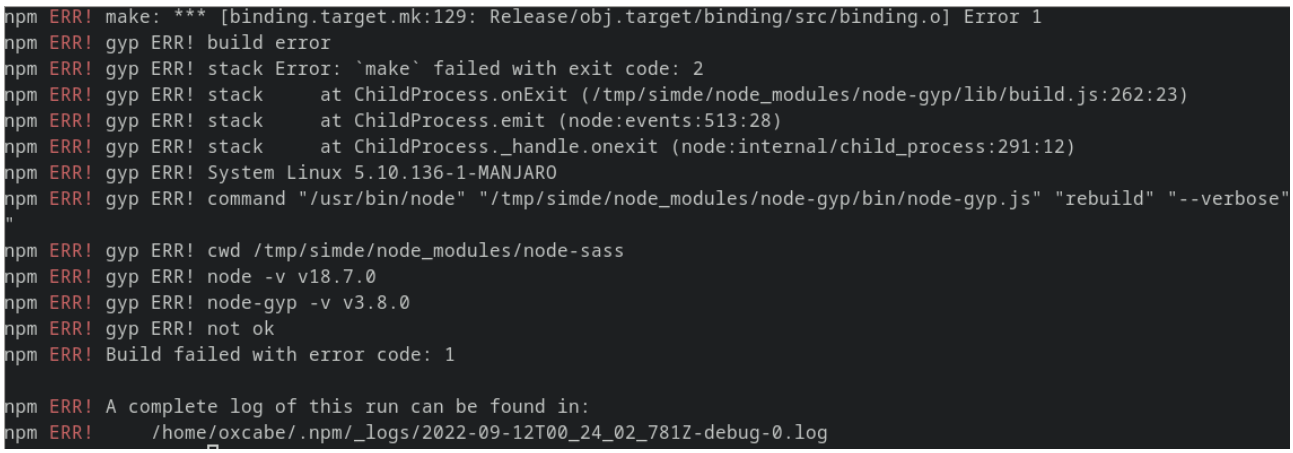


Figura 4.1: Error de instalación causado por la dependencia *node-sass*

Una vez resulta posible realizar la instalación, se procede con la actualización manual de las dependencias, asignando mayor prioridad a las que se consideren esenciales, como *React*, *TypeScript* y *Webpack*. En algunos casos puntuales, ha sido necesario reemplazar dependencias abandonadas por otras de similar funcionalidad y, en un caso en particular, implementar de nuevo el funcionamiento de la dependencia dentro de la base de código del simulador, a falta de mejores alternativas.

Cabe destacar, finalmente, que algunos módulos obsoletos pero compatibles han tenido que ser retirados y reemplazados debido a las vulnerabilidades que estos presentaron tras realizar una auditoría de seguridad en las dependencias, cuyos resultados pueden visualizarse en la Figura 4.2.

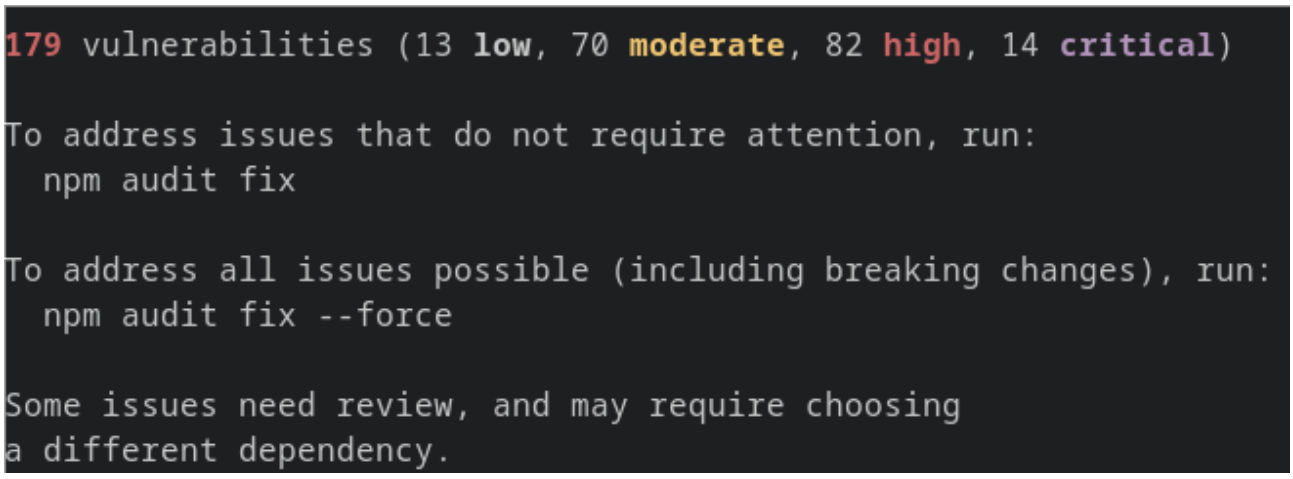


Figura 4.2: Resultado de la auditoría sobre dependencias vulnerables

4.1.2. Refactorización del código afectado por la actualización de dependencias

Una vez las dependencias fueron actualizadas, era necesario adaptar los posibles cambios en las funciones y abstracciones que estas otorgaban a la base de código.

La refactorización del código comenzó con la configuración de *Webpack*, puesto que era necesario conseguir que funcionase para realizar compilaciones y comprobar la funcionalidad de la aplicación.

El caso más complicado de gestionar ha sido, con diferencia, la resolución de las múltiples dependencias relacionadas con *i18next*[32], puesto que las funciones de internacionalización, por naturaleza, se encuentran repartida a lo largo de toda la base de código.

4.1.3. Sustitución del servicio de gestión de dependencias

Se realiza el reemplazamiento de *dependabot* por *Renovate*. Esto conlleva desactivar *dependabot* en el menú de *grafo de dependencias* en las opciones del repositorio en *GitHub*, cuyo menú se muestra en la Figura 4.3. A continuación se conecta el servicio de aplicación de *Renovate* contra el repositorio del proyecto, se aplica la configuración y comienza el funcionamiento del servicio.

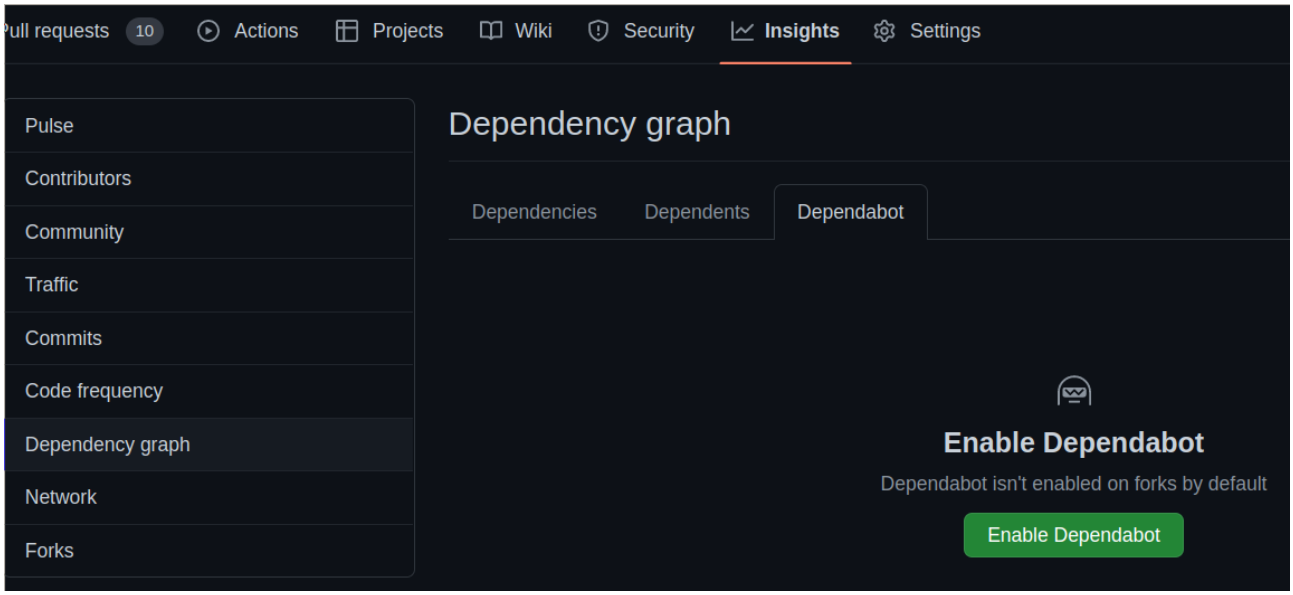


Figura 4.3: Interfaz de gestión de *dependabot* en *GitHub*

4.1.4. Configuración del entorno de desarrollo

Se configura el entorno de desarrollo combinando las facilidades hacia el desarrollo virtualizado proporcionadas por *DevContainers* junto con el modo de depuración configurable del editor de código *Visual Studio Code*. Este editor permite integrarse para funcionar con el navegador, de manera que habilite la posibilidad de establecer puntos de ruptura y examinar tanto las variables como el estado durante la ejecución. El entorno de desarrollo utilizado es mostrado en la Figura 4.4.

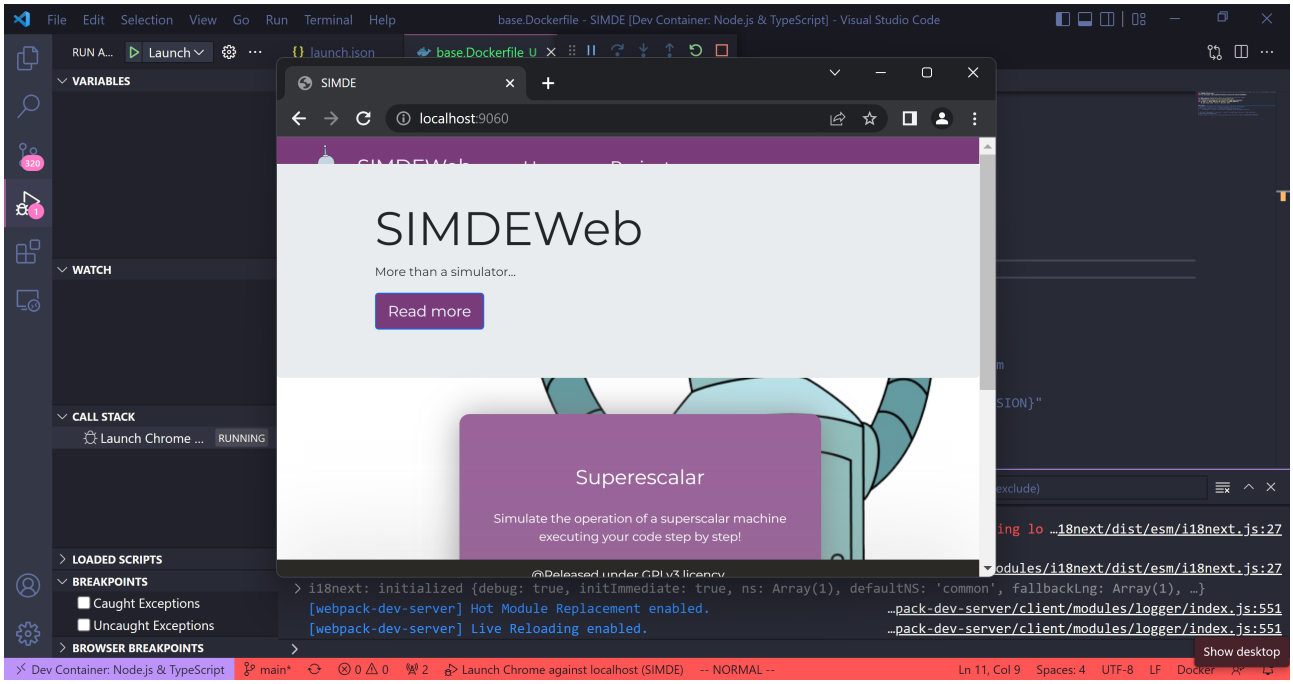


Figura 4.4: Depuración en el entorno de desarrollo virtualizado con *DevContainers*

4.2. Revisión de las tareas de integración continua

La configuración para las tareas de integración contínua establecidas en *GitHub Actions* son examinadas para verificar que:

- La imagen de Docker para el sistema en producción es construida.
- Las dependencias del proyecto son instaladas correctamente.
- Las pruebas unitarias son ejecutadas.

Dado que la configuración de las tareas se encontraba configurada apropiadamente, es a continuación y, a través de la implementación de la funcionalidad de *Drag & Drop*, que la línea de montaje puede ser ejecutada y validada para su posterior uso.

4.3. Implementación de funcionalidad Drag & Drop

La funcionalidad de *Drag & Drop* fue implementada entre las tablas de instrucciones del simulador de arquitectura *VLIW*. Dicha funcionalidad fue facilitada por la librería *react-dnd*[33], la cual permitió diseñar funcionalidad reactiva al arrastrar el índice de la instrucción hacia la tabla de instrucciones *VLIW*. Se presenta dicha funcionalidad en la Figura 4.5.

Librerías similares que fueron barajadas, como *react-beautiful-dnd*[34], poseían limitaciones que complicaban la implementación en nuestro caso de uso específico, dada la naturaleza de alto nivel orientada a listas a la que aspiraban dichas alternativas.

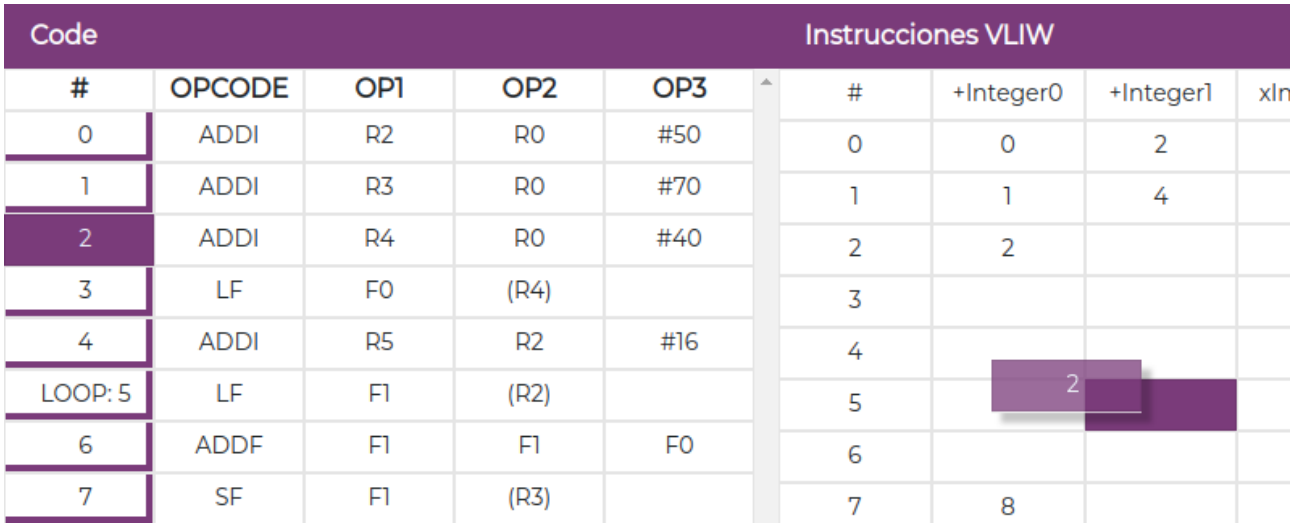


Figura 4.5: Funcionalidad *Drag & Drop* en el simulador VLIW

4.4. Solución de errores en el estado de la máquina superescalar

4.4.1. Estado de la máquina superescalar

Durante el desarrollo del proyecto, fueron encontrados varios errores de estado de máquina que se iban propagando según la ejecución progresaba. Estos errores, una vez corregidos, sirven para formar pruebas de regresión y, en el caso de que el alcance del trabajo acogiese una línea de desarrollo dirigida con más énfasis hacia las pruebas, hubiesen sido añadidas a un supuesto sistema para las regresiones y posteriormente comprobadas.

4.4.2. Problemas en la recarga de página y enrutamiento

Otra situación que causaba incomodidades durante el desarrollo era la pobre gestión de la ruta del navegador, la cual no reconocía directamente las máquinas, ni permitía actualizar la página, proceso que se repite a menudo durante el desarrollo y corrección de características del software, como se puede ver en la Figura 4.6.

La solución tomada fue indicarle a *Webpack*, a través de la configuración, la necesidad de retroceder a la ruta funcional a la hora de gestionar esos casos, considerados como errores por la herramienta.

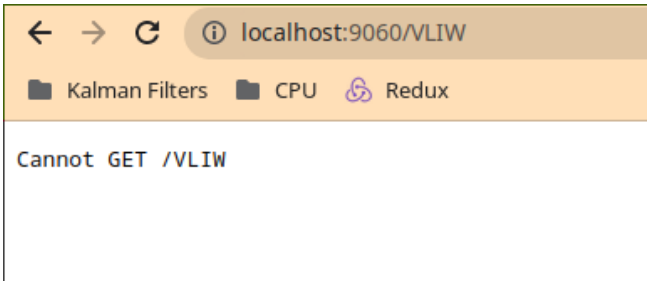
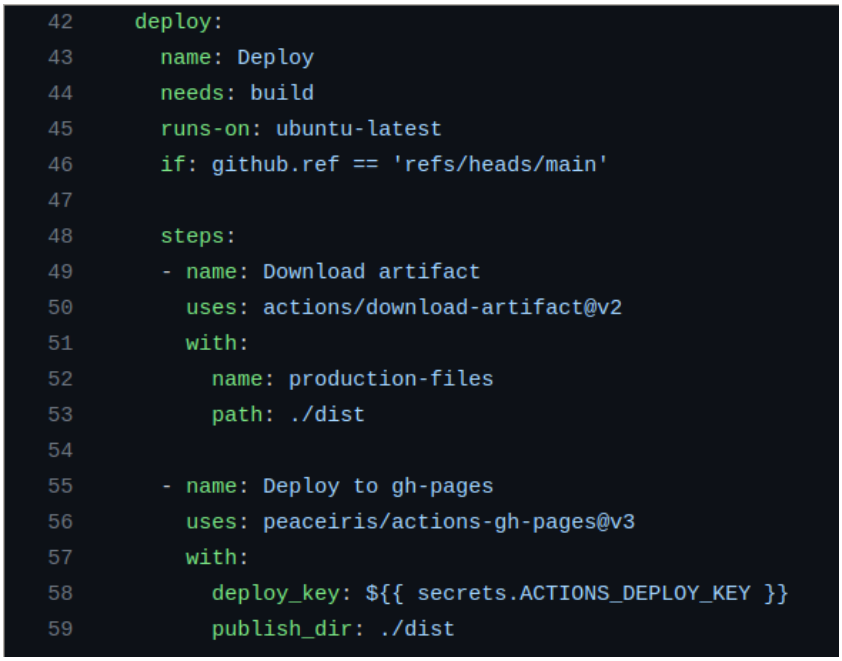


Figura 4.6: Error al recargar o acceder directamente al simulador *VLIW*

4.5. Finalización del desarrollo

4.5.1. Configuración del entorno de producción en GitHub Actions

Se finaliza el desarrollo práctico del trabajo verificando la correcta configuración y funcionamiento del despliegue en producción del sistema, el cual también es operado como una tarea de la línea de montaje que *GitHub Actions* maneja. Dicha configuración puede apreciarse en la Figura 4.7.



```
42    deploy:
43      name: Deploy
44      needs: build
45      runs-on: ubuntu-latest
46      if: github.ref == 'refs/heads/main'
47
48      steps:
49      - name: Download artifact
50        uses: actions/download-artifact@v2
51        with:
52          name: production-files
53          path: ./dist
54
55      - name: Deploy to gh-pages
56        uses: peaceiris/actions-gh-pages@v3
57        with:
58          deploy_key: ${{ secrets.ACTIONS_DEPLOY_KEY }}
59          publish_dir: ./dist
```

Figura 4.7: Configuración del despliegue

Capítulo 5

Conclusiones y líneas futuras

5.1. Conclusiones

A través de la investigación y posterior desarrollo del trabajo se ha logrado actualizar el estado del desarrollo del proyecto SIMDE. Una vez habilitado el proyecto, también se ha logrado establecer un entorno de buenas prácticas y de servicios de mantenimiento, soporte e integración, con mayor facilidad gracias a la automatización de procesos. Se ha maximizado la longevidad del proyecto y simplificada la capacidad de contemplación de futuras líneas de desarrollo.

Si bien este trabajo sentará las mecánicas de desarrollo del SIMDE para los próximos años, es inevitable que, dado el ritmo al que evolucionan las tecnologías Web y el desarrollo moderno de software, vuelva a presentarse la necesidad de realizar un nuevo enfoque que sustituya e incluso cuestione las líneas de pensamiento que conforman este trabajo. Si bien, el hecho de que este proceso ocurra manera repentina o gradualmente, a lo largo del tiempo y con los futuros avances seguirá dependiendo de un factor humano.

5.2. Líneas de trabajo futuras

5.2.1. Mejoras en la visualización de información y entrada de datos de usuario

Es de gran interés avanzar el desarrollo y la investigación sobre las formas que puede tener un usuario de interactuar con la plataforma:

- Adición de gráficos y elementos visuales informativos, como mapas de calor del recorrido de las instrucciones y de la memoria, que ayudarían

a entender mejor la importancia de los diferentes tipos de localidad de referencia.

- Mejorar los campos de entrada de datos del usuario, principalmente en lo que se refiere a la carga y modificación de código pseudo-ensamblador y formato textual de la tabla *Very Long Instruction Word*.

5.2.2. Desarrollo e integración de la plataforma de ludificación

Una clara línea de trabajo a futuro trata sobre adaptar e integrar el trabajo de Antonio López, previamente mencionado en la introducción: *Plataforma de ludificación de un simulador didáctico de arquitectura de computadores* [4].

La iteración más simple de este proyecto consistiría en adaptar el trabajo mencionado en la memoria sobre la línea principal del desarrollo del proyecto SIMDE. Sin embargo, una opción más completa y avanzada consistiría en divergir hacia una implementación propia del back-end, puesto que nos permitiría tener más control e integridad sobre la legitimidad de los resultados en las actividades asociadas a la ludificación.

Si existe la intención de proceder con el caso más complejo, sería necesario realizar un proyecto de API REST que permita validar, a través de una máquina idéntica al simulador utilizado, el código y propiedades de la máquina relacionadas con la participación del usuario.

5.2.3. Simulación de una memoria caché

Este punto trataría sobre la posibilidad de agregar una memoria caché en el modelo general de gestión de la memoria. Sería entonces de gran utilidad establecer la posibilidad de configurar la memoria caché con diferentes algoritmos de caché, como el *“First In, First Out”*, conocido también como *FIFO* o el popular *“Least Recently Used”*, abreviado como *LRU*.

5.2.4. Modularización de las máquinas y adaptación a WebAssembly

Modularizar las máquinas conllevaría un trabajo considerable y algo desafiante, puesto que consistiría en separar de manera más significativa

la implementación de la lógica o “core” del simulador respecto a su interfaz gráfica, o “vista”.

Esta adaptación conlleva un aumento considerable en la complejidad del desarrollo, puesto que se adaptarían partes del proyecto a un lenguaje como C++ o Rust, pero resultaría en un aumento en el rendimiento, seguridad y calidad de la arquitectura subyacente.

5.2.5. Implementación de máquinas con arquitecturas monociclo y vectorial

Como punto final se podría considerar, tras un vistazo a la taxonomía de Flynn [35], varias implementaciones para otros tipos de máquinas, con diferentes arquitecturas de computadores:

- Máquina con arquitectura monociclo. Una máquina sencilla, limitada a un funcionamiento secuencial y un rendimiento de 1 Ciclo por Instrucción. Utilizaría el mismo repertorio de instrucciones que el resto de máquinas existentes para la fecha de este trabajo.
- Máquina vectorial (SIMD). Implementación de una arquitectura de propósito específico, especializada en el procesamiento aritmético de estructuras de datos secuenciales. Esta máquina contemplaría un nuevo conjunto de instrucciones dada su peculiar naturaleza.

Ambas implementaciones abrirían al SIMDE a nuevas aplicaciones didácticas y fundamentales, como el aprendizaje del desarrollo de programas en ensamblador de arquitecturas RISC [36] y la realización y adaptación de programas para explotar el paralelismo y eficiencia de una arquitectura vectorial adquiriendo, en el proceso, conocimientos sobre las bases de diseño y el funcionamiento interno de la respectiva máquina.

Capítulo 6

Summary and Conclusions

6.1. Conclusions

Through the research and subsequent development of the work, it has been possible to update the development status of the SIMDE project. Once the project has been enabled, it has also been possible to establish an environment of best practices and maintenance, support and integration services, with greater ease thanks to the automation of processes. The longevity of the project has been maximized and the ability to contemplate future lines of development has been simplified.

While this work will set the mechanics of SIMDE development for years to come, it is inevitable that, given the pace at which Web technologies and modern software development are evolving, the need for a new approach to replace and even challenge the lines of thinking that make up this work will recur. Whether this process occurs suddenly or gradually, over time and with future advances, it will continue to depend on a human factor.

6.2. Future lines of work

6.2.1. Improvements in information display and user data input

It is of great interest to advance the development and research on how the user can interact with the platform:

- Addition of graphics and informative visuals, such as heat maps of the instruction and memory path, that would help to better understand the importance of different types of locality of reference.
- Improve user input fields, mainly regarding the loading and modification of pseudo-assembler code and formatting of the VLIW table.

6.2.2. Gamification platform development and integration

A clear line of future work deals with adapting and integrating the work of Antonio López, previously mentioned in the introduction: *Ludification platform for a didactic simulator of computer architecture* [4].

The simplest iteration of this project would consist in adapting the work mentioned in the report on the main line of development of the SIMDE project. However, a more complete and advanced option would be to diverge towards a back-end implementation, since it would allow us to have more control and integrity over the legitimacy of the results in the activities associated with gamification.

If there is the intention to proceed with the more complex case, it would be necessary to develop a REST API project that allows to validate, through a machine identical to the simulator used, the code and properties of the machine related to each user's participation.

6.2.3. Cache memory simulation

This point revolves around the possibility of adding a cache memory in the general memory management model. It would then be very useful to establish the capability of configuring the cache memory with different caching algorithms, such as *"First In, First Out"*, also known as *FIFO* or the popular *"Least Recently Used"*, abbreviated as *LRU*.

6.2.4. Modularization of machines and adaptation to WebAssembly

Modularizing the machines would entail considerable and somewhat challenging work, since it would consist of separating more significantly the implementation of the logic or "core" of the simulator from its graphical interface, or "view".

This adaptation would result a considerable increase in development complexity, since parts of the project is to be converted to a language such as C++ or Rust, but would result in an increase in performance, security and quality of the underlying architecture.

6.2.5. Implementation of machines with single-cycle and vector architectures.

As a final point one could consider, after a look on Flynn's taxonomy [35], some different implementations for other types of machines, with different computer architectures:

- Machine with monocycle architecture. A simple machine, limited to sequential operation and a performance of 1 Cycle per Instruction. It would use the same instruction set as the rest of the existing machines by the date of this work.
- Vector Machine (SIMD). Implementation of a purpose-specific architecture, specialized in arithmetic processing of sequential data structures. This machine would contemplate a new set of instructions given its peculiar nature.

Both implementations would open SIMDE to new didactic and fundamental applications, such as learning the development of assembly programs for RISC architectures [36] and the development and adaptation of programs to exploit the parallelism and efficiency of a vector architecture while aiding, in the process, to acquire knowledge about the design bases and the inner workings of the respective machine.

Capítulo 7

Presupuesto

7.1. Presupuesto general

Se presenta en la Tabla 7.1 el presupuesto estimado para la realización del proyecto asociado a este Trabajo de Fin de Grado.

Descripción	Coste por hora	Coste total
168 horas de trabajo	40€	6.720€
Nota: El coste por hora incluye la amortización del equipo y la cuota de autónomo.		

Tabla 7.1: Presupuesto general

Bibliografía

[1] Adrián Abreu González, Melissa Díaz Arteaga e Iván Castilla Rodríguez. *Simulador para Planificación Dinámica y Estática*. https://github.com/etsiiull/SIMDE. 2017.

[2] Iván Castilla Rodríguez y col. «SIMDE: Un Simulador para el Apoyo Docente en la Enseñanza de las Arquitecturas ILP con Planificación Dinámica y Estática». En: jul. de 2004.

[3] Adrián Abreu González. «Simulador didáctico de arquitectura de computadores». En: jun. de 2017.

[4] Antonio Jesús López Garnier. «Plataforma de ludificación de un simulador didáctico de arquitectura de computadores». En: jun. de 2018.

[5] John L. Hennessy y David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. isbn: 012383872X.

[6] Melissa Díaz Arteaga. «Simulador didáctico de una arquitectura de planificación estática». En: ago. de 2018.

[7] Tobias Koppers y Webpack contributors. *Webpack is a module bundler*. https://github.com/webpack/webpack. 2014.

[8] Simone Scalco y col. «On the feasibility of detecting injections in malicious npm packages». En: jun. de 2022. doi: 10.1145/3538969.3543815.

[9] Mend. *Mend Renovate: Automated Dependency Updates*. https://renovatebot.com/. 2016.

[10] Isaac Schlueter y npm contributors. *npm - a JavaScript package manager*. https://www.npmjs.com/package/npm. 2009.

[11] José Román Hernández Martín (Manz). *Scripts de NPM*. https://lenguajejs.com/npm/administracion/scripts-de-npm. 2015.

[12] Github Inc. *GitHub Pages: Websites for you and your projects, hosted directly from your GitHub repository*. https://pages.github.com. 2008.

[13] Adrián Abreu González, Melissa Díaz Arteaga e Iván Castilla Rodríguez. *SIMDE Web en GitHub Pages*. https://etsiiull.github.io/SIMDE. 2017.

[14] Solomon Hykes y Docker Inc. *Docker overview*. https://docs.docker.com/get-started/overview. 2013.

[15] Microsoft Corporation. *Developing inside a Container*. https://code.visualstudio.com/docs/remote/containers. 2019.

[16] Microsoft Corporation. *TypeScript: JavaScript With Syntax For Types*. https://www.typescriptlang.org. 2012.

[17] Jordan Walke y Meta Platforms Inc. *React – A JavaScript library for building user interfaces*. https://reactjs.org. 2013.

[18] Dan Abramov y Andrew Clarke. *Redux - A predictable state container for JavaScript apps*. https://redux.js.org. 2015.

[19] Wikimedia Commons. *File:Ngrx-redux-pattern-diagram.png — Wikimedia Commons, the free media repository*. [Online; accessed 12-September-2022]. 2020. url: %5Curl%7Bhttps://commons.wikimedia.org/w/index.php?title=File:Ngrx-redux-pattern-diagram.png&oldid=47658546%7D.

[20] Erik Wittern, Philippe Suter y Shriram Rajagopalan. «A Look at the Dynamics of the JavaScript Package Ecosystem». En: *Proceedings of the 13th International Conference on Mining Software Repositories*. MSR '16. Austin, Texas: Association for Computing Machinery, 2016, págs. 351-361. isbn: 9781450341868. doi: 10.1145/2901739.2901743. url: https://doi.org/10.1145/2901739.2901743.

[21] Vinay Singh y col. «The Transition from Centralized (Subversion) VCS to Decentralized (Git) VCS: A Holistic Approach.» En: *IUP Journal of Electrical & Electronics Engineering* 12.1 (2019).

[22] Atlassian Corporation Plc. *Gitflow Workflow*. https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow. 2014.

[23] G Brown. *Software Chart*. [Online; accessed 12-September-2022]. 2017. url: %5Curl%7Bhttps://openclipart.org/detail/278052/software-chart%7D.

[24] Paul Hammant y friends. *Trunk Based Development*. https://trunkbaseddevelopment.com. 2017.

[25] P. Runeson. «A survey of unit testing practices». En: *IEEE Software* 23.4 (2006), págs. 22-29. doi: 10.1109/MS.2006.91.

[26] Wei-Tek Tsai y col. «End-to-end integration testing design». En: *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*. IEEE. 2001, págs. 166-171.

[27] Microsoft Corporation. *Playwright is a framework for Web Testing and Automation*. https://playwright.dev. 2019.

[28] H.K.N. Leung y L. White. «A study of integration testing and software regression at the integration level». En: *Proceedings. Conference on Software Maintenance 1990*. 1990, págs. 290-301. doi: 10.1109/ICSM.1990.131377.

[29] H.K.N. Leung y L. White. «Insights into regression testing (software testing)». En: *Proceedings. Conference on Software Maintenance - 1989*. 1989, págs. 60-69. doi: 10.1109/ICSM.1989.65194.

[30] GitHub Inc. *Dependabot - Automated dependency updates built into GitHub*. https://github.com/dependabot. 2017.

[31] Hampton Lintorn-Catlin. *Sass: Syntactically Awesome Style Sheets*. https://sass-lang.com. 2006.

[32] Jan Mühlemann e i18next Community. *i18next: learn once - translate everywhere*. https://github.com/i18next/i18next. 2011.

[33] Chris Trevino. *react-dnd: Drag and Drop for React*. https://github.com/react-dnd/react-dnd. 2014.

[34] Atlassian Corporation Plc. *react-beautiful-dnd: Beautiful and accessible drag and drop for lists with React*. https://github.com/atlassian/react-beautiful-dnd. 2017.

[35] M.J. Flynn. «Very high-speed computing systems». En: *Proceedings of the IEEE* 54.12 (1966), págs. 1901-1909. doi: 10.1109/PROC.1966.5273.

[36] E.D. Reilly. *Milestones in Computer Science and Information Technology*. Greenwood Press, 2003. isbn: 9781573565219. url: https://books.google.com/books?id=JTYPKxug49IC.