# Backlog de Cada Sprint

**Proyecto:** APT "CarMatch"  
**Integrantes:** Mario Francisco Fica Sánchez; Ulises Rosales Alejos  
**Carrera:** Ingeniería Informática  
**Sede:** Padre Alonso de Ovalle (PAO)  
**Fecha:** Octubre 2025

## Introducción

El proyecto *CarMatch* se desarrolló utilizando la metodología ágil **Scrum**, organizando el trabajo en iteraciones cortas llamadas *sprints*. En Scrum, un producto se construye a través de una serie de iteraciones (*sprints*) que dividen un proyecto grande en partes manejables[[1]](https://www.atlassian.com/es/agile/scrum/sprints#:~:text=%26quot%3BCon%20scrum%2C%20un%20producto%20se,producto%20de%20Jira%20de%20Atlassian). Cada sprint tiene una duración fija (en este caso, sprints semanales) y al finalizar cada iteración el equipo entrega un incremento funcional del producto para su revisión. Durante la planificación de cada sprint se seleccionaron ciertas **historias de usuario** desde el backlog del producto para desarrollarlas[[2]](https://www.innovaciondigital360.com/cio/metodologia-scrum-concepto-beneficios-e-implementaciones/#:~:text=En%20Scrum%2C%20el%20trabajo%20se,producto%20para%20trabajar%20en%20ellos), definiendo tareas técnicas específicas para completarlas. Esto permitió **entregar valor de forma incremental** y adaptarse a retroalimentación o cambios de requerimientos de un sprint a otro, asegurando un progreso consistente hacia los objetivos del proyecto.

En *CarMatch*, el equipo estuvo conformado por dos estudiantes desarrolladores con roles complementarios: **Mario Francisco Fica Sánchez**, encargado de infraestructura (servidor en la nube), base de datos, monitoreo y seguridad; y **Ulises Rosales Alejos**, encargado del desarrollo de funcionalidades de scraping y automatización (N8N), backend (Django) y frontend (Next.js). Siguiendo un enfoque iterativo por fases, los 11 sprints abarcaron desde la **planificación inicial** del proyecto, la **implementación de la infraestructura cloud**, el **desarrollo del backend** en Django, la configuración de **flujos automáticos** con N8N, el desarrollo del **frontend** en Next.js, hasta la **integración** de componentes. A continuación se presenta el **backlog de cada sprint** (semanas 1 a 11), detallando el objetivo de cada sprint, las historias de usuario abordadas, las tareas técnicas realizadas, los responsables, los criterios de aceptación y los entregables obtenidos.

## Sprint 1: Inicio del Proyecto

**Objetivo del Sprint:** Definir el alcance del proyecto CarMatch, documentar los requisitos iniciales y establecer la base conceptual para el desarrollo.

| Historias de Usuario | Tareas Técnicas | Responsables | Criterios de Aceptación | Entregables |
| --- | --- | --- | --- | --- |
| **HU1:** Como **patrocinador del proyecto**, quiero que el equipo defina y documente claramente el **alcance del proyecto** CarMatch, de manera que todos los involucrados tengan una visión común y aprobada del objetivo a lograr. | • Reunir requerimientos generales con el cliente/tutor.<br>• Elaborar el **Documento de Definición del Proyecto** (visión, objetivos, alcance, restricciones).<br>• Revisar y ajustar el documento con el feedback del tutor. | • **Mario F. Fica** – Redacción y estructuración del documento de definición del proyecto.<br>• **Ulises R. Alejos** – Colaboración en definición de alcance y revisión. | • Documento de definición completo, describiendo objetivo, alcance, participantes y hitos principales del proyecto.<br>• Aprobación formal del documento por el tutor/profesor, indicando conformidad con la definición del proyecto. | • Documento de Definición del Proyecto (visión y alcance) aprobado. |
| **HU2:** Como **product owner**, quiero contar con una **Especificación de Requerimientos de Software (ERS)** detallada y validada, para asegurar que todas las necesidades y funcionalidades esperadas del sistema estén documentadas conforme a estándares, sirviendo de guía para el desarrollo. | • Identificar y listar todos los requisitos funcionales y no funcionales del sistema.<br>• Redactar la **ERS** siguiendo el estándar IEEE/ISO, incluyendo descripción de requisitos, criterios de calidad, etc.<br>• **Revisar y cerrar** la ERS mediante una reunión de validación, incorporando correcciones hasta obtener la versión final. | • **Mario F. Fica** – Desarrollo del documento ERS inicial y ajuste según revisiones.<br>• **Ulises R. Alejos** – Revisión cruzada de requisitos, aseguramiento de consistencia con la visión del proyecto.<br>• **Equipo** – Validación conjunta del contenido final de la ERS. | • Documento ERS completo y estructurado, cubriendo todos los requisitos del proyecto (funcionales y no funcionales).<br>• ERS revisado y firmado/aprobado por el tutor o cliente, sin observaciones pendientes. | • Especificación de Requerimientos de Software (ERS) versión final.<br>• Lista priorizada de requisitos aprobados. |
| **HU3:** Como **analista del proyecto**, quiero generar una **matriz de trazabilidad** que vincule cada requisito con sus casos de uso correspondientes, así como elaborar **casos de uso UML** de alta calidad, de modo que podamos garantizar la cobertura completa de los requisitos en el diseño del sistema. | • Derivar **casos de uso** a partir de los requisitos identificados (modelado de casos de uso en UML).<br>• Construir la **Matriz de Trazabilidad** que relaciona cada requisito del ERS con uno o más casos de uso.<br>• Revisar la coherencia entre la matriz y el ERS, asegurando que no queden requisitos sin caso de uso asociado. | • **Ulises R. Alejos** – Identificación y documentación de casos de uso; elaboración de diagramas UML.<br>• **Mario F. Fica** – Apoyo en verificación de trazabilidad y consistencia con requisitos.<br>• **Ulises R. & Mario F.** – Revisión conjunta de la matriz final. | • Matriz de trazabilidad completa, cubriendo el 100% de los requisitos del proyecto con al menos un caso de uso asociado a cada uno.<br>• Diagramas de caso de uso en UML legibles y validados, que describen correctamente las interacciones de los actores con el sistema conforme a los requisitos. | • Matriz de Trazabilidad de Requisitos vs. Casos de Uso.<br>• Diagramas UML de Casos de Uso del sistema. |

## Sprint 2: Arquitectura y Diseño de Prototipos

**Objetivo del Sprint:** Definir la arquitectura técnica de la solución e iniciar el diseño de la interfaz de usuario, proporcionando una visión clara de cómo se estructurará el sistema y cómo lucirá la aplicación.

| Historias de Usuario | Tareas Técnicas | Responsables | Criterios de Aceptación | Entregables |
| --- | --- | --- | --- | --- |
| **HU1:** Como **arquitecto de software**, quiero **diseñar la arquitectura general** del sistema (componentes principales y su interacción), para garantizar que la solución esté bien estructurada y que todos los módulos (backend, frontend, base de datos, scraping) puedan integrarse de forma eficiente. | • Definir la **arquitectura lógica** del sistema: diagramar módulos (scraping/N8N, API Django, base de datos, frontend) y sus interacciones.<br>• Definir la **arquitectura física**: decidir la infraestructura de despliegue (servidor cloud, base de datos en la nube), incluyendo diagrama de despliegue (servidor, conexiones, puertos).<br>• Documentar decisiones de arquitectura (patrones utilizados, tecnologías seleccionadas para cada componente). | • **Mario F. Fica** – Diseño de la arquitectura lógica (componentes de software) y física (infraestructura en la nube), elaboración de diagramas arquitectónicos.<br>• **Ulises R. Alejos** – Colaboración en definición de componentes de software (especialmente integración N8N, frontend) y revisión de la documentación de arquitectura. | • Documento de arquitectura del sistema completado, incluyendo diagramas de arquitectura lógica (estructura de módulos) y física (despliegue en servidor, BD, etc.).<br>• La arquitectura propuesta ha sido revisada y aprobada por el equipo/tutor, verificando que cumple con los requerimientos (escalabilidad, seguridad básica, coherencia con el ERS). | • Documento de Arquitectura del Sistema (diagramas y especificaciones técnicas de alto nivel).<br>• Diagramas arquitectónicos (componentes lógicos y distribución física) avalados por el tutor. |
| **HU2:** Como **diseñador UX/UI**, quiero desarrollar **wireframes y prototipos de alta fidelidad** de la interfaz de usuario de CarMatch, para visualizar la experiencia de uso y obtener retroalimentación temprana antes de la implementación. | • Diseñar **wireframes** de las pantallas principales de la aplicación (baja fidelidad, para flujo básico: página de búsqueda, listado de resultados, detalles de repuesto/servicio).<br>• Crear **mockups de alta fidelidad** basados en los wireframes, aplicando lineamientos de usabilidad y accesibilidad, definiendo la identidad visual de la plataforma.<br>• Presentar los prototipos al tutor/usuarios para retroalimentación y refinar el diseño según comentarios (iteración sobre los mockups). | • **Ulises R. Alejos** – Creación de wireframes iniciales y mockups en herramienta de diseño (por ejemplo, Figma), asegurando alineamiento con los requisitos de UX.<br>• **Mario F. Fica** – Revisión de las propuestas de interfaz, aportando consideraciones de viabilidad técnica e integración con backend. | • Wireframes que cubren todos los casos de uso principales, validados por el equipo (navegación coherente y completa).<br>• Prototipos de alta fidelidad con diseño UI consistente y atractivo, revisados con feedback incorporado del tutor/usuarios (cumplimiento de principios de usabilidad). | • Set de Wireframes de las principales pantallas de CarMatch.<br>• Mockups/Prototipos de interfaz en alta fidelidad (diseño UI final) para las pantallas clave. |
| **HU3:** Como **equipo de desarrollo**, quiero contar con un **plan de infraestructura** y configuración inicial para nuestros entornos (desarrollo, pruebas y producción), de modo que tengamos claridad sobre cómo y dónde se desplegarán los componentes (frontend, backend, base de datos, N8N) y podamos preparar dichos entornos con anticipación. | • Decidir la plataforma de despliegue: elección de proveedor cloud (por ejemplo, GCP) y especificaciones de máquinas/servicios a utilizar (VM, PostgreSQL administrado, etc.).<br>• Planificar la configuración inicial del servidor (SO, dependencias) y de la base de datos en la nube, asegurando compatibilidad con Django y N8N.<br>• Documentar un **plan de infraestructura** que incluya recursos necesarios, accesos, y consideraciones de red para el futuro despliegue. | • **Mario F. Fica** – Selección de tecnologías de infraestructura (ej: VM en GCP para backend/N8N, Cloud SQL para PostgreSQL), elaboración del plan de despliegue y configuración.<br>• **Ulises R. Alejos** – Revisión del plan para asegurar que soporta las necesidades de scraping y frontend, identificando cualquier requisito adicional. | • Decisiones de infraestructura tomadas (documentadas): proveedor cloud definido, tamaño de instancias, configuraciones base acordadas.<br>• Plan de infraestructura revisado por ambos integrantes, asegurando que abarca consideraciones de seguridad, escalabilidad básica y presupuesto disponible (uso de cuentas gratuitas, etc.). | • Plan de Infraestructura Cloud (documento con detalles de entornos de desarrollo/pruebas/producción, configuración de servidor, base de datos y N8N).<br>• Lista de recursos y accesos necesarios preparados para la siguiente fase de implementación. |

## Sprint 3: Infraestructura Inicial (N8N y BD)

**Objetivo del Sprint:** Implementar la infraestructura básica del sistema, incluyendo la instalación de la herramienta de automatización de flujos (*N8N*) y la configuración inicial de la base de datos y modelo de datos, sentando las bases para el procesamiento y almacenamiento de la información de CarMatch.

| Historias de Usuario | Tareas Técnicas | Responsables | Criterios de Aceptación | Entregables |
| --- | --- | --- | --- | --- |
| **HU1:** Como **desarrollador de automatización**, quiero **instalar y configurar N8N en el entorno local/servidor**, para poder diseñar y ejecutar flujos automáticos de recolección y procesamiento de datos (web scraping y ETL) que alimenten la plataforma CarMatch. | • Instalar la herramienta **N8N** en el servidor local (o VM cloud) con los componentes necesarios (Docker o instalación directa).<br>• Configurar N8N para que esté operativo: creación de usuario administrador, ajustes de entorno (puertos, credenciales básicas).<br>• Verificar la ejecución correcta de N8N con un flujo de prueba simple (hello world) para asegurar que la plataforma de automatización funciona sin errores. | • **Ulises R. Alejos** – Instalación y configuración técnica de N8N; pruebas iniciales de funcionamiento de flujos en N8N.<br>• **Mario F. Fica** – Soporte en configuración del entorno (ej. apertura de puertos en firewall local para N8N, provisión de máquina) y validación post-instalación. | • Instancia de N8N instalada y accesible en el entorno designado (ej. servidor local o cloud), con una cuenta administradora creada.<br>• Ejecución exitosa de un flujo de prueba en N8N, confirmando que la herramienta opera correctamente y puede conectarse a internet/BD según sea necesario. | • Plataforma N8N operativa en entorno de desarrollo (o servidor designado) lista para crear flujos.<br>• Captura de pantalla o evidencia del flujo de prueba exitoso en N8N. |
| **HU2:** Como **administrador de datos**, quiero **configurar la base de datos** inicial del proyecto (PostgreSQL en la nube) y definir el **modelo de datos normalizado**, para almacenar de forma consistente la información de repuestos y servicios recopilada, asegurando integridad y eficiencia en las consultas. | • Deploy de instancia **PostgreSQL** en la nube (por ejemplo, creación de una base de datos en Google Cloud SQL) o configuración local temporal para desarrollo.<br>• Definir el **modelo de datos**: diseñar el esquema de la base de datos con tablas normalizadas (ej.: tabla de repuestos, proveedores, historial de precios, etc.) según los requisitos identificados.<br>• Elaborar el **diccionario de datos**, documentando cada tabla, campo, tipo de dato y relaciones (PK, FK) de acuerdo al modelo diseñado. | • **Mario F. Fica** – Configuración de la instancia de base de datos en GCP (o local), incluyendo creación de usuarios/roles y ajustes de seguridad básicos; diseño del modelo entidad-relación y normalización.<br>• **Ulises R. Alejos** – Revisión del modelo de datos propuesto contra los requisitos (verificando que todas las entidades necesarias estén consideradas); aportes para optimizar la estructura según las necesidades de búsqueda/comparación. | • Base de datos PostgreSQL accesible y funcionando (localmente o en cloud) para el proyecto, con las conexiones verificadas.<br>• Modelo de datos relacional diseñado y documentado, cumpliendo formas normales (sin redundancias innecesarias) y abarcando los datos clave del dominio CarMatch (catálogo de repuestos/servicios, proveedores, precios, etc.).<br>• Diccionario de datos completo y validado, que describe claramente la estructura de la base de datos. | • Modelo Entidad-Relación (diagrama) del sistema CarMatch.<br>• Diccionario de Datos del proyecto (descripción de tablas y campos).<br>• Instancia de Base de Datos inicial configurada (con esquema vacío listo para migraciones). |
| **HU3:** Como **ingeniero de datos**, quiero establecer un **procedimiento inicial de carga de datos de ejemplo** (dataset de repuestos) en la base de datos, para validar que el modelo funciona correctamente y preparar un subconjunto de datos de prueba con el cual desarrollar y probar las funcionalidades del sistema. | • Preparar un conjunto de **datos de prueba** representativo (ej.: lista de repuestos con precios de distintas fuentes) en formato CSV u otro.<br>• Cargar los datos de ejemplo en la base de datos recientemente configurada, ya sea manualmente (scripts SQL) o usando N8N para insertar registros, verificando que se respeten las relaciones y restricciones del modelo.<br>• Realizar **consultas de verificación** en la base de datos (SELECTs) para comprobar que los datos se almacenaron correctamente y que las tablas se relacionan según lo esperado. | • **Ulises R. Alejos** – Selección o generación de datos de ejemplo realistas; uso de N8N o scripts para insertar los datos en la base de datos.<br>• **Mario F. Fica** – Verificación de la integridad de los datos cargados; ejecución de consultas de prueba y ajuste de esquema si alguna incidencia es detectada. | • Base de datos poblada con un dataset de prueba (p. ej., al menos 20 repuestos con sus respectivos precios y proveedores) sin errores de integridad referencial.<br>• Resultados de consultas de prueba correctos (los datos de ejemplo pueden ser recuperados y combinados desde múltiples tablas según el modelo).<br>• Confirmación de que el modelo de datos soporta las operaciones básicas requeridas (ej.: listar repuestos con sus precios y fuentes). | • Datos de prueba cargados en la BD (registro de inserción o CSV importado).<br>• Informe breve de verificación de la BD (consultas ejecutadas y resultados esperados/obtenidos). |

## Sprint 4: Depuración de Datos y Prueba de Rendimiento

**Objetivo del Sprint:** Desarrollar procesos iniciales de depuración y enriquecimiento de datos (ETL) para garantizar la calidad de la información, e implementar pruebas básicas de rendimiento sobre la infraestructura montada, con el fin de evaluar la capacidad del sistema y asegurar que se cumplen criterios mínimos de eficiencia.

| Historias de Usuario | Tareas Técnicas | Responsables | Criterios de Aceptación | Entregables |
| --- | --- | --- | --- | --- |
| **HU1:** Como **administrador de la plataforma**, quiero que los **datos de repuestos** recopilados estén **limpios y normalizados**, eliminando duplicados y formateando la información de manera uniforme, para asegurar que los usuarios vean datos consistentes y confiables al realizar comparaciones en CarMatch. | • Desarrollar un flujo de **ETL de deduplicación/normalización** en N8N: identificar registros duplicados de repuestos (mismo código o descripción) provenientes de distintas fuentes y consolidarlos en uno solo; homogenizar unidades, formatos de precios/moneda, etc.<br>• Implementar reglas de limpieza de datos (por ejemplo, eliminar caracteres especiales, unificar mayúsculas/minúsculas) para las entradas de repuestos y proveedores.<br>• Probar el flujo ETL con los datos de ejemplo cargados, verificando que al final del proceso los datos resultantes no tengan duplicados y siguen siendo íntegros. | • **Ulises R. Alejos** – Configuración de workflow N8N para ETL de limpieza de datos: nodos de lectura, transformación (scripts o funciones en N8N) y escritura en BD; ejecución y ajuste del proceso hasta lograr la limpieza deseada.<br>• **Mario F. Fica** – Validación de la calidad de datos antes y después del ETL; verificación manual de algunos registros para confirmar la correcta deduplicación y consistencia. | • Flujo de ETL configurado en N8N que toma datos brutos de repuestos y produce datos limpios (sin duplicados ni formatos inconsistentes).<br>• Datos de prueba actualizados en la base de datos: no existen entradas duplicadas para un mismo repuesto/proveedor; campos normalizados (ej.: mismas unidades de medida, formato numérico unificado).<br>• Evidencia de ejecución exitosa del proceso ETL (log de N8N o reporte de cambios realizados). | • Workflow N8N de limpieza de datos (documentado, con capturas o descripción de pasos).<br>• Base de datos depurada según las reglas definidas (datos consistentes post-ETL). |
| **HU2:** Como **analista de negocio**, quiero que el sistema mantenga un **histórico de precios y cambios** en los repuestos a lo largo del tiempo, de modo que se pueda analizar la evolución de los precios y brindar información transparente al usuario sobre tendencias o variaciones en el mercado automotriz. | • Extender el modelo de datos para incluir una estructura de **historial de precios** (por ejemplo, tabla de historial con id de repuesto, fecha, precio, fuente).<br>• Modificar los flujos de actualización de datos (N8N) para que en lugar de sobrescribir siempre el precio de un repuesto, se guarde el registro histórico cada vez que hay un cambio o una nueva lectura de precio.<br>• Realizar pruebas simuladas: actualizar manualmente algunos precios de ejemplo mediante el flujo y verificar que los cambios quedan registrados en la tabla de historial sin duplicar entradas previas inalteradas. | • **Ulises R. Alejos** – Implementación en N8N de la lógica de registro histórico (condiciones para insertar en tabla de historial cuando detecte variación de precio o nuevo dato); ajustes en la base de datos (creación de tabla historial si no existía, índices si necesarios).<br>• **Mario F. Fica** – Diseño de la tabla de historial y aseguramiento de que la performance de la BD es adecuada con la estrategia de historial; verificación de consistencia tras actualizaciones (que el precio vigente también se refleja donde corresponda). | • Tabla de historial de precios creada y vinculada correctamente con la tabla principal de repuestos (clave foránea, etc.).<br>• Flujos de actualización de precios ajustados: al correrlos, los precios nuevos se agregan como nuevos registros en historial sin errores ni duplicados; los precios sin cambios no generan entradas repetidas.<br>• Capacidad de generar un reporte básico de historial por repuesto (ej., consultas muestran múltiples precios antiguos ordenados por fecha para un repuesto dado). | • Esquema de base de datos actualizado con tabla de Historial de Precios.<br>• Funcionalidad de registro histórico integrada en los workflows de actualización (N8N).<br>• Ejemplos de datos de historial capturados (p. ej., CSV o consulta de historial por item). |
| **HU3:** Como **arquitecto del sistema**, quiero realizar una **prueba de rendimiento inicial** del backend y la base de datos utilizando datos de muestra, para evaluar si la configuración actual soporta la carga esperada (consultas de búsqueda y comparación) y detectar tempranamente cuellos de botella o la necesidad de optimizar algún componente. | • Preparar un conjunto de **escenarios de prueba de carga**: por ejemplo, múltiples solicitudes concurrentes al API de comparación de precios, consultas intensivas a la base de datos simulando múltiples usuarios.<br>• Utilizar herramientas o scripts (puede ser un flujo N8N o una herramienta como JMeter/Postman) para lanzar las peticiones concurrentes simuladas al sistema (API/Django) y medir tiempos de respuesta, uso de CPU/memoria en el servidor, etc.<br>• Analizar los resultados de la prueba y elaborar un **informe de rendimiento** inicial con métricas (latencia promedio, throughput, utilización de recursos) y posibles recomendaciones (p. ej., ajustar parámetros de la BD, considerar caching más agresivo) de ser necesario. | • **Ulises R. Alejos** – Diseño e implementación de la prueba de carga (scripts de prueba, configuración de N8N para estrés simulado u otra herramienta); ejecución de las pruebas y recolección de métricas.<br>• **Mario F. Fica** – Monitoreo del comportamiento del servidor y base de datos durante las pruebas; análisis conjunto de los resultados e identificación de límites o fallas de rendimiento. | • Prueba de rendimiento completada sin que el sistema colapse: el backend Django responde a las peticiones de prueba dentro de tiempos aceptables (definidos en requisitos no funcionales, e.g., <2s por consulta en promedio bajo carga ligera).<br>• Métricas de performance documentadas, incluyendo identificación de cualquier consulta lenta o recurso sobrecargado.<br>• Validación de que la infraestructura actual puede manejar al menos la carga básica esperada; si hubo hallazgos críticos, estos quedan documentados con plan de mitigación. | • Informe de Pruebas de Rendimiento Inicial (resultados y análisis de carga simulada).<br>• Scripts o configuraciones utilizadas para pruebas de carga (anexados).<br>• Ajustes realizados o recomendaciones para mejorar rendimiento (en caso necesario). |

## Sprint 5: Seguridad de Infraestructura

**Objetivo del Sprint:** Configurar la red y la seguridad en la infraestructura del servidor, estableciendo mecanismos de protección y control de acceso que resguarden la plataforma CarMatch y su información frente a accesos no autorizados o vulnerabilidades en esta etapa inicial.

| Historias de Usuario | Tareas Técnicas | Responsables | Criterios de Aceptación | Entregables |
| --- | --- | --- | --- | --- |
| **HU1:** Como **administrador de sistemas**, quiero configurar las **políticas de red y cortafuegos** en el servidor cloud, de manera que solo los puertos y servicios necesarios (p. ej., SSH, puerto de API, N8N, BD) estén accesibles y se bloquee todo tráfico no requerido, para minimizar la superficie de ataque. | • Configurar las **reglas de firewall** en la plataforma cloud: habilitar únicamente puertos seguros (por ej., 22 para SSH con IPs confiables, 80/443 si aplica para API/frontend, 5678 para N8N si requiere acceso, etc.) y denegar tráfico entrante no necesario.<br>• Verificar la conexión a cada servicio desde fuentes autorizadas (p.ej., cliente SSH desde la red del desarrollador, conexión a BD desde backend) para asegurar que las reglas permiten el funcionamiento normal.<br>• Registrar la configuración de red aplicada (listado de puertos abiertos/cerrados y origen permitido) en la documentación del proyecto. | • **Mario F. Fica** – Configuración de reglas de firewall/VPC en la nube; pruebas de acceso a servicios tras la configuración; documentación de la configuración realizada.<br>• **Ulises R. Alejos** – Soporte en pruebas (intentar accesos con puertos correctos/incorrectos para validar el firewall); confirmación de que la aplicación sigue funcionando con las nuevas reglas. | • Servidor con firewall activo acorde a las mejores prácticas: puertos innecesarios cerrados, únicamente accesos imprescindibles permitidos (por protocolo, puerto y origen).<br>• Pruebas de conectividad superadas: se puede acceder a los servicios permitidos (ej.: el backend/SSH responden a conexiones permitidas) y se confirmó que peticiones por puertos bloqueados no son atendidas.<br>• Documentación clara de la configuración de red, facilitando futuras revisiones o cambios. | • Configuración de Firewall/Seguridad de red (capturas de pantalla o lista de reglas aplicadas).<br>• Informe breve de validación de accesos (resultado de pruebas de puertos). |
| **HU2:** Como **usuario de la plataforma**, quiero que la comunicación con el sistema sea **segura**, de modo que cualquier dato que ingrese o consulte (por ejemplo, credenciales o información personal) viaje cifrado y protegido contra interceptaciones. | • **Instalar certificados SSL** y habilitar HTTPS para los servicios web: si el frontend o API estarán expuestos, configurar un certificado (auto-firmado en desarrollo o de Let's Encrypt en producción) en el servidor web o proxy.<br>• Configurar la **conexión segura a la base de datos** (por túnel SSL o conexiones autenticadas fuertemente) si aplica, para proteger la información en tránsito entre backend y BD.<br>• Realizar pruebas de acceso vía HTTPS al servicio (en caso de tener front/API ya accesible) verificando que el certificado es válido y que el tráfico está cifrado; comprobar que no es posible acceder vía HTTP plano (redirecciones o bloqueos establecidos). | • **Mario F. Fica** – Instalación y configuración de certificados SSL en el entorno (por ej., usar Nginx como proxy inverso con HTTPS habilitado para backend); aseguramiento de conexiones cifradas en todos los puntos posibles.<br>• **Ulises R. Alejos** – Pruebas funcionales tras la habilitación de SSL (verificar que las solicitudes al backend/front se atienden correctamente bajo HTTPS); ajuste de cualquier endpoint de integración (N8N, etc.) para usar URLs seguras si es necesario. | • Comunicación cifrada implementada: el acceso al frontend (si disponible) y/o API se realiza mediante HTTPS sin advertencias de seguridad.<br>• Certificado SSL en funcionamiento (en entorno de desarrollo puede ser auto-firmado, pero probado).<br>• Conexiones entre servicios internos seguras o locales (BD solo accesible internamente o mediante credenciales robustas), garantizando que datos sensibles no viajan en texto plano. | • Certificado SSL instalado (detalle del certificado o cadena).<br>• Configuraciones de servidor/proxy habilitando HTTPS (archivos de configuración o descripción).<br>• Evidencia de acceso exitoso vía HTTPS (captura de navegador o herramienta sin errores de certificado). |
| **HU3:** Como **responsable de seguridad**, quiero implementar **mecanismos básicos de protección en el servidor** (por ejemplo, actualización de paquetes, deshabilitar accesos por contraseña en SSH, instalación de fail2ban u otras herramientas), para reducir vulnerabilidades comunes y proteger el entorno antes de avanzar con desarrollos más complejos. | • Actualizar el sistema operativo y paquetes del servidor a la última versión (parches de seguridad aplicados).<br>• Configurar **SSH seguro**: deshabilitar login de *root*, obligar uso de llaves SSH en lugar de contraseña; opcionalmente cambiar puerto de SSH para ofuscar.<br>• Instalar herramientas de **monitoreo de intrusiones o anti-fuerza bruta** (ej.: *fail2ban* para bloquear IPs después de intentos fallidos) y configurar alertas básicas de seguridad si es posible.<br>• Probar que tras estos cambios, los servicios siguen accesibles para el equipo (llave SSH funciona, etc.) y que los mecanismos de protección actúan (ej., intentar un login SSH con contraseña para ver si es rechazado). | • **Mario F. Fica** – Ejecución de actualizaciones del sistema; configuración de SSH hardened (usuarios, llaves, etc.); instalación y configuración de herramientas de seguridad adicionales.<br>• **Ulises R. Alejos** – Verificación post-configuración: confirmar acceso con nuevos parámetros (uso de llave SSH), simular escenarios de intento fallido para observar respuesta de seguridad (fail2ban u otros). | • Servidor actualizado y endurecido: todos los paquetes al día, sin servicios innecesarios corriendo; acceso SSH únicamente mediante clave (comprobado) y usuarios limitados.<br>• Herramientas de protección activas (por ejemplo, fail2ban bloqueando IP tras X intentos fallidos, comprobado en logs).<br>• No se introdujeron regresiones: la aplicación y herramientas (N8N, etc.) siguen funcionando correctamente tras las medidas de seguridad. | • Lista de medidas de hardening aplicadas (actualizaciones realizadas, config SSH, herramientas instaladas).<br>• Registros o evidencias de funcionamiento de las medidas (ej.: log de fail2ban mostrando un bloqueo de prueba).<br>• Reporte de estado de seguridad del servidor tras la configuración. |

## Sprint 6: Modelado del Backend

**Objetivo del Sprint:** Construir la base del backend de la aplicación mediante el framework Django, definiendo los modelos de datos dentro del proyecto backend y preparando la estructura del API, de forma que el sistema pueda comenzar a manejar la información central (repuestos, proveedores, etc.) de acuerdo al diseño establecido.

| Historias de Usuario | Tareas Técnicas | Responsables | Criterios de Aceptación | Entregables |
| --- | --- | --- | --- | --- |
| **HU1:** Como **desarrollador backend**, quiero **iniciar el proyecto Django** e integrar la base de datos al entorno de desarrollo, para contar con una estructura de proyecto lista donde pueda implementar las funcionalidades del lado del servidor de CarMatch. | • Crear un nuevo proyecto Django (django-admin startproject) y una aplicación principal para CarMatch (por ejemplo, app "core").<br>• Configurar la conexión de Django con la base de datos PostgreSQL previamente creada: ajustar settings.py con las credenciales y parámetros de la BD; verificar la conectividad.<br>• Ejecutar migraciones iniciales de Django (incluyendo las automáticas de Django auth/admin) para confirmar que el proyecto reconoce el esquema. | • **Ulises R. Alejos** – Creación del proyecto Django y configuración de base de datos en el código; ejecución de migraciones y resolución de cualquier error de conexión/configuración.<br>• **Mario F. Fica** – Revisión de la configuración (especialmente asegurando que no se exponen credenciales sensibles en código); verificación de que el proyecto corre en local/servidor dev sin problemas tras configuración. | • Proyecto Django creado exitosamente, sin errores al ejecutar el servidor de desarrollo (manage.py runserver).<br>• Conexión Django-PostgreSQL establecida: las migraciones iniciales aplicadas correctamente a la base de datos (tablas de Django auth/admin creadas, etc.).<br>• Estructura de proyecto bajo control de versiones (código subido al repositorio) para seguimiento del desarrollo. | • Código base del proyecto Django "CarMatch" (estructura inicial de archivos, configuración de settings con BD).<br>• Registro de migraciones aplicadas (entry de migración inicial).<br>• Documento de configuración del entorno de desarrollo (variables de entorno para DB, etc.). |
| **HU2:** Como **arquitecto de datos**, quiero **implementar en Django los modelos de datos** definidos (correspondientes a repuestos, proveedores, precios, etc.), reflejando el modelo entidad-relación planificado, de manera que el ORM de Django gestione correctamente las tablas y podamos manipular los datos desde el código de forma sencilla. | • Definir en models.py de la aplicación Django las clases de modelo para las principales entidades: por ejemplo, Repuesto, Proveedor, Precio (u HistorialPrecio), etc., incluyendo campos con sus tipos y restricciones según el diseño (longitudes, opciones, relaciones ForeignKey o ManyToMany donde aplique).<br>• Ejecutar makemigrations y migrate para generar las migraciones correspondientes a estos modelos y aplicarlas en la base de datos, creando así las tablas definidas desde Django.<br>• Verificar la correspondencia entre los modelos Django y el esquema original planificado: revisar que todos los campos importantes están incluidos y que las relaciones en BD coinciden (claves foráneas correctamente creadas). | • **Ulises R. Alejos** – Codificación de los modelos en Django; generación y aplicación de migraciones; ajuste de modelos en caso de encontrar inconsistencias o necesidades (tipos de campo adecuados, e.g. DecimalField para precios, etc.).<br>• **Mario F. Fica** – Revisión del código de modelos asegurando que se cumplen las reglas de normalización y restricciones (por ejemplo, unique constraints si necesarios); comparación contra el diccionario de datos original. | • Modelos Django implementados para todas las entidades clave, con nombres de campos, tipos de datos y relaciones coherentes con el modelo de la base de datos diseñado previamente.<br>• Migraciones aplicadas satisfactoriamente: las tablas y columnas en la base de datos reflejan los modelos sin error. (La base queda estructurada mediante Django ORM, lista para operaciones).<br>• Se puede acceder a los modelos vía Django shell u admin y crear/leer objetos, demostrando que la integración ORM-BD es funcional. | • Código fuente de modelos Django (archivo models.py con definiciones de clases de modelo).<br>• Migraciones de Django generadas para la creación de tablas de negocio (archivos de migración .py).<br>• Tablas correspondientes en la base de datos (verificables con \d en psql o equivalente). |
| **HU3:** Como **Product Owner**, quiero contar con un **panel de administración básico** (Django Admin) para visualizar y verificar los datos ingresados en el sistema (repuestos, proveedores, precios), de manera de tener un control de calidad inicial de la información y facilitar la carga o corrección manual si fuera necesario. | • Habilitar el **Django Admin** para los nuevos modelos: registrar los modelos creados en el admin (admin.py), definiendo representaciones amigables y filtros de ser útil.<br>• Crear un superusuario en Django para tener acceso al panel administrativo.<br>• Probar el acceso al panel admin (/admin) en el servidor de desarrollo: loguearse, comprobar que los modelos aparecen y los datos de ejemplo (cargados previamente en BD) se visualizan correctamente; realizar inserciones o ediciones desde el admin para verificar su funcionamiento. | • **Mario F. Fica** – Configuración del Django Admin (registro de modelos, personalización mínima); creación de usuario admin.<br>• **Ulises R. Alejos** – Pruebas de uso del admin: navegación por registros de repuestos, proveedores, etc.; comprobación de que los cambios hechos vía admin reflejan en la BD y viceversa. | • Panel de administración Django operativo y accesible, mostrando los modelos de datos relevantes de CarMatch.<br>• Los datos de ejemplo se pueden visualizar desde el admin sin errores, y es posible agregar/editar registros manualmente.<br>• El admin cuenta con los permisos adecuados (solo accesible a superusuario) y provee un respaldo para gestionar datos mientras no exista una interfaz de usuario específica para esas funciones. | • Django Admin habilitado para modelos principales (evidencia: captura de pantalla de la interfaz admin listando objetos).<br>• Usuario administrador creado (credenciales documentadas de forma segura para entorno de desarrollo).<br>• Registro de pruebas realizadas en el admin (p. ej., creación manual de un objeto de prueba y verificación en BD). |

## Sprint 7: Desarrollo de API REST

**Objetivo del Sprint:** Desarrollar la API RESTful del sistema CarMatch utilizando Django (Django REST Framework), implementando los endpoints necesarios que permitirán ofrecer a los usuarios las funcionalidades de búsqueda y comparación de repuestos a través del frontend. Esto incluye la lógica de negocio básica para gestionar los datos de repuestos y sus precios desde el backend.

| Historias de Usuario | Tareas Técnicas | Responsables | Criterios de Aceptación | Entregables |
| --- | --- | --- | --- | --- |
| **HU1:** Como **usuario final de CarMatch**, quiero **buscar repuestos o servicios automotrices** en la plataforma por nombre o tipo, para obtener un listado de opciones disponibles junto con sus precios y proveedores, ahorrando tiempo en la comparación manual. | • Implementar en el backend un **endpoint de búsqueda/listado de repuestos** (ej: GET /api/repuestos?query=<texto>): crear vista/controlador en Django que filtre los modelos de repuestos por texto (nombre, descripción) y devuelva un JSON con la lista de coincidencias, incluyendo detalles relevantes (nombre, modelo, etc.) y tal vez precio mínimo actual.<br>• Incluir en la respuesta datos necesarios para la comparación rápida (por ejemplo, cantidad de proveedores encontrados, rango de precios, etc. si se define así).<br>• Probar el endpoint manualmente (via curl o herramienta REST) con datos de ejemplo para asegurar que filtra correctamente y formatea la respuesta según lo definido. | • **Ulises R. Alejos** – Desarrollo del endpoint de búsqueda de repuestos (vista Django o DRF ViewSet); implementación de la lógica de filtrado y serialización de datos a JSON.<br>• **Mario F. Fica** – Revisión de la performance de la consulta (asegurando uso eficiente de ORM, e.g. evitando N+1 queries); validación de la precisión de los resultados devueltos con diferentes consultas de prueba. | • Endpoint GET /api/repuestos operativo: al enviar una consulta con un término, devuelve un JSON con los repuestos que contienen ese término, con campos esenciales (por ejemplo: id, nombre, categoría, precio mínimo, etc.).<br>• Se maneja el caso de no encontrar resultados (JSON vacío o mensaje apropiado).<br>• Pruebas realizadas: consultas con distintos criterios devuelven los resultados esperados (ej., buscar por nombre parcial trae los repuestos correctos). | • Código del endpoint de búsqueda (vista/serializer en Django REST).<br>• Ejemplos de respuesta JSON del endpoint con datos de prueba.<br>• Documentación inicial de la API (descripción del endpoint de búsqueda y sus parámetros). |
| **HU2:** Como **usuario**, quiero ver el **detalle de un repuesto específico**, incluyendo todas sus ofertas o precios disponibles de distintos proveedores, para poder comparar y elegir la mejor opción. | • Implementar un **endpoint de detalle de repuesto** (ej: GET /api/repuestos/<id>): devuelve información detallada de un repuesto específico, incluyendo sus atributos y una lista embebida o relacionada de precios por proveedor (por ejemplo: proveedor X - $Precio Y, fecha última actualización, etc.).<br>• Asegurarse de incluir en la respuesta datos como descripción completa, especificaciones si las hay, y un listado estructurado de ofertas/precios (ordenado por precio o por fecha).<br>• Probar el endpoint con un ID de ejemplo para validar que trae los datos correctos y que maneja casos de error (ej.: ID no existente retorna 404 apropiado). | • **Ulises R. Alejos** – Desarrollo del endpoint de detalle (uso de Django ORM para traer repuesto y relacionadas, o DRF relations/serializers anidados); incorporación de historial de precios o precios actuales en la respuesta.<br>• **Mario F. Fica** – Verificación de que la respuesta incluye todos los datos necesarios para la comparación; pruebas de respuesta con repuestos que tienen múltiples precios vs. aquellos con uno solo; aseguramiento de formato consistente. | • Endpoint GET /api/repuestos/<id> operativo: dado el id de un repuesto existente, retorna un JSON con la información completa del repuesto (campos básicos) + array de precios por proveedor asociados a ese repuesto.<br>• Si el repuesto no existe, retorna un mensaje de error/404 JSON correctamente.<br>• La estructura de datos facilita su representación en el frontend (ej., lista de precios con proveedor y fecha). | • Código del endpoint de detalle de repuesto (vista/serializer).<br>• Ejemplo de respuesta JSON con detalle y ofertas para un repuesto (con datos simulados).<br>• Actualización de la documentación de API (sección de detalle de repuesto). |
| **HU3:** Como **desarrollador frontend**, quiero disponer de un **API REST autenticada** para operaciones seguras (por ejemplo, agregar nuevos datos o acceder a información sensible), de forma que el acceso al backend esté protegido y solo usuarios autorizados puedan realizar ciertas acciones en CarMatch. | • **Configurar autenticación en la API**: decidir método (p. ej., autenticación token JWT o token simple de DRF); implementar la obtención de token (endpoint de login) y requerir token en endpoints sensibles.<br>• Añadir **restricciones y permisos** en los endpoints ya creados según corresponda: posiblemente los endpoints de consulta de repuestos sean públicos, pero se prepara la estructura para que endpoints futuros de modificación requieran auth (y, por ejemplo, el admin panel permanece protegido por Django admin de por sí).<br>• Probar la autenticación: obtener un token con un usuario de prueba y luego hacer llamadas autenticadas a un endpoint protegido verificando que sin token rechaza (HTTP 401) y con token permite. | • **Mario F. Fica** – Implementación de la capa de autenticación en el backend (instalación/configuración de JWT u otro método; ajustes en settings.py y creación de endpoints de auth si es JWT).<br>• **Ulises R. Alejos** – Pruebas integrales de los endpoints con y sin autenticación; asegurarse de documentar para frontend cómo enviar el token en las solicitudes. | • Mecanismo de autenticación funcionando: es posible obtener un token de autenticación válido mediante credenciales de usuario, y las rutas designadas lo exigen correctamente (responden 401 No autorizado si no se provee o es inválido).<br>• No se ha roto el acceso a endpoints públicos (los de búsqueda/detalle de repuestos siguen accesibles sin token si así se definió).<br>• El sistema está preparado para proteger futuros endpoints de creación/edición, garantizando seguridad básica en la API. | • Configuración de Autenticación en Django/DRF (por ejemplo, JSON Web Token o TokenAuth configurado).<br>• Endpoint(s) de login o obtención de token funcionando (ejemplo de petición/response).<br>• Documento actualizado de la API indicando esquemas de autenticación y permisos. |

## Sprint 8: Comparador de Precios y Seguridad API

**Objetivo del Sprint:** Finalizar las funcionalidades avanzadas del backend, implementando el endpoint específico de comparación de precios de un repuesto entre múltiples fuentes (núcleo de la propuesta CarMatch) y robusteciendo la API con medidas de seguridad adicionales (autenticación, limitación de solicitudes, prevención de ataques comunes), asegurando así un servicio confiable y protegido.

| Historias de Usuario | Tareas Técnicas | Responsables | Criterios de Aceptación | Entregables |
| --- | --- | --- | --- | --- |
| **HU1:** Como **usuario de CarMatch**, quiero **comparar el precio** de un repuesto específico entre distintos proveedores en tiempo real, para identificar rápidamente cuál es la opción más conveniente de compra. | • Implementar un **endpoint de comparación de precios** (ej: GET /api/comparacion/<id\_repuesto>): al invocarlo, el backend recopila (o lee de la BD actualizada) los precios más recientes de ese repuesto en todos los proveedores disponibles, y devuelve un resumen con el precio mínimo, máximo, promedio y listado de cada proveedor con su precio actual.<br>• Integrar este endpoint con la lógica existente: si la BD ya tiene los precios actualizados vía N8N, simplemente formatear los datos; si no, potencialmente desencadenar una actualización on-demand (según alcance).<br>• Implementar **caché selectivo** para este endpoint de comparación: de modo que sucesivas solicitudes del mismo repuesto en un corto periodo no rehagan cálculos innecesarios (usar caching de Django o un mecanismo simple en memoria con expiración, para los resultados de comparación). | • **Ulises R. Alejos** – Desarrollo de la vista/función del endpoint de comparación; cálculo de métricas (min, max, avg) y estructuración de la respuesta JSON; implementación de caché utilizando la capa de caching de Django (por clave de repuesto) o almacenamiento temporal en servidor.<br>• **Mario F. Fica** – Configuración del sistema de caché (por ejemplo, Memcached o Redis si disponible, o fallback a cache local de Django); pruebas de carga del endpoint de comparación para verificar mejoras de rendimiento con caché versus sin caché. | • Endpoint GET /api/comparacion/<id> disponible y funcionando: dado un id de repuesto válido, retorna en JSON los datos comparativos (por ejemplo: "min\_price": X, "max\_price": Y, "providers": [ {prov1, precio1}, ... ]).<br>• Las respuestas consecutivas para el mismo <id> dentro de un intervalo corto usan la respuesta en caché (verificado por tiempos de respuesta menores o logs), mientras que cambios en los datos de precio se reflejan tras expirar la caché.<br>• Si se consulta un repuesto inexistente o sin precios, el sistema responde adecuadamente (mensaje de "sin datos" o similar). | • Código del endpoint de comparación con lógica de negocio y caching.<br>• Ejemplo de respuesta JSON de comparación (con datos simulados de varios proveedores).<br>• Configuración/documentación de la estrategia de caché (duración, invalidación). |
| **HU2:** Como **administrador**, quiero que la **API esté protegida contra un uso abusivo** (muchas solicitudes en poco tiempo) o ataques de fuerza bruta, para garantizar la disponibilidad del servicio y la estabilidad del sistema incluso cuando crezca el número de usuarios. | • Implementar **Rate Limiting** en la API: configurar un límite de solicitudes por minuto por IP o por token de usuario (p. ej., usando funcionalidades de Django REST Framework throttling o integrando una librería) para los endpoints principales.<br>• Añadir monitorización básica de cuántas veces se llega al límite (logs) para futuros análisis.<br>• Probar el comportamiento: simular más llamadas que el límite permitido desde una misma fuente y verificar que a partir de cierto punto el API responde con código de estado 429 (Too Many Requests) con el mensaje correspondiente, mientras que un ritmo normal de peticiones sigue funcionando. | • **Mario F. Fica** – Configuración de las políticas de *throttling* en DRF o middleware de rate limit; ajuste de umbrales (p. ej., X requests/minuto anónimas, Y requests/min autenticadas).<br>• **Ulises R. Alejos** – Pruebas de estrés focalizado desde un mismo cliente para confirmar que el límite se aplica; revisión de que usuarios diferentes no se ven afectados indebidamente (limite por IP/token individual, no global). | • Límite de peticiones implementado: el sistema controla la frecuencia de uso y devuelve error 429 con mensaje "Too many requests" cuando se excede (verificado en pruebas).<br>• Bajo uso normal, no hay impacto perceptible en los usuarios (los límites establecidos cubren holgadamente la expectativa de uso legítimo).<br>• Configuración flexible para ajustar en producción si fuese necesario, documentada. | • Configuraciones de Throttling/Rate Limit en el código (DRF settings o middleware custom).<br>• Resultado de prueba de límite (captura de respuesta 429 con headers de limit reset, etc., si aplica).<br>• Documentación en la API (políticas de tasa de solicitud para clientes). |
| **HU3:** Como **especialista en seguridad**, quiero asegurarme de que la API esté **protegida contra ataques comunes** como **CSRF o XSS**, y que los datos estén válidos, de forma que el sistema no sea vulnerable a inyecciones o scripts maliciosos, preservando la integridad y seguridad de la plataforma. | • Revisar la configuración de **CSRF** de Django: dado que la API REST podría usarse desde un frontend separado, habilitar medidas apropiadas (tokens CSRF si se usan sesiones, o asegurarse de usar métodos seguros desde frontend con token JWT, etc.). Si se usan formularios o Django templates en admin, confirmar protección CSRF activa allí.<br>• Implementar **validaciones y sanitización** en los campos de entrada del API: por ejemplo, asegurar que campos de texto no permiten scripts (escapado adecuado), longitudes máximas definidas en modelos y validadas, rechazar entradas fuera de rango, etc.<br>• Realizar pruebas de penetración básicas: intentar enviar payloads maliciosos a la API (ej: scripts <script> en campos texto, SQL injection patterns en campos de consulta) y verificar que o bien se escapan o no producen efectos negativos (el sistema no se rompe ni almacena código ejecutable). | • **Mario F. Fica** – Configuración/Verificación de la protección CSRF en los componentes necesarios (quizá no estrictamente necesario para pure API JSON if using tokens, pero asegurado en panel admin y cualquier formulario); implementación de validaciones adicionales en modelos/serializers para entradas.<br>• **Ulises R. Alejos** – Pruebas manuales de envío de datos maliciosos; inspección del resultado en BD o en respuesta para confirmar que no se almacenan ni ejecutan códigos indeseados; ajuste de código en caso de detectar vulnerabilidades. | • El sistema demuestra ser robusto frente a entradas maliciosas simples: no es posible inyectar código (los campos de texto devueltos por la API muestran, por ejemplo, <script> escapado como texto plano en lugar de ejecutarlo).<br>• Las medidas de seguridad de Django (CSRF tokens, validaciones) están habilitadas donde corresponden (ej: panel admin protegido, API no vulnerable a CSRF porque usa token auth, etc.).<br>• No se encontraron vectores de ataque inmediatos durante las pruebas de penetración básicas, o se mitigaron durante este sprint si se halló alguno. | • Lista de verificaciones de seguridad aplicadas (CSRF, XSS, validaciones de input).<br>• Resultados de pruebas de seguridad (descripción de intentos de ataque y comportamiento observado).<br>• Código actualizado con parches de seguridad (si se hicieron cambios específicos en serializadores o modelos). |

## Sprint 9: Workflows Automáticos y Accesibilidad

**Objetivo del Sprint:** Diseñar los flujos automáticos de obtención y actualización de datos mediante N8N (web scraping/ETL) integrándolos conceptualmente al sistema, y simultáneamente garantizar que el diseño front-end cumpla con estándares de accesibilidad y sea responsive, sentando las bases para una interfaz usable por cualquier persona y en cualquier dispositivo.

| Historias de Usuario | Tareas Técnicas | Responsables | Criterios de Aceptación | Entregables |
| --- | --- | --- | --- | --- |
| **HU1:** Como **ingeniero de scraping**, quiero **diseñar los flujos de trabajo en N8N** que extraerán datos de repuestos y servicios desde los sitios web de interés, de modo que la plataforma CarMatch cuente con datos actualizados automáticamente sin intervención manual. | • Identificar las **fuentes de datos** para repuestos/servicios (por ejemplo, sitios web o APIs de tiendas automotrices en línea).<br>• En N8N, diseñar un **workflow de scraping** por cada fuente: utilizar nodos HTTP Request o específicos para extraer datos (o APIs si existen), luego transformar los datos al formato interno (campos como nombre, precio, proveedor, etc.) y almacenarlos en la base de datos (mediante llamadas a la API Django o escribiendo directo en BD).<br>• Configurar la **planificación** (scheduling) de estos workflows en N8N para que se ejecuten periódicamente (por ejemplo, diariamente o semanalmente), asegurando la actualización continua de la información. | • **Ulises R. Alejos** – Desarrollo de los flujos N8N: scraping de al menos 2 fuentes diferentes de repuestos; uso de nodos de procesamiento para limpiar datos; integración con la BD (vía API REST del backend, probablemente, utilizando el endpoint de creación de precios/repuestos o un webhook dedicado).<br>• **Mario F. Fica** – Revisión de los flujos en términos de eficiencia y carga (evitar scrapings muy pesados en poco tiempo); asegurar que las credenciales o keys necesarias para APIs externas estén seguras en N8N (usando credenciales encripatadas o variables). | • Workflows de N8N creados y funcionando para fuentes definidas: al ejecutarlos manualmente, logran obtener datos reales de repuestos y almacenarlos en la BD de CarMatch con el formato correcto (nuevos repuestos o actualizaciones de precios/historial).<br>• Los flujos están documentados (qué fuente cubren, frecuencia de ejecución) y programados en el scheduler de N8N con la periodicidad acordada.<br>• Si alguna fuente requiere manejo de anti-bot (delays, headers), está contemplado en el diseño del flujo para no ser bloqueado. | • Diagramas o capturas de los workflows N8N diseñados (con anotaciones de los pasos clave).<br>• Descripción de cada flujo: fuente de datos, qué repuestos extrae, frecuencia programada.<br>• Evidencia de ejecución exitosa: datos poblados en BD provenientes de al menos una ejecución de cada flujo. |
| **HU2:** Como **usuario con discapacidad**, quiero que la **interfaz de CarMatch sea accesible**, cumpliendo las pautas WCAG AA, para poder navegar y usar la plataforma sin barreras (por ejemplo, con lectores de pantalla o solo con el teclado). | • Revisar los **diseños de la interfaz** (wireframes/mockups) bajo la óptica de accesibilidad: asegurar contraste de colores suficiente, tamaños de fuente legibles, y estructura semántica planificada (encabezados, etc.).<br>• Preparar el **entorno front-end** (framework Next.js) incorporando librerías o configuraciones para accesibilidad: por ejemplo, usar componentes estándar de HTML en lugar de solo divs, instalar **linters de accesibilidad** o herramientas que ayuden a detectar problemas durante el desarrollo.<br>• Documentar un conjunto de **lineamientos de accesibilidad** a seguir en la implementación (atributos ARIA necesarios, manejo de foco al cambiar de página, textos alternativos para imágenes, etc.) y aplicar correcciones al diseño donde haga falta (p.ej., añadir descripciones a iconos, ajustar contrastes en paleta de colores). | • **Ulises R. Alejos** – Análisis de accesibilidad en la fase de diseño; configuración de Next.js con herramientas (ej: ESLint plugin para JSX a11y); creación de un documento guía de accesibilidad para el proyecto.<br>• **Mario F. Fica** – Validación independiente de los criterios: usando alguna checklist WCAG, simular navega con solo teclado en prototipo o con herramientas de auditoría (lighthouse a11y) para señalar potenciales mejoras; revisar que el equipo las incorpora. | • Se garantiza que los diseños cumplen con **WCAG 2.1 nivel AA**: contraste de colores verificado (ej. ratio >= 4.5:1 en texto normal), fuentes suficientemente grandes, no se usan exclusivamente colores para transmitir información, etc.<br>• Preparación técnica: el proyecto Next.js tiene integrado un linter o configuraciones que advierten sobre problemas de accesibilidad en el código (por ejemplo, alt faltante en imágenes generaría warning).<br>• Existe un documento de pautas de accesibilidad adoptadas por el equipo, que guía a que todo elemento UI tenga su correspondiente atributo/consideración de accesibilidad en la implementación. | • Informe de revisión de accesibilidad de los diseños (con lista de verificación WCAG y cumplimiento).<br>• Configuración del proyecto Next.js con herramientas de accesibilidad (por ejemplo, eslint-plugin-jsx-a11y activado).<br>• Guía/Pautas de Accesibilidad para CarMatch (documento interno con recomendaciones y decisiones tomadas para asegurar inclusión). |
| **HU3:** Como **usuario móvil**, quiero que la plataforma CarMatch tenga un **diseño responsive**, adaptándose correctamente a la pantalla de mi smartphone o tablet, para poder usarla cómodamente desde cualquier dispositivo sin perder funcionalidad. | • Definir puntos de ruptura (**breakpoints**) de diseño responsivo basados en los mockups: identificar cómo se reorganizarán los elementos de la interfaz en tamaños pequeño (móvil), mediano (tablet) y grande (desktop).<br>• Asegurarse de que los componentes UI previstos (menús, listas de resultados, tablas de comparación) tengan un comportamiento responsive: por ejemplo, menú colapsable en móvil, tarjetas en vez de tabla para lista de repuestos en pantallas estrechas, etc. (esto se planifica en CSS/Design stage).<br>• Incluir en la documentación de diseño o en el propio código front inicial los **meta tags** y viewport necesarios para un correcto escalado en dispositivos móviles, y planificar el uso de unidades relativas (rem/em, % en contenedores) en la implementación. | • **Ulises R. Alejos** – Adaptación de diseños a formato móvil: creación de versiones móviles de los mockups si no se tenían, o notas sobre cómo cambia la disposición; preparar estilos base responsive (ej: usar un framework CSS responsive o diseño propio con flex/grid adaptable).<br>• **Mario F. Fica** – Verificación teórica de que todos los elementos serán accesibles en pantallas pequeñas (nada crítico queda oculto sin manera de acceder); revisión de la configuración de Next.js para mobile (meta viewport tag en páginas \_document or Head). | • Los diseños de CarMatch contemplan su uso en distintos tamaños: existe claridad sobre cómo se verá en móvil vs desktop (p. ej., botones suficientemente grandes en móvil, columnas que se apilan en lugar de tabla, menú hamburguesa implementado para navegación móvil, etc.).<br>• El front-end tendrá la meta etiqueta de viewport correcta (<meta name="viewport" content="width=device-width, initial-scale=1">) y usará medidas fluidas para acomodar distintos anchos.<br>• Se documentó en los entregables de diseño o notas del sprint la estrategia responsive para futuros desarrolladores. | • Mockups o esquemas de la interfaz en versión móvil (si corresponde) de las principales pantallas.<br>• Anotaciones de responsive design (lista de breakpoints definidos, cambios de layout descriptos).<br>• Configuración base del proyecto para responsive (meta viewport en el código fuente). |

## Sprint 10: Integración y Monitoreo

**Objetivo del Sprint:** Integrar completamente los componentes desarrollados – conectando los flujos automáticos de N8N con el backend Django – y desarrollar herramientas de administración y monitoreo para la plataforma. Esto implica crear un panel de administración de CarMatch para gestionar fuentes de datos y usuarios, así como establecer un sistema de monitoreo y alertas que supervise la salud de la aplicación.

| Historias de Usuario | Tareas Técnicas | Responsables | Criterios de Aceptación | Entregables |
| --- | --- | --- | --- | --- |
| **HU1:** Como **administrador de CarMatch**, quiero que los **flujos automáticos (N8N) estén integrados con el backend** del sistema, de forma que cada vez que se obtengan nuevos datos o actualizaciones mediante scraping, estos se **ingresen automáticamente** en la base de datos a través de la API, manteniendo la información del sistema siempre actualizada. | • Configurar **webhooks o conectores** en N8N para que, al finalizar un flujo de scraping/ETL, envíen los datos obtenidos al backend Django: puede ser invocando endpoints de la API (por ejemplo, un POST a /api/repuestos o a /api/precios para actualizar valores).<br>• Asegurar que las llamadas desde N8N al API utilizan la autenticación necesaria (por ejemplo, usando un token de servicio embebido en las peticiones) para ser aceptadas por el backend seguro.<br>• Probar la integración end-to-end: ejecutar un flujo N8N real (o simulado) que traiga uno o dos nuevos registros y observar que esos registros aparecen en la base de datos del backend (verificados vía admin o API GET) inmediatamente después. | • **Ulises R. Alejos** – Implementación de la lógica en N8N para llamadas a la API (uso de nodo HTTP Request en N8N con método POST/PUT hacia Django); configurar tokens o credenciales seguras en N8N para la autenticación en la API.<br>• **Mario F. Fica** – Ajustes en el lado del backend si necesarios (por ejemplo, crear endpoints específicos para recibir lotes de datos de N8N, o relajar CSRF para ciertas rutas API ya que vendrán de un agente confiable); verificación de que no hay datos duplicados ni conflictos tras la inserción automática. | • Los flujos N8N ahora **alimentan directamente el sistema**: al correr un flujo, se observa que los datos viajan hasta el backend sin intervención manual y se almacenan correctamente.<br>• Autenticación entre N8N y backend confirmada: las peticiones de N8N llevan un token válido y solo mediante él se permite la inserción (no abierto al público).<br>• No se generan registros inconsistentes: si el repuesto ya existía, se actualiza su precio/historial, si es nuevo, se crea un nuevo registro, todo según la lógica esperada. | • Workflows N8N actualizados con nodos de integración a API (captura o documentación de estos nodos configurados).<br>• Token o credencial de servicio para N8N (documentada de manera segura).<br>• Evidencia de nuevos datos en el sistema tras ejecución de flujos (por ejemplo, logs de Django mostrando recepción de POST de N8N, o captura de admin mostrando nuevos ítems). |
| **HU2:** Como **administrador**, quiero disponer de un **Panel de Administración interno** en la plataforma (más allá del Django admin genérico) para gestionar las **fuentes de datos, usuarios y logs** del sistema CarMatch, con una interfaz amigable, de modo que pueda supervisar y configurar la plataforma sin interactuar directamente con la base de datos o herramientas de desarrollo. | • Crear en el frontend (o como parte de Django admin custom) un **módulo de Panel Admin CarMatch**: por ejemplo, una sección donde se liste las **fuentes de datos** integradas (sitios web scrapeados), permitiendo activar/desactivar alguna o ver fecha de última actualización; lista de **usuarios** registrados en la plataforma (si hubiese gestión de usuarios finales) con posibilidad de bloquear/permitir; y **logs** de actividad o de flujos (p. ej., resultados recientes de scraping, alertas de errores en flujos, etc.).<br>• Esto puede ser implementado como vistas adicionales en Django (si opta por interfaz simple server-rendered) o como páginas protegidas en el frontend Next.js que consuman endpoints admin específicos.<br>• Asegurar que el acceso a este panel esté protegido (solo cuentas admin) y que las operaciones (ej.: desactivar una fuente) tengan efecto en el sistema (ej., N8N puede leer un flag de "fuente activa/inactiva"). | • **Mario F. Fica** – Desarrollo de la lógica de backend para el panel admin: modelos o settings para fuentes de datos (ej., tabla Fuente con campos nombre, url, activa/inactiva), vistas o endpoints para actualizar dichos estados; integración con N8N (posiblemente mediante API de N8N o variable global) para no ejecutar flujos desactivados.<br>• **Ulises R. Alejos** – Desarrollo de la interfaz del panel: si es en frontend Next, construir página admin con secciones de Fuentes, Usuarios, Logs, consumiendo los endpoints preparados; si es en Django templates, crear las plantillas correspondientes con estilos básicos. | • Existe un panel administrativo dedicado de CarMatch: el administrador puede ver un listado de fuentes de datos configuradas y cambiar su estado (ej.: "Pausar scraping de X tienda" y N8N no la ejecutará más si está off).<br>• Se puede visualizar información de usuarios (al menos los administrativos o de prueba) y los logs recientes del sistema (ej. últimos flujos ejecutados, último ingreso de un admin, etc.), lo que facilita la supervisión del sistema en ejecución.<br>• Todas estas funcionalidades están restringidas correctamente: solo accesibles tras autenticación como admin (verificado que un usuario no logueado o sin permisos no puede cargar esas vistas). | • Código fuente del módulo de administración (vistas Django o páginas Next.js, controladores, etc.).<br>• Capturas del Panel de Administración en funcionamiento (lista de fuentes de datos con sus estados, etc.).<br>• Manual breve de uso del panel (explicando cómo activar/desactivar fuentes, interpretar logs). |
| **HU3:** Como **operador de TI**, quiero contar con un **sistema de monitoreo y alertas** que supervise la disponibilidad de los componentes de CarMatch (servidor, base de datos, servicio N8N, API) y notifique al equipo en caso de fallos o comportamientos anómalos, para poder reaccionar rápidamente ante incidentes y minimizar tiempos de inactividad. | • Configurar **monitoreo del servidor**: instalar/agregar herramientas que midan recursos (CPU, RAM, uso de disco) y disponibilidad (uptime) del servidor; podría usarse un servicio externo (Pingdom, UptimeRobot para pings HTTP) y/o instalar *Prometheus + Grafana* para métricas internas si lo amerita el alcance, o algo más sencillo como scripts con cron que registren datos.<br>• Configurar **monitoreo de la aplicación**: habilitar logging detallado de la aplicación Django y N8N, y posiblemente integrar con un servicio de alertas. Ejemplo: usar los *hooks* de error en N8N para que envíen correo si un flujo falla; en Django, configurar email o notificación en caso de error 500 en producción.<br>• Establecer **alertas**: definir qué situaciones generan alerta (ej.: si servidor no responde a ping en 5 min, si uso CPU > 90% por 10 min, si un flujo N8N falla consecutivamente, etc.), y configurar envíos de correo electrónico o mensajes (por ej., a correo de los desarrolladores) cuando ocurran. | • **Ulises R. Alejos** – Configuración de N8N para alertas de fallos (nodos de error-handling en workflows que manden email al equipo en caso de excepción); utilización de un servicio web para monitorear la URL de la API periódicamente.<br>• **Mario F. Fica** – Implementación de herramientas de monitoreo en el servidor (posiblemente instalación de *Netdata* para métricas del servidor o Cloud Monitoring de GCP; scripts custom para chequeos); creación de una cuenta de correo o canal donde recibir las notificaciones; pruebas de que las alertas se envían correctamente. | • El sistema cuenta con supervisión activa: por ejemplo, un servicio de terceros verifica cada 5 minutos que la API responde y notifica si cae; las métricas de sistema están recolectándose (se puede ver un dashboard simple de CPU/memoria, o al menos logs de estos parámetros, para diagnóstico futuro).<br>• Se configuraron alertas efectivas: simulado un escenario de falla (detener el servicio intencionalmente o forzar un error), se recibió la notificación correspondiente en el canal definido (correo o mensaje), validando el mecanismo.<br>• La presencia de monitoreo y alertas permite mayor confianza en la operación continua, con capacidad de respuesta rápida por parte del equipo en caso de problemas. | • Descripción de la solución de monitoreo implementada (herramientas y servicios utilizados).<br>• Captura de un dashboard o log de métricas del sistema en tiempo real (si aplica).<br>• Ejemplo de alerta recibida (correo o captura de notificación) ante una falla simulada. |

## Sprint 11: Implementación de Interfaz Frontend

**Objetivo del Sprint:** Desarrollar la interfaz de usuario de CarMatch utilizando Next.js, implementando las pantallas principales de la aplicación (búsqueda, resultados, detalle de repuesto, etc.) según los diseños realizados, y asegurando que la interfaz sea funcional de manera independiente con datos simulados, quedando lista para integrar con la API.

| Historias de Usuario | Tareas Técnicas | Responsables | Criterios de Aceptación | Entregables |
| --- | --- | --- | --- | --- |
| **HU1:** Como **usuario final**, quiero una **página de inicio/búsqueda** en CarMatch donde pueda ingresar criterios (nombre de repuesto, categoría, etc.) y lanzar la búsqueda, para comenzar el proceso de encontrar el repuesto que necesito de manera sencilla. | • Implementar la **página de inicio con barra de búsqueda**: en Next.js, crear la página principal que incluya un campo de búsqueda y filtros básicos (si aplica), y un botón o acción para enviar la consulta.<br>• Agregar elementos de UI amigables: instrucciones breves, ejemplos de búsqueda o un diseño atractivo que refleje la marca CarMatch.<br>• Programar la lógica de captura del input del usuario y redirección a la página de resultados (por ahora, sin consumir API real, pero dejando preparado para hacerlo). | • **Ulises R. Alejos** – Desarrollo de la página index.js de Next.js con el formulario de búsqueda; estilo CSS responsivo para que se vea bien en distintos tamaños; implementación de estado local para manejar el input de búsqueda.<br>• **Mario F. Fica** – Revisión del código y aseguramiento de buenas prácticas (componentes separados si necesario, cumplimiento de pautas de accesibilidad en el formulario); testeo básico de la UI en diferentes navegadores/tamaños. | • La página de inicio se carga correctamente en un navegador, mostrando el branding de CarMatch y un cuadro de búsqueda central.<br>• Al ingresar un texto y activar la búsqueda, la aplicación navega a la pantalla de resultados (pasando el término de búsqueda en la ruta o estado).<br>• Diseño validado: elementos visibles y bien distribuidos tanto en desktop como en móvil, con textos e instrucciones claras para el usuario. | • Código fuente de la página de Inicio (React/Next.js).<br>• Captura de pantalla de la pantalla de búsqueda en desktop y móvil. |
| **HU2:** Como **usuario**, quiero ver una **lista de resultados** con los repuestos que coinciden con mi búsqueda, mostrando para cada repuesto información resumida (nombre, quizás imagen si disponible, precio mínimo encontrado), para poder comparar múltiples opciones de un vistazo. | • Implementar la **página de resultados de búsqueda**: en Next.js, crear una página que reciba el término de búsqueda (ej. /resultados?query=X) y muestre una lista de resultados en forma de tarjetas o lista.<br>• Cada **tarjeta de resultado** debe mostrar nombre del repuesto, una pequeña descripción o categoría, y opcionalmente el precio más bajo disponible o número de ofertas (según lo que la API puede proveer). También incluir un botón o link "Ver detalle" que lleve a la página de detalle de ese repuesto.<br>• Por ahora, usar datos simulados/mock para la lista (ej., un array estático de objetos JSON dentro del código o fetch a un JSON local) hasta integrar con la API real. | • **Ulises R. Alejos** – Desarrollo del componente de lista de resultados y de cada tarjeta; aplicar estilos CSS para un layout responsive (p.ej., grid flexible que se adapte a pantallas más pequeñas con tarjetas en columnas).<br>• **Mario F. Fica** – Verificación de la navegabilidad (que el query se capture de la URL, se muestre, y que al hacer clic en detalle lleve a la ruta de detalle correspondiente); revisión de performance básica (lista renderiza eficientemente) y de que los datos de ejemplo representan adecuadamente lo que vendrá del backend. | • Al realizar una búsqueda desde la página principal, el usuario es dirigido a la página de resultados y ve poblada una lista de repuestos coincidentes (tomados de datos simulados).<br>• El diseño de la lista es limpio y fácil de leer: cada item destaca el nombre del repuesto y su precio más atractivo, con un enlace claro al detalle.<br>• Si la búsqueda devuelve cero resultados (caso simulado), se muestra un mensaje indicando "sin resultados" en lugar de una lista vacía. | • Código fuente de la página de Resultados y componentes asociados (tarjeta de repuesto).<br>• Captura de pantalla de un ejemplo de resultados con varios items listados. |
| **HU3:** Como **usuario**, quiero ver la **pantalla de detalle** de un repuesto seleccionado, con la lista completa de precios por proveedor y demás detalles (descripción, especificaciones), para poder decidir dónde comprar. | • Implementar la **página de detalle de repuesto** en Next.js: página dinámica (ej: /repuesto/[id].js) que muestre información detallada. Incluir campos como nombre, descripción amplia, quizá imagen si está en alcance, y una sección listando todas las ofertas: por cada proveedor, su nombre, precio, y link si existiera (podría abrir sitio del proveedor, futuro alcance).<br>• Añadir también en esta página algún elemento comparativo clave, por ejemplo resaltar el precio mínimo y el proveedor que lo ofrece, y quizás graficar historial de precios si hubiera (aunque sea textual ahora, dependiendo del alcance de datos).<br>• Utilizar datos de ejemplo para rellenar la página: crear un objeto simulado con detalle y lista de precios. Posteriormente, prever integración con la llamada real al endpoint de detalle. | • **Ulises R. Alejos** – Desarrollo del componente de detalle, estructurando la información en secciones claras (info general del repuesto arriba, tabla o lista de precios abajo); estilado apropiado, incluyendo formato de moneda para precios y quizás colores para destacar el mejor precio.<br>• **Mario F. Fica** – Revisión de la exactitud de los datos mostrados vs lo que la API entregaría (para facilitar la futura integración: e.g., usar mismas nomenclaturas de campos que la API); prueba de la navegación (desde la lista de resultados al detalle y quizás un botón "volver" a resultados). | • La página de detalle se carga (usando un id de repuesto ficticio) y presenta toda la información relevante del repuesto en un formato organizado.<br>• La lista de precios por proveedor es clara y ordenada, permitiendo ver fácilmente quién vende más barato, etc. No hay datos duros sin descripción (todo campo mostrado está identificado, ej. "Proveedor X: $Y").<br>• Se validó que en mobile la tabla/lista de precios es desplazable o adaptativa para no desbordar la pantalla. | • Código fuente de la página de Detalle de repuesto.<br>• Captura de pantalla de la página de detalle con datos de ejemplo.<br>• (Opcional) Captura de la vista mobile del detalle para evidenciar responsive design. |
| **HU4:** Como **desarrollador**, quiero preparar la **integración del frontend con la API** (aunque sea básica en esta etapa), para comprobar que la estructura desarrollada puede conectarse al backend y obtener datos reales, acercándonos a un producto funcional completo. | • Configurar en Next.js llamadas fetch hacia la API del backend para los datos de búsqueda y detalle, usando getServerSideProps o fetch en useEffect según se decida: quizá inicialmente apuntando a un endpoint mock JSON si el backend no está desplegado aún, pero estructurar el código de forma que sea fácil cambiar la URL base de la API.<br>• Probar una integración parcial: por ejemplo, habilitar que la búsqueda en vez de datos estáticos haga un fetch a un JSON local (simulando latencia) o, si es viable, a la API real en desarrollo (requiere backend corriendo y CORS habilitado).<br>• Manejar casos de error en fetch (mostrar mensajes de error al usuario si la API no responde). | • **Ulises R. Alejos** – Implementación del uso de fetch para obtener datos en las páginas de resultados y detalle (dejando la URL de la API configurable); asegurar tratar respuestas JSON correctamente y mapear a props de los componentes.<br>• **Mario F. Fica** – Ajustes necesarios en backend (habilitar CORS para el origen del front dev, por ejemplo, con Django Cors Headers); pruebas integrando realmente si es posible: correr backend local y front local, realizar una búsqueda real y ver resultados en la interfaz. | • El frontend tiene el código listo para consumir la API: hay funciones definidas para obtener datos de búsqueda y detalle desde endpoint (actualmente contra un mock o entorno dev) que devuelven la estructura esperada.<br>• Se comprobó que cambiando una bandera, la aplicación podría apuntar al API real y funcionaría (lo cual sienta las bases para el siguiente paso de integración completa).<br>• Los posibles fallos de comunicación están manejados: si la API no responde, el usuario vería un mensaje ("Error al obtener datos, intente más tarde") en lugar de la app colgarse. | • Fragmentos de código donde se realizan las llamadas a API (indicando endpoints, etc.).<br>• Documentación/comentarios en el código sobre cómo configurar la URL de la API (por ejemplo, variable de entorno NEXT\_PUBLIC\_API\_URL).<br>• (Si se probó) Captura de la interfaz obteniendo datos de la API real en entorno de pruebas. |
|  |  |  |  |  |

[[1]](https://www.atlassian.com/es/agile/scrum/sprints#:~:text=%26quot%3BCon%20scrum%2C%20un%20producto%20se,producto%20de%20Jira%20de%20Atlassian) Sprints de scrum Article | Agile

<https://www.atlassian.com/es/agile/scrum/sprints>

[[2]](https://www.innovaciondigital360.com/cio/metodologia-scrum-concepto-beneficios-e-implementaciones/#:~:text=En%20Scrum%2C%20el%20trabajo%20se,producto%20para%20trabajar%20en%20ellos) Metodología Scrum: concepto, beneficios e implementaciones | InnovaciónDigital360

<https://www.innovaciondigital360.com/cio/metodologia-scrum-concepto-beneficios-e-implementaciones/>